Abstraction and Counterexample-Guided Refinement in Model Checking of Hybrid Systems

Edmund Clarke Ansgar Fehnker Zhi Han Bruce Krogh
Joël Ouaknine Olaf Stursberg Michael Theobald

January 2003
CMU-CS-03-104

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Edmund Clarke, Joël Ouaknine, and Michael Theobald are with the Computer Science Department at Carnegie Mellon University. Ansgar Fehnker, Zhi Han, and Bruce Krogh are with the Electrical and Computer Engineering Department at Carnegie Mellon University. Olaf Stursberg is with the Process Control Lab at University of Dortmund, Germany.
This research was supported by the Defense Advanced Research Project Agency (DARPA) MoBIES project under contracts no. F3361500C1701 and F33615-02-C-0429, by the Army Research Office (ARO) under contract no. DAAD19-01-1-0485, by the National Science Foundation (NSF) under grants no. CCR-0121547 and CCR-0098072, by the Office of Naval Research (ONR) under contract no. N00014-95-1-0520. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, ARO, ONR, NSF, the U.S. Government or any other entity.

Abstract

Hybrid dynamic systems include both continuous and discrete state variables. Properties of hybrid systems, which have an infinite state space, can often be verified using ordinary model checking together with a finite-state abstraction. Model checking can be inconclusive, however, in which case the abstraction must be refined. This paper presents a new procedure to perform this refinement operation for abstractions of hybrid systems. Following an approach originally developed for finite-state systems [1, 2], the refinement procedure constructs a new abstraction that eliminates a counterexample generated by the model checker. For hybrid systems, analysis of the counterexample requires the computation of sets of reachable states in the continuous state space. We show how such reachability computations with varying degrees of complexity can be used to refine hybrid system abstractions efficiently. Examples illustrate our counterexample-guided refinement procedure and experimental results for a prototype implementation of the procedure indicate significant advantages over existing methods.

Keywords: Formal Verification, Abstraction, Model Checking, Hybrid Systems, Refinement, Counterexamples

1 Introduction

Hybrid systems are formal models that include both continuous and discrete state variables. With the increasing use of hybrid systems to design embedded controllers for complex systems such as manufacturing processes, automobiles, and transportation networks, there is an urgent need for more powerful analysis tools, especially for safety critical applications. Tools developed so far for the automated analysis of hybrid systems are restricted to low-dimensional continuous dynamics [3]. The reason for this limitation is the difficulty of representing and computing sets of reachable states for continuous dynamic systems. Recent publications have proposed two general approaches to deal with the complexity of hybrid system analysis, namely modular analysis (e.g., [4, 5]) and abstraction (e.g., [6, 7, 8]). This paper focuses on the latter approach.

Abstraction maps a given model into a less complex model that retains the behaviors of interest [6]. In the context of hybrid system verification, abstraction transforms the inherently infinite state system into a finite-state model [7, 8]. Existing tools often do not take into account the specification itself when building an abstract model. Rather, an abstract representation is constructed for the entire hybrid system using a degree of detail which seems to be appropriate. If the abstraction is not suitable to analyze the property, then the abstract model is globally refined [9].

As an alternative, we suggest a procedure that (a) starts from a coarse abstract model and a safety property, (b) identifies parts of the hybrid system which potentially violate the property, and (c) iteratively refines the abstract model until verification reveals whether or not the property in question is satisfied. A framework that follows this general scheme of abstraction, refinement, and analysis, is *counterexample-guided abstraction refinement (CEGAR)* [1, 10, 2]: For a given system the initial abstraction leads to a conservative model that is guaranteed to include all behaviors of the original system. Model checking is then applied to the abstract model. If the property is violated, the model checker produces a *counterexample* as an *execution path* of the abstract model for which the property is not true. If this counterexample corresponds to a genuine behavior of the original system, then the property does not hold for the original system. Otherwise, the information provided by the counterexample is then used to *refine* the abstract model, i.e., some detail is added to the abstract model in order to obtain a more accurate, yet conservative, representation of the original model. In particular, the refined model is constructed so as to exclude the *spurious* counterexample. The procedure of alternating between model checking and refinement is continued until the property is confirmed or refuted.

This procedure has recently been applied successfully to finite-state systems in a variety of areas, and in particular in the verification of digital circuits [1, 10]. Earlier work based on the use of counterexamples includes the localization reduction in the context of concurrent systems [2], and recent work has seen the application of the technique to the verification of C programs [11, 12].

This paper extends counterexample-guided model refinement to hybrid systems, which include both continuous and discrete state variables and thus have an infinite state space. We provide effective means of coping with the difficulties of computing reachable sets for hybrid systems. In particular, we employ reachable set computations with varying degrees of accuracy to refine hybrid system abstractions efficiently. This flexibility cannot easily be achieved with other verification tools for hybrid systems.

The paper is structured as follows. Section 2 presents preliminaries on abstraction and counterexample-guided refinement. In Section 3 we describe the CEGAR verification approach that refines abstract models based on counterexamples. We introduce hybrid systems in Section 4, and apply CEGAR to hybrid systems in Section 5. Section 6 summarizes the contributions of this paper.

2 Preliminaries

We introduce the notions of abstraction and counterexample-guided refinement for general transition systems, defined as follows:

Definition 1 *Transition System.* A *transition system* is a triple $TS = (S, S_0, E)$ with a (possibly infinite) state set S, an initial set $S_0 \subseteq S$, and a set of transitions $E \subseteq S \times S$. $\diamond$

A *path* of a transition system is a finite sequence $(s_0, s_1, \ldots, s_m)$ with $s_0 \in S_0$, each $s_i \in S$, and each pair of successive states $(s_i, s_{i+1}) \in E$.

Given two transition systems A and C, A is said to be an *abstract model* of C if the following relation can be established.

Definition 2 *Abstraction.* A transition system $A = (\hat{S}, \hat{S}_0, \hat{E})$ with a finite set of states $\hat{S}$ is an *abstract model* of a transition system $C = (S, S_0, E)$, denoted $A \succeq C$, if there exists an *abstraction function* $\alpha : S \to \hat{S}$ such that:

- the initial set is $\hat{S}_0 = \alpha(S_0) = \{\hat{s}_0 \mid \exists s_0 \in S_0 : \hat{s}_0 = \alpha(s_0)\}$, and
- $\hat{E} \supseteq \alpha(E) = \{(\hat{s}_1, \hat{s}_2) \mid \exists s_1, s_2 \in S : (s_1, s_2) \in E, \hat{s}_1 = \alpha(s_1), \hat{s}_2 = \alpha(s_2)\}$.$\diamond$

Note: In general, it is possible—and sometimes desirable—to consider an abstraction *relation* α rather than a mere abstraction function. The work presented here can easily be adapted to this more general case, however for simplicity we shall stick to the above definition.

Sometimes the term *simulation* is used in the literature to describe the abstraction relation. In contrast to the definitions of abstraction in [1, 10], Defn. 2 allows A to include *spurious transitions*, i.e., the set $\hat{E}$ may contain elements that do not correspond to transitions in C. Spurious transitions arise in the construction of abstractions of hybrid systems because in most cases sets of reachable states for continuous systems cannot be represented and computed exactly [9].

Abstract models will be used to analyze properties of a given transition system. Throughout the paper, we will the given system C the *concrete system*.

In order to construct a more detailed model from a given abstract model, we define the following concept of *model refinement*.

Definition 3 *Refinement of Abstract Models.* Given a concrete system $C = (S, S_0, E)$ and an abstract model $A = (\hat{S}, \hat{S}_0, \hat{E})$ such that $C \preceq A$, with abstraction function $\alpha : S \to \hat{S}$, a model $A' = (\hat{S}', \hat{S}'_0, \hat{E}')$ is called a *refined abstract model of C with* *respect to* A if there are two abstraction functions $\alpha' : S \rightarrow \hat{S}'$ and $\alpha'' : \hat{S}' \rightarrow \hat{S}$, i.e., if $C \preceq A' \preceq A$. ⋄

Properties (or specifications) are verified for the concrete model C using an abstract model A. In this paper we consider the verification of safety properties, defined as follows.

Definition 4 *Safety.* Given a transition system $TS = (S, S_0, E)$, let the set $B \subseteq S \setminus S_0$ specify a set of *bad states*. We say that *TS is safe with respect to* B, denoted by $TS \models \mathbf{AG}\neg B$ iff there is no path in the transition system from an initial state in S_0 to a bad state in B. Otherwise we say *TS is unsafe*, denoted by $TS \not\models \mathbf{AG}\neg B$. ⋄

Definition 5 *Counterexamples.* A path $\sigma = (s_0, s_1, \ldots, s_m)$ of $TS = (S, S_0, E)$ with $s_m \in B$ is called a *counterexample* of TS with respect to the safety property $TS \models \mathbf{AG}\neg B$. Given a concrete transition system C, an abstract transition system A, and a counterexample σ in C, we say that $\hat{\sigma} = (\hat{s}_0, \hat{s}_1, \hat{s}_2, \ldots, \hat{s}_m)$ is the *corresponding abstract counterexample* of the abstract system A, if $\hat{s}_i = \alpha(s_i)$ holds for all $i \in \{0, \ldots, m\}$. Given a counterexample $\hat{\sigma}$ of A, σ is called a *corresponding concrete counterexample* if for all i, $\hat{s}_i = \alpha(s_i)$ and $(s_i, s_{i+1}) \in E$. If a counterexample $\hat{\sigma}$ of A has no corresponding concrete counterexample for C, $\hat{\sigma}$ is called a *spurious counterexample*. ⋄

Lemma 1 Given a concrete model $C = (S, S_0, E)$, and an abstract model $A = (\hat{S}, \hat{S}_0, \hat{E})$ of C with an abstraction function α, let $B \subseteq S \setminus S_0$, and choose $\hat{B} \subset \hat{S}$ such that $\hat{B} \supseteq \alpha(B) = \{\hat{b} \mid \exists\, b \in B : \hat{b} = \alpha(b)\}$. If $A \models \mathbf{AG}\neg\hat{B}$, then $C \models \mathbf{AG}\neg B$. □

If $A \models \mathbf{AG}\neg\hat{B}$ can be verified, it can immediately be concluded from Lemma 1 (i.e., without applying verification to the concrete system C) that $C \models \mathbf{AG}\neg B$. On the other hand, the converse of Lemma 1 with respect to the **AG**-property does not hold. If the verification of A reveals $A \not\models \mathbf{AG}\neg\hat{B}$, then we cannot conclude that C is not safe with respect to B, since the counterexample for A may be spurious. We call a method that checks whether or not a counterexample is spurious a *validation method*. If the validation method discovers that the counterexample is spurious, then the counterexample is used to refine A. We now introduce a scheme for *counterexample-guided abstraction refinement (CEGAR)* to verify safety properties for a given concrete model. The basic principle is to repeat the following sequence of steps until the property is verified or refuted [1]. The starting point is a concrete model C and an abstract model A (we propose in Sec. 5.1 a specific way to obtain an initial abstract model for hybrid systems). The first step is then to analyze $A \models \mathbf{AG}\neg\hat{B}$ by model checking. If this property holds it can immediately be concluded from Lemma 1 that C is safe, too. Otherwise a counterexample is obtained, and we must verify whether it has a corresponding real counterexample in C. If so, then the safety property does not hold for C. Otherwise, i.e., when the counterexample is spurious, the counterexample is used to refine the model A. That is, a new and more detailed model A' with $C \preceq A' \preceq A$ is produced, which excludes the spurious counterexample.

The procedure of model checking, validation of the counterexample, and refinement of the abstract model is repeated until the safety property is proved or refuted for C. The pseudo-code in Fig. 1 summarizes this procedure:

> **ALGORITHM:** Counterexample-Guided Abstraction Refinement: CEGAR
> **INPUT:** Concrete model C and a set of bad states B
> **OUTPUT:** B is (or is not) reachable
>
> Generate initial abstract model A (bad states are called $\hat{B}$)
> Generate counterexample $\hat{\sigma}$ (if one exists) by model checking A wrt $\hat{B}$
> `WHILE` $\hat{\sigma}$ exists `DO`
>   Validation of $\hat{\sigma}$
>   `IF` $\hat{\sigma}$ validated `THEN` terminate with "B reachable"
>   `ELSE`
>   Generate refined model A' using counterexample $\hat{\sigma}$
> $A := A'$
>   Generate next $\hat{\sigma}$ by model checking A wrt $\hat{B}$
>   `ENDIF`
> `ENDDO`
> Terminate with "B not reachable"

Figure 1: CEGAR: Scheme for verifying/falsifying $C \models$ **AG**$\neg B$ based on counterexample-guided abstraction refinement

The crucial steps in the CEGAR procedure are *model checking*, *validation*, and *refinement*. With respect to model checking, standard algorithms for AG-properties can be used [13].

For validating a counterexample, the important ingredient is the computation of successors of states. We define an operator $succ$ that determines the successor states from a given set $\tilde{S} \subseteq S$ by $succ(\tilde{S}) = \{s \in S \mid \exists \tilde{s} \in \tilde{S} : (\tilde{s}, s) \in E\}$. This set may not be exactly computable for a given concrete model C, i.e. only over-approximations $\overline{succ}(\tilde{S}) \supset succ(\tilde{S})$ may be available. We first assume that $succ(\tilde{S})$ is computable.

A counterexample $\hat{\sigma} = (\hat{s}_0, \ldots, \hat{s}_m)$ of A is then validated as follows: Let $S_k = \alpha^{-1}(\hat{s}_k)$, $k \in \{0, \ldots, m\}$ denote the sets of concrete states corresponding to an element of $\hat{\sigma}$. The reachable parts of these sets are recursively defined by $S_0^{reach} := S_0$, $S_k^{reach} := succ(S_{k-1}^{reach}) \cap S_k$, $k \in \{1, \ldots, m\}$. The counterexample is spurious iff $S_k^{reach} = \emptyset$ for at least one k, and we say *the counterexample is refuted*. Otherwise, the counterexample is *validated*, and B is reachable.

If the counterexample is refuted with $S_k^{reach} = \emptyset$, the model A is refined into a new finite abstract model $A' = (\hat{S}', \hat{S}_0', \hat{E}')$ (cf. Defn. 3). The refined model should take into account that there are no concrete transitions from states in S_{k-1}^{reach} to states in S_k. We therefore require that the set $\hat{E}'$ of A' **not** contain transitions in the set $\{(\alpha'(s_1), \alpha'(s_2)) \mid \exists\ s_1 \in S_{k-1}^{reach}, s_2 \in S_k\}$. Thus, successive refined models will exclude previously explored counterexamples. A method for the refinement of abstract models for infinite-state systems will be presented in the next section.

3 Refinement of Abstract Models

This section presents a specific method for refining an abstract model A. The main idea is to directly use the information obtained from the validation procedure to refine certain abstract states. Assume that the abstract model includes a transition between $\hat{s}_1$ and $\hat{s}_2$, while the validation of the counterexample has revealed that only a subset of concrete states in $S_2 := \alpha^{-1}(\hat{s}_2)$ is reachable from concrete states in $S_1 := \alpha^{-1}(\hat{s}_1)$. In this case we refine A by splitting $\hat{s}_2$ into two new states. The first one, denoted by $\hat{s}_2^{reach}$, represents the reachable subset of S_2, given by $S_2^{reach} := succ(S_1) \cap S_2$. The second one, denoted by $\hat{s}_2^{comp}$, represents the complement of the reachable part, given by $S_2^{comp} := S_2 \setminus S_2^{reach}$. In addition, the abstraction function that maps concrete states to abstract ones also has to be refined.

Definition 6 *State Splitting.* Consider a concrete model $C = (S, S_0, E)$ and an abstract model $A = (\hat{S}, \hat{S}_0, \hat{E})$ with an abstraction function $\alpha : S \to \hat{S}$. Let $(\hat{s}_1, \hat{s}_2) \in \hat{E}$ be a transition of a counterexample $\hat{\sigma}$. Then, we define ρ_{split} to be a function that maps A, α, and $(\hat{s}_1, \hat{s}_2) \in \hat{E}$ onto both an abstract model $A' = (\hat{S}', \hat{S}'_0, \hat{E}')$ and an abstraction function $\alpha' : S \to \hat{S}'$, i.e., $(A', \alpha') = \rho_{split}(A, \alpha, (\hat{s}_1, \hat{s}_2))$, defined as follows:

- $\hat{S}' = (\hat{S} \setminus \{\hat{s}_2\}) \cup \{\hat{s}_2^{reach}, \hat{s}_2^{comp}\}$
- $\alpha'(s) = \begin{cases} \alpha(s) & \text{if } s \notin S_2 \\ \hat{s}_2^{reach} & \text{if } s \in S_2^{reach} \\ \hat{s}_2^{comp} & \text{if } s \in S_2^{comp} \end{cases}$
- $\hat{S}'_0 = \{\hat{s}' \in \hat{S}' | \alpha''(\hat{s}') \in \hat{S}_0\}$
- $\hat{E}' = \{(\hat{s}'_1, \hat{s}'_2) \in \hat{S}' \times \hat{S}' | \exists \hat{s}_1, \hat{s}_2 \in \hat{S} : (\hat{s}_1, \hat{s}_2) \in \hat{E} \wedge \hat{s}_1 = \alpha''(\hat{s}'_1) \wedge \hat{s}_2 = \alpha''(\hat{s}'_2)\} \setminus \{(\hat{s}_1, \hat{s}_2^{comp})\}$

  where $\alpha'' : \hat{S}' \to \hat{S}$ maps $\hat{s}'$ to itself if $\hat{s}' \notin \{\hat{s}_2^{reach}, \hat{s}_2^{comp}\}$, and to $\hat{s}_2$ otherwise. $\diamond$

Lemma 2 Let $A = (\hat{S}, \hat{S}_0, \hat{E})$ be an abstract model of $C = (S, S_0, E)$ with abstraction function $\alpha : S \to \hat{S}$. For a given transition $(\hat{s}_1, \hat{s}_2) \in \hat{E}$, assume that $S_2^{reach} \neq \emptyset$. Then $(A', \alpha') := \rho_{split}(A, \alpha, (\hat{s}_1, \hat{s}_2))$ is a refinement of A, i.e., $A \succeq A' \succeq C$. $\square$

As a next step, we consider the case where the set of successors of S_1 and the set S_2 are disjoint. In this case, we can simply omit the corresponding abstract transition.

Definition 7 *Transition Purging.* The function ρ_{purge} maps an abstract model $A = (\hat{S}, \hat{S}_0, \hat{E})$, an abstraction function $\alpha : S \to \hat{S}$ and a transition $(\hat{s}_1, \hat{s}_2) \in \hat{E}$ to $A' = (\hat{S}, \hat{S}_0, \hat{E}')$ with $\hat{E}' = \hat{E} \setminus \{(\hat{s}_1, \hat{s}_2)\}$. $\diamond$

Lemma 3 Let $A = (\hat{S}, \hat{S}_0, \hat{E})$ be an abstract model of $C = (S, S_0, E)$ with the abstraction function $\alpha : S \to \hat{S}$. For a given transition $(\hat{s}_1, \hat{s}_2) \in \hat{E}$, assume that $S_2^{reach} = \emptyset$. Then $A' := \rho_{purge}(A, \alpha, (\hat{s}_1, \hat{s}_2))$ is a refinement of A, i.e., $A \succeq A' \succeq C$. $\square$

Based on these results, we now present a more specific formulation of the CEGAR algorithm in Fig. 2, called INFINITE-STATE-CEGAR, which uses the functions ρ_{split} and ρ_{purge} for refinement.

```
ALGORITHM: INFINITE-STATE-CEGAR
INPUT: Concrete model C and a set of bad states B
OUTPUT: B is (or is not) reachable

Generate initial abstract model A and abstraction function α
B̂ := α(B)
Generate counterexample σ̂ = (ŝ_0, ..., ŝ_m) by model checking of A wrt B̂
S_0^reach := α^{-1}(ŝ_0)
WHILE σ̂ exists DO
    // validation of counterexample
    k := 0
    WHILE S_k^reach ≠ ∅ AND k < m DO
        k := k + 1
        S_k^reach := succ(S_{k-1}^reach) ∩ α^{-1}(ŝ_k)
    ENDDO
    // if counterexample is validated, then terminate, else refine
    IF S_k^reach ∩ B ≠ ∅ THEN terminate with "B reachable"
    ELSE
        FOR l = 1, ..., k
            // split abstract state ŝ_l into two: one that corresponds
            // to S_l^reach and one that corresponds to α^{-1}(ŝ_l) \ S_l^reach
            IF S_l^reach ≠ α^{-1}(ŝ_l)
            THEN (A, α) := ρ_split(A, α, (ŝ_{l-1}, ŝ_l))
            ENDIF
        ENDFOR
        // remove spurious transition between ŝ_{k-1} and ŝ_k
        A := ρ_purge(A, α, (ŝ_{k-1}, ŝ_k))
        Generate σ̂ by model checking of A wrt B̂
    ENDIF
ENDDO
Terminate with "B not reachable"
```

Figure 2: INFINITE-STATE-CEGAR.

Correctness of the algorithm is implied by the following lemma.[1] Note that termination of the algorithm cannot be guaranteed as the number of states in the concrete

[1]The proofs of all lemmas in the paper can be found in the appendix.

model may be infinite, and a finite abstract model to verify (or disprove) the given property may not exist [14].

Lemma 4 If the algorithm terminates with "B reachable", then $C \not\models$ **AG**$\neg B$, and if the algorithm terminates with "B not reachable", then $C \models$ **AG**$\neg B$. □

The proposed procedure of validating counterexamples and refining abstract models is based on the computation of successor states. Alternatively, one could formulate a similar algorithm that uses sets of predecessors, or even a combination of both as presented in [1] and [10].

The INFINITE-STATE-CEGAR algorithm in Fig. 2 is based on the assumption that sets of successor states are exactly computable. Unfortunately, this rarely occurs in practice for hybrid systems, and one must settle for an *over-approximation* $\overline{succ}$ to the successor function $succ$. In this case, the counterexample validation step may become overly conservative, in that the algorithm may fail to refute a spurious counterexample.[2] On the other hand, we have:

Lemma 5 If the INFINITE-STATE-CEGAR algorithm using over-approximations in computing successor states terminates with "B not reachable", then $C \models$ **AG**$\neg B$. □

3.1 Example

Let us borrow Hofstadter's "MU-puzzle"[15] to illustrate the salient issues at hand.

The MIU-system is a simple rewrite system over alphabet $\Sigma = \{\mathsf{M}, \mathsf{I}, \mathsf{U}\}$, with initial string MI, and production rules

1. $x\mathsf{I} \longrightarrow x\mathsf{IU}$
2. $\mathsf{M}x \longrightarrow \mathsf{M}xx$
3. $x\mathsf{III}y \longrightarrow x\mathsf{U}y$
4. $x\mathsf{UU}y \longrightarrow xy$

where $x, y \in \Sigma^*$ are arbitrary finite strings, and string concatenation is denoted as simple juxtaposition. For example, from the initial string MI, one can derive the new string MIU through an application of Rule 1.

The MU-puzzle asks whether this rewrite system can ever derive the string MU.

We model this as a safety property over an infinite transition system $C = (S, S_0, E)$, as follows. Let $S = \Sigma^*$, $S_0 = \{\mathsf{MI}\}$, and

$$E = \{(x\mathsf{I}, x\mathsf{IU}), (\mathsf{M}x, \mathsf{M}xx), (x\mathsf{III}y, x\mathsf{U}y), (x\mathsf{UU}y, xy) \mid x, y \in \Sigma^*\} .$$

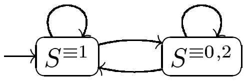
Let $B = \{\mathsf{MU}\}$. It is clear that $C \models \mathbf{AG}\neg B$ if and only if the MU-puzzle cannot be solved, in other words if the string MU cannot be derived in the MIU-system.

The abstract models of C that we shall consider 'lump together' states (i.e., Σ-strings) of S. The first step is to choose an initial abstract model. The only obligatory

[2] We discuss this point in greater detail in the next section.

requirement is that this model should separate the initial state(s) from the bad state(s). An additional desirable property of the initial partition is that it should also be reasonably coarse, so as to minimize the number of abstract states and correspondingly allow for efficient model checking.

Let us first introduce some auxiliary definitions. For $x \in \Sigma^*$, let $\sharp_{\mathsf{I}} x$ represent the number of times the symbol I appears in x, modulo 3. Next, for $j = 0, 1, 2$, let $S^{\equiv j} = \{s \in S \mid \sharp_{\mathsf{I}} s = j\}$. Our initial abstract model is $A_1 = (\{S^{\equiv 1}, S^{\equiv 0,2}\}, \{S^{\equiv 1}\}, E_1)$, where $S^{\equiv 0,2} = S^{\equiv 0} \cup S^{\equiv 2}$ and the transition relation E_1 is depicted below:

 .

The abstraction function $\alpha_1 : S \to \{S^{\equiv 1}, S^{\equiv 0,2}\}$ satisfies $\alpha_1(s) = S^{\equiv 1}$ if $\sharp_{\mathsf{I}} s = 1$, and $\alpha_1(s) = S^{\equiv 0,2}$ otherwise. Our set of abstract bad states is $B_1 = \alpha_1(B) = \alpha_1(\{\mathsf{MU}\}) = \{S^{\equiv 0,2}\}$.

We now observe that $A_1 \nvDash \mathbf{AG}\neg B_1$ since there is a path (consisting of a single transition) from the initial state $S^{\equiv 1}$ to the bad state $S^{\equiv 0,2} \in B_1$. However, upon validation over the concrete system C, we find that this counterexample is in fact spurious, since the only one-step transitions from the single initial state $\mathsf{MI} \in S_0$ are $\mathsf{MI} \longrightarrow \mathsf{MIU}$ (as per Rule 1) and $\mathsf{MI} \longrightarrow \mathsf{MII}$ (Rule 2). In other words, $\mathsf{MU} \in B$ is not reachable in one step.

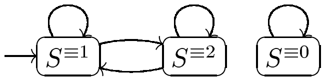
We must now refine our initial abstraction in such a way as to exclude this counterexample. As discussed above, we would normally base our next refinement on the successor function $succ$. Unfortunately, not only is $succ(S^{\equiv 1})$ difficult to compute, but in fact it turns out that iterating the refinement-counterexample-validation cycle with $succ$ would never terminate, and thus would never allow us to decide whether $C \vDash \mathbf{AG}\neg B$ or not.

Fortunately, we are able to rely on an over-approximation $\overline{succ}$ of the successor states: $\overline{succ}(s) = \{u \in S \mid \sharp_{\mathsf{I}} u = \sharp_{\mathsf{I}} s \vee \sharp_{\mathsf{I}} u \equiv 2\sharp_{\mathsf{I}} s\}$. Glancing at the production Rules 1–4, it is clear that $\overline{succ}$ is indeed an over-approximation of $succ$; for example, Rule 3 removes three I's from one term to the next (and therefore leaves the same number of I's modulo 3), whereas Rule 2 doubles the number of I's of a term.

Applying INFINITE-STATE-CEGAR leads to the second abstraction

$$A_2 = (\{S^{\equiv 0}, S^{\equiv 1}, S^{\equiv 2}\}, \{S^{\equiv 1}\}, E_2),$$

where E_2 is depicted below:

 .

The abstraction function $\alpha_2 : S \to \{S^{\equiv 0}, S^{\equiv 1}, S^{\equiv 2}\}$ takes $s \in S$ to $S^{\equiv \sharp_{\mathsf{I}} s}$. We have split the previous abstract state $S^{\equiv 0,2}$ into the two states $S^{\equiv 0}$ and $S^{\equiv 2}$, and updated our transition relation accordingly. Our set of abstract bad states is now $B_2 = \alpha_2(B) = \alpha_2(\{\mathsf{MU}\}) = \{S^{\equiv 0}\}$.

We observe straightaway that $A_2 \vDash \mathbf{AG}\neg B_2$. Lemma 5 then implies that $A \vDash \mathbf{AG}\neg B$, and hence that the MIU-system cannot derive the string MU.

In general, there are no hard and fast rules to decide on an initial abstraction or a suitable over-approximation to the successor function. As this example demonstrates, these choices may require a good deal of insight. However, we show in Section 5 that for hybrid systems one can find effective heuristics to derive candidate initial abstractions and successor functions.

4 Hybrid Systems

Hybrid systems are a class of infinite state systems that include both continuous and discrete state variables. This section presents the syntax and semantics of hybrid automata, which are used to model hybrid systems. We will illustrate these definitions with an example that models a simple car controller. The same example will be used in later sections to illustrate the CEGAR approach to the verification of hybrid systems.

4.1 Definition of Hybrid Automata

Definition 8 *Syntax of the Hybrid Automaton HA.* A *hybrid automaton* is a tuple $HA = (Z, z_0, X, inv, X_0, T, g, j, f)$ where

- Z is a finite set of *locations* with an *initial location* $z_0 \in Z$.
- $X \subseteq \mathbb{R}^n$ is the continuous state space.
- $inv : Z \to 2^X$ assigns to each location $z \in Z$ an invariant of the form $inv(z) \subseteq X$.
- $X_0 \subseteq X$ is the set of initial continuous states. The set of initial hybrid states of *HA* is thus given by the set of states $\{z_0\} \times X_0$.
- $T \subseteq Z \times Z$ is the set of *discrete transitions* between locations.
- $g : T \to 2^X$ assigns a *guard* set $g((z_1, z_2)) \subseteq X$ to $(z_1, z_2) \in T$.
- $j : T \times X \to 2^X$ assigns to each pair $(z_1, z_2) \in T$ and $x \in g((z_1, z_2))$ a *jump* set $j((z_1, z_2), x) \subseteq X$.
- $f : Z \to (X \to \mathbb{R}^n)$ assigns to each location $z \in Z$ a continuous vector field $f(z)$. We use the notation f_z for $f(z)$. The evolution of the continuous behavior in location z is governed by the differential equation $\dot{\chi}(t) = f_z(\chi(t))$. We assume that the differential equation has a unique solution for each initial value $\chi(0) \in inv(z)$. $\diamond$

The semantics of HA is defined by means of a trace transition system. Each state (z, x) in the trace transition system corresponds to a continuous state x within location z. Two such states, (z_1, x_1) and (z_2, x_2), are connected by a transition in the trace transition system if and only if state (z_2, x_2) can be reached from state (z_1, x_1) by a continuous evolution within location z_1 followed by a discrete transition to location z_2.

Definition 9 *Semantics of the Hybrid Automaton HA.* The semantics of a hybrid automaton HA is a *transition system* $TTS = (S, S_0, E)$ with:

- the set of all *hybrid states* (z, x) of HA,

$$S = \bigcup_{z \in Z} \bigcup_{x \in inv(z)} \{(z, x)\} \tag{1}$$

- the set of *initial hybrid states* $S_0 = \{z_0\} \times X_0$,
- transitions $(s_1, s_2) \in E$ with $s_1 = (z_1, x_1)$, $s_2 = (z_2, x_2)$, iff there exists $(z_1, z_2) \in T$ and a trajectory $\chi : [0, \tau] \to X$ for some $\tau \in \mathbb{R}^{>0}$ such that:
  - $\chi(0) = x_1$, $\chi(\tau) \in g((z_1, z_2))$,
  - $x_2 \in j((z_1, z_2), \chi(\tau))$,
  - $\dot{\chi}(t) = f_{z_1}(\chi(t))$ for $t \in [0, \tau]$,
  - $\chi(t) \in inv(z_1)$ for $t \in [0, \tau]$,
  - $x_2 \in inv(z_2)$.

A path $\sigma = (s_0, s_1, s_2, \ldots, s_m)$ of TTS is called a *trace* of HA, and we refer to TTS as the *trace transition system* of HA. $\diamond$

Definition 10 *Safety of a Hybrid Automaton.* For a hybrid automaton HA with a semantics as in Defn. 9, let $z_b \in Z \setminus \{z_0\}$ denote an *unsafe* location. HA is said to be *safe* with respect to z_b, denoted by $TTS \models \mathbf{AG}\neg z_b$ iff for all traces σ there is no $s \in \sigma$ with $s = (z_b, x)$ for some $x \in X$. We write $TTS \not\models \mathbf{AG}\neg z_b$ otherwise. $\diamond$

The extension of the analysis task to multiple initial locations and/or multiple unsafe locations is straightforward but is omitted here for simplicity.

4.2 Example

As a motivating example, we consider a simple controller that steers a car along a straight road. The car is assumed to drive at a constant speed $r = 2$, and its motion is modeled by the distance x from the middle of the road ($x = 0$ corresponds to the middle) and the heading angle γ ($\gamma = 0$ corresponds to moving straight ahead). Fig. 3 shows a scenario in which the car is initially on the road. The controller is able to detect whether the car is on the left or right border (i.e. $x \leq -1, x \geq 1$). Whenever the car enters the left border, the controller forces it to turn right until the car is back on the road again. Then a left turn is initiated, and continued until the car is again going straight ahead in the direction of the road, i.e. when the heading is aligned with the road ($\gamma = 0$). A similar strategy is employed when the car enters the right border.

Fig. 4 shows a hybrid automaton model for this example. Besides the position x and the heading angle γ, the description includes an internal timer c, that the controller uses to time the steering manoeuvres. The differential equations for these three continous variables depend on the location: we have $\dot{x} = -r \cdot sin(\gamma)$ in all locations except

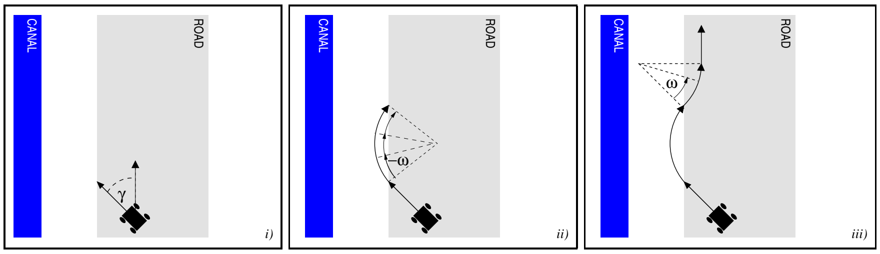


Figure 3: *i)* Initially, the car drives on the road with heading angle γ. *ii)* If the controller detects that the car has left the road, it corrects the heading by turning right to avoid the canal. *iii)* Once the car is back on the road, a left turn is initiated until the car moves straight again.

`in_canal`. The derivative of γ varies when a border is reached. On the border the motion of the car describes an arc with the angular velocity $\dot{\gamma} = -\omega = -\pi/4$ (or $\omega = \pi/4$ respectively), i.e., the arc is part of a circle with radius r/ω. The timer c measures the time period which the car spends on the border. In the correction modes the timer decreases with double rate, i.e., the correction takes half the time as that spent previously by the car on the border. Since the sign of $\dot{\gamma}$ is reversed when the car moves back on the road, the angle has the value zero when the correction mode is left ($c = 0$), i.e., the car then moves along the road. During this correction it might, however, happen that the other border is reached, which means that the controller then switches to the strategy of the corresponding location.

The three continuous variables are initialized to $-1 \leq x \leq 1$ (the car is on the road), $-\pi/4 \leq \gamma \leq \pi/4$, and $c = 0$. It has to be verified for this set of initial states whether the given control strategy guarantees that the unsafe location `in_canal` (z_b) is never reached. The following sections explain how this task can be solved by abstraction-based and counterexample-guided verification.

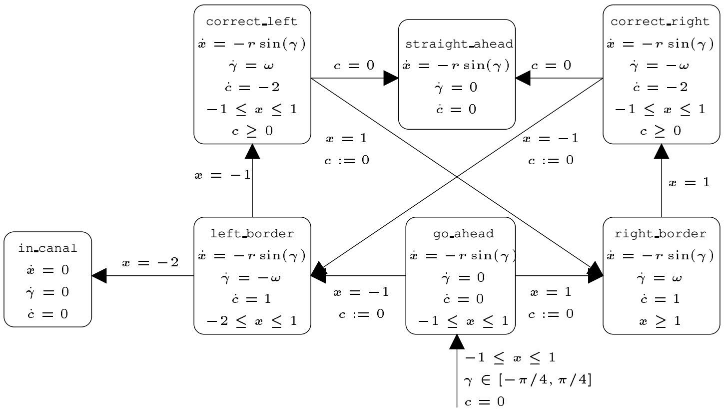


Figure 4: Hybrid automaton that models the car steering example.

5 Refinement of Abstractions for Hybrid Systems

This section applies the general concepts of Section 3 to hybrid systems. We present specific solutions for the two crucial steps in INFINITE-STATE-CEGAR, *validation* and *refinement*. The key to the validation step is the computation of successor states for a given set of states in the trace transition system. Starting from the initial set, the validation procedure computes the successors along the counterexample until either the unsafe location z_b is reached or a transition is determined to be spurious. The computation of sets of successor states is usually the most expensive step in hybrid system verification. Successor sets can be computed and represented *exactly* only for certain sub-classes of hybrid systems [16, 17]. However, several approaches to over-approximate successor sets have been published, as e.g., successor set approximations by hyper-rectangles [18], general polyhedra [19], projections to lower dimensional polyhedra [20], or ellipsoids [21]. Most of these approaches aim at providing an efficient way to obtain conservative but tight approximations to sets of reachable states for hybrid systems.

We note that the main difficulties introduced by hybrid systems—as opposed to finite-state or discrete infinite-state systems such as the MIU-system—originate from the fact that the transition relation for hybrid systems is *implicit*, derived from differential equations which in general do not even have analytical solutions. Even when analytical solutions are available, the representation and computation of successor sets is non-trivial, making it difficult to manufacture reasonably tight over-approximations to the successor function. Consequently, given an abstraction function, one has to construct the transition relation by focusing on one transition at a time in the abstract system. By contrast, in the finite-state and discrete infinite-state cases, one can usually define the transition relation for an abstraction function globally—many transitions can be constructed simultaneously.

The verification framework presented here can include different techniques to over-approximate the set of successors. The idea of using different methods is motivated by the trade-off between the accuracy and the computational complexity of different methods. If, e.g., a faster but maybe less accurate technique is sufficient to refute a counterexample, then there is no need to use a more computationally expensive method.

In the following, we first describe how an initial abstraction for a hybrid automaton can be obtained, and then focus on the validation of counterexamples and refinement of abstract models based on the use of different methods for computing successor states.

5.1 Abstraction of Hybrid Systems

For the first step of the INFINITE-STATE-CEGAR algorithm, the construction of an initial abstraction, we introduce one abstract state for each location of HA. This means that two hybrid states (z_i, x_i) and (z_j, x_j) of TTS are mapped to the same abstract state if and only if $z_i = z_j$. This rule applies for all but the initial location, for which we introduce one abstract state $\hat{s}_0$ to represent all initial hybrid states of TTS, and another one ($\hat{s}'_0$) to represent the remaining hybrid states corresponding to the location z_0:

Definition 11 *Initial Abstraction of Hybrid Systems.* Given a hybrid automaton HA with $Z = \{z_0, z_1, \ldots, z_{n_z}\}$, let S denote the set of hybrid states as defined in (1). For $i \in \{0, 1, \ldots, n_z\}$, we define the abstraction function $\alpha : S \to \hat{S}$ by:

$$\alpha(z_i, x) = \begin{cases} \hat{s}_0 & \text{if } i = 0 \wedge x \in X_0 \\ \hat{s}_0' & \text{if } i = 0 \wedge x \notin X_0 \\ \hat{s}_i & \text{otherwise} \end{cases} \tag{2}$$

and the initial abstract model $A = (\hat{S}, \hat{S}_0, \hat{E})$ is defined by ($i \in \{0, 1, \ldots, n\}$, $j \in \{0, 1, \ldots, n_z\}$):

- $\hat{S} = \{\hat{s}_0', \hat{s}_0, \hat{s}_1, \ldots, \hat{s}_n\}$
- $\hat{S}_0 = \{\hat{s}_0\}$
- $\hat{E} = \{(\hat{s}_i, \hat{s}_j) | (z_i, z_j) \in T\} \cup \{(\hat{s}_0', \hat{s}_j) | (z_0, z_j) \in T\} \cup \{(\hat{s}_i, \hat{s}_0') | (z_i, z_0) \in T\}$

$\diamond$

The initial abstract model represents the discrete structure of the hybrid system without regard to the continuous dynamics and guards. Given this definition, it has to be shown that A is indeed an abstract model of the underlying trace transition system, i.e., that it fulfills Defn. 2:

Lemma 6 For *HA* with trace transition system $TTS = (S, S_0, E)$, let $A = (\hat{S}, \hat{S}_0, \hat{E})$ denote the initial abstract model for TTS. Then, $A \succeq TTS$. $\square$

Example (cont.) Fig. 5 depicts the initial abstract model of the hybrid system in Fig. 4. It is a copy of the discrete part of the hybrid system, except that the initial location is divided into two parts: $\hat{s}_0$ represents the states in location `go_ahead` with $x \in [-1, 1]$, $\gamma \in [-\pi/4, \pi/4]$ and $c = 0$, and $\hat{s}_0'$ all other states in `go_ahead`. The abstract states $\hat{s}_1$ to $\hat{s}_6$ represent the hybrid states of the other locations (`left_border`, `right_border`, `correct_left`, `correct_right`, `straight_ahead` and `in_canal`, respectively). ⧫

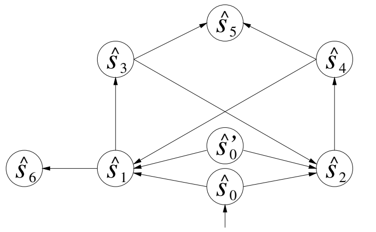


Fig 5. Initial abstract model of the hybrid system depicted in Fig. 4

5.2 Over-approximation of the Sets of Successors

We now turn to the question of computing sets of successor states, as required in the validation and refinement steps. The goal is to use different over-approximations with different precisions and different computational requirements. For technical reasons it is convenient to define $\overline{succ}$ in terms of pairs $S_1, S_2 \subseteq S$, where S_1 is a set of *source states* and S_2 is a set of *potential successor states*. $\overline{succ}(S_1, S_2)$ is a conservative approximation of those sucessors of states in S_1 that lie in S_2.

Definition 12 *Over-approximation of successor states.* Let HA be a hybrid automaton with the trace transition system $TTS = (S, S_0, E)$, and let A and α be defined as in Defn. 11. For a transition $(\hat{s}_1, \hat{s}_2) \in \hat{E}$ of A, we call $S_1 := \alpha^{-1}(\hat{s}_1)$ the set of *hybrid source states* and $S_2 := \alpha^{-1}(\hat{s}_2)$ the set of *potential hybrid successor states*. Then $\overline{succ} : (2^S \times 2^S) \rightarrow 2^S$ is an *over-approximation* of the hybrid successor states in S_2 iff the following holds:

- $\overline{succ}(S_1, S_2) \subseteq S_2$,
- $\overline{succ}(S_1, S_2) \supseteq succ(S_1) \cap S_2$. $\diamond$

A possible explicit realization of the operator $\overline{succ}$ with respect to a given set S_2 combines the following steps: (a) By approximating the continuous evolution for all states in S_1, the reachable subset of the guard set $g(t)$ is determined, where $t = (z_1, z_2) \in T$ is the transition of HA that corresponds to the transition $(\hat{s}_1, \hat{s}_2) \in \hat{E}$ of A. Usually, this step is the most costly of the whole verification procedure; (b) the jump function $j(t, x)$ is applied to all hybrid states (z_1, x) which are in the reachable subset of $g(t)$; (c) the image of $j(t, x)$ is intersected with the set S_2 of potential hybrid successor states.

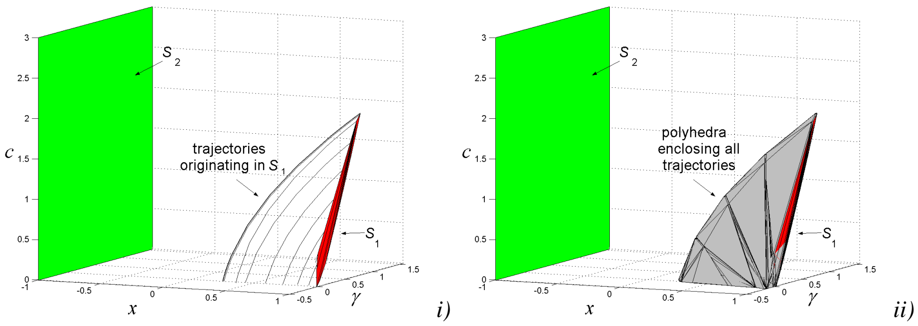


Figure 6: All trajectories that originate in S_1 leave the invariant when $c = 0$, and none of them comes close to S_2. Figure (*i*) shows the result of the optimization method. Figure (*ii*) the result of the method that enclose the trajectories by polyhedra.

Example (cont.) Our prototype implementation uses two different methods, $\overline{succ}_{coarse}$ and $\overline{succ}_{tight}$, to over-approximate the set of successor states. Fig. 6 illustrates these two methods for the discrete transition from `correct_right` to `left_border`. For location `correct_right` we choose S_1 as subset of the plane $x = 1$, and S_2 as all states of location `left_border` that satisfy the invariant $-2 \leq x \leq -1$. Fig. 6 depicts S_1 and the face of S_2 that coincides with the guard $x = -1$. The transition is not spurious if there exists a trajectory that starts in S_1 and ends in S_2 without leaving the invariant of `correct_right` ($-1 \leq x \leq 1 \wedge c \geq 0$). Fig. 6 (i) depicts a number of trajectories that start in S_1, none of which reach S_2.

The first method $\overline{succ}_{coarse}$ poses the existence question for a trajectory between S_1 and S_2 as an optimization problem. The distance between a trajectory and S_2 is defined as the minimum distance between all points on the trajectory and S_2. If the global minimum over all trajectories that start in S_1 is strictly greater than zero, then no successor state of S_1 exists in S_2. In this case $\overline{succ}_{coarse}(S_1, S_2)$ returns an empty set. If the minimum distance is zero, at least one corresponding concrete path exists, and $\overline{succ}_{coarse}(S_1, S_2)$ returns the entire set S_2 as an over-approximation of the set of successor states. The bold trajectory in Fig. 6 (i) is the optimal trajectory. Its distance to S_2 is greater than zero, and there is hence no trajectory from S_1 to S_2.

The second method $\overline{succ}_{tight}$ computes polyhedra that enclose all trajectories that originate in S_1. This over-approximation with polyhedra is based on work presented in [19]. The set of successor states $\overline{succ}_{tight}(S_1, S_2)$ is then obtained by intersecting the polyhedra with S_2. Fig. 6 (ii) shows that this intersection is empty, i.e. there are no successors of S_1 in S_2. ♦

5.3 Validation and Refinement

The INFINITE-STATE-CEGAR algorithm makes a clear distinction between the validation of a counterexample, and the refinement of the abstract model. For hybrid systems, we propose a slightly different approach, in which the steps of validation and refinement are interleaved. We assume to have a set of over-approximation techniques $\overline{succ}_1, \ldots, \overline{succ}_n$ that can (but not necessarily need to) establish a hierarchy of coarse to tight approximations.

The proposed algorithm for the combined validation and refinement steps of a counterexample is shown in Fig. 7. Let $\sigma = (\hat{s}_0, \ldots, \hat{s}_m)$ denote a counterexample of the abstract model A. The algorithm consists of two nested loops. The outer loop corresponds to checking each transition of the counterexample. The inner loop applies each of the over-approximation techniques to the current transition of the counterexample, and, depending on the result, one of the two refinement operations is executed: If an over-approximation technique $\overline{succ}_l$ reveals that the current transition is spurious, i.e. $S_k^{reach} = \emptyset$, then the transition is removed from the abstract model by ρ_{purge}. When a transition is removed, the set of behaviors of A does not include the current counterexample anymore, and thus the combined validation and refinement of the current counterexample is completed.

If on the other hand, $\overline{succ}_l$ returns a non-empty set S^k_{reach} and this set is a true subset of the states corresponding to $\hat{s}_k$, the function ρ_{split} divides $\hat{s}_k$ into two states $\hat{s}_k^{reach}$ and $\hat{s}_k^{comp}$ (cf. Defn. 6). In this case $\sigma = (\hat{s}_0, \ldots, \ldots, \hat{s}_{k-1}, \hat{s}_k^{reach}, \hat{s}_{k+1} \ldots, \hat{s}_m)$ remains a counterexample of the refined model. Thus the algorithm continues with the next transition $(k + 1)$ until either $S_k^{reach} = \emptyset$ or until the last transition of the counterexample is validated.

There is some freedom in combining the steps of validation and refinement, i.e., the scheme in Fig. 7 is just one possible implementation. One interesting alternative is to apply the coarsest method for validation first to all transitions in the abstract counterexample, or to apply state splitting (ρ_{split}) only based on the result of the most accurate approximation method $\overline{succ}_n$.

The algorithm as proposed in Fig. 7 has two possible outcomes: either it is proved that a forbidden state cannot be reached or that there exists a counterexample that cannot be refuted. Since the validation procedure relies on over-approximations, it can-

```
FOR k = 1,...,m
    FOR l = 1,...,n
        S_k^reach := succ_l(S_{k-1}^reach, α^{-1}(ŝ_k))
        IF S_k^reach = ∅
            A := ρ_purge(A, α, (ŝ_{k-1}, ŝ_k))
            RETURN //jump out of both loops
        ELSEIF S_k^reach ⊊ α^{-1}(ŝ_k)
            (A, α) := ρ_split(A, α, (ŝ_{k-1}, ŝ_k))
        ENDIF
    ENDFOR
ENDFOR
```

Figure 7: Refinement and validation steps for hybrid systems.

not be guaranteed that this abstract counterexample corresponds to a concrete one. In this case, under-approximations of sets of successor states can possibly be used to prove that a counterexample exists: Assume that the procedure terminates with a counterexample $\sigma = (\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_k, \ldots, s_m)$, no transition of which could be refuted. Similarly to Defn. 12, we can define an *under-approximation* of successor states $S_k^{reach} = \underline{succ}(S_{k-1}^{reach}, \alpha^{-1}(\hat{s}_k))$ which returns a set $S_k^{reach} \subseteq \alpha^{-1}(\hat{s}_k)$ guaranteed to contain only true successors of S_{k-1}^{reach}. If this operator is applied along the counterexample (from $k = 1$ to $k = m$) and $S_n^{reach} \neq \emptyset$, there exists at least one path for the hybrid system which violates the safety property.

As noted earlier, when using over-approximations, there is no guarantee that a spurious counterexample can be refuted. The likelihood of refuting spurious counterexamples can be increased, however, by using tighter polyhedral approximations. When the over-approximations are tight, the presence of an unrefuted yet spurious counterexample is indicative of a very slim error margin separating the reachable states from the bad ones. We would argue that when an unrefuted spurious counterexample is encountered, it may be better to redesign the implementation of our hybrid system so as to increase the error margin, rather than risk facing an actual failure in a real-world implementation of this system.

If we compare the verification algorithm for hybrid systems presented here to similar approaches in the literature such as [9], we note that the main advantage of our method is that, in relying on spurious counterexamples to refine our successive abstract models, we are focusing on the *local* properties of our system that are relevant to establish or invalidate a *particular* specification. This leaves us free, for instance, to employ cheap gross over-approximations of successor states in irrelevant areas of the hybrid system.

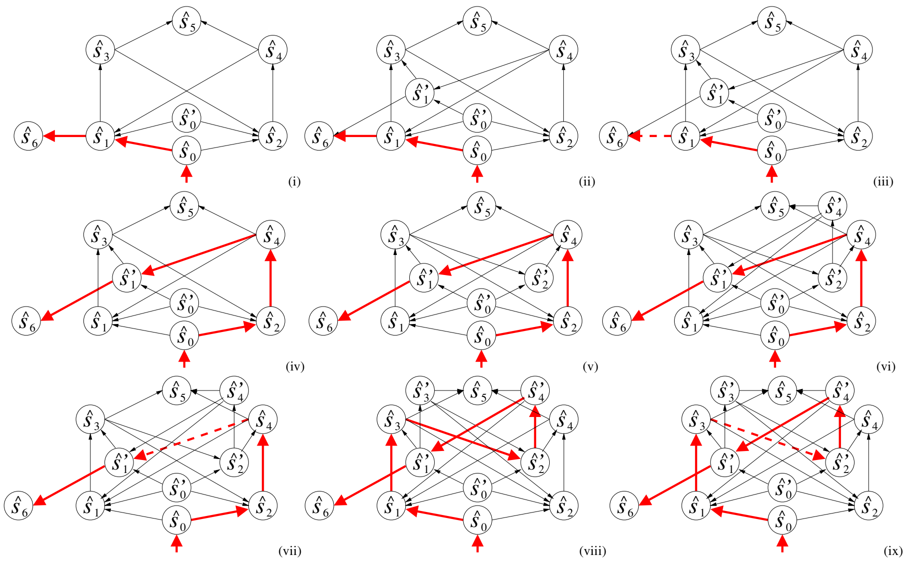


Figure 8: Counterexample-guided abstraction illustrated for the car steering problem.

Example (cont.) The requirement that the hybrid model in Fig. 4 should never enter the location `in_canal` translates into the reachability question for state $\hat{s}_6$ of the abstract model in Fig. 5. The first counterexample for the initial abstract model is $\sigma_1 = (\hat{s}_0, \hat{s}_1, \hat{s}_6)$ (see Fig. 8 (i)). The validation procedure considers first the transition $(\hat{s}_0, \hat{s}_1)$ which corresponds to the transition between `go_ahead` and `left_border` in the hybrid automaton. As a first step, $\overline{succ}_{coarse}(S_0, \alpha^{-1}(\hat{s}_1))$ is computed with the result that the minimum distance over all initial states is zero. This is obvious from the fact that those states of the initial set for which $x = -1$ enable the transition guard immediately. Thus, $\overline{succ}_{coarse}$ returns the entire invariant of location `left_border` as set S_2. The next step is to compute $S_2^{reach} = \overline{succ}_{tight}(S_0, \alpha^{-1}(\hat{s}_1))$. The algorithm then splits $\hat{s}_1$ so that $\hat{s}_1$ represents the set S_2^{reach}, and the new abstract state $\hat{s}_1'$ represents $S_2 \setminus S_2^{reach}$ (Fig. 8 (ii)).

Since the counterexample has not been eliminated yet, the transition $(\hat{s}_1, \hat{s}_6)$ is considered next. Method $\overline{succ}_{coarse}$ finds that the minimal distance between the trajectories that start in S_2^{reach} and the guard $x = -2$ is greater than zero. This means that no trajectory reaches the guard, and the corresponding transition is removed (Fig. 8 (iii)).

The procedure continues with the next counterexample $\sigma_2 = (\hat{s}_0, \hat{s}_2, \hat{s}_4, \hat{s}_1', \hat{s}_6)$, as depicted in Fig. 8 (iv). As for the first counterexample, the abstract state $\hat{s}_2$ is split into the states that are reachable from the initial set S_0, and the remainder (Fig. 8 (v)). Then, the procedure moves forward one transition and splits state $\hat{s}_4$ as a result of applying $\overline{succ}_{tight}$. The reachable part is represented by $\hat{s}_4$ in Fig. 8 (vi). Method $\overline{succ}_{coarse}$ then finds that one cannot reach any state that is represented by $\hat{s}_1'$ from this set, and the transition $(\hat{s}_4, \hat{s}_1')$ can be deleted from A (Fig. 8 (vii)).

The final counterexample is $\sigma_3 = (\hat{s}_0, \hat{s}_1, \hat{s}_3, \hat{s}_2', \hat{s}_4', \hat{s}_1', \hat{s}_6)$. The state $\hat{s}_1$ was already split for the first counterexample. Similarly to the procedure for the counterex-

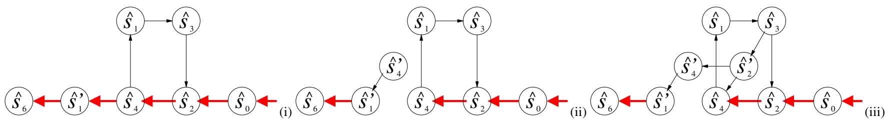


Figure 9: Validation an refinement of fragments illustrated for a counterexample of the car steering problem.

ample σ_2, abstract state $\hat{s}_3$ is split as depicted in Fig. 8 (viii). It can then be shown that transition $(\hat{s}_3, \hat{s}'_2)$ is spurious, which eliminates the last counterexample (Fig. 8 (ix)). Consequently, the abstract state $\hat{s}_6$ is not reachable, and thus the same applies for the location `in_canal` of the hybrid automaton. ♦

5.4 Validation and Refinement of Fragments of Counterexamples

The initial abstraction of the example in Fig. 5 contains only *two* counterexamples without cycles, $(\hat{s}_0, \hat{s}_1, \hat{s}_6)$ and $(\hat{s}_0, \hat{s}_2, \hat{s}_4, \hat{s}_1, \hat{s}_6)$. However, to show that no bad state is reachable, *three* counterexamples in the series of abstractions were considered and refuted (cf. Fig. 8). Hence, refining an abstract model, to eliminate a particular counterexample, may introduce new counterexamples. In this subsection we show that considering fragments of counterexamples, rather than complete counterexamples, can reduce the total number of counterexample that have to be considered. This often results in a significant speed-up of the verification process.

The main reason for considering fragments is as follows. The validation and refinement routine that we presented in the previous subsection typically refutes a counterexample (indeed, when a counterexample is not refuted, the algorithm stops). The counterexample refutation case can be made more efficient by the following observation. In the previous subsection, a (spurious) counterexample $(\hat{s}_0, \ldots, \hat{s}_m)$ is refuted by showing that no corresponding concrete path $(s_0, \ldots, s_m)$ exists. Interestingly, showing that any one of the transitions $(\hat{s}_i, \hat{s}_{i+1})$ in the counterexample is spurious is a sufficient condition for the non-existence of a corresponding concrete path.

Alternatively, we can also conclude that a counterexample is spurious if one of the fragments $(\hat{s}_i, \hat{s}_{i+1}, \hat{s}_{i+2})$ is spurious, in other words if there is no corresponding concrete path (s_i, s_{i+1}, s_{i+2}) in the concrete model. In general, one can define spurious fragments of length n. Validation and refinement of such fragments of counterexamples can be done in a similar way as for complete counterexamples.

We now illustrate that validation and refinement of short fragments can increase the efficiency of the verification process. Clearly, if one can refute a fragment of a counterexample, e.g., a single transition, then the entire counterexample is spurious. If a counterexample can be refuted by considering a fragment of length n, it can surely be refuted by considering fragments of length $n+1$. However, using a fragment of length $n+1$ may have the undesirable side-effect of introducing new counterexamples, or at least more counterexamples than the method based on fragments of length n.

Example (cont.) Consider as an example Fig. 9 (i), which depicts part of the abstract model in Fig. 8 (iv) and contains the counterexample. Note that there is a loop that enters the counterexample at $\hat{s}_2$ and leaves it at $\hat{s}_4$. For this car steering example it can be shown that the fragment $(\hat{s}_2, \hat{s}_4, \hat{s}'_1)$ is spurious, even though neither of the transitions is spurious on its own. This means that validation and refinement of fragments of length 2 removes the counterexample as depicted in Fig. 9 (ii).

If we consider the complete counterexample instead, we also find that the counterexample is spurious. But in this case we would also split $\hat{s}_2$, which introduces an additional counterexample that exploits the loop, as shown in Fig. 9(iii). In general, whenever we split all abstract states between the entry and exit points of a loop, it will 'open' the loop, and inevitably create an additional counterexample.

There is little choice if these states have to be split to refute the counterexample. Consider for instance the first counterexample in Fig. 8 (i). This counterexample can only be eliminated by splitting $\hat{s}_1$. But if it is possible to refute a short fragment, rather than a long one, this should be preferred. If we apply validation and refinement to fragments of length 2 of the counterexample in Fig. 8 (iv), we are guaranteed that it will not introduce new counterexamples. If it then succeeds, we can be sure that the number of counterexamples decreases. In this particular case, refuting fragment $(\hat{s}_2, \hat{s}_4, \hat{s}'_1)$ eliminates all other counterexamples, as they also include this fragment.

5.5 Experimental Results

Experimental results for a prototype implementation of the procedure indicate its advantages over existing methods. We apply the prototype first to the car steering example that was discussed throughout this paper. Then a larger and more challenging example on an adaptive cruise control system that was put forward in the MoBIES project [22] is discussed.

5.5.1 Car Steering Example

For the car steering example we take as baseline INFINITE-STATE-CEGAR as described in Subsection 5.3 with the only successor operator $\overline{succ}_{tight}$. We refer to this method as INFINITE-STATE-CEGAR-I. For the car steering example this method computes the same number of $\overline{succ}_{tight}$ operations as a breadth–first application of the successor operator. Breadth-first application is the most prevalent method used for model checking hybrid systems.

We compare this method with two other instances of INFINITE-STATE-CEGAR. INFINITE-STATE-CEGAR-II refines and validates complete counterexamples using the two different methods, as described in Subsection 5.3. The third instance INFINITE-STATE-CEGAR-III first validates single transitions using $\overline{succ}_{coarse}$. Next, it considers fragments of length 2, using $\overline{succ}_{coarse}$. Finally, the third validation and refinement scheme considers fragments of length 2, but uses $\overline{succ}_{tight}$ for the first transition, and $\overline{succ}_{coarse}$ for the second. If these three schemes fail to refute the counterexample, the complete counterexample is considered, using the same routine as the second instance of INFINITE-STATE-CEGAR.

For the car steering example the following results are obtained when run on a Pentium 4, 1.4GHz. INFINITE-STATE-CEGAR-I considers three counterexamples, computes $\overline{succ}_{tight}$ five times, and takes 117 seconds to verify that the car steering example is safe. INFINITE-STATE-CEGAR-II considers the same counterexamples but computes $\overline{succ}_{tight}$ only three times, and finishes in 70 seconds. INFINITE-STATE-CEGAR-III considers only two counterexamples, and computes $\overline{succ}_{tight}$ only once. Since, this particular successor was easy to compute, the overall time drops to 10 seconds.

5.5.2 MoBIES Adaptive Cruise Control System

The model that we use for the adaptive control experiments is based on a Simulink/ Stateflow model [23]. The adaptive cruise control is part of a vehicle-to-vehicle coordination system. The part of this system that we consider comprises two modes: the cruise control mode (cc-mode) in which a car tries to keep a constant speed, and an adaptive cruise control mode (acc-mode), in which the car tries to stay a safe distance behind a vehicle ahead of it. The acc-controller switches into acc-mode whenever the distance between the car and a vehicle ahead falls below a certain threshold. This threshold depends linearly on car speed.

The system also includes an automatic transmission system with four gears. Depending on the speed of the car it will switch between the different gears. The hybrid automaton that models both the acc-controller and the automatic transmission has 8 locations for the normal operation and one additional state that is entered on collisions, when the distance between the cars is zero. Obviously, this is the location that should not be reachable. The model takes into account the distance between two cars, their relative velocity and the velocity of the following car. The differential equations that describe the continuous behavior are non-linear, mainly due to saturation; for each gear there are upper and lower bounds on the possible acceleration.

For the adaptive cruise control example the hybrid model checker CheckMate[3] is used as a baseline, since it is possible for this case study to generate a CheckMate model that exhibits the exact same behavior as our model. CheckMate takes 770 seconds to verify that the system is safe. We compare this result to our two approaches INFINITE-STATE-CEGAR-II and INFINITE-STATE-CEGAR-III. INFINITE-STATE-CEGAR-II considers 46 counterexamples, and computes 11 times $\overline{succ}_{tight}$, in 450 seconds. The resulting safe abstraction has 29 states. INFINITE-STATE-CEGAR-III only considers 10 potential counterexamples, computes $\overline{succ}_{tight}$ just once, and takes only 39 seconds. The resulting abstraction has just 15 states. Five of the counterexamples have been refuted by considering single transitions; for example, when the following car is in first gear and in acc-mode, then it cannot collide with the leading car. All other counterexamples were refuted by considering segments of length 2. For example, one such refuted counterexample corresponds to the case when the car is in third gear and switches to acc-mode—this cannot lead to a collision.

6 Conclusions

This paper presents a new method for using counterexamples to refine abstractions of hybrid systems. The principal alternative for verifying the safety properties considered in this paper is to compute the reachable states for the hybrid system using a breadth-first application of the successor operator $succ$. It is apparent that the INFINITE-STATE-CEGAR procedure can be faster than breadth-first reachability when the safety property does not hold for the concrete system, since in this case it is possible for the model checker to quickly find a true counterexample. On the other hand, if the safety property holds, refuting one counterexample may implicitly refute others. However, the INFINITE-STATE-CEGAR procedure continues until all possible counterexamples have been explored (and indeed, may not terminate), which is in some cases equivalent to the breadth-first reachability computation. Nevertheless, we have shown here that INFINITE-STATE-CEGAR offers the possibility of using multiple methods for computing approximations to the successor states.

References

[1] Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. Volume 1855 of LNCS., Springer (2000) 154–169

[2] Kurshan, R.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press (1994)

[3] Silva, B., Stursberg, O., Krogh, B., Engell, S.: An assessment of the current status of algorithmic approaches to the verification of hybrid systems. In: IEEE Conf. on Decision and Control. (2001) 2867–2874

[4] Henzinger, T., Minea, M., Prabhu, V.: Assume-guarantee reasoning for hierarchical hybrid systems. In: HSCC. Volume 2034 of LNCS., Springer (2001) 275–290

[5] Frehse, G., Stursberg, O., Engell, S., Huuck, R., Lukoschus, B.: Modular analysis of discrete controllers for distributed hybrid systems. In: IFAC World Congress. (2002)

[6] Alur, R., Henzinger, T., Lafferriere, G., Pappas, G.: Discrete abstractions of hybrid systems. Proceedings of the IEEE **88** (2000) 971–984

[7] Alur, R., Dang, T., Ivancic, F.: Reachability analysis of hybrid systems via predicate abstraction. In: HSCC. Volume 2289 of LNCS., Springer (2002) 35–48

[8] Tiwari, A., Khanna, G.: Series of abstractions for hybrid automata. In: HSCC. Volume 2289 of LNCS., Springer (2002) 465–478

[9] Chutinan, A., Krogh, B.: Verification of infinite-state dynamic systems using approximate quotient transition systems. IEEE Transactions on Automatic Control **46** (2001) 1401–1410

[10] Clarke, E., Gupta, A., Kukula, J., Strichman, O.: Sat based abstraction-refinement using ilp and machine learning techniques. In: CAV. LNCS, Springer (2002)

[11] Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of c programs. In: PLDI. SIGPLAN 36(5) (2001)

[12] Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Symp. on Principles of Programming Languages, ACM Press (2002) 58–70

[13] Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)

[14] Henzinger, T.: The theory of hybrid automata. In: Symp. on Logic in Computer Science, IEEE Computer Society Press (1996) 278–292

[15] Hofstadter, D.: Gödel, Escher, Bach: An Eternal Golden Braid. Basic Books (1989)

[16] Lafferriere, G., Pappas, G., Yovine, S.: A new class of decidable hybrid systems. In: HSCC. LNCS 1569, Springer (1999) 103–116

[17] Henzinger, T., Kopke, P., Puri, A., Varaiya, P.: What's decidable about hybrid automata? In: Symposium on Theory of Computing, ACM Press (1995) 373–382

[18] Dang, T., Maler, O.: Reachability analysis via face lifting. In: HSCC. LNCS 1386, Springer (1998) 96–109

[19] Chutinan, A., Krogh, B.: Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In: HSCC. LNCS 1569, Springer Verlag (1999) 76–90

[20] Greenstreet, M., Mitchell, I.: Reachability analysis using polygonal projections. In: HSCC. LNCS 1569, Springer (1999) 103–116

[21] Kurzhanski, A., Varaiya, P.: Ellipsoidal techniques for reachability analysis. In: HSCC. LNCS 1790, Springer (2000) 203–213

[22] MoBIES: (DARPA project.) http://vehicle.me.berkeley.edu/mobies/vehicle/.

[23] Girard, A.R., de Souza, J.B., Misener, J.A., Hedrick, J.K.: A control architecture for integrated cooperative cruise control and collision warning systems. In: Proccedings of the Control and Decision Conference, Florida (2001) 1–1

Appendix

Proof of Lemma 1.
Proof. By contradiction: If $C \not\models$ **AG**$\neg B$, then at least one path $\sigma = (s_0, s_1, \ldots, b)$ with $b \in B$ must exist for C. From Defn. 2, it follows that the corresponding abstract counterexample $\hat{\sigma} = (\hat{s}_0, \hat{s}_1, \ldots, \hat{b})$ of A is a counterexample which contradicts the premise $A \models$ **AG**$\neg \hat{B}$. ■

Proof of Lemma 2.
Proof. (i) $A \succeq A'$. It follows straightforwardly that A is an abstract model of A' with abstraction function α'' as defined in Defn. 6.
(ii) $A' \succeq C$. From the above definitions of $A' = (\hat{S}', \hat{S}'_0, \hat{E}')$ and α', it follows that A' would be an abstract model of C, if $\hat{E}'$ also included the transition $(\hat{s}_1, \hat{s}_2^{comp})$. However, since S_2^{reach} and S_2^{comp} are disjoint, this abstract transition does not correspond to any concrete transition and can therefore be omitted. ■

Proof of Lemma 3.
Proof. (i) $A \succeq A'$. The corresponding abstraction function is the identity. Since A has just an additional transition it is an abstract model of A'.
(ii) $A' \succeq C$. The abstraction function for this abstraction is α. We can then omit the abstract transition $(\hat{s}_1, \hat{s}_2)$, since it does not correspond to any concrete transition. ■

Proof of Lemma 4.
Proof. (i) If the algorithm terminates with "B reachable", then the set of reachable states in the concrete model is non-empty along the path of the last checked counterexample. Formally, $S_k^{reach} \neq \emptyset$, $k = 0, \ldots, m$ due to the conditions in the IF statement ($S_k^{reach} \cap B \neq \emptyset$) and the WHILE statement ($S_k^{reach} \neq \emptyset$ AND $k < m$).

We can now show that the last checked counterexample in the algorithm is not spurious. To do so, we first show that for each k, all $s_k \in S_k^{reach}$ can be reached by paths in the concrete model. The proof is done by induction on k. For $k = 0$, each $s_0 \in S_0^{reach}$ can be reached by a path of length zero. For $k > 0$, for each $s_k \in S_k^{reach}$ there exists an $s_{k-1} \in S_{k-1}^{reach}$ such that $(s_{k-1}, s_k) \in E$ (by definition of the *succ* operator). By induction, s_{k-1} is reachable by some concrete path $(s_0, \ldots, s_{k-1})$, hence s_k is reachable via the concrete path $(s_0, \ldots, s_k)$.

Since for each k, all $s_k \in S_k^{reach}$ can be reached by paths in the concrete model, there are paths $(s_0, s_1, \ldots, s_m)$ with $s_m \in S_m^{reach} \cap B$. Each such path corresponds to a counterexample in the concrete model. Thus, $C \models$ **AG**$\neg B$.
(ii) If the algorithm terminates with "B not reachable", then it was not possible to find any counterexample for the current abstract model A. But since A is in each step an abstraction of C we can conclude by Lemma 1 that $C \models$ **AG**$\neg B$ holds. ■

Proof of Lemma 5.
Proof. The proof follows the same lines as the corresponding case of Lemma 4 and is therefore omitted.

Proof of Lemma 6.

Proof. We show that α as defined in Defn. 11 is an abstraction function. The first condition in Defn. 2 follows directly from the definition of α. To show the second condition, it must be proved that

$\hat{E} = \{(\hat{s}_i, \hat{s}_j) | (z_i, z_j) \in T\} \cup \{(\hat{s}'_0, \hat{s}_j) | (z_0, z_j) \in T\} \cup \{(\hat{s}_i, \hat{s}'_0) | (z_i, z_0) \in T\} \supseteq \{(\hat{s}_i, \hat{s}_j) | \exists s_i, s_j \in S : (s_i, s_j) \in E, \hat{s}_i = \alpha(s_i), \hat{s}_j = \alpha(s_j)\}$.

Assume $(s_i, s_j) \in E$, and $s_i = (z_i, x_i)$ and $s_j = (z_j, x_j)$ with $x_i, x_j \in X$ and $i, j \neq 0$. Then, it follows from the definition of E in Defn. 9 that $(z_i, z_j) \in T$. Thus, $(\hat{s}_i, \hat{s}_j) \in \hat{E}$. The other cases ($i = 0$ or $j = 0$) can be shown in a similar way. ■