Hybrid Automata-based CEGAR for Rectangular Hybrid Systems

Pavithra Prabhakar, Sridhar Duggirala, Sayan Mitra, Mahesh Viswanathan

Abstract. In this paper we present a framework for carrying out counterexample guided abstraction-refinement (CEGAR) for systems modelled as rectangular hybrid automata. The main difference, between our approach and previous proposals for CEGAR for hybrid automata, is that we consider the abstractions to be hybrid automata as well. We show that the CEGAR scheme is semi-complete for the class of rectangular hybrid automata and complete for the subclass of initialized rectangular automata. We have implemented the CEGAR based algorithm in a tool called Hare, that makes calls to HyTech to analyze the abstract models and validate the counterexamples. Our experiments demonstrate the usefulness of the approach.

1 Introduction

Direct model checking of realistic hybrid systems is in general undecidable and often foiled by the state-space explosion problem. Hence, one has to rely on some sort of abstraction. Finding the right abstraction is in itself a difficult problem. To this end, the *counterexample guided abstraction refinement* (CEGAR) [7] technique (see Section 3) which combines automatic refinement with model checking has gained preeminence in a number of contexts [4, 19, 8, 14] including in timed and hybrid systems [2, 6, 5, 24, 11, 23, 9, 20].

The space over which CEGAR performs the abstractions and refinements is key in determining both the efficiency (of model checking) and the completeness of the procedure. For example, in [2, 6, 5, 24, 23] abstraction-refinement is carried out in the space of finite-state discrete transition systems. Computing the transitions for this abstract finite state machine involves computing the unbounded time reachable states from the states in the concrete system corresponding to a particular state in the abstract system, which is difficult in practice for hybrid systems with complex continuous dynamics.

In this paper, we investigate CEGAR in the context of abstractions which are hybrid systems as well. When compared to using finite-state abstractions, using hybrid abstractions in a CEGAR framework requires carrying out computationally simpler tasks when constructing abstractions, refining them and validating counterexamples. Instead of the computationally expensive unbounded time successor computation, constructing hybrid abstractions only involves making local checks about flow equations. Moreover when validating counterexamples in a hybrid CEGAR scheme, one is only required to compute time-bounded successors (see 4.4). Computing time-bounded successors is often computationally easier than computing time-unbounded successors; for example, for automata with linear differential dynamics, time-bounded posts can be efficiently approximated, while no such algorithms exist for time-unbounded posts.

In this paper, we focus on *hybrid abstraction*-based CEGAR framework for rectangular hybrid systems. We abstract such automata using *initialized rectangular hybrid automata* [16]. The choice of initialized rectangular hybrid automata as the abstract space is motivated by the desire to have a rich space of abstractions, with a decidable model checking problem, and effective tools to analyze them. Our abstraction consists of the following operations: collapsing the control states and transitions, dropping the continuous variables and scaling the variables. Variable scaling changes the constants that appear in the abstract hybrid automaton which in turn can positively influence the efficiency of model checking the abstract automaton. Our refinement algorithm involves splitting control states/transitions, and/or adding variables that may have new dynamics (due to scaling).

Our main results in this paper are *complete/semi-complete* CEGAR algorithms for rectangular hybrid systems - (semi-completeness) if the hybrid system we are analyzing is a *rectangular* hybrid automaton and is faulty, then our CEGAR algorithm will terminate by demonstrating the error; (completeness) on the other hand, if the hybrid system is an *initialized rectangular* hybrid automaton then our CEGAR algorithm will *always* terminate. Such completeness results are usually difficult to obtain. In our case, one challenge is the fact that the collection of abstract counterexamples is not enumerable as the executions of an abstract hybrid system are uncountable. Thus, in order to argue that all abstract counterexamples are eventually considered, we need to change the notion of a counterexample to be a (infinite) set of executions, rather than a single execution. This change in the definition of counterexample also forces the validation algorithms to be different. The completeness proof then exploits topological properties, like compactness, to argue for termination.

Another highlight of our presentation is that we view our CEGAR framework as a composition of a few CEGAR loops. To carry this out, we identify some concepts and obtain a few results about the composition of CEGAR loops. Such a compositional approach simplifies the presentation of the refinement algorithm in our context, which is otherwise unwieldy, and helps identify more clearly the subtle concepts needed for the completeness proof to go through.

We have implemented our CEGAR based algorithm for rectangular hybrid automata in a tool that we call **H**ybrid **A**bstraction **R**efinement **E**ngine (HARE[1]). HARE makes calls to HyTech [17] to analyze abstract models and generate counterexamples; we considered PHAVer [13] and SpaceEx [12], but at the time of writing they do not produce counterexamples. We analyzed the tool on several benchmark examples which illustrate that its total running time is comparable with that of HyTech, and on some examples HARE is a couple of orders of magnitude faster. Moreover, in some cases HARE can prove safety with dramatically small abstractions. Fair running-time comparison of HARE with other Matlab-based tools, such as d/dt [3] and checkmate [5], is not possible because of the differences in the runtime environments. Experimental comparison of finite-state and hybrid abstractions was also not possible because to the best of our knowledge, the tools implementing finite-state abstractions are not publicly available. We believe that in terms of efficiency, the approaches of finite state abstractions and hybrid abstractions are incomparable.

Related Work. CEGAR with hybrid abstractions have been investigated in [9, 20] where the abstractions are constructed by ignoring certain variables. Counterexamples are used to identify new variables to be added to the abstraction. This approach has been carried out for timed automata [9] and linear hybrid automata [20]. In comparison to the above, our abstractions (may) change both the control graph and variable dynamics, and are not restricted to only forgetting continuous variables. In contrast, the scheme in [20] considers a more general class of hybrid automata, though the abstractions in that scheme are not progressively refined. Finally, in [10] hybrid systems with flows described by linear differential equations are approximated by rectangular hybrid automata. Even though, their scheme progressively refines abstractions, the refinements are not guided by counter-examples.

2 Preliminaries

Notation, Images and Inverse Images. Let $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{R}_{\geq 0}$ denote the set of natural numbers, integers, rationals, reals and non-negative reals, respectively. Given a function $f : A \to B$ and a subset $A' \subseteq A$, $f(A')$ is defined to be the set $\{f(x) \,|\, x \in A'\}$. Similarly, for $B' \subseteq B$, $f^{-1}(B')$ is the set $\{x \,|\, \exists y \in B', f(x) = y\}$. When B' is a singleton set $\{y\}$, we also use $f^{-1}(y)$ to denote $f^{-1}(\{y\})$.

Transition Systems. A *transition system* $\mathcal{T}$ is a tuple $(S, S^0, \Sigma, \longrightarrow)$, where S is a set of states, $S^0 \subseteq S$ is a set of initial states, Σ is a set of transition labels, and $\longrightarrow \subseteq S \times \Sigma \times S$ is a transition relation. We call $(s, a, s') \in \longrightarrow$ a transition of $\mathcal{T}$ and denote it as $s \xrightarrow{a} s'$. We denote the elements of a transition system using appropriate subscripts. For example, the set of states of a transition system $\mathcal{T}_i$ is denoted by S_i.

An *execution fragment* σ of a transition system $\mathcal{T}$ is a sequence of transitions $t_0 t_1 t_2 t_3 \cdots t_n$, such that $s'_i = s_{i+1}$ for $0 < i < n$, where t_i is given by $s_i \xrightarrow{a_i} s'_i$. We denote the above execution fragment by $\sigma = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \cdots s_{n-1} \xrightarrow{a_{n-1}} s_n$. We say that the length of σ is n, denoted by, $|\sigma|$. An *execution* of σ is an execution fragment starting from a state in S^0. We denote the set of all execution fragments of $\mathcal{T}$ by $\mathit{ExecF}(\mathcal{T})$ and the set of all executions by $\mathit{Exec}(\mathcal{T})$.

Given a set of states $S' \subseteq S$ and a symbol $a \in \Sigma$, $\mathit{Pre}_{\mathcal{T}}(S', a)$ is defined to be the set $\{s_1 \,|\, \exists s_2 \in S' : s_1 \xrightarrow{a} s_2\}$ and $\mathit{Post}_{\mathcal{T}}(S', a)$ as $\{s_2 \,|\, \exists s_1 \in S' : s_1 \xrightarrow{a} s_2\}$. Given a subset Σ' of Σ, $\mathit{Pre}_{\mathcal{T}}(S', \Sigma') = \bigcup_{a \in \Sigma'} \mathit{Pre}_{\mathcal{T}}(S', a)$ and $\mathit{Post}_{\mathcal{T}}(S', \Sigma') = \bigcup_{a \in \Sigma'} \mathit{Post}_{\mathcal{T}}(S', a)$.

Hybrid Automata A hybrid system is a system which exhibits mixed discrete-continuous behaviors. A popular model for representing hybrid systems is that of hybrid automata [16], which combine finite state automata modeling the discrete dynamics, and differential equations or inclusions modeling the continuous dynamics. An execution of such a system begins in a state of the automaton with some values to the variables representing the continuous dynamics. It then either evolves continuously, where in, only the values of the continuous variables change according to the continuous dynamics associated with the current discrete state of the automaton, or takes a discrete transition, where potentially both the discrete and continuous state of the automaton can change. During the latter, the state of the automaton changes from the source of an edge to its target, and the discrete transition is enabled only if the continuous state before the transition satisfies the enabling condition associated with the edge. The value of the continuous state after the transition either remains the same or is reset to some other value.

Definition 1. *A* hybrid automaton $\mathcal{H}$ *is a tuple* (Loc, $Edges$, $Source$, $Target$, q^{init}, n, $Cont_0$, inv, $flow$, $jump$)*, where*

- *Loc is a finite set of* (discrete) control states *or* locations.
- *$Edges$ is a finite set of* edges.
- *$Source$, $Target$: $Edges \to Loc$ are functions which associate a source and a target location to every edge, respectively.*
- $q^{init} \in Loc$ *is the* initial location*. The components above represent the discrete part of the automaton.*
- *$n \in \mathbb{N}$ is the dimension of $\mathcal{H}$, also denoted by $Dim(\mathcal{H})$, which represents the number of continuous variables in the system. The set of continuous states is given by $Cont = \mathbb{R}^n$.*
- *$Cont_0 \subseteq Cont$ is the* initial set of continuous states.
- *inv: $Loc \to 2^{Cont}$ associates with every location an* invariant *set. The continuous state of the system belongs to the invariant of a location as long as the control is in that location.*
- *$flow$: $Loc \to 2^{Traj(Cont)}$ associates with every location a set of trajectories, where $Traj(Cont)$ is the set of continuous functions from $\mathbb{R}_{\geq 0} \to Cont$.*
- *$jump$: $Edges \to 2^{Cont \times Cont}$, associates with every edge a set of pair of states describing the value of the continuous state before and after the edge is taken.*

Next we present the semantics of a hybrid automaton, as a transition system it represents. The semantics of a hybrid automaton $\mathcal{H}$ is defined in terms of the transition system $[\![\mathcal{H}]\!] = (Q, Q^0, \Sigma, \longrightarrow)$ over $\Sigma = \mathbb{R}_{\geq 0} \cup Edges$, where $Q = Loc \times Cont$, $Q^0 = q^{init} \times Cont_0$, and the transition relation $\longrightarrow$ is given by:

- *Continuous transitions* - For $t \in \mathbb{R}_{\geq 0}$, $(q_1, x_1) \xrightarrow{t} (q_2, x_2)$ iff $q_1 = q_2 = q$ and there exists a function $f \in flow(q)$ such that $x_1 = f(0)$, $x_2 = f(t)$ and for all $t' \in [0, t]$, $f(t') \in inv(q)$.
- *Discrete transitions* - For $e \in Edges$, $(q_1, x_1) \xrightarrow{e} (q_2, x_2)$ iff $q_1 = Source(e)$, $q_2 = Target(e)$, $x_1 \in inv(q_1)$, $x_2 \in inv(q_2)$ and $(x_1, x_2) \in jump(e)$.

We focus on the problem of *control state reachability*, namely, given a hybrid automaton $\mathcal{H}$ and a location $q \neq q^{init}$, is q reachable in $\mathcal{H}$, or equivalently does there exist an execution of $\mathcal{H}$ from a state in $\{q^{init}\} \times \mathit{Cont}_0$ to $\{q\} \times \mathit{Cont}$? Typically, q is a "bad" or "unsafe" location that we do not want to reach, and we are interested in determining the safety of the system, namely, no execution reaches the unsafe location.

3 CEGAR framework

A counter-example guided abstraction refinement algorithm consists of the four steps, namely, *abstraction*, *model-checking*, *validation* and *refinement*. We focus on safety verification here. CEGAR loop begins with the construction of an abstraction (an overapproximation) of the original system (also called the concrete system). The abstract system is then model-checked to determine if there exists an execution from the initial location to an unsafe location. Such a path if one exists is called an *abstract counter-example*. If the abstract system has no counter-examples, then it can be deduced from the properties of abstraction that even the concrete system does not have any counter-examples, and hence is safe. However, if an abstract counter-example is returned in the model-checking phase, then one cannot in general make any conclusions about the safety of the concrete system, and the counter-exapmle is validated to determine if there exist a counter-example in the concrete system corresponding to it. If a concrete counter-example is found, then the concrete system is unsafe, and the concrete counter-example exhibits a bug in the system. Otherwise, the analysis in validating the abstract counter-example is used to construct a new abstract system which is a refinement of the current abstract system. The CEGAR algorithm continues with the model-checking of the new abstract system. In general, the CEGAR loop might not terminate.

The purpose of this section is to fix notation and highlight some differences with the standard CEGAR loop. We present several CEGAR algorithms in the next section, which provide various guarantees about the termination, namely, completeness and semi-completeness. Completeness refers to the fact that there are only finitely many iterations of the CEGAR loop in any execution; and semi-completeness refers to the fact that the CEGAR loop always terminates on a faulty machine exhibiting a counter-example. Semi-completeness is easy to guarantee when the set of concrete executions of a system are (efficiently) enumerable, which is lacking for the class of systems we consider. Hence, we need to change the notion of a counter-example to encapsulate a possibly (uncountable) number of counter-examples, and perform the validation simultaneously on this infinite set. This requires us to perform validation in a slightly different manner, namely, we first need to find the actual set of abstract executions corresponding to the counter-example, and then perform the standard validation on the computed set. Further, in order to guarantee termination, we need to at the least guarantee that we make progress in each iteration of the CEGAR loop.

In the rest of the section, we setup notation and explain our notion of counter-example, the modified validation algorithm, and distill some local condition which ensure progress of the CEGAR loop.

We fix some notation for the rest of this section. Our concrete systems and abstract systems are both hybrid automata. Let $\mathcal{H}_C$ be a concrete hybrid automaton and let $\mathcal{H}_A$ be an abstract hybrid automaton. Let $\mathcal{T}_C = [\![\mathcal{H}_C]\!] = (S_C, S_C^0, \Sigma_C, \longrightarrow_C)$ and $\mathcal{T}_A = [\![\mathcal{H}_A]\!] = (S_A, S_A^0, \Sigma_A, \longrightarrow_A)$ be the transition systems associated with $\mathcal{H}_C$ and $\mathcal{H}_A$, respectively. Let us fix an unsafe location $q_C^{unsafe} \neq q_C^{init}$ in $\mathcal{H}_C$, and we want to check if q_C^{unsafe} is reachable in $\mathcal{H}_C$.

3.1 Abstraction

In the next section, we present several methods for constructing abstractions. Here, we define formally the relation that holds between a system and its abstraction.

Given transition systems $\mathcal{T}_1 = (S_1, S_1^0, \Sigma_1, \longrightarrow_1)$ and $\mathcal{T}_2 = (S_2, S_2^0, \Sigma_2, \longrightarrow_2)$, an *abstraction* or *simulation* function from $\mathcal{T}_1$ to $\mathcal{T}_2$ is a pair of functions $\alpha = (\alpha_S, \alpha_\Sigma)$, where $\alpha_S : S_1 \to S_2$ and $\alpha_\Sigma : \Sigma_1 \to \Sigma_2$ such that

- $\alpha_S(S_1^0) \subseteq S_2^0$, and
- for every $s_1, s_1' \in S_1$ and $a_1 \in \Sigma_1$, $s_1 \xrightarrow{a_1}_1 s_1'$ implies $\alpha_S(s_1) \xrightarrow{\alpha_\Sigma(a_1)}_2 \alpha_S(s_1')$.

We say that $\mathcal{T}_2$ is an abstraction of $\mathcal{T}_1$, denoted by $\mathcal{T}_1 \preceq \mathcal{T}_2$. We denote the fact that α is an abstraction function from $\mathcal{T}_1$ to $\mathcal{T}_2$ by $\mathcal{T}_1 \preceq_\alpha \mathcal{T}_2$.
Notation. Given an abstraction function $\alpha = (\alpha_S, \alpha_\Sigma)$, we will drop the subscripts S and Σ when it is clear from the context. For example, for a state s, we will use $\alpha(s)$ to mean $\alpha_S(s)$. Note that if α is an abstraction function from $\mathcal{T}_1$ to $\mathcal{T}_2$ and $\sigma = s_1 \xrightarrow{a_1}_1 \cdots s_n$ is an execution fragment of $\mathcal{T}_1$, $\sigma' = \alpha(s_1) \xrightarrow{\alpha(a_1)}_2 \cdots \alpha(s_n)$ is an execution fragment of $\mathcal{T}_2$. Let us denote $\alpha(s_1) \xrightarrow{\alpha(a_1)}_2 \cdots \alpha(s_n)$ by $\alpha(\sigma)$.

Let us fix an abstraction function α from the concrete system $\mathcal{T}_C$ to the abstract system $\mathcal{T}_A$. Since, we are interested in control state reachability, we need certain consistency conditions on α to ensure that property is preserved. We assume that there exists a location $q_A^{unsafe} \neq q_A^{init}$ in $\mathcal{H}_A$ satisfying $\alpha(\{q_C^{unsafe}\} \times \mathit{Cont}_C) \subseteq \{q_A^{unsafe}\} \times \mathit{Cont}_A$, that is, α maps the elements of the unsafe set of concrete states to states corresponding to an unique location of $\mathcal{H}_A$. Let $\mathit{Unsafe}_C = \{s_C\} \times \mathit{Cont}_C$ and $\mathit{Unsafe}_A = \{s_A\} \times \mathit{Cont}_A$. Note that if q_A^{unsafe} is not reachable in the abstract hybrid automaton $\mathcal{H}_A$, then q_C^{unsafe} is not reachable in the concrete hybrid automaton $\mathcal{H}_C$.

3.2 Counter-examples

If q_A^{unsafe} is reachable in $\mathcal{H}_A$, then the model-checker returns a counter-example. In order to guarantee semi-completeness, we need a set of counter-examples which are enumerable, and spawn the set of all executions of the system. Hence, we define a counter-example to be a path in the control flow graph of the abstract hybrid automaton, which is feasible, that is, has an execution corresponding to it. Note that such a counter-example exhibits potentially an infinite set of unsafe abstract executions.

Given an element $\pi = q_0 e_0 q_1 \cdots q_n \in (\mathit{LocEdges})^* \mathit{Loc}$, define $\mathit{PathToExecF}(\pi)$ to be the set of all execution $\sigma = (q_0, x_0) \xrightarrow{t_0} (q_0, y_0) \xrightarrow{e_0} (q_1, x_1) \xrightarrow{t_1} (q_1, y_1) \cdots (q_{n-1}, y_{n-1}) \xrightarrow{e_{n-1}} (q_n, x_n) \xrightarrow{t_n} (q_n, y_n)$. Formally, a *counter-example* of a hybrid automaton $\mathcal{H}$ given an unsafe location q^{unsafe} is an alternating sequence of locations and edges, that is, an element π of $q^{init} \mathit{Edges}(\mathit{LocEdges})^*\ q^{unsafe}$ such that $\mathit{PathToExecF}(\pi)$ is not empty. The length of a counter-example is the number of elements in the sequence. We will call a counter-example of $\mathcal{H}_C$ with unsafe location q_C^{unsafe}, a *concrete counter-example*, and a counter-example of $\mathcal{H}_A$ with unsafe location q_C^{unsafe}, an *abstract counter-example.*

3.3 Validation

We think of an abstract counter-example as representing a possibly infinite set of abstract unsafe executions. Hence, validation needs to check if there is a concrete unsafe execution corresponding to any of the abstract unsafe executions represented by the abstraction counter-example. This requires us to perform validation in a slightly different manner. Validation takes place in two phases: In the first phase, a forward analysis is done to compute a representation of the abstract executions corresponding to the counter-example. In particular, the set of abstract states reached by traversing the abstract counter-example is computed. In the next phase, a backward reachability computation is performed in the concrete system, along the potentially infinite set of abstract executions computed in the previous step, and represented by a sequence of sets of abstract states. The precise algorithm is given in Figure 1. There exists a concrete unsafe execution corresponding to the abstract counter-example π_A iff $\mathit{Reach}_{\pi_A, \alpha}(0) \cap S_C^0 \neq \emptyset$.

Remark 1. Observe that just running a standard backward reachability algorithm on the counter-example does not suffice, since the reach sets thus obtained might contain concrete state which do not correspond to an actual abstract execution. This is because, running a backward reachability would correspond to "validating" all abstract execution fragments corresponding to any "subpath" of the counter-example, simultaneously.

Given an execution fragment σ' of $\mathcal{T}_A$ and an abstraction function α from $\mathcal{T}_C$ to $\mathcal{T}_A$, we denote by $\mathit{Conc}_\alpha(\sigma')$, the set of execution fragments in $\mathcal{T}_C$ corresponding to $\mathcal{T}_A$, namely, the set $\{\sigma \in \mathit{ExecF}(\mathcal{T}_C) \mid \alpha(\sigma) = \sigma'\}$. Validation is the process of checking if $\mathit{Conc}_\alpha(\mathit{PathToExecF}(\pi_A))$ contains an execution of $\mathcal{T}_C$ reaching s_C.

Proposition 1. *$\mathit{Conc}_\alpha(\mathit{PathToExecF}(\pi_A))$ contains an execution of $\mathcal{T}_C$ reaching s_C iff $\mathit{Reach}_{\pi_A, \alpha}(0) \cap S_C^0 \neq \emptyset$.*

If $\mathit{Reach}_{\pi_A, \alpha}(0) \cap S_C^0 \neq \emptyset$, then we call π_A a *spurious* counter-example.

Input: π_A, an abstract counter-example in $\mathcal{H}_A$ of length l.

Phase 1: Forward reachability in the abstract automaton.

Compute $\mathit{FReach}_{\pi_A}(i)$, for $0 \leq i \leq 2l+1$.

- $\mathit{FReach}_{\pi_A}(0) = S_A^0$.
- $\mathit{FReach}_{\pi_A}(i+1) = \mathit{Post}_{\mathcal{T}_A}(\mathit{FReach}_{\pi_A}(i), a)$, where
  - (Time Elapse) $a = \mathbb{R}_{\geq 0}$ if i is even, and
  - (Edge) $e'_{(i-1)/2}$ if i is odd, for $0 \leq i < 2l+1$.

Phase 2: Backward reachability in the concrete automaton.

Compute S_k for $0 \leq i \leq 2l+1$:

$$S_k = \alpha^{-1}(\mathit{FReach}_{\pi_A}(k)), 0 \leq i < 2l+1$$

$$S_{2l+1} = \alpha^{-1}(\mathit{FReach}_{\pi_A}(k)) \cap (q_C^{\mathit{unsafe}} \times \mathit{Cont}_C)$$

Compute $\mathit{Reach}_{\pi_A,\alpha}(k)$, $0 \leq k \leq 2l+1$.

- $\mathit{Reach}_{\pi_A,\alpha}(2l+1) = S_{2l+1}$.
- $\mathit{Reach}_{\sigma_A,\alpha}(k) = S_k \cap \mathit{Pre}_{\mathcal{T}_C}(\mathit{Reach}_{\pi_A,\alpha}(k+1), a)$, where
  - (Time elapse) $a = \mathbb{R}_{\geq 0}$ if k is even, and
  - (Edge) $\alpha^{-1}(e'_{(k-1)/2})$, if k is odd.

Fig. 1. Validation Algorithm

3.4 Refinement

We formalize the conditions which ensure that the refinement is making progress by "eliminating" spurious abstract counter-examples. The refinement algorithms we present in the next section ensure the progress conditions presented here.

Definition 2. *Given two transition systems $\mathcal{T}_1$ and $\mathcal{T}_2$ such that $\mathcal{T}_1 \preceq \mathcal{T}_2$, a transition system $\mathcal{T}_3$ is said to be a* refinement *of $\mathcal{T}_2$ with respect to $\mathcal{T}_1$, if $\mathcal{T}_1 \preceq \mathcal{T}_3 \preceq \mathcal{T}_2$.*

Notation. We will say $\mathcal{H}_3$ is a refinement of $\mathcal{H}_2$ with respect to $\mathcal{H}_1$ to mean that $[\![\mathcal{H}_3]\!]$ is a refinement of $[\![\mathcal{H}_2]\!]$ with respect to $[\![\mathcal{H}_1]\!]$, and denote it by $\mathcal{H}_1 \preceq \mathcal{H}_3 \preceq \mathcal{H}_2$.

Our goal is to find a refinement $\mathcal{H}_R$ such that $\mathcal{H}_C \preceq \mathcal{H}_R \preceq \mathcal{H}_A$. Note that $\mathcal{H}_R = \mathcal{H}_A$ is such a system, however, we want to make progress by eliminating the spurious counter-example. Hence, we define good refinements to be those in which some "potential" execution fragment of the counter-example in the current abstraction is not a "potential" execution fragment of any counter-example in the refinement.

To formalize progress, we need some definitions. Given a transition system $\mathcal{T}$, a *potential execution fragment* of $\mathcal{T}$ is a sequence $\rho = s_0 a_0 s_1 a_1 \cdots s_{l-1} a_{l-1} s_l$ alternating between elements of S and Σ. Further, given an abstraction function γ from $\mathcal{T}_1$ to $\mathcal{T}_2$, and an execution fragment $\sigma' = s'_0 \xrightarrow{a'_0}_2 s'_1 \xrightarrow{a'_1}_2 \cdots s'_{l-1} \xrightarrow{a'_{l-1}}_2 s'_l$ of $\mathcal{T}_2$, $\mathit{Potential}_\gamma(\sigma')$ is the set of all potential execution fragments $\rho = s_0 a_0 s_1 a_1 \cdots s_{l-1} a_{l-1} s_l$ of $\mathcal{T}_1$ such that $\gamma(s_i) = s'_i$ and $\gamma(a_i) = a'_i$. Note that a potential execution fragment might not correspond to an actual execution fragment.

Definition 3. *Let $\mathcal{H}_C \preceq_\alpha \mathcal{H}_A$ and $\mathcal{H}_C \preceq_\beta \mathcal{H}_R$. Given a spurious counter-example π_A of $\mathcal{H}_A$, a refinement $\mathcal{H}_R$ of $\mathcal{H}_A$ with respect to $\mathcal{H}_C$ is said to be* good *with respect to π_A if there exists a $\rho \in \mathit{Potential}_\alpha(\mathit{PathToExecF}(\pi_A))$ such that $\rho \notin \mathit{Potential}_\beta(\mathit{PathToExecF}(\pi_R))$ for any counter-example π_R of $\mathcal{H}_R$.*

In validating a spurious counter-example π_A of $\mathcal{H}_A$, we see that $Reach_{\pi_A,\alpha}(0)$ $\cap S_C^0$ is empty. Note that if $Reach_{\pi_A,\alpha}(k) = \emptyset$ for some k, then $Reach_{\pi_A,\alpha}(i)$ is empty for all $i \leq k$. Let $\hat{k}$ be the largest integer such that $Reach_{\pi_A,\alpha}(\hat{k})$ is empty, if $Reach\pi_A, \alpha(k)$ is empty for some k, otherwise, let $\hat{k} = 0$ (since $Reach_{\pi_A,\alpha}(0) \cap S_C^0$ is definitely empty). We call $\hat{k}$ the *infeasibility index* of π_A with respect to α. The next proposition states a local sufficient condition to ensure that a refinement is good.

Proposition 2. *Let α be the an abstraction function which is surjective. Let $\pi_A = s'_0 e'_0 s'_1 \cdots s'_{l-1} e'_{l-1} s'_l$ be a spurious counter-example of $\mathcal{H}_A$ with an infeasibility index $\hat{k}$ with respect to α. Suppose that $\mathcal{H}_R \preceq_\beta \mathcal{H}_A$ is a refinement of $\mathcal{H}_A$ with respect to $\mathcal{H}_C$ satisfying:*

$$Post_{[\![\mathcal{H}_R]\!]}(\beta(S_{\hat{k}}), \beta(a_{\hat{k}})) \cap \beta(Reach_{\pi_A,\alpha}(\hat{k}+1)) = \emptyset, \tag{1}$$

where $S_{\hat{k}}$ is as defined in Figure 1 and $a_{\hat{k}} = \mathbb{R}_{\geq 0}$ if $\hat{k}$ is even and is $\{\alpha^{-1}(e'_{(\hat{k}-1)/2})\}$ otherwise. Then $\mathcal{H}_R$ is a refinement of $\mathcal{H}_A$ with respect to $\mathcal{H}_C$ which is good with respect to π_A.

Remark 2. Validation can be done by checking if $Conc_\alpha(PathToExecF(\pi_A))$ contains an execution of $\mathcal{T}_C$ reaching s_C, which can be verified by performing a backward reachability in the concrete system with respect to the abstract counter-example and checking if the reach set becomes empty. However, in order to guarantee progress in the refinement step, in particular, to be able to perform refinement which satisfies Equation 1, we need to identify and eliminate a potential concrete execution of some abstract unsafe execution.

3.5 Completeness, Semi-completeness and Composition

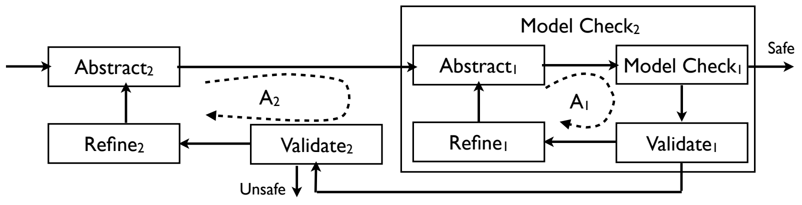

Fig. 2. $A_2[A_1]$: Composition of CEGAR Algorithm A1 with A2

A CEGAR algorithm for a class of systems takes as input a finite representation of a system from the class and (a) either outputs "YES" or (b) outputs "NO" and returns a counter-example of the system, or (c) does not terminate. If it outputs "YES", then it is guaranteed that the system is safe, that is, the unsafe location is not reachable, and if it outputs “NO”, then it is unsafe. We will assume that the different phases of any loop of the CEGAR algorithm terminate, but CEGAR algorithm itself may or may not terminate. Next, we define the notions of completeness and semi-completeness of a CEGAR algorithm.

Definition 4. *A CEGAR algorithm is said to be* complete *for a subclass C of hybrid automata if it terminates for all inputs from the class C. It is said to be* semi-complete *if it terminates at the least for inputs from C which are unsafe, that is, have an execution to an unsafe location.*

We say that a CEGAR algorithm is *fair*, if it returns a smallest length counter-example whenever it terminates on an unsafe system, under the assumption that the model-checker always returns a smallest length counter-example.

Note that a CEGAR algorithm is essentially a model-checking algorithm. Hence, we can compose CEGAR algorithms by using a CEGAR algorithm A_1 as a model-checker for a CEGAR algorithm A_2 as shown in Figure 2. We denote the composed algorithm by $A_2[A_1]$.

Proposition 3. *Let A_i be a CEGAR algorithm with input and abstraction spaces C_i and D_i respectively, for $i = 1, 2$, such that $D_2 \subseteq C_1$. Then:*

- *If A_1 and A_2 are complete and fair CEGAR algorithms, then $A_2[A_1]$ is complete and fair.*
- *If A_2 is semi-complete and fair, and if A_1 is complete and fair, then $A_2[A_1]$ is also semi-complete and fair.*

4 CEGAR for Rectangular Hybrid Automata

In this section, we focus on a subclass of hybrid automata called rectangular hybrid automata. We present three CEGAR algorithms for this class. In order to keep the presentation simple, we choose to illustrate the ideas in the algorithms using examples. The formal description and details can be found in [22].

We begin with a brief overview of the class of rectangular hybrid systems. A rectangular (hybrid) automaton is a hybrid automaton in which the invariants, guards, jumps and flow are specified using rectangular constraints. A rectangular constraint is of the form $x \in I$, where x is a variable and I is an interval whose finite end-points are integers. Figure 3 shows a rectangular hybrid automaton.

It has four locations, namely, l_1, l_2, l_3 and l_4, as shown by the circles and four edges e_1, e_2, e_3 and e_4, as shown by the arrows between the circles. The invariant at location l_3 is given by $x \in [-1, 1]$. To keep the diagram simple, we have omitted the constraints $x \in [-10, 10]$ and $y \in [-10, 10]$ from the invariants of every location. The flow is specified by rectangular differential inclusions of the form $\dot{x} \in I$. The flow associated with it are all functions whose projection to the x-component is such that the derivative with respect to time belongs to the interval I at all times.

The jump relation is specified using two kinds of constraints, namely, guards and resets. A guard specifies the enabling condition on the edge, and the reset

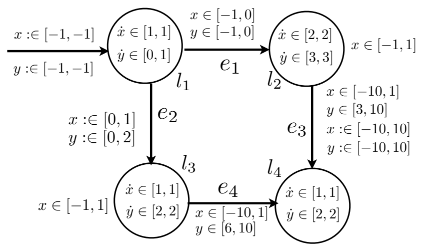

Fig. 3. $\mathcal{H}_1$: An example of a rectangular hybrid automaton

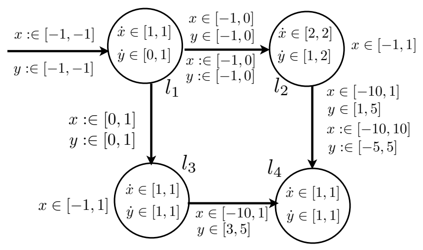

Fig. 4. $\mathcal{H}_5$: Flow abstraction of $\mathcal{H}_2$ scaling down y by a factor of 2

specifies the value of the continuous state after the edge is taken. An edge is labelled by a constraint of the form $x \in I$, which specifies that the edge can be taken when the value of x belongs to the interval I. If an edge is labelled by a constraint of the form $x :\in I$, it means that the value of x after the edge is taken is reset non-deterministically to some value in the interval I. When a constraint of the form $x :\in I$ is absent, it means that the value of a variable remains the same after taking an edge. We call a reset of the first form as *strong reset* and a reset of the second form as an *identity reset.* Note that the jump relation associated with a constraint $x \in I_1$, $x :\in I_2$ is $I_1 \times I_2$, where as that associated with just $x \in I$ is $\{(x,x) \,|\, x \in I\}$.

The control state reachability problem is undecidable for the class of rectangular hybrid automata [15]. Hence, to ensure that the model-checking phase of a CEGAR loop terminates, we consider as the abstraction space, a subclass of rectangular hybrid automata called *initialized rectangular hybrid automata*, which have the property that whenever the differential inclusions associated with a variable is different for the source and target of an edge, then the edge is necessarily labelled by a reset constraint for that variable, that is, the value of the variable is non-deterministically reset to some value in an interval (as opposed to carrying over the value from previous location). The control state reachability problem has been shown to be decidable for the class of initialized rectangular hybrid automata [15]. For example, the variable x is non-initialized along the edge e_1, since the constraints associated with $\dot{x}$ in locations l_1 and l_2 are different, but the edge e_1 does not have a reset constraint for x.

Before presenting the CEGAR algorithm, let us discuss briefly the computability of the validation step. For a transition system $\mathcal{T}$ arising from a rectangular hybrid automaton $\mathcal{H}$, one can compute $Pre_{\mathcal{T}}(S,a)$ and $Post_{\mathcal{T}}(S,a)$ where S is any linear set and a is either the set of non-negative reals $\mathbb{R}_{\geq 0}$ or a subset of edges *Edges* of the hybrid automaton $\mathcal{H}$. Further, the resulting sets are linear too. This implies that $FReach_{\pi}(i)$ can be computed for any counter-example π of $\mathcal{H}$ and any position i in π. Further $FReach_{\pi}(i)$ is a linear set. We distill the conditions for $Reach_{\pi_A,\alpha}(i)$ to be computable in the following proposition.

Proposition 4. *Let α be an abstraction function from $\mathcal{H}_C$ to $\mathcal{H}_A$. Suppose that for any linear subset S of the state space, $\alpha^{-1}(S)$ is a linear set and can be com-*

puted. Then $Reach_{\pi_A,\alpha}(k)$ is a linear set and is computable for each k. Further, $Reach_{\pi_A,\alpha}(k)$ is a compact set.

Proof. Compactness follows from the fact that the invariants associated with the locations are compact (closed and bounded).

In this section, we present three CEGAR algorithms. For each of these we present the "hybrid abstraction" which can be thought of as a symbolic representation of the abstraction function, and a specific method to construct an abstract system using the hybrid abstraction. Further, the resulting abstraction function will be such that it satisfies the hypothesis of Proposition 4. Hence, we can effectively carry out the validation phase. We will present our refinement algorithm which ensures progress as given by Equation 1.

The first CEGAR algorithm abstracts a rectangular hybrid automaton to an initialized rectangular hybrid automaton by abstracting the identity resets of the edges which violate the initialization condition by strong resets. This algorithm is semi-complete for the class of rectangular automata. Next, we present two CEGAR algorithms for the class of initialized rectangular automata, which are complete for this class. One abstracts a system, by merging together different locations/edges and the other abstracts by dropping/scaling variables. All these algorithms are fair. Hence, they can be composed as sketched in Proposition 3 to obtain more sophisticated CEGAR algorithms which are complete and semi-complete, respectively.

4.1 Strong Reset Abstraction based CEGAR

In this section, we present a semi-complete CEGAR algorithm for the class of rectangular hybrid automata.

Abstraction The broad idea is to abstract a rectangular hybrid automaton to an initialized rectangular hybrid automaton by abstracting an identity reset which violates the initialization condition by a strong reset. A naive approach is to replace a constraint $x \in I$ associated with a pair non-initialized edge e and variable x, by the constraint $x \in I, x :\in I$. It transforms an identity reset to a strong reset, and is such that the jump relation associated with the new constraint $\{(v_1, v_2) \mid v_1, v_2 \in I\}$ is a superset of the jump relation before the transformation $\{(v, v) \mid v \in I\}$. Observe that one can interpret the identity reset $\{(v, v) \mid v \in I\}$ as an infinite set of strong resets $(\{(v, v)\})_{v \in I}$. We choose to abstract an identity reset by a finite set of strong resets for the verification to be computationally feasible. More generally, we abstract a constraint $x \in I$ by a set of constraints $x \in J_i, x :\in J_i, i \in K$, where $J_i, i \in K$ is a finite partition of I.

Consider the rectangular automaton $\mathcal{H}_1$ in Figure 3. The only non-initialized edge in the automaton is e_1. The strong rest abstraction of $\mathcal{H}_1$ is exactly the same as $\mathcal{H}_1$, except that the constraint associated with edge e_1 is $x \in [-1, 0]$, $x :\in [-1, 0]$, $y \in [-1, 0], y :\in [-1, 0]$. Let us call this automaton $\mathcal{H}_2$.

Refinement The refinement step constructs a new abstraction by replacing the jump relation associated with a non-initialized edge by a smaller set. The validation step identifies the two sets as given by Equation 1 that need to be separated. One can show that the infeasibility index always corresponds to a non-initialized edge of the concrete automaton. The refinement corresponds to refining the partition used in transforming the non-initialized edge to an initialized edge.

Let us consider the counter-example $l_1e_1l_2e_3l_4$ of the strong reset abstraction of $\mathcal{H}_1$, namely, $\mathcal{H}_2$, where l_4 is the unsafe state. Validation step returns that the sets $A = \{(-1, 0)\}$ and $B = \{(v_1, v_2) \mid v_1 \in [-1, 0], v_2 \in [-1, 0], v_1 \geq v_2\}$. The minimum distance (Euclidean) between A and B is $\sqrt{2}/2$. Hence, any square whose sides are $1/2$ units will not overlap with both A and B. Therefore, we partition the guard $x \in [-1, 0]$, $y \in [-1, 0]$ into square chunks of width $1/2$. So the edge e_1 in the refinement is labelled by the multi-rectangular constraints corresponding to the partition $(x \in [-1, -1/2], y \in [-1, -1/2])$, $(x \in [-1/2, 0], y \in [-1, -1/2])$, $(x \in [-1, -1/2], y \in [-1/2, 0])$, and $(x \in [-1/2, 0], y \in [-1/2, 0])$. Note that the refinement step could force us to use rational end-points, even if we begin with integer end-points for all the intervals which appear in the constraints.

Remark 3. Compactness of the reach set is a crucial property we exploit here, since this guarantees that there exists a minimum distance between the two sets that need to be separated according to Equation 1.

It is easy to see that if the model-checker always return a smallest counter-example in the abstract system when one exists, then the CEGAR algorithm will find a smallest length counter-example if it terminates. However, the CEGAR loop might not terminate in general (a consequence of the undecidability of control state reachability for rectangular automata). Hence, we obtain the following partial guarantee about the termination of the CEGAR loop.

Theorem 1. *The strong reset abstraction based CEGAR algorithm is semi-complete for the class of rectangular hybrid automata, and is fair.*

Remark 4. Note that the semi-completeness depends crucially on our choice of the notion of a counter-example. For example, the above theorem would not hold had we chose an abstract execution fragment as the notion of counter-example.

4.2 Control Abstraction based CEGAR

We present a CEGAR algorithm for the class of initialized rectangular automata. Hence, this CEGAR loop can be used as a model-checker for the strong reset abstraction based CEGAR algorithm of the previous section.

Abstraction As the name suggests, we abstract the underlying control flow graph of the initialized rectangular automaton. More precisely, we define a consistent partition of locations/edges and merge the locations/edges in the each of the partitions. We define the constraints for the invariants, differential inclusions, guards and resets to be a the smallest rectangular constraints which contain the corresponding constraints for the elements in each partition.

Figure 5 shows an abstraction of the initialized rectangular automaton $\mathcal{H}_2$ in which location l_2 and l_3 are merged and edges e_3 and e_4 are merged.

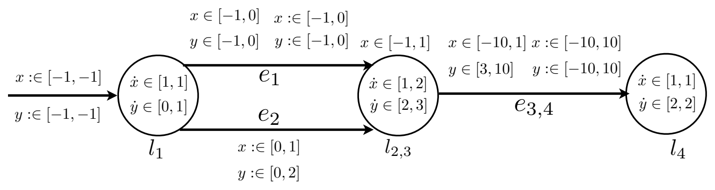

Fig. 5. $\mathcal{H}_3$: Control Abstraction of the Hybrid Automaton $\mathcal{H}_2$

The constraint on $\dot{x}$ in $l_{2,3}$ is a constraint which corresponds to the smallest rectangular set containing $[2,2]$ and $[1,1]$, the constraints on $\dot{x}$ in l_2 and l_3, respectively. Similarly, the guard for the variable y on the edge $e_{3,4}$ is the union of the sets $[3,10]$ and $[6,10]$.

We call the smallest rectangular set containing a given set, the *rectangular hull* of the set. We use the following rules in constructing the constraints for an abstract edge E. If all the concrete edges corresponding to E have identity resets, then E is an identity reset and the constraint for the guard is the rectangular hull of the constraints on the concrete edges. If at least one the edges does not have an identity reset, then the edges with identity reset are transformed into the naive strong reset explained in the previous section. Then the guard and reset are obtained by taking the rectangular hull of the the guards and resets, respectively, of the concrete strong reset transformed edges. The only exception to the above rule is when E is a singleton edge, in which case, the last step of taking the rectangular hull is skipped. This is to ensure that after finite number of refinement (essentially, when the abstract locations and edges are singleton sets), the abstract automaton is identical to the concrete automaton.

Refinement The refinement algorithm essentially constructs a new control abstraction by splitting the equivalence class of locations and edges near the point of infeasibility. We present a specific method to refine along the above lines. Let us define L_2 to be the set of locations appearing in $\mathit{Reach}_{\pi_A,\alpha}(\hat{k}+1)$. If the infeasibility edge corresponds to an abstract edge e, then we also define E and L_1. E is the set of concrete edges in $\alpha^{-1}(e)$, whose target is a location in L_2. And L_1 is the locations appearing in $\mathit{Reach}_{\pi_A,\alpha}(\hat{k})$ which is a source of some edge in E. We then split the equivalence classes such that the elements in L_1, L_2 and E appear in singleton sets. The intuition behind the above construction is that the only elements which effect satisfaction of Equation 1 are those in the above sets. The above splitting will force the refined system to be equivalent to the concrete system locally, and hence Equation 1 is trivially satisfied.

Let us consider the counter-example $\pi_A = l_1 e_2 l_{2,3} e_{3,4} l_4$. The infeasibility index $\hat{k}$ corresponds to the time transition from $l_{2,3}$ to $l_{2,3}$. In more detail, $FReach_{\pi_A}(\hat{k})$ and $FReach_{\pi_A}(\hat{k}+1)$ are the regions C given by the constraints $x \in [0,1], y \in [0,1]$ and D given by the constraints $x \in [0,1], y \in [0,4], x \geq y-3$, respectively. However, $Reach_{\pi_A,\alpha}(\hat{k}+1)$ is given by $l_2 \times \{(1,4)\}$. And taking the predecessor of $l_2 \times \{(1,4)\}$ with respect to $\dot{x} \in [2,2]$ and $\dot{y} \in [3,3]$, has a non-empty intersection with the region C. Hence, we split the equivalence class $\{l_2, l_3\}$ such that l_2 is in a singleton equivalence class. We will also need to split the edges e_3 and e_4 to obtain a consistent partition. Therefore, the refinement step results in the concrete automaton.

The control abstraction based CEGAR will always terminate, since starting with any partition of locations and edges, it is possible to refine the partitions only finitely many times due to the fact that the set of locations and edges is finite. Hence, we obtain a complete CEGAR algorithm for the class of initialized rectangular automata. Further, it is fair.

Theorem 2. *The control abstraction based CEGAR algorithm is complete for the class of initialized rectangular hybrid automata, and is fair.*

4.3 Flow Abstraction based CEGAR

We present another complete algorithm for the class of initialized rectangular automata which abstracts the continuous dynamics by dropping certain variables or applying a limited form of linear transformation, namely, scaling on the variables.

Abstraction A flow abstraction preserves the underlying control flow graph. A specification of a flow abstraction provides a subset of the variables of the concrete automaton and a scaling factor, a natural number, for each of the variables in the subset. The abstraction is constructed by first dropping the variables not in the specified subset, that is, only the constraints corresponding to the variables in the subset are retained in the invariants, guards, resets and differential inclusions. Next, the constraints are scaled according to the scaling factors provided. A scaling factor of $k \in \mathbb{N}$ for a variable x involves, replacing every constant c appearing in the constraints involving x by c/k. Note that this step may result in an automaton with rational end-points. Hence, we take the rectangular hull of the sets obtained. The only exception is when the scaling factor is 1, in which case we do not take the rectangular hulls. This is to guarantee that when all the variables are included with a scaling factor of 1, we obtain the concrete automaton. Scaling factor helps in reducing the granularity for the purpose of analyzing the system.

Let us consider the automaton $\mathcal{H}_1$, and a flow abstraction which keeps the variable x with scaling factor of 1. The resulting abstract system is obtained by removing all the constraints involving y from $\mathcal{H}_1$. Let us consider another abstraction in which we keep both the variables, x with a scaling factor of 1 and y with a scaling factor of 2. The resulting abstract automaton is shown in Figure 4. The flow of y in $\mathcal{H}_2$ is given by $\dot{y} \in [3, 3]$, which when scaled down by 2 gives the $\dot{y} \in [1.5, 1.5]$, and then taking the rectangular hull of the set gives $\dot{y} \in [1, 2]$. Similarly, the guard on e_3 is transformed from $y \in [3, 10]$ to $y \in [1, 5]$.

Refinement The refinement algorithm consists of two steps. Broadly, in the first step we choose a subset of variables, and in the second step assign appropriate scaling factors to the chosen variables. We iterate over all subsets of variable in the increasing order of size and check if Equation 1 holds, when the scaling factor for all the variables is taken to be 1. Then we assign a scaling factor for a variable in the chosen set, to be the g.c.d of the previous scaling factor and all the constants appearing in the constraints involving the variable locally, that is, the constants appearing the invariants and flows of the location, if the infeasibility index corresponds to a time transition, otherwise, one considers the constants appearing in the guards and reset of the edge corresponding to the infeasibility index. These conditions ensure that Equation 1 is satisfied.

Let us consider the abstraction $\mathcal{H}_4$ and the counter-example $l_1 e_2 l_3 e_4 l_4$. We observe that the infeasibility index corresponds to transition from l_3 to l_3 with time elapse. Hence, we need to add y. However, since the g.c.d of the constants corresponding to location l_3, namely, the differential inclusion $\dot{y} \in [2, 2]$ and the invariant $y \in [-10, 10]$, is 2, we assign a scaling factor of 2 with y. The resulting refinement is the automaton $\mathcal{H}_5$ shown in Figure 4.

Again, the flow abstraction based CEGAR algorithm always terminates, because there are only finite number of refinements starting from any abstraction. Every refinement entails adding a variable or changing the scaling associated with a variable. Note that the new scaling factor is necessarily lower than or equal to the previous scaling factor, because by definition the new scaling factor is a divisor of the previous scaling factor. Since the number of variable is finite, and the scaling factor associated with a newly introduced variable is finite, we obtain that the CEGAR algorithm terminates in a finite number of iterations. Also, the CEGAR algorithm is fair.

Theorem 3. *The flow abstraction based CEGAR algorithm is complete for the class of initialized rectangular hybrid automata, and is fair.*

4.4 Discussion

We obtain a semi-complete algorithm for the class of rectangular hybrid automata by composing all the three algorithms, and a complete algorithm for the class of initialized rectangular automata by composing the last two algorithms. The compositional CEGAR framework provides a convenient method to describe and implement CEGAR algorithms in a modular fashion.

The hybrid abstraction based CEGAR algorithms have the advantage that the various phases of the CEGAR loop are more efficient. As said before, construction of the abstraction is simpler because one can avoid expensive unbounded *Post* computations with respect to time. However, the validation phase as described requires computing unbounded *Pre*. One can avoid unbounded *Pre* computations, by using a small trick. One can add a new clock (a variable x with $\dot{x} = 1$), which forces taking a discrete transition every τ time units. This is achieved by adding self loops on the locations with a guard $x \in [\tau, \tau]$ and reset $x :\in [0, 0]$ and adding the constraint $x \in [0, \tau]$ to the invariant. The new system is equivalent to the old system in terms of checking safety. However, validating a counter-example of the new system requires computing *Pre* with respect to the time interval $[0, \tau]$.

5 Implementation and Experimental Results

The tool, which we call **H**ybrid **A**bstraction **R**efinement **E**ngine (HARE), implements the CEGAR algorithm in C++. HARE input consists of a hybrid automaton and a single initial abstraction function (which combines all the three different types of abstractions). This function defines the initial abstract hybrid automaton. The default initial abstract automaton has no variables and has three locations—an initial location, an unsafe location, and a third location corresponding to all the other locations of the concrete automaton. This abstract automaton is automatically translated to the input language for HyTech [18] and then model-checked. If HyTech does not produce a counterexample for the safety property, HARE returns the current abstraction. Otherwise, the counterexample is parsed and validated. In order to validate the counter-example, we need to compute *Pre* and *Post* with respect to certain edges and/or time. However, HyTech allows taking *Pre* and *Post* only with respect to the set of all the transitions. Thus, our implementation of the validation involves construction of new hybrid automata corresponding to the counter-example, and calls HyTech's *Pre* and *Post* functions on these automata. These calls to HyTech, at least in part, contribute to the relatively large time that HARE spends in the validation step for some of the case studies. Our implementation consists of a single CEGAR algorithm which is a combination of the the three CEGAR schemes presented in Section 4. It does not correspond to any particular composition of the algorithms, and our presentation of the algorithm as a composition is purely for highlighting the ideas in the implementation in a readable manner.

5.1 Experimental Results

Our experimental evaluation of HARE (see Table 1) is based on five classes of examples:

1. BILL_n models a ball in an n-dimensional bounded reflective rectangle. The unsafe set is a particular point in the bounded rectangle.

2. NAV_n models the motion of a point robot in an $n \times n$ grid where each region in the grid is associated with a rectangular vector field. When the robot is in a region, its motion is described by the flow equations of that region. The unsafe set is a particular set of regions. NAV_n_A and NAV_n_B represent the two different configurations of the vector fields, the initial and the unsafe regions. NAV_n_C models two robots on the same the $n \times n$ grid with different initial conditions; the unsafe set being the set of states where the two robots simultaneously reach the same unsafe region.
3. SATS_n models a distributed air traffic control protocol with n aircraft presented in [21]. The model of each aircraft captures several (8 or 10) physical regions in the airspace where the aircraft can be located, such as the left holding region at 3K feet, the left approach region, the right missed-approach region, the runway, etc. The continuous evolution of the aircraft are described by rectangular dynamics within each region. An aircraft transitions from one to another region based on the rules defined by the traffic control protocol, which involves the state of the current aircraft and also other aircrafts. Thus, the automata for the different aircraft communicate through shared variables. The safety property requires that the distance between two aircraft is never less than a safety constant c. We have worked on two variants of SATS: SATS_n_S models just one side of the airspace and the full SATS_n_C has two sides.
4. FISME_n models Fischer's timing-based mutual exclusion algorithm with n concurrent processes.
5. ZENO is a variant of the well-known 2D bouncing ball system where the system has zeno executions.

It is clear from the above table that Hare produces relatively small abstractions: in some cases with two orders of magnitude reduction in the number of locations, and often reducing the continuous state space by one or two dimensions. In the extreme case of NAV_n_A, an abstraction with 6 discrete states is found in 4 iterations, independent of the size of the grid. This is not too surprising in hindsight because the final abstraction clearly illustrates why only a constant number of control locations can reach the unsafe region in this example, and it successfully lumps all the unreachable locations together. Yet, the total verification time is better for HyTech for NAV_n_A and NAV_n_B than Hare primarily because, as discussed earlier, Hare makes numerous calls to HyTech not only for model checking the abstract automaton but also for the validation and the refinement refinement steps. Note that the time taken for abstraction refinement is comparable to that of the time taken for direct verification by HyTech. Hare 's advantage is apparent in the case of NAV_C_*, SATS, and FISME, where the system consists of several automata evolving in parallel. In NAV_C, for example, since the motion of each of the robots can be abstracted into a simpler automaton with less number of discrete locations, the state space of the composition of these abstract automaton is reduced dramatically (exponentially in the number of robots) and this is apparent in the differences in the running time.

Problem	Conc. size (locs, vars)	Abst. size (locs, vars)	Iter.	Validation (sec)	Abstraction Refinement(sec)	Hare (sec)	HyTech (sec)
BILL_2_A	(6,2)	(4, 1)	1	0.02	0.04	0.06	0.03
BILL_3_A	(8,3)	(4, 1)	1	0.04	0.06	0.1	0.04
NAV_10_A	(100,2)	(6, 2)	4	0.64	0.16	0.8	0.16
NAV_15_A	(225,2)	(6, 2)	4	1.07	0.18	1.25	0.27
NAV_10_B	(100,2)	(5, 1)	4	0.67	0.16	0.83	0.24
NAV_15_B	(225,2)	(5, 1)	4	1.84	0.29	2.13	0.52
NAV_8_C	(64^2,4)	(7^2, 4)	5	1.45	1.39	2.84	23.54
NAV_10_C	(100^2,4)	(7^2, 4)	5	2.41	1.51	3.92	58.24
NAV_14_C	(196^2,4)	(7^2, 4)	5	5.38	1.74	7.12	346.83
SATS_3_S	(512,3)	(320, 2)	4	0.48	1.92	2.40	2.64
SATS_4_S	(4096,4)	(1600, 2)	4	5.25	15.38	20.63	23.75
SATS_5_S	(32786,5)	(8000,2)	4	45.79	106.58	154.17	189.65
SATS_3_C	(1000,4)	(500, 2)	5	2.04	3.82	5.86	6.26
SATS_4_C	(10000,5)	(2500, 2)	5	22.25	41.37	63.98	76.63
FISME_2	(4^2,4)	(9, 4)	4	0.03	0.07	0.1	0.02
FISME_3	(4^3,5)	(36, 4)	4	0.44	1.34	1.78	1.98
FISME_4	(4^4,6)	(144, 4)	4	28.27	22.21	50.48	78.23
ZENO_BOX	(7,2)	(5,1)	1	0.04	0.04	0.08	—

Table 1. The columns (from left) show the problem name, sizes of the concrete and final abstract hybrid automaton, number of CEGAR iterations, time taken for validation, time taken for refinement, total time by Hare and the time taken by HyTech

In SATS, the time taken for validation is less compared to the time taken for abstraction and refinement steps. This is primarily because of the nature of the system. In SATS case study, the time taken to verify the abstractions is considerable while compared to other case studies.

The advantage of variable-hiding abstraction is apparent in ZENO (HyTech does not terminate in this case), as a subset of variables are sufficient to infer the safety of the system. We believe that in a complex hybrid automaton, with several components, adding the sufficient number of variables and abstracting the state space of hybrid automaton will yield better abstractions. All of this suggests, a direction of research, one we plan on pursuing, where the model-checker is more closely integrated with an abstraction refinement tool such as Hare.

References

1. HARE. In *https://wiki.cites.uiuc.edu/wiki/display/MitraResearch/HARE.*
2. R. Alur, T. Dang, and F. Ivancic. Counter-Example Guided Predicate Abstraction of Hybrid Systems. In *TACAS 2003*, pages 208–223, 2003.
3. Euene Asarin, Thao Dang, and Oded Maler. The d/dt tool for verification of hybrid system, 2002.

4. T. Ball and S. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *Proc. of the SPIN*, pages 113–130, 2000.
5. E.M. 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. *JFCS*, 14(4):583–604, 2003.
6. E.M. Clarke, A. Fehnker, Z. Han, B. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald. Verification of Hybrid Systems Based on Counterexmple-Guided Abstraction Refinement. In *TACAS*, pages 192–207, 2003.
7. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *CAV*, pages 154–169, 2000.
8. J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE*, pages 439–448, 2000.
9. H. Dierks, S. Kupferschmid, and K.G. Larsen. Automatic Abstraction Refinement for Timed Automata. In *FORMATS*, pages 114–129, 2007.
10. Laurent Doyen, Thomas A. Henzinger, and Jean franois Raskin. Automatic rectangular refinement of affine hybrid systems. In *in FORMATS05, vol. 3829 in LNCS*, pages 144–161. Springer, 2005.
11. A. Fehnker, E.M. Clarke, S. Jha, and B. Krogh. Refining Abstractions of Hybrid Systems using Counterexample Fragments. In *HSCC 2005*, pages 242–257, 2005.
12. G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable Verification of Hybrid Systems. In *Proc. of CAV*, 2011.
13. Goran Frehse. Phaver: Algorithmic verification of hybrid systems past hytech. In *HSCC*, pages 258–273, 2005.
14. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *POPL 2002*, pages 58–70, 2002.
15. T.A. Henzinger, P.W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *Proc. of STOC*, pages 373–382, 1995.
16. Thomas A. Henzinger. The theory of hybrid automata. In *LICS*, pages 278–292, 1996.
17. Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech: A model checker for hybrid systems. In *CAV*, pages 460–483, 1997.
18. Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech: A model checker for hybrid systems. In *CAV*, pages 460–483, 1997.
19. G. Holzmann and M. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, 2000.
20. Sumit Kumar Jha, Bruce H. Krogh, James E. Weimer, and Edmund M. Clarke. Reachability for linear hybrid automata using iterative relaxation abstraction. In *HSCC 2007*, pages 287–300, 2007.
21. Cesar A. Munoz, Gilles Dowek, and Vctor Carreo. Modeling and verification of an air traffic concept of operations. In *ISSTA*, pages 175–182, 2004.
22. Pavithra Prabhakar, Sridhar Duggirala, Sayan Mitra, and Mahesh Viswanathan. Hybrid automata-based cegar for rectangular hybrid automata. In *http://www.its.caltech.edu/ pavithra/Papers/rtss2012tr.pdf*.
23. M. Segelken. Abstraction and Counterexample-guided Construction of Omega-Automata for Model Checking of Step-discrete linear Hybrid Models. In *CAV*, pages 433–448, 2007.
24. M. Sorea. Lazy approximation for dense real-time systems. In *Proc. of FORMATS/FTRFTS*, pages 363–378, 2004.