

Language-Based Abstraction Refinement for Hybrid System Verification^{*}

Felix Klaedtke¹, Stefan Ratschan², and Zhikun She³

¹ ETH Zurich, Computer Science Department, Zurich, Switzerland

² Institute of Computer Science, Czech Academy of Sciences, Prague, Czech Republic

³ Max-Planck-Institut für Informatik, Saarbrücken, Germany

Abstract. The standard counterexample-guided abstraction-refinement (CEGAR) approach uses finite transition systems as abstractions of concrete systems. We present an approach to represent and refine abstractions of infinite-state systems that uses regular languages instead of finite transition systems. The advantage of using languages over transition systems is that we can store more fine-grained information in the abstraction and thus reduce the number of abstract states. Based on this language-based approach for CEGAR, we present new abstraction-refinement algorithms for hybrid system verification. Moreover, we evaluate our approach by verifying various non-linear hybrid systems.

1 Introduction

The verification of infinite-state systems is often done by abstracting the concrete system to an abstract finite state system. The abstract system over-approximates the concrete one, i.e., it includes all the behaviors of the concrete system. However, it can include behaviors that do not correspond to behaviors of the concrete system. Such behaviors are called *spurious*. In the *counterexample-guided abstraction-refinement* (CEGAR) paradigm [6] one usually starts with a very coarse abstraction and uses spurious counterexamples to iteratively refine the abstraction until verification reveals whether or not the property in question holds. CEGAR has been successfully used for verifying many different classes of infinite-state systems. For instance, CEGAR has been adopted and used for the verification of hybrid systems [1, 5, 17]. In this paper, we focus on the safety verification of hybrid systems. However, in principle, the presented method also applies to other infinite-state systems.

The standard CEGAR approach uses finite transition systems as abstractions of the concrete systems. The use of transition systems has the following disadvantages. Assume that we have abstract states a, b, c with transitions from a to b and from b to c (we write $a \rightarrow b$ and $b \rightarrow c$ with the obvious meaning). Even if $a \rightarrow b$ and $b \rightarrow c$ are not spurious, it is not necessarily the case that $a \rightarrow b \rightarrow c$ is not spurious. If $a \rightarrow b \rightarrow c$ is spurious, we can refine the abstraction by splitting

^{*} This work was partly supported by the German Research Foundation (DFG) and the Swiss National Science Foundation (SNF).

at least one of the abstract states such that the concrete system is reflected more closely. Splitting abstract states in such a way that the spurious sequence $a \rightarrow b \rightarrow c$ does not appear in the abstraction anymore can be difficult. Furthermore, the splitting of an abstract state introduces additional abstract states and it can introduce new spurious counterexamples. This can lead to a state space explosion of the abstract systems. For instance, in the worst case, the number of abstract states can double in each refinement step of the CEGAR algorithm for hybrid system [5]. One method [10] to address this problem splits abstract states based on graph-topological properties of the finite transition system.

As a means to reduce the number of abstract states we present an alternative to finite transition systems as abstractions in the CEGAR paradigm. Namely, we use languages to over-approximate the behaviors of concrete systems. In our approach, a language contains at least the sequences of abstract states that correspond to runs of the concrete system. The use of languages allows us to store more fine-grained information in the abstraction, and to refine the abstraction, we do not necessarily need to split abstract states. For instance, in the example above, if the sequence $a \rightarrow b \rightarrow c$ of abstract states is spurious, we can remove all sequences from the language that contain $a \rightarrow b \rightarrow c$ as a subsequence. For checking whether a sequence of abstract states is spurious, we present an extension of the method that is used in the hybrid system verifier HSOLVER [18, 19]. As in the standard CEGAR approach, we also allow to split abstract states to refine the abstraction. We use deterministic finite automata to represent the languages and present automata operations for refining the abstractions. We evaluate our language-based approach on various non-linear hybrid systems using a prototype implementation. Our experiments demonstrate that the use of languages often reduces the number of splittings significantly.

To our knowledge, the use of languages for representing and refining abstractions in the context of CEGAR for hybrid system verification is novel. The observation that regular languages/automata can be used to improve abstraction-refinement algorithms already appeared in [7]. However, [7] mainly focuses on the completeness of abstraction-refinement algorithms for infinite-state systems in general and does not provide concrete algorithms that beneficially make use of finite state automata in practice. Related to our work with respect to languages for representing the behavior of hybrid systems are [2] and [3, 4]. Roughly speaking, Asarin et al. [2] use languages to show decidability of the reachability problem of a certain class of hybrid systems, and Brihaye et al. [3, 4] use words of languages to construct bisimilar finite transition systems for so-called o-minimal hybrid systems. Other methods for verifying systems with continuous dynamics by over-approximating their behaviors, and automatically and incrementally refining the abstractions appeared recently in [9, 14]. In [9], rectangular hybrid automata [12] are used to over-approximate the behavior, and [14] uses finite transition systems. Both methods only apply to hybrid systems with restricted (linear, multi-affine) continuous dynamics.

We proceed as follows. In §2, we give the definition of hybrid systems that we use in this paper and the verification problem that we address. In §3, we

describe our method for using languages as abstractions. In §4, we give details on how we manipulate the languages that represent our abstractions and in §5, we present methods to check if sequences of abstract states are spurious. In §6, we report on experimental results. Finally, in §7, we draw conclusions.

2 Verification of Hybrid Systems

Hybrid systems are systems with continuous and discrete state variables. In this paper, we use the mathematical model from [17, 19], which we briefly recall in this section. It captures many relevant classes of hybrid systems, and many other formalisms for hybrid systems in the literature are special cases of it.

We use a set S to denote the modes of a hybrid system, where S is finite and nonempty. $I_1, \dots, I_k \subseteq \mathbb{R}$ are compact intervals over which the continuous variables of a hybrid system range. Φ denotes the state space of a hybrid system, i.e., $\Phi = S \times I_1 \times \dots \times I_k$. Note that it is not a severe practical restriction that the continuous variables have to range over compact intervals because, in most applications, the variable ranges are bounded and engineers use their experience to choose reasonable values for the interval bounds.

Definition 1. A hybrid system H is a tuple $(Flow, Jump, Init, Unsafe)$, where $Flow \subseteq \Phi \times \mathbb{R}^k$, $Jump \subseteq \Phi \times \Phi$, $Init \subseteq \Phi$, and $Unsafe \subseteq \Phi$.

Informally speaking, the predicate $Init$ specifies the initial states of a hybrid system $H = (Flow, Jump, Init, Unsafe)$ and $Unsafe$ the states that should not be reachable from an initial state. The relation $Flow$ specifies how the system may develop continuously by relating each state to the corresponding derivative, and $Jump$ specifies how H may change states discontinuously by relating each state to its successor states. Formally, the behavior of H is defined as follows:

Definition 2. A flow of length $l \geq 0$ in a mode $s \in S$ is a function $r : [0, l] \rightarrow \Phi$ such that the projection of r to its continuous part is differentiable and for all $t \in [0, l]$, the mode of $r(t)$ is s . A trajectory of H is a sequence of flows r_0, \dots, r_p of lengths l_0, \dots, l_p such that for all $i \in \{0, \dots, p\}$,

- (i) if $i > 0$ then $(r_{i-1}(l_{i-1}), r_i(0)) \in Jump$, and
- (ii) if $l_i > 0$ then $(r_i(t), \dot{r}_i(t)) \in Flow$, for all $t \in [0, l_i]$, where \dot{r}_i is the derivative of the projection of r_i to its continuous component.

In the following, we denote the length of a flow r by $|r|$. Moreover, we address the state $r_i(t)$ in a trajectory by the pair (i, t) . This naturally gives us a (lexicographical) order \preceq on the states in a trajectory.

Definition 3. A (concrete) counterexample of H is a trajectory r_0, \dots, r_p of H such that $r_0(0) \in Init$ and $r_p(|r_p|) \in Unsafe$. H is safe if it does not have a counterexample.

We use the following constraint language to describe hybrid systems. The variable s ranges over S and the variables x_1, \dots, x_k range over I_1, \dots, I_k , respectively. In addition, to denote the derivatives of x_1, \dots, x_k we use the vari-

ables $\dot{x}_1, \dots, \dot{x}_k$ that range over \mathbb{R} ,⁴ and to denote the targets of jumps, we use the primed variables s', x'_1, \dots, x'_k that range over S and I_1, \dots, I_k , respectively. Constraints are arbitrary Boolean combinations of equalities and inequalities over terms that may contain function symbols like $+$, \times , \exp , \sin , and \cos .

We assume in the remainder of the text that a hybrid system is described by our constraint language. That means, the flows of a hybrid system are given by a constraint $flow(s, x_1, \dots, x_k, \dot{x}_1, \dots, \dot{x}_k)$, the jumps are given by a constraint $jump(s, x_1, \dots, x_k, s', x'_1, \dots, x'_k)$, the initial states are given by a constraint $init(s, x_1, \dots, x_k)$, and a constraint $unsafe(s, x_1, \dots, x_k)$ describes the unsafe states. To simplify notation, we do not distinguish between a constraint and the set it represents.

Example 1. For illustrating the above definitions, consider the following simple hybrid system. The hybrid system has two modes m_1, m_2 and the continuous variables x_1 and x_2 , where x_1 ranges over the interval $[0, 2]$ and x_2 over $[0, 1]$, i.e., $\Phi = \{m_1, m_2\} \times [0, 2] \times [0, 1]$.

The set of initial states are given by the constraint $init(s, x_1, x_2) = (s = m_1 \wedge x_1 = 0 \wedge x_2 = 0)$ and $unsafe(s, x_1, x_2) = (x_1 > 1 \wedge x_2 = 1)$ describes the unsafe states. The hybrid system can switch modes from m_1 to m_2 if $x_2 = 1$, i.e., $jump(s, x_1, x_2, s', x'_1, x'_2) = (s = m_1 \wedge x_2 = 1 \rightarrow s' = m_2 \wedge x'_1 = x_1 \wedge x'_2 = x_2)$.

The continuous behavior is quite simple: In mode m_1 , the values of the variables x_1, x_2 change with slope 1; in mode m_2 , the slope of x_1 is 1 and x_2 has slope -1 . For a flow in mode m_1 , the constraint $0 \leq x_1 \leq 1$ must hold and in mode m_2 , $1 \leq x_1 \leq 2$ must hold. The corresponding flow constraint is

$$flow(s, x_1, x_2, \dot{x}_1, \dot{x}_2) = (s = m_1 \rightarrow \dot{x}_1 = 1 \wedge \dot{x}_2 = 1 \wedge 0 \leq x_1 \leq 1) \wedge (s = m_2 \rightarrow \dot{x}_1 = 1 \wedge \dot{x}_2 = -1 \wedge 1 \leq x_1 \leq 2).$$

Note that the constraint $0 \leq x_1 \leq 1$ in $flow$ forces a jump from mode m_1 to m_2 if x_1 becomes 1. Otherwise, the system makes no progress. In general, an invariant that has to hold in a mode can be modeled by formulating a flow constraint that does not allow a continuous behavior in certain parts of the state space.

A trajectory of the hybrid system starting from the initial state $(m_1, (0, 0))$ is r_0, r_1 , where the flows $r_1, r_2 : [0, 1] \rightarrow \Phi$ are given by

$$r_0(t) = (m_1, (t, t)) \quad \text{and} \quad r_1(t) = (m_2, (t + 1, 1 - t)).$$

Obviously, this hybrid system is safe.

The verification problem we address in the following is to prove automatically that a hybrid system is safe. Note that the reachability problem for hybrid systems is undecidable [12]. So, it is only possible to give semi-algorithms for this problem that (hopefully) terminate on many relevant problem instances. In this paper, our focus is on verifying that a given hybrid system is safe and on efficiency for instances of practical relevance. For the sake of readability, we use in the remainder of the paper the term ‘‘algorithm’’ liberally in the sense that we do not require that an algorithm terminates for every input instance.

⁴ The dot does not have any special meaning here; it is only used to distinguish dotted from undotted variables.

3 Language-Based Abstractions

In this section, we present our language-based approach of abstracting hybrid systems and refining abstractions. For the remainder of the text, let $H = (\text{Flow}, \text{Jump}, \text{Init}, \text{Unsafe})$ be a hybrid system.

As in many abstraction techniques we cover H 's state space Φ by using finitely many subsets of Φ , which we call *regions*. We identify the regions by naming them with the symbols of an alphabet Ω and a function γ that assigns to every element of Ω a region and that *covers* Φ , i.e., $\Phi = \bigcup_{b \in \Omega} \gamma(b)$.

Intuitively speaking, we use a word $b_1 \dots b_n \in \Omega^+$ to represent all the trajectories of H that pass in the order of the occurrences of the symbols through the regions $\gamma(b_1), \dots, \gamma(b_n)$.

Definition 4. Let $w = b_1 \dots b_n \in \Omega^+$ with $n \geq 1$. A trajectory r_1, \dots, r_p of H follows w if there exists a non-decreasing sequence $(i_1, t_1), \dots, (i_{n+1}, t_{n+1})$ with respect to the ordering \preceq such that

- (i) $i_1 = 1$ and $t_1 = 0$,
- (ii) $0 \leq t_j \leq |r_{i_j}|$, for all $j \in \{1, \dots, n\}$,
- (iii) $i_{n+1} = p$, $t_{n+1} = |r_p|$, and $r_p(|r_p|) \in \gamma(b_n)$, and
- (iv) for all $j \in \{1, \dots, n\}$ and all (i', t') , if $(i_j, t_j) \preceq (i', t') \prec (i_{j+1}, t_{j+1})$ then $r_{i'}(t') \in \gamma(b_j)$.

Note that different trajectories can follow the same word and a trajectory can follow different words.

Example 2. Consider again the hybrid system from Example 1. Assume that its state space is covered by $\gamma : \{a, b, c, d\} \rightarrow \Phi$ with $\gamma(a) = \{m_1\} \times [0, \frac{1}{2}] \times [0, 1]$, $\gamma(b) = \{m_1\} \times [\frac{1}{2}, 1] \times [0, 1]$, $\gamma(c) = \{m_2\} \times [1, \frac{3}{2}] \times [0, 1]$, and $\gamma(d) = \{m_2\} \times [\frac{3}{2}, 2] \times [0, 1]$. Let r_1, r_2 be the trajectory from $(m_1, (\frac{1}{4}, \frac{1}{4}))$ to $(m_2, (\frac{7}{4}, \frac{1}{4}))$ with

$$r_1(t) = (m_1, (\frac{1}{4} + t, \frac{1}{4} + t)) \quad \text{and} \quad r_2(t) = (m_2, (1 + t, 1 - t)),$$

for $t \in [0, \frac{3}{4}]$. The trajectory r_1, r_2 follows the word $abcd$, since the sequence $(1, 0), (1, \frac{1}{4}), (1, \frac{3}{4}), (2, \frac{1}{2}), (2, \frac{3}{4})$ is non-decreasing with respect to \preceq and satisfies the conditions (i)–(iv) in Definition 4.

Definition 5. We call a word $w \in \Omega^+$ *prefix-spurious* (*suffix-spurious*, *respectively*) if there is no trajectory r_1, \dots, r_p that follows w and starts in *Init*, i.e., $r_1(0) \in \text{Init}$ (*ends in Unsafe*, i.e., $r_p(|r_p|) \in \text{Unsafe}$, *respectively*). Moreover, we call w *midfix-spurious* if there is no trajectory that follows w . For the sake of brevity, we use the following abbreviations: *p-spurious* for *prefix-spurious*, *s-spurious* for *suffix-spurious*, and *m-spurious* for *midfix-spurious*.

Note that if a word is *m-spurious* then it is *p-spurious* and *s-spurious*. However, if a word is *p-spurious* or *s-spurious*, we cannot conclude that it is *m-spurious*.

We assume that we can check if a word is *p-spurious*, *s-spurious*, or *m-spurious*. Since the reachability problem for hybrid systems is undecidable [12] such a check has to be over-approximating, i.e., it returns either “spurious” or “don’t know” (see §5 for more details on mechanizing such a check). Additionally,

Algorithm 1 Language-based abstraction-refinement algorithm

Input: hybrid system H

- 1: $\Omega \leftarrow \{b\}$, where b is some symbol
- 2: $L \leftarrow \Omega^+$
- 3: let $\gamma : \Omega \rightarrow 2^\Phi$ be the function with $\gamma(b) = \Phi$
- 4: **while** $L \neq \emptyset$ **do**
- 5: **for all** $b \in \Omega$ with $L \cap \{bv \mid v \in \Omega^*\} \neq \emptyset$ and b is p-spurious **do**
- 6: $L \leftarrow L \setminus \text{ext}(\{bv \mid v \in \Omega^*\})$
- 7: **end for**
- 8: **for all** $b \in \Omega$ with $L \cap \{ub \mid u \in \Omega^*\} \neq \emptyset$ and b is s-spurious **do**
- 9: $L \leftarrow L \setminus \text{ext}(\{ub \mid u \in \Omega^*\})$
- 10: **end for**
- 11: **for all** $w \in C(\Omega)$ with $|w| = \text{maxlen}$ and $L \cap \{uvw \mid u, v \in \Omega^*\} \neq \emptyset$ **do**
- 12: $L \leftarrow \text{removeMidfixSpurious}(H, L, w)$
- 13: **end for**
- 14: split region $\gamma(a)$ in regions U and V , for some $a \in \Omega$
- 15: $\Omega \leftarrow \Omega \cup \{b\}$, where b is a fresh symbol
- 16: $L \leftarrow L^{a \sim b}$
- 17: update γ , i.e., $\gamma(a) = U$, $\gamma(b) = V$, and $\gamma(c)$ is not altered for $c \in \Omega \setminus \{a, b\}$
- 18: **end while**

such a check may detect that a trajectory follows the given word, and returns “not spurious” in that case. This additional information can be used to further optimize the following algorithms. However, in order to keep the exposition simple, we do not consider this further.

To verify that the hybrid system H is safe, we use languages L such that every counterexample of H follows a word in L . We iteratively choose words $w \in L$ and check whether w is p-spurious, s-spurious, or m-spurious. Assume that $L \subseteq \Omega^+$, where Ω is some alphabet. If we can show that w is p-spurious, we remove the words of the form wv with $v \in \Omega^*$ from L , if we can show that w is s-spurious, we remove the words uw with $u \in \Omega^*$ from L , and if we can show that w is m-spurious, we remove the words uvw with $u, v \in \Omega^*$ from L . If the language becomes empty, we know that H is safe. In addition to the checks if a word is p-spurious, s-spurious, or m-spurious, we can split a region. For reflecting a split, we add a new symbol b to Ω , update γ , and add certain words to L .

Various details are left open in the description above. In the following, we provide the details by describing the language-based counterexample-abstraction-refinement algorithm in Alg. 1.

Note that the language L can be infinite. So, instead of checking all words in L in an iteration (lines 4–18 of Alg. 1), we only check the words up to a given maximal length maxlen , which we fix in advance. Moreover, we restrict ourselves to *contracted* words, i.e., words in which a subword of the form bb with $b \in \Omega$ does not occur. The reason for this is that if, e.g., $ubbv$ is m-spurious then ubv is also m-spurious. Let $C(\Omega)$ denote the set of contracted words in Ω^+ . The *contraction* of a word w is the word w' , where we remove repeated symbols, i.e., we replace the maximal subwords of the form $b \dots b$ in w by b . For $K \subseteq \Omega^*$,

Algorithm 2 removeMidfixSpurious

Input: hybrid system H , language L , word w

- 1: $N \leftarrow \emptyset$
- 2: **for** $l \leftarrow \text{length of } w$ **downto** 2 **do**
- 3: **for all** subwords w' of w of length l and w' is not a subword of a word in N **do**
- 4: **if** w' is m-spurious **then** $L \leftarrow L \setminus \text{ext}(\{uw'v \mid u, v \in \Omega^*\})$
- 5: **else** $N \leftarrow N \cup \{w'\}$
- 6: **end for**
- 7: **end for**
- 8: **return** L

we define $\text{ext}(K)$ as the language that contains a word $u \in \Omega^*$ iff there is some $v \in K$ such that the contractions of u and v are the same.

The decision in which order to check the words in $C(\Omega)$ of length at most maxlen is non-trivial. In particular, should we check short words or long words first? On the one hand, a longer word is more likely to be identified, e.g., as m-spurious since at most as many trajectories follow a word as any of its subwords. On the other hand, if we identify a short word w , e.g., as m-spurious then we do not need to check longer words in which w occurs. In each iteration (lines 4–18), Alg. 1 checks the words in the following order. First, we check words if they are p-spurious or s-spurious (lines 5–7 and 8–10, respectively). We only check words of length 1, i.e., for a region, we check if it contains initial and unsafe states. If we identify $b \in \Omega$ as p-spurious, we remove $\text{ext}(\{bu \mid u \in \Omega^*\})$ from L . Analogously, if we identify $b \in \Omega$ as s-spurious, we remove $\text{ext}(\{ub \mid u \in \Omega^*\})$. Then, for every word $w \in C(\Omega)$ of length maxlen , we use Alg. 2 to check if subwords w' of w are m-spurious. If we identify w' as m-spurious, we remove $\text{ext}(\{uw'v \mid u, v \in \Omega^*\})$ from L . Furthermore, Alg. 2 maintains a set N to avoid unnecessary checks whether a word w' is m-spurious. We do not check whether w' is m-spurious if w' is a subword of a word \tilde{w} for which we could not prove in an earlier iteration (lines 2–7 of Alg. 2) that \tilde{w} is m-spurious (i.e., the used solver has returned “don’t know” for \tilde{w}). We assume here that the solver will then also return “don’t know” for w' . Note that this is a reasonable assumption since intuitively it is easier to show that a word is m-spurious than to show that one of its subwords is m-spurious.

After checking contracted words in L of length at most maxlen , Alg. 1 splits a region according to some heuristic, extends the alphabet Ω , and updates the language L and the function γ (lines 14–18). For reflecting a split of a region named by $a \in \Omega$, we need to specify that a fresh symbol behaves exactly like a .

Definition 6. For $K \subseteq \Omega^*$ and symbols a, b , we define $K^{a \sim b}$ as the smallest set such that if $a_1 \dots a_n \in K$ then $a'_1 \dots a'_n \in K^{a \sim b}$, where $a'_i = a_i$ if $a_i \neq a$ and $a'_i \in \{a, b\}$ if $a_i = a$, for all $i \in \{1, \dots, n\}$.

The correctness of Alg. 1 follows from the invariant that every trajectory from *Init* to *Unsafe* follows a word in L . Note that we only remove words from L that are p-spurious, s-spurious, or m-spurious.

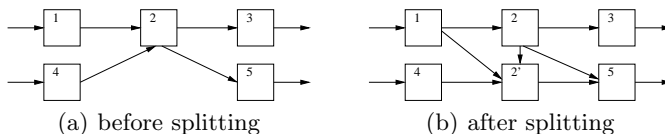


Fig. 1. Abstraction refinement

Theorem 1. *If Alg. 1 terminates then H is safe.*

Before we describe how we represent and manipulate the languages L that describe the abstractions (§4) and how we check if a word is p-spurious, s-spurious, or m-spurious (§5), we relate our language-based approach to the standard approach of using finite transition systems in CEGAR to abstract the concrete system. Furthermore, we present optimizations of Alg. 1 (§3.1 and §3.2).

In the standard CEGAR approach one uses finite transition systems to over-approximate the behavior of a concrete system. From a finite transition system T we can obtain a language $L_T \subseteq S^+$, where S is the set of states of T . Assume that we associate to each state s of T a region $\eta(s)$ of H 's state space. We define L_T as the language that consists of the words $s_0 \dots s_n \in S^+$, where $s_0 \rightarrow \dots \rightarrow s_n$ is a path in T with $\eta(s_0) \cap \text{Init} \neq \emptyset$ and $\eta(s_n) \cap \text{Unsafe} \neq \emptyset$. Note that T and L_T represent the same abstract counterexamples and deleting a transition $s \rightarrow s'$ in T corresponds to removing all words from L_T in which ss' occurs as a subword.

The other direction is as follows. Let $L \subseteq \Omega^+$ be a language in our language-based approach. The set of states of the finite transition system T_L is Ω . A state $b \in \Omega$ is initial iff $\gamma(b) \cap \text{Init} \neq \emptyset$ and b is an unsafe state in T_L iff $\gamma(b) \cap \text{Unsafe} \neq \emptyset$. We have a transition from state $b \in \Omega$ to $b' \in \Omega$ in T_L iff bb' is a subword of a word in L . Note that the abstraction L can be more accurate than T_L . In the case where $\text{maxlen} = 2$, the words that we check in line 11 of Alg. 1 are of the form ab , where $a, b \in \Omega$ and $a \neq b$. So, the elimination of words from the language L in line 12 corresponds to the elimination of edges between states in the transition system T_L .

We illustrate that choosing maxlen larger than 2 can be beneficial. Consider the abstraction in Figure 1(a) and assume that the abstract states 1 and 4 are reachable from an initial state in the abstraction, and the states 3 and 5 lead to an unsafe state in the abstraction. Assume further that the sequence $4 \rightarrow 2 \rightarrow 3$ is m-spurious because there is no trajectory from the abstract state 4 over 2 to 3. In our approach, we just remove $\text{ext}(\{u423v \mid u, v \in \Omega^*\})$ from the language. When using finite transition systems, we try to split the abstract state 2 into two new abstract states, which we name 2 and $2'$, in such a way that there is no trajectory from 4 to 2, no trajectory from $2'$ to 3, and no trajectory from $2'$ to 2 (see Figure 1(b)). The refined abstraction contains neither the sequence $4 \rightarrow 2 \rightarrow 3$ nor $4 \rightarrow 2' \rightarrow 3$. However, the splitting might introduce new fragments of spurious counterexamples, e.g., $1 \rightarrow 2' \rightarrow 5$. Moreover, to prove that $4 \rightarrow 2$, $2' \rightarrow 3$, and $2' \rightarrow 2$ are m-spurious, it might be necessary to split the region of the original abstract state 2 into two parts of a very complex form. It might even be the case that our data structure for representing regions is not flexible enough

to allow a split such that the edges $4 \rightarrow 2$, $2' \rightarrow 3$, and $2' \rightarrow 2$ can be removed. In this case, we have to split the regions 2 and 2' further. The experimental results in §6 demonstrate that such situations arise in practice and taking into account sequences of abstract states with more than two states can pay off.

3.1 Region Pruning

In Alg. 1, we only split regions and remove words from the language L to refine the abstraction. For instance, we remove a word if we can show that there is no trajectory that follows this word. In this subsection, we optimize our verification method by a complementary method: if we can prove that a part of a region is not reachable then we remove this part of the region, i.e., we prune certain states from regions. Observe that the regions may not cover the state space Φ anymore. However, since we only remove unreachable states, the regions still cover the part of Φ in which there might be a counterexample.

Definition 7. A state $y \in \Phi$ is reachable from a state $x \in \Phi$ if there is a trajectory r_0, \dots, r_p with $r_0(0) = x$ and $r_p(|r_p|) = y$. For $w \in \Omega^+$, $y \in \Phi$ is w -reachable from $x \in \Phi$ if there is a trajectory r_1, \dots, r_p that follows w , $r_0(0) = x$, and $r_p(|r_p|) = y$.

The following lemma allows us to remove states from regions that are neither w -reachable from an initial state, for all contracted words w of length less than $maxlen$ nor w -reachable from some other state, for all contracted words w of length $maxlen$. Due to space limitations, we omit its proof.

Lemma 1. Assume that γ covers the reachable states of Φ , i.e., for every $y \in \Phi$, if y is reachable from some initial state then $y \in \gamma(b)$, for some $b \in \Omega$. For every $l \geq 1$, if the state $y \in \Phi$ is reachable from an initial state then there is a word $w \in C(\Omega)$ such that

- $|w| < l$ and y is w -reachable from some state $x \in Init$, or
- $|w| = l$ and y is w -reachable from some state $x \in \Phi$.

In particular, for $l = 2$, it holds that if $y \in \gamma(b)$ is reachable from some initial state then y is either b -reachable from some $x \in Init$ or y is ab -reachable from some $x \in \gamma(a)$, where $a \neq b$. In [19], we already used the special case $l = 2$ to remove unreachable states from regions.

The optimized algorithm is Alg. 3, where $reach(H, b_1 \dots b_n)$ computes a superset of the states in the region $\gamma(b_n)$ for which there is a trajectory of H that follows $b_1 \dots b_n$, and, analogously, $reachInit(H, b_1 \dots b_n)$ computes a superset of the states in the region $\gamma(b_n)$ for which there is a trajectory of H that starts in $Init$ and follows $b_1 \dots b_n$. Details on implementing $reach$ and $reachInit$ are in §5.

3.2 Incremental Computation

Our second optimization exploits the following fact: if we split a region then we only need to re-check the words in which a symbol b occurs such that $\gamma(b)$ was

Algorithm 3 Optimized algorithm with region pruning

Input: hybrid system H

- 1: $\Omega \leftarrow \{b\}$, where b is some symbol
- 2: $L \leftarrow \Omega^+$
- 3: let $\gamma : \Omega \rightarrow 2^\Phi$ be the function with $\gamma(b) = \Phi$
- 4: **while** $L \neq \emptyset$ **do**
- 5: **for all** $b \in \Omega$ **do**
- 6: $R \leftarrow \emptyset$
- 7: **for all** $wb \in C(\Omega)$ with $|wb| < \text{maxlen}$ and $L \cap \{wbv \mid v \in \Omega^*\} \neq \emptyset$ **do**
- 8: **if** $\text{reachInit}(H, wb) \neq \emptyset$ **then** $R \leftarrow R \cup \text{reachInit}(H, wb)$
- 9: **else** $L \leftarrow L \setminus \text{ext}(\{wbv \mid v \in \Omega^*\})$ // wb is p -spurious
- 10: **end for**
- 11: **if** $L \cap \{ub \mid u \in \Omega^*\} \neq \emptyset$ and b is s-spurious **then**
- 12: $L \leftarrow L \setminus \text{ext}(\{ub \mid u \in \Omega^*\})$
- 13: **end if**
- 14: **for all** $wb \in C(\Omega)$ with $|wb| = \text{maxlen}$ and $L \cap \{uwbv \mid u, v \in \Omega^*\} \neq \emptyset$ **do**
- 15: $K \leftarrow \text{removeMidfixSpurious}(H, L, wb)$
- 16: **if** $K = L$ **then** $R \leftarrow R \cup \text{reach}(H, wb)$
- 17: **else** $L \leftarrow K$
- 18: **end for**
- 19: **if** $R = \emptyset$ **then** $\Omega \leftarrow \Omega \setminus \{b\}$ // no reachable states in region $\gamma(b)$
- 20: update γ , i.e., restrict γ to the domain Ω , $\gamma(b) = R$ if $R \neq \emptyset$, and $\gamma(c)$ is not altered for $c \in \Omega \setminus \{b\}$
- 21: **end for**
- 22: split region $\gamma(a)$ in regions U and V , for some $a \in \Omega$
- 23: $\Omega \leftarrow \Omega \cup \{b\}$, where b is a fresh symbol
- 24: $L \leftarrow L^{a \sim b}$
- 25: update γ , i.e., $\gamma(a) = U$, $\gamma(b) = V$, and $\gamma(c)$ is not altered for $c \in \Omega \setminus \{a, b\}$
- 26: **end while**

involved in this split. We do this by maintaining a set $Q \subseteq \Omega$: We only iterate the for-loops (lines 5–7, 8–10, and 11–13 of Alg. 1) for $b \in \Omega$ if it is in Q . After we have processed a symbol in Q , we remove it from Q . At the beginning of an iteration of the while-loop (lines 4–18 of Alg. 1) Q consists of the symbols for the regions that were involved in the split in the previous iteration, and if it is the first iteration, Q consists of the symbol chosen in line 1 of Alg. 1.

We can improve Alg. 3 in a similar way by iterating the for-loop (lines 5–21) for only the symbols in Q , and adding symbols b to Q for which the for-loop (lines 5–21) might be successful in changing the region $\gamma(b)$ or removing words from the language that contain b . Whenever a region $\gamma(b')$ has been changed, we add a symbol c to Q if there is a word w in which b' occurs, and either $|wc| = \text{maxlen}$ and $L \cap \{uwc \mid u, v \in \Omega^*\} \neq \emptyset$, or $|wc| < \text{maxlen}$ and $L \cap \{wcv \mid v \in \Omega^*\} \neq \emptyset$. Note that pruning $\gamma(b')$ as well as splitting $\gamma(b')$ may add symbols to Q .

4 Language Representation and Manipulation

In this section, we discuss how we represent and manipulate the languages $L \subseteq \Omega^*$ in the algorithms presented in the previous section. First, observe that the algorithms satisfy the following invariants:

Lemma 2. *Throughout Alg. 1 and Alg. 3, L is regular and $L = \text{ext}(L)$.*

We use minimal DFAs to represent the languages L and our implementation uses the DFA library from the MONA tool [13]. All the DFA operations used in our algorithms are rather straightforward. For instance, for checking the non-emptiness of $L \cap \{wv \mid v \in \Omega^*\}$ for a word $w \in \Omega^+$ in line 7 of Alg. 3, it suffices to check whether we reach a non-rejecting sink state in the minimal DFA that represents L when reading w . For the operation $L^{a \sim b}$, we define the DFA $A^{a \sim b} = (Q, \Omega \cup \{b\}, \delta', q_0, F)$ with $\delta'(q, b) = \delta(q, a)$ and $\delta'(q, x) = \delta(q, x)$, for all $q \in Q$ and $x \in \Omega$, where the DFA $A = (Q, \Omega, \delta, q_0, F)$ accepts L . Obviously, $A^{a \sim b}$ accepts the language $L^{a \sim b}$ when b is fresh, i.e., $b \notin \Omega$.

Although the used automata operations are all fairly simple, it turned out that they dominate the running times. To reduce the number of performed automata operations, we maintain a list in which we store words that we have identified as m-spurious. We update the DFA and empty the list before we split a region because otherwise the list would become too long. Moreover, we remove words w from the list if there is another word w' in the list such that w' is a subword of w . Whenever we have a query about a word, we search in this list before we query the DFA.

5 Counterexample Checking

The presented language-based approach is independent of the method for checking if a word is spurious and for pruning unreachable states from the region of the last symbol of a word. In this section, we present a method that accomplishes these two tasks. It extends a method that is used in the verification tool HSOLVER [17–19] and it has advantages over other methods: pruning is inherent to the method and it handles non-linear differential equations so that the correctness of the results are not interfered with rounding errors due to floating-point arithmetic.

As in other approaches (e.g. [14]), we use hyper-rectangles (*boxes*) for decomposing the state-space. That is, we require that the regions that cover the reachable states in Φ of the hybrid system H are of the form (s, B) , where $s \in S$ is a mode and $B \subseteq \mathbb{R}^k$ is a box, i.e. $B = [\ell_1, u_1] \times \cdots \times [\ell_k, u_k]$. Recall that k is the number of the continuous variables of H . In principle, regions can be represented by more complex geometrical shapes. However, since we utilize a constraint solver that uses boxes, we restrict ourselves in the following to boxes.

We need the following definitions. Let B be the box $[\ell_1, u_1] \times \cdots \times [\ell_k, u_k]$. The *ith lower face* of B is the box $[\ell_1, u_1] \times \cdots \times [\ell_{i-1}, u_{i-1}] \times [\ell_i, \ell_i] \times [\ell_{i+1}, u_{i+1}] \times \cdots \times [\ell_k, u_k]$ and the *ith upper face* of B is the box $[\ell_1, u_1] \times \cdots \times [\ell_{i-1}, u_{i-1}] \times [u_i, u_i] \times [\ell_{i+1}, u_{i+1}] \times \cdots \times [\ell_k, u_k]$. Assume that $\text{flow}_{(s,B)}(x, y)$ denotes a constraint that

models the fact that there is a flow from $x \in \mathbb{R}^k$ to $y \in \mathbb{R}^k$ in mode $s \in S$ and box $B \subseteq \mathbb{R}^k$. Furthermore, to make the solving of such a constraint easier, we assume that $flow_{(s,B)}(x, y)$ does not contain differentiation symbols and the bound variables are existentially quantified. There are various possibilities to achieve these assumptions, e.g., by using linearization [20]. We use $flow_{(s,B)}(x, y)$ to define the following constraints. Let $s, s' \in S$ be modes and $B, B' \subseteq \mathbb{R}^k$ boxes.

- The constraint $reach_{(s,B),(s',B')}^J(x, y)$ models the fact that there is a jump from $x \in B$ and mode s to $y \in B'$ and mode s' , i.e.,

$$x \in B \wedge y \in B' \wedge \exists x' \in B'. \text{Jump}(s, x, s', x') \wedge flow_{(s',B')}(x', y).$$

- The constraint $reach_{(s,B),(s',B')}^F(x, y)$ models the fact that there is a continuous flow in mode s from $x \in B \cap B'$ to $y \in B'$, i.e.,

$$s = s' \wedge x \in B \cap B' \wedge y \in B' \wedge flow_{(s',B')}(x, y) \wedge \bigwedge_{F \text{ face of } B'} \left[x \in F \rightarrow incoming_{(s,B'),F}(x) \right],$$

where $incoming_{(s,B'),F}(x) = \exists \dot{x}_1, \dots, \dot{x}_k \in \mathbb{R}. Flow(s, x, (\dot{x}_1, \dots, \dot{x}_k)) \wedge \dot{x}_j \geq 0$ if F is the j th lower face of B' , and if F is the j th upper face of B' , $incoming_{(s,B'),F}(x) = \exists \dot{x}_1, \dots, \dot{x}_k \in \mathbb{R}. Flow(s, x, (\dot{x}_1, \dots, \dot{x}_k)) \wedge \dot{x}_j \leq 0$. Note that we need the conjuncts $[x \in F \rightarrow incoming_{(s,B'),F}(x)]$, for the faces F of B' to ensure that a flow starting in $x \in F$ stays in box B' , i.e., the derivative in $x \in F$ does not point out of box B' .

A word $w \in \Omega^+$ has a *self-jump* if there are $(s, x), (s, x') \in \gamma(b)$ such that $(s, x, s, x') \in \text{Jump}$, for some symbol b in w .

In the following, let $w = b_0 \dots b_n \in C(\Omega)$ with $\gamma(b_i) = (s_i, B_i)$, for $i \in \{0, \dots, n\}$. The following theorem extends an earlier result [17, 19]. Its proof is similar and we omit it.

Theorem 2. *Assume that w does not have self-jumps. If a state $(s_n, y_n) \in \gamma(b_n)$ is w -reachable from some state in Φ then the constraint $\exists y_0 \in \mathbb{R}^k. reach_w(y_0, y_n)$ is satisfiable, where the constraint $reach_w(y_0, y_n)$ is defined as*

$$\exists y_1, \dots, y_{n-1} \in \mathbb{R}^k. \bigwedge_{1 \leq i \leq n} \left[reach_{\gamma(b_{i-1}), \gamma(b_i)}^J(y_{i-1}, y_i) \vee reach_{\gamma(b_{i-1}), \gamma(b_i)}^F(y_{i-1}, y_i) \right].$$

So, if we can disprove the constraint $\exists y_0 \in \mathbb{R}^k. reach_w(y_0, y_n)$ and w does not have self-jumps, then w is m-spurious. We use the solver RSOLVER [15, 16], which is based on interval-constraint-propagation techniques [8]. Constraints as, e.g., the constraint in Thm. 2 can be solved efficiently by RSOLVER, where the correctness is not affected by rounding errors. We use the following feature of RSOLVER: take as input a constraint ϕ and a box B and output a box B' such that $B' \subseteq B$, where B' contains the solutions of ϕ in B . We denote this algorithm by $Prune(\phi, B)$. Note that if B' is empty, we know that ϕ has no solution in B .

If the constraint of Thm. 2 does have a solution, we are interested in a sub-box of B_n that contains all its solutions. However, RSOLVER would spend time to compute such sub-boxes not only for B_n but for the boxes B_1, \dots, B_n . To

Algorithm 4 Counterexample checking without self-jumps

Input: word $b_0 \dots b_n$ without self-jumps, where $\gamma(b_i) = (s_i, B_i)$, for $i \in \{0, \dots, n\}$
Output: sub-box of $\gamma(b_n)$ containing the states in $\gamma(b_n)$ that are reachable via a trajectory following $b_0 \dots b_n$

- 1: $B' \leftarrow B_0$
- 2: $F' \leftarrow B_0 \cap B_1$
- 3: **for** $1 \leq i \leq n - 1$ **do**
- 4: $B'_F \leftarrow \text{proj}_2(\text{Prune}(\text{reach}_{(s_{i-1}, F'), (s_i, B_i)}^F, F' \times B_i))$
- 5: $B'_J \leftarrow \text{proj}_2(\text{Prune}(\text{reach}_{(s_{i-1}, B'), (s_i, B_i)}^J, B' \times B_i))$
- 6: **if** $B'_F = \emptyset$ and $B'_J = \emptyset$ **then return** \emptyset
- 7: $F' \leftarrow \text{proj}_2(\text{Prune}(\text{reach}_{(s_{i-1}, F'), (s_i, B'_F)}^F, F' \times (B'_F \cap B_{i+1}))) \cup$
 $\text{proj}_2(\text{Prune}(\text{reach}_{(s_{i-1}, B'), (s_i, B'_J)}^J, B' \times (B'_J \cap B_{i+1})))$
- 8: $B' \leftarrow B'_F \cup B'_J$
- 9: **end for**
- 10: **return** $\text{proj}_2(\text{Prune}(\text{reach}_{(s_{n-1}, F'), (s_n, B_n)}^F, F' \times B_n)) \cup$
 $\text{proj}_2(\text{Prune}(\text{reach}_{(s_{n-1}, B'), (s_n, B_n)}^J, B' \times B_n))$

avoid this superfluous work and to deal later with words that have self-jumps, we apply RSOLVER not to the whole constraint but only to constituent pieces. We compute an over-approximation of the reachable states starting from (s_0, B_0) and in each iteration, we propagate the over-approximation to the box of the next symbol in the word. The details are in Alg. 4, where proj_2 denotes the function $\text{proj}_2(B) = \{y \in \mathbb{R}^k \mid (x, y) \in B, \text{ for some } x \in \mathbb{R}^k\}$, for a box $B \subseteq \mathbb{R}^{2k}$. Note that in each iteration we first compute an over-approximation B' of the set of reachable elements in the box (this is needed for following jumps) and then an over-approximation F' of the set of reachable elements in the intersection of this box and the next box (this is needed for following flows).

Corollary 1. *Assume that w does not have self-jumps. If $(s_n, y_n) \in \gamma(b_n)$ is w -reachable from an initial state in $\gamma(b_0)$ then the following constraint is satisfiable*

$$\exists y, y_0 \in B_0. \text{Init}(s_0, y) \wedge \text{flow}_{\gamma(b_0)}(y, y_0) \wedge \text{reach}_w(y_0, y_n).$$

So, if we can disprove the constraint in Cor. 1 for w and w does not have self-jumps, then w is p-spurious. Analogously, we can check if w is s-spurious. The corresponding adaptations of Alg. 4 are straightforward.

Now, we are left with the question of how to deal with self-jumps. The problem is that such a jump might occur arbitrarily often within a box. Note that splitting eventually removes all self-jumps that do not occur between the same point. Hence, we solve the problem by using the whole box to over-approximate the reachable information in such an abstract state. We adapt the loop in Alg. 4 in such a way that we check whether there is a jump from B_i to B_i before assigning new values to F' and B' . If such a jump exists, then we use the assignments $B' \leftarrow B$ and $F' \leftarrow B_i \cap B_{i+1}$ instead of the current assignments to F' and B' . Similarly, we deal with contracted words that might have self-jumps and where the region of the first letter contains an initial state.

hybrid system	$maxlen = 2$			$maxlen = 3$			$maxlen = 4$		
	time	splits	size	time	splits	size	time	splits	size
2-tanks	0.40	34	20	0.08	6	5	0.13	6	5
car	0.31	0	1	0.39		1	0.50	0	1
clock	1.16	44	32	0.68	27	12	3.47	27	15
convoi-1	0.70	0	1	0.69	0	1	0.69	0	1
convoi	281.37	362	357	79.27	89	87	∞		
eco	0.97	45	48	8.45	43	66	127.58	43	71
focus	1.04	66	59	0.15	6	8	0.16	5	7
mixing	31.57	269	145	58.14	59	134	∞		
real-eigen	0.11	2	3	0.15	5	3	0.16	5	3
s-focus	0.07	2	4	0.13	2	4	0.19	2	4
trivial-hard	∞			0.04	5	6	0.06	5	6
van-der-pole	51.66	687	156	0.77	12	12	1.83	15	10

Table 1. Experimental results

6 Experimental Results

We used our problem database⁵ of hybrid systems from the literature as well as some new hybrid systems to evaluate our approach. Our implementation uses the following heuristic to split regions (see line 22 of Alg. 3). We choose a box with maximal side length. We split this box by bisecting one of its sides, where we use a round-robin strategy to choose the box side.

The experimental results are summarized in Tab.1 for different values of $maxlen$ (the running times are in seconds; the symbol ∞ means “more than 600 seconds”; the columns “size” show the peak automata sizes). We used a computer with an Intel Pentium 2.60 GHz CPU with 512 Mbytes of main memory running Linux. Four hybrid system in our database (**1-flow**, **circuit**, **heating**, **navigation**) could not be verified within the time limit with any of the values of $maxlen$. They are not listed in the table.

First, recall that for $maxlen = 2$ the language-based approach is closely related to the standard approach of using transition systems (see discussion after Thm. 1 in §3). In particular, when applying the same heuristics, the number of splittings is identical for transition systems and $maxlen = 2$. Second, observe that for almost all test cases, the number of splittings decreases considerably when choosing $maxlen = 3$ instead of $maxlen = 2$. That means, (i) checking the existence of trajectories between two and three regions and (ii) exploiting this information when there are no trajectories is effective in reducing the number of region splittings. Third, observe that the number of splittings is approx. the same for $maxlen = 3$ and $maxlen = 4$. This is somewhat surprising. Fourth, observe that a smaller number of splittings does not necessarily result in better running times. For instance, for the test case **mixing**, the version with $maxlen = 2$ is faster than the version with $maxlen = 3$ by a factor of nearly 2 although approx. 4.5 times more splittings are needed. This can be explained as follows: For larger values of $maxlen$, significantly more words are analyzed in an iteration of the algorithm. Profiling of our prototype implementation revealed that the time consumed by

⁵ See <http://hsolver.sourceforge.net/benchmarks> for details and references.

the solver for checking if a word is spurious remains rather small in comparison to the time consumed by the operations that maintain and refine the abstractions. Finally, we remark that the sizes of the DFAs remain rather small.

In summary, the experimental results suggest that $maxlen = 4$ is not a good choice, since the version with $maxlen = 4$ is always outperformed by one of the versions with $maxlen = 2$ or $maxlen = 3$. For larger values of $maxlen$, we expect that the running times increase further. Between $maxlen = 2$ and $maxlen = 3$ there is no clear winner. However, the version with $maxlen = 3$ seems to be more robust: for some test cases, the running times for $maxlen = 3$ are significantly faster than for $maxlen = 2$ and in the cases where $maxlen = 2$ is faster, the running times for $maxlen = 3$ increase only moderately (except the test case `eco`, where the number of splittings varies only slightly and the version with $maxlen = 2$ outperforms the version with $maxlen = 3$ by a factor of approx. 8).

We see mainly two reasons why our algorithm fails on the four examples `1-flow`, `circuit`, `heating`, and `navigation`. First, since the used solver wraps the solutions for a given constraint into boxes, we obtain an over-approximation when solving a constraint. Sometimes these over-approximations are too coarse such that many splits are needed before we are able to identify a word as spurious. Second, some of the examples are not robustly safe, i.e., they become unsafe under some small perturbation [11]. Sometimes we can succeed by using another splitting heuristic. For example, our algorithm would verify `1-flow` easily (for all values of $maxlen$) if we bisect in the first iteration the box on the third side and not on the first.

7 Conclusion

We presented a language-based approach to represent and refine abstractions for hybrid system verification. The advantage of using languages as abstractions instead of finite transition systems is that languages can over-approximate the behaviors of hybrid systems more accurately. On the one hand, the costs to maintain and refine these abstractions increase. On the other hand, our experiments show that the number of abstract states often reduces significantly. Moreover, we generalized the method that is used in the verification tool `HSOLVER` [17–19] to analyze non-empty sequences of abstract states of arbitrary finite length.

Future work includes the design of more sophisticated data structures to maintain such language-based abstractions and to investigate termination issues of this approach. It is also future work to incorporate better heuristics (e.g., for splitting regions and the words that are analyzed in an iteration). It is open if the techniques in [10] can be used for the language-based approach. Finally, we want to investigate the use of ω -languages and Büchi automata as abstractions for hybrid systems for verifying progress properties, like stability of hybrid systems.

References

1. R. Alur, T. Dang, and F. Ivančić. Predicate abstraction for reachability analysis of hybrid systems. *ACM Trans. Embedded Comput. Syst.*, 5(1):152–199, 2006.

2. E. Asarin, G. Schneider, and S. Yovine. On the decidability of the reachability problem for planar differential inclusions. In *Hybrid Systems: Computation and Control (HSCC'01)*, volume 2034 of *Lect. Notes Comput. Sci.*, pages 89–104, 2001.
3. T. Brihaye and C. Michaux. On the expressiveness and decidability of o-minimal hybrid systems. *J. Complexity*, 21(4):447–478, 2005.
4. T. Brihaye, C. Michaux, C. Rivière, and C. Troestler. On o-minimal hybrid systems. In *Hybrid Systems: Computation and Control (HSCC'04)*, volume 2993 of *Lect. Notes Comput. Sci.*, pages 219–233, 2004.
5. E. Clarke, A. Fehnker, Z. Han, B. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald. Abstraction and counterexample-guided refinement in model checking of hybrid systems. *Internat. J. Found. Comput. Sci.*, 14(4):583–604, 2003.
6. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
7. D. Dams. Comparing abstraction refinement algorithms. In *Proc. of 2nd Workshop on Software Model Checking (SoftMC'03)*, volume 89(3) of *Electr. Notes Theor. Comput. Sci.*, pages 405–416, 2003.
8. E. Davis. Constraint propagation with interval labels. *Artif. Intell.*, 32(3):281–331, 1987.
9. L. Doyen, T. A. Henzinger, and J.-F. Raskin. Automatic rectangular refinement of affine hybrid systems. In *Formal Modeling and Analysis of Timed Systems (FORMATS'05)*, volume 3829 of *Lect. Notes Comput. Sci.*, pages 144–161, 2005.
10. A. Fehnker, E. Clarke, S. Jha, and B. Krogh. Refining abstractions of hybrid systems using counterexample fragments. In *Hybrid Systems: Computation and Control (HSCC'05)*, volume 3414 of *Lect. Notes Comput. Sci.*, pages 242–257, 2005.
11. M. Fränzle. Analysis of hybrid systems: An ounce of realism can save an infinity of states. In *Computer Science Logic (CSL'99)*, volume 1683 of *Lect. Notes Comput. Sci.*, pages 126–140, 1999.
12. T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata. *J. Comput. System Sci.*, 57:94–124, 1998.
13. N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. *Internat. J. Found. Comput. Sci.*, 13(4):571–586, 2002.
14. M. Kloetzer and C. Belta. Reachability analysis of multi-affine systems. In *Hybrid Systems: Computation and Control (HSCC'06)*, volume 3927 of *Lect. Notes Comput. Sci.*, pages 348–362, 2006.
15. S. Ratschan. RSOLVER. <http://rsolver.sourceforge.net>. Software package.
16. S. Ratschan. Continuous first-order constraint satisfaction. In *Artificial Intelligence, Automated Reasoning, and Symbolic Computation (AISC'02)*, volume 2385 of *Lect. Notes Comput. Sci.*, pages 181–195, 2002.
17. S. Ratschan and Z. She. Safety verification of hybrid systems by constraint propagation based abstraction refinement. *ACM Trans. Embedded Comput. Syst.* To appear.
18. S. Ratschan and Z. She. HSOLVER. <http://hsolver.sourceforge.net>. Software package.
19. S. Ratschan and Z. She. Safety verification of hybrid systems by constraint propagation based abstraction refinement. In *Hybrid Systems: Computation and Control (HSCC'05)*, volume 3414 of *Lect. Notes Comput. Sci.*, pages 573–589, 2005.
20. S. Ratschan and Z. She. Constraints for continuous reachability in the verification of hybrid systems. In *Artificial Intelligence and Symbolic Computation (AISC'06)*, volume 4120 of *Lect. Notes Comput. Sci.*, pages 196–210, 2006.