Efficient Dynamic Error Reduction for Hybrid Systems Reachability Analysis

Stefan Schupp(✉) and Erika Ábrahám

RWTH Aachen University, Aachen, Germany
stefan.schupp@cs.rwth-aachen.de

Abstract. To decide whether a set of states is reachable in a hybrid system, over-approximative symbolic successor computations can be used, where the symbolic representation of state sets as well as the successor computations have several parameters which determine the efficiency and the precision of the computations. Naturally, faster computations come with less precision and more spurious counterexamples. To remove a spurious counterexample, the only possibility offered by current tools is to reduce the error by re-starting the complete search with different parameters. In this paper we propose a CEGAR approach that takes as input a user-defined ordered list of search configurations, which are used to dynamically refine the search tree along potentially spurious counterexamples. Dedicated datastructures allow to extract as much useful information as possible from previous computations in order to reduce the refinement overhead.

1 Introduction

As the correct behavior of *hybrid systems* with mixed discrete-continuous behavior is often safety critical, a lot of effort was put into the development and implementation of techniques for their analysis. In this paper we focus on techniques for proving unreachability of a given set of unsafe states. Besides methods based on theorem proving [11,21,25], logical encoding [13,15,22,26] and validated simulation [12,28], *flowpipe-construction-based methods* [2,7,9,17–20,27] show increasing performance and usability. These methods over-approximate the set of states that are reachable in a hybrid system from a given set of initial states by executing an iterative forward reachability analysis algorithm. The result is a sequence of state sets whose union contains all system paths starting in any initial state (usually for bounded time duration and a bounded number of discrete steps, unless a fixedpoint could be detected).

If the resulting over-approximation does not intersect with the unsafe state set then the verification task is successfully completed. However, if the intersection is not empty, due to the over-approximation the results are not conclusive. In this case the only possibility for achieving a conclusive answer is to change

This work was supported by the German research council (DFG) in the context of the HyPro project and the DFG Research Training Group 2236 UnRAVeL.

some analysis parameters to reduce the approximation error. As a smaller error typically comes with a higher computational effort, the choice of suitable parameters by the user can be a tedious task.

Most tools do not support the dynamic change of those parameters, thus after the modification of the parameters the user has to re-start the whole computation. One of the few tools implementing some hard-coded dynamic parameter adaptations is the STC mode [16] of SpaceEx [17], which dynamically adapts the time-step size during reachability analysis to detect the enabledness of discrete events more precisely. Another parameter (the degree of Taylor approximations) is dynamically adapted in the Flow* tool [9]. The method [5], also implemented in SpaceEx, uses cheap (but stronger over-approximating) computations to detect potentially unsafe paths and use this information to guide more precise (and more time-consuming) computations. In [6] the authors present a method to automatically derive template directions when using template polyhedra as a state set representation in a CEGAR refinement fashion during analysis. As a last example, in [24] the authors use model abstraction to hide model details and apply model refinement if potential counterexamples are detected; after each refinement, the approach makes use of previous reachability analysis results and adapts them for the refined model, instead of a complete restart.

However, none of the available tools supports the dynamic adjustments of several parameters by a more elaborate strategy, which is either defined by the user or chosen from a pre-defined set. In this paper we propose such an approach, provide an implementation based on the HyPro [27] programming library, present some use cases to demonstrate its applicability and advantages, and discuss ideas for further extensions and improvements. Our main contributions are:

– the definition of *search strategies* to specify the dynamic adjustment of parameter configurations;
– the formalization of a general *reachability analysis algorithm with dynamic configuration adjustment* following a search strategy, where *dynamic* means that adjustments are triggered during the analysis process in a fully automated manner only for parts of the search where they are needed to achieve conclusive analysis results;
– the identification of information, collected during reachability analysis, which can be *re-used* after a parameter adjustment to reduce the computational effort of forthcoming analysis steps;
– a *datatype* to store information about previously completed analysis steps, including information about re-usability, and supporting dynamic parameter adjustments according to a given strategy;
– the *implementation* of the reachability analysis algorithm using dynamic parameter adjustment and supporting information re-usage;
– the *evaluation* of our method on some case studies.

Outline. In Sect. 2 we recall some preliminaries on flowpipe-construction-based reachability analysis, before presenting our algorithm for the dynamic adjustment of parameter configurations in Sect. 3. In Sect. 4 we provide some experimental results and conclude the paper in Sect. 5.

2 Preliminaries

In this work we develop a method to dynamically adjust the parameters of a verification method for *autonomous linear hybrid systems* whose continuous dynamics can be described by *ordinary differential equations* (*ODEs*) of the form $\dot{x}(t) = A \cdot x(t)$, but our approach can be naturally extended to methods for non-autonomous hybrid systems with external input or non-linear dynamics.

Hybrid automata [3] are one of the modeling formalisms for hybrid systems. Similarly to discrete transition systems, nodes (called *locations* or *control modi*) model the discrete part of the state space (e.g. the states of a discrete controller) and transitions between the nodes (called *jumps*) labeled with guards and reset functions model discrete state changes. To model the continuous dynamics between discrete state changes, *flows* in the form of ordinary differential equation (ODE) systems, and *invariants* in the form of predicates over the model variables are attached to the locations. The ODEs specify the evolution of the continuous quantities over time (called the *flowpipe*), where the control is forced to leave the current location before its invariant gets violated. Initial predicates attached to the locations specify the initial states.

A *state* $\sigma = (\ell, \nu)$ of a hybrid automaton consists of a location l and a variable valuation ν. A *region* is a set of states $(\ell, P) = \{\ell\} \times P$. A *path* π of a hybrid automaton is a sequence $\pi = \sigma_0 \xrightarrow{t_0} \sigma_1 \xrightarrow{e_1} \sigma_2 \xrightarrow{t_2} \ldots$ of time steps $\sigma_i \xrightarrow{t_i} \sigma_{i+1}$ of duration t_i and discrete steps $\sigma_k \xrightarrow{e_k} \sigma_{k+1}$ following a jump, where $\sigma_0 = (\ell_0, \nu_0)$ is an initial state. A state is called *reachable* if there exists a path leading to it.

Flowpipe-construction-based reachability analysis aims at determining the states that are reachable in (a model of) a hybrid system, in order to show that certain unsafe states cannot be reached. Since the reachability problem for hybrid systems is in general undecidable, these methods usually *over-approximate* the set of states that are reachable along paths with a bounded number of jumps (called the *jump depth*) J and a bounded time duration T (called the *time horizon*) between two jumps. We explain the basic ideas needed to understand our contributions; for further reading we refer to, e.g., [8,23].
Starting from an initial region (ℓ_0, V_0), the analysis over-approximates flowpipes and jump successors iteratively. Due to non-determinism, this generates a *tree*, whose nodes n_i are either *unprocessed* leafs storing a tuple $(\pi_i;\ \ell_i, V_i;\ \bot)$, or *processed* inner nodes storing $(\pi_i;\ \ell_i, V_i;\ V_{i,0}, \ldots, V_{i,k_i})$.

The pair (ℓ_i, V_i) is the node's *initial region*, which is (ℓ_0, V_0) for the *root*. By $\pi_i = I_{i,0}, e_{i,0}, \ldots, I_{i,d_i}, e_{i,d_i}$, with $I_{i,l}$ being intervals and $e_{i,l}$ being jumps, we encode a set $\{\sigma_0 \xrightarrow{t_0} \sigma_0' \xrightarrow{e_{i,0}} \sigma_1 \ldots \xrightarrow{e_{i,d_i}} \sigma_{d_i+1} \mid \sigma_0 \in (\ell_0, V_0), t_l \in I_{i,l}\}$ of paths along which (ℓ_i, V_i) is reachable.

To process a node $(\pi_i;\ \ell_i, V_i;\ \bot)$, we divide the time horizon $[0, T]$ into segments $[t_{i,0}, t_{i,1}], \ldots, [t_{i,k_i}, t_{i,k_i+1}]$ with $t_{i,0} = 0$ and $t_{i,k_i+1} = T$, and for each segment $[t_{i,j}, t_{i,j+1}]$ we compute an over-approximation $V_{i,j}$ of the states reachable from V_i in ℓ_i within time $[t_{i,j}, t_{i,j+1}]$. I.e., $R_i = \cup_{j=0}^{k_i} V_{i,j}$ contains all valuations reachable in location ℓ_i from V_i within time T. The segmentation is usually homogeneous, meaning that the *time-step size* $t_{i,j+1} - t_{i,j}$ is constant, but there are also approaches for dynamic adaptations.

The processing is completed by computing for each *flowpipe segment* $V_{i,j}$ and each jump e from ℓ_i to some ℓ'_i an over-approximation $V^e_{i,j}$ of the valuations reachable from $V_{i,j}$ by executing e. To store the jump successors, either we add a child node $(\pi_i, [t_{i,j}, t_{i,j+1}], e;\ \ell'_i, V^e_{i,j};\ \bot)$ to n_i for each $V^e_{i,j} \neq \emptyset$, or we *aggregate* successors along a jump e into a single child node $(\pi_i, [t_{i,j}, t_{i,j'}], e;\ \ell'_i, R^e_i;\ \bot)$ with $V^e_{i,l} = \emptyset$ for all $l \notin [j, j'-1]$ and $\cup_e \cup_{j'' \in [j,j'-1]} V^e_{i,j''} \subseteq R^e_i$, or we *cluster* successors along a jump into a fixed number of child nodes (see Fig. 3).

For illustration purposes, above we stored all flowpipe segments $V_{i,j}$ in the nodes. In practice they are too numerous and if they contain no unsafe states then they are deleted. In the following, we assume that each node stores a tuple $(\pi_i;\ \ell_i, V_i;\ p)$, where the flag p is 1 for processed nodes and 0 otherwise. (For a simple reachability analysis, we need to store neither the path nor the processed flag, but we will make use of the information stored in them later on. Furthermore, we could even delete the initial regions of processed nodes, however, besides counterexample and further output generation, they might be also useful for fixedpoint detection.)

State set representations are one of the core components in the above analysis procedure. Additionally to the storage of state sets, these datatypes need to provide certain (over-approximative) operations (union, intersection, linear transformation, Minkowski sum etc.) on states sets. Besides geometric representations (e.g., boxes/hyperrectangles, oriented rectangular hulls, convex polyhedra, template polyhedra, orthogonal polyhedra, zonotopes, ellipsoids) also symbolic representations (e.g., support functions or Taylor models) can be used for this purpose. The variety of representations is rooted in the general problem of deciding between computational effort and precision. Generally, faster computations often come at the cost of precision loss and vice versa, more precise computations need higher computational effort. The representations might differ in their size, i.e., the required memory consumption, which has a further influence on the computational costs for operations on these representations.

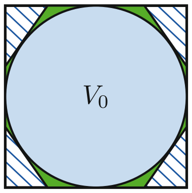
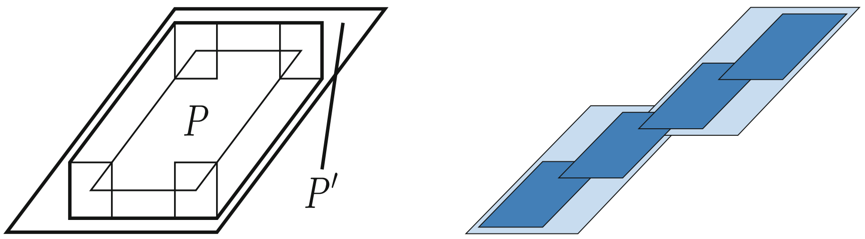


Fig. 1. Polytope (green) and box (hatched) approx. of state set V_0. (Color figure online)

3 CEGAR-Based Reachability Analysis

If potential reachability of an unsafe state is detected by over-approximative computations, in order to achieve a conclusive verification result, we need to *reduce the over-approximation error* to an extent that allows to determine that the counterexample is spurious.



(a) Over-approximating the Minkowksi sum of a polytope P and a box by a less complex polytope. P'.

(b) Smaller time steps typically lead to more precise computations (dark blue) than larger time steps (light blue).

Fig. 2. Reduction and time-step size influence the flowpipe over-approximation error. (Color figure online)

Search parameters, parameter configurations and search strategies. The size of the over-approximation error depends on various search parameters, which influence besides the precision also the computational effort of the performed analysis:

1. *State set representation*: The choice of the state set representation has a very strong influence on both the error and the running time of the computations. For example, boxes are very efficient but introduce large over-approximations, whereas convex polyhedra are in general more precise but computationally much more expensive (see Fig. 1).
2. *Reductions*: Some of the state set representations can grow in the representation size during the computations. For example, during the analysis we need to compute the Minkowski sum $A \oplus B = \{a + b \mid a \in A \wedge b \in B\}$ of two state sets A and B. Figure 2(a) shows a 2-dimensional example to illustrate how the representation size of a polytope P in the vertex representation (storing the vertices of the polytope) increases from 4 to 6 when building the Minkowski sum with a box. Another source of growing representation sizes are large enumerators and/or denominators when using rationals to describe for instance coefficients of vectors. When the size of a representation gets too large we can try to reduce it on the cost of additional over-approximation. Thus the precision/cost is dependent also on the fact whether such reductions take place.
3. *Time-step size*: The time-step size for the flowpipe construction can be constant or dynamically adapted. In the constant case it directly determines the number of flowpipe segments that need to be over-approximated and for which jump successors need to be computed. In the case of dynamic adaptation, the adaptation heuristics determines the number of segments and thus the computational effort. In both cases, smaller time-step sizes often lead to more precise computations on the cost of higher computational effort as more segments are computed (see Fig. 2(b)).
4. *Aggregation and clustering*: The precision is higher if no aggregation takes place or if the number of clusters increases (see Fig. 3). However, completely

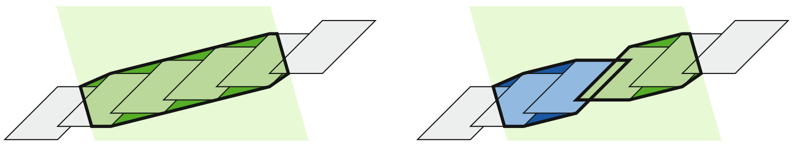


Fig. 3. Six sets (gray), a guard (light green), the aggregation of their intersections (left, thick line), and the clustering of their intersections into two sets (right, thick lines); both aggregation and clustering introduces additional error (dark green and dark blue). (Color figure online)

switching off both aggregation and clustering often leads to practically intractable computational costs. Increasing the precision by allowing a larger number of clusters can improve the precision by managable increase in the running times, but the number of clusters should be carefully chosen considering also the size of the time steps (as they determine the number of flowpipe segments and thus the number of state sets to be clustered).

5. *Splitting initial sets*: Large initial state sets might be challenging for the reachability analysis. If the algorithm cannot find a conclusive answer, we can split the initial set into several subsets and apply reachability analysis to each of the subsets. Besides the enabling/disabling of initial state set splitting, also the splitting heuristics is relevant for the precision. In general, a fewer number of initial state sets is less precise but more cheap to compute with. Furthermore, it might be also relevant where the splitting takes place.

Most flowpipe-construction-based tools allow the user to define a *search parameter configuration*, fixing values for the above-listed search parameters. Aside from a few exceptions mentioned in the introduction, this configuration remains constant during the whole analysis. Whenever an unsafe state is detected to be potentially reachable, the user can re-start the analysis with a different parameter configuration to reduce the over-approximation error.

As the executions with different parameter configurations are completely independent, potentially useful information from previous search processes gets lost. To enable the exploitation of such information, we propose an approach to build a connection between executions with different parameter configurations.

Instead of a single configuration, we propose to define an ordered sequence $c_0, \ldots, c_n$ of search parameter configurations, which we call a *search strategy*, whereas the position of a parameter configuration within a search strategy is called its *refinement level*. Configurations at higher refinement levels should typically lead to more precise computations, but this is not a soundness requirement.

Dynamic configuration adaptation. We start the analysis with the first configuration in the search strategy, i.e. the one at refinement level 0. If the analysis with this configuration can prove safety then the process is completed.

Otherwise, if the reachability computation detects a (potentially spurious) counterexample then the search with the current configuration is paused; note that at this point there might be unprocessed nodes whose successors were not yet computed. Now, our goal is to exclude the detected counterexample by doing as few computations as possible using configurations at higher refinement levels and, if we succeed, process those yet unprocessed nodes further at refinement level 0. For the first counterexample this means intuitively re-computing reachability only along the counterexample path with the configuration at refinement level 1; we say that we *refine the path.* Note that the result of a path refinement can be a tree, e.g. if the refinement switched off aggregation. If the counterexample could be excluded by the path refinement, then we switch back to the previous refinement level to process the remaining, yet unprocessed nodes. Otherwise, if the counterexample could not be excluded then we get another, refined counterexample; in this case we recursively try to exclude this counterexample by switching to the configuration at the second refinement level etc.

Let us first clarify what we mean by *refining a counterexample path.* We define a counterexample to be a path in the search tree. If the configuration, which created the counterexample, used aggregation then it means determining the flowpipes and the jump successors for the given sequence of locations (as stored in the nodes on the path) and jumps (as stored on the edges) with the configuration at the next-higher refinement level. However, if the previous configuration did not aggregate then we need to determine only a subset of the jump successors, namely those whose time point is covered by the counterexample.

Now let us discuss what it means to refine a path *by doing as few computations as possible.* If we find a counterexample at a refinement level i then we need a refinement for the whole path at level $i+1$. However, another counterexample detected previously at level i might share a prefix with the current one; if the previous counterexample has already been refined then we need to refine only the not-yet-refined postfix of the current counterexample.

The analysis at refinement level 0 and each path refinement computation generates a search tree. To reduce the computational effort as much as possible, we have to exchange information between these search trees. For example, for a given counterexample found at refinement level i we need to know whether a prefix of it was already refined at level $i+1$. To allow such information exchange, we could store each search tree separately and extract information from the trees when needed by traversing them. This option requires the least management overhead during reachability computations but it has major drawbacks from the point of computational costs for tree traversal. Alternatively, we could store each search tree separately but store in addition refinement relations between their nodes, allowing to relate paths and retrieve information more easily. However, we would have high costs for setting up and storing all node relations. Instead, we decided to collect all information in a single *refinement tree.* Tree updates require a careful management of the refinement nodes and their successors, but the advantage is that information about previous searches is easier accessible.

Next we first discuss how nodes of the refinement tree are processed, how paths in the refinement tree are refined, and finally we explain our dynamical parameter refinement algorithm.

The algorithm. Each refinement tree node n_i is a kind of "meta-node" that contains an ordered sequence $(n_i^0, \ldots, n_i^{u_i})$ with $0 \le u_i \le n$, where $n+1$ is the size of the search strategy, and each entry n_i^j has the form $(\pi;\ \ell, V;\ p)$ as explained in Sect. 2.

Assume for simplicity that the model has a single initial region (ℓ_0, X_0), and let $V_{0,i}$ represent X_0 according to the state set representation of refinement level i. The refinement tree is initialized with a root node $n_0 = (n_0^0, \ldots, n_0^n)$ with $n_0^i = (\epsilon;\ \ell_0, V_{0,i};\ 0)$.

We additionally introduce a *task list* which is initialized to contain $(n_0; 0; \epsilon)$ only. Elements $(n_i; j; \pi)$ in the task list store the fact that we need to compute successors for the jth element of the refinement node n_i at level j. If $\pi = \epsilon$ then we are not refining and we need to consider all the successors for further computations, otherwise we are at a refinement level $j > 0$ and only the successors along the counterexample-path π need to be considered.

We remove and process elements from the task list one by one. Assume we consider the task list element $(n_i; j; \pi')$ with $n_i^j = (\pi;\ \ell, V;\ p)$.

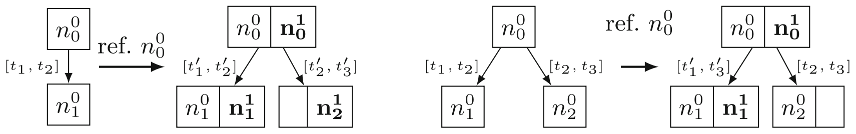
If $p = 0$ then we over-approximate the flowpipe starting from V in ℓ for the time horizon T, using the configuration at level j in the search strategy.

If the computed flowpipe segments contain no bad states and the jump depth J is not yet reached then we compute also the jump successors. Depending on the clustering/aggregation settings at level j, this yields a set of jump successor regions $R_1, \ldots, R_m$ with $R_k = (\ell_k, V_k)$ over time intervals $I_1, \ldots, I_m$ along jumps $e_1, \ldots, e_m$. If the number of children m' of n_i is less than m then we add $m - m'$ new children; if $m' > 0$ then we add to the newly created children as many dummy entries (containing empty sets) as the other children have, in order to bring all children to the same refinement level. After that, we select for each $k = 1, \ldots, m$ a different child $\hat{n}_k$ of n_i and append $(\pi, I_k, e_k;\ \ell_k, V_k;\ 0)$ to the child's entry sequence (see Fig. 4). If $m' > m$ then we add to all not selected children (to which no new entry was added) a dummy entry. Finally, we set p to 1.

If the node could be processed without discovering any bad states (or if p was already 1 and thus processing was not needed) then we update the task list as follows:

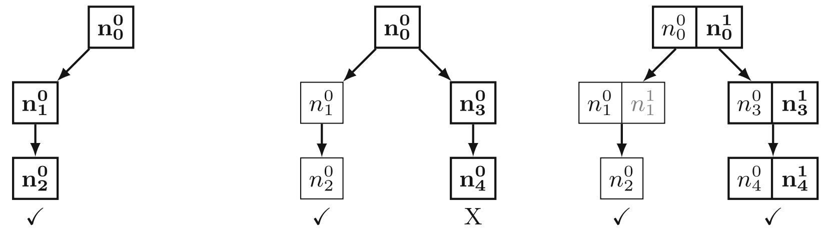
- If $\pi' = \epsilon$ then we have to process all successor nodes at the level j' determined by the number of entries E in each of the nodes $\hat{n}_k$. We add $(\hat{n}_k; E; \epsilon)$ to the task list for all $k = 1, \ldots, m$.
- Otherwise, if $\pi' = I, e, \pi''$ then we add $(\hat{n}_k; j; \pi'')$ for all $k = 1, \ldots, m$ for which $I_k \cap I \neq \emptyset$ and $e_k = e$.

Note that if $\pi' = \epsilon$ but $j > 0$ then we just succeeded to refine a spurious counterexample from level $j - 1$ to a safe path at level j and can continue further successor computations using a lower level configuration. This switch to a lower level happens because the children $\hat{n}_k$ of n_i have less then j entries in their queues. Now the processing is completed and the next element from the task list can be approached.



(a) Node refinement adds child nodes. (b) Node refinement removes child nodes.

Fig. 4. Tree update after node refinement with changing number of child nodes and transition timing refinement.



(a) Safe path. (b) Counterexample. (c) Refinement.

Fig. 5. Partial tree refinement to remove a spurious counterexample.

If during processing $(n_i; j; \pi')$ with $n_i^j = (\pi;\ \ell, V;\ p)$ the computed flowpipe had a non-empty intersection with the set of unsafe states then we have found a counterexample at level j. If $j = n$ then the highest refinement level has been reached and the algorithmus terminates without any conclusive answer. Otherwise, if $j < n$, we repeat the computations along the counterexample path with a higher-level configuration (see Fig. 5). This is implemented by adding $(n_0; j+1; \pi, \pi')$ to the task list.

The main structure of the algorithm is shown in Algorithm 1.1.

3.1 Incrementality

The efficiency of the presented approach can be further improved by implementing *incrementality*: already available book-keeping and additional information gained throughout the computation can be exploited to speed up later refinements.

For example, the presented approach already keeps track of time intervals where jumps were enabled, i.e. the time intervals during which the intersection of a state set and the guard condition was non-empty. Assume we process $(n; i; \pi')$ at level i with $n_i = (\pi; \ell, V; p)$ being the ith entry in n. Let I be the union of all the time intervals for all flowpipe segments for which a non-empty jump successor was computed along a jump e. Later, when processing $(\hat{n}; j; \hat{\pi}')$ at level $j > i$ with $\hat{n}_j = (\hat{\pi}; \ell, \hat{V}; \hat{p})$ being the jth entry in $\hat{n}$, if the path set encoded by $\hat{\pi}$ is included in the path set encoded by π then we need to compute jump successors along e only for flowpipe segments over time intervals that have a non-empty intersection with I.

```
analyze(){
  while (true) do
    if (task list is empty) then
      return safe
    fi;
    take an element $(n_i; j; \pi')$ with $n_i^j = (\pi;\ \ell, V;\ p)$ from task list;
    if $(p = 0)$ then
      $R$ := computeFlowpipeSegments$(\ell, V, j)$
    fi;
    if $(p = 0$ and $R$ contains unsafe states) then
      if $(j = n)$ then return unknown;
      addToTaskList($(n_0; j + 1; \pi, \pi')$)
    else
      if (jump depth not yet reached) then
        computeJumpSuccessorsAndUpdateTaskList$(n_i, j, \pi', R)$
      fi
    fi
  od
}
```

Algorithm 1.1. Reachability analysis algorithm with backtracking and refinement.

Table 1. Strategies s_i with different refinement levels (lvl.). Strategies vary time step size (δ) and state set representation (box, sf = support function). Strategy s_5 changes aggregation and clustering (n = no aggregation, c:max. number of successor nodes).

	Strategies																
	s_0			s_1				s_2				s_3		s_4		s_5		
lvl.	0	1	2	0	1	2	3	0	1	2	3	0	1	0	1	0	1	2
δ	.01	.001	.0001	.01	.001	.01	.001	.01	.001	.01	.0001	.1	.001	.1	.001	.1	.001	.001
rep.	box	box	sf	box	box	sf	sf	box	box	sf	sf	box	sf	box	poly	box	box	sf
agg.	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	n	c:3	c:3

Similarly, if (ℓ, V) contains no unsafe states but $(\ell, \hat{V})$ does then we know that the latter counterexample is spurious if the path set encoded by $\hat{\pi}$ is included in the path set encoded by π.

A similar observation holds for flowpipe segments: if a segment in the flowpipe of $(\ell, \hat{V})$ is empty, what happens if the invariant is violated, then we know that the same segment of the flowpipe from $(\ell, \hat{V})$ will also be empty.

4 Experimental Results

In order to show the general applicability of our approach we have conducted several experiments on an implementation of the method presented in Sect. 3. We have used our implementation to verify safety of several well-known benchmarks using different strategies (see Table 1). All experiments were carried on an Intel Core i7 (4×4 GHz) CPU with 16 GB RAM. Results for the used strategies can be found in Table 2.

Benchmarks. Different benchmarks from the area of hybrid systems verification are selected: The well-known bouncing ball benchmark models the height and velocity of a falling ball bouncing off the ground. The added set of bad states constrains the height of the ball after the first of 4 bounces. This benchmark already exhibits most properties more challenging benchmarks cover while being simple enough to be a sanity check for our method.

The 5-D switching system [10] is an artificially created model with 5 locations and 5 variables which shows more complex dynamic and is well-suited to show the differences in over-approximation error between the used state set representations. We added a set of bad states in the last location where the system's trajectories converge to a certain point.

The navigation benchmark [14] models the velocity and position of a point mass moving through cells on a two-dimensional plane (we used variations of instances 9 and 11). Each cell (location) exhibits different dynamic influencing the acceleration of the mass. The goal is to show that a set of good states can potentially be reached while a set of bad states will always be avoided (see Fig. 6(b)). The initial position of the mass is chosen from a set, such that this benchmark demonstrates non-determinism for the discrete transitions which results in a more complex search tree.

The platoon benchmark [1,4] models a vehicle platoon of three cars where two controlled cars follow the first one while keeping the distance e_i between each other within a certain threshold (see Fig. 6(a)). This benchmark was chosen, as it unifies a higher dimension of the state space with a more complex dynamic.

Strategies. During the development of our approach we tested several strategies with varying parameters (a) the state set representation, (b) the time step size and (c) aggregation settings. In general, other parameters (e.g. initial set splitting) could be also considered but our prototype currently does not yet support these. For this evaluation we selected six strategies $s_0, \ldots, s_5$ which mostly vary (a) and (b) (see Table 1). Changing aggregation settings has shown to be challenging for the tree update mechanism but the exponential blow-up of the number of tree nodes did not render this method effective in practice. Furthermore for disabled aggregation settings, the largest precision gain can be observed for boxes while for all other tested state set representations the effect can be neglected. Note that our prototype implements the general case where time step sizes are not necessarily monotonically decreasing and multiples of each other which implies refinement starting from the root node.

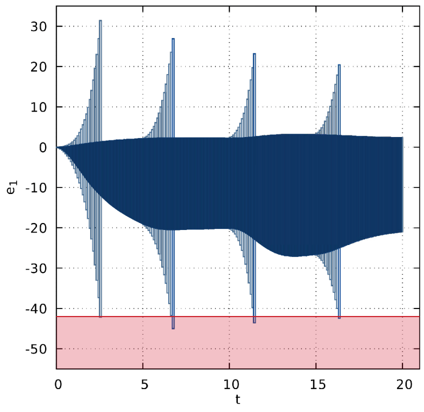
Comparison. We compare our refinement algorithm (1) with a classic approach where no refinement is performed. To achieve this, we specify only a single strategy element for our algorithm. We give results for (2) the fasted successful setting (of the respective strategy), an experienced user would choose and for (3) the setting with the highest precision level, a conservative user would select. The three entries per cell in Table 2 show the running times for our dynamical approach (gray), the fastest successful setting and the conservative approach. The numbers in brackets show the number of nodes in the search tree; for refinement strategies we give the number of nodes for each refinement level.

Table 2. Experimental results in seconds for different strategies. Timeout (TO) was set to 10 min, memout (MO) to 4 GB, (err) marks numerical errors. Three results per cell: (1) dynamic refinement (gray), (2) fastest successful setting only, (3) most precise setting. In brackets: Number of nodes in the search tree, refinement runs give the number of nodes on each level.

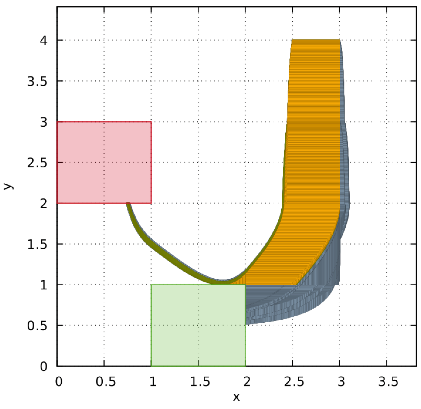
Bm.	Strategy					
	s_0	s_1	s_2	s_3	s_4	s_5
BBl	**0.15** (5|2|0)	**0.15** (5|2|0|0)	**0.15** (5|2|0|0)	0.46 (5|2)	1.58 (5|2)	0.21 (29|4|0)
	0.22 (5)	0.18 (5)	0.18 (5)	0.97 (5)	3.45 (5)	1.71 (121)
	11.93 (5)	0.97 (5)	9.90 (5)	0.97 (5)	3.45 (5)	9.47 (63)
Na09	TO	5.76 (279|6|4|0)	**5.09** (317|17|6|0)	549	err	TO
	TO	118 (244)	118 (244)	TO	TO	MO
	TO	TO	TO	TO	TO	TO
Na11	TO	7.15 (45|8|7|0)	7.61 (75|16|7|0)	63.4 (73|11)	err	120 (75|4168|0)
	TO	**6.4** (24)	**6.4** (24)	395 (24)	TO	130 (4170)
	TO	395 (24)	TO	395 (24)	TO	TO
5DS.	2.27 (5|5|5)	0.49(5|5|5|5)	2.3 (5|5|5|5)	0.39 (5|5)	15.31(5|5)	0.45(5|64|5)
	2.35 (5)	0.38 (5)	2.36 (5)	0.38 (5)	TO (5)	**0.37** (5)
	2.35 (5)	0.38 (5)	2.36 (5)	0.38 (5)	TO (5)	**0.37** (5)
Plat.	173 (5|4|4)	3.67 (5|4|4|0)	3.6 (5|4|4|5)	18.7 (5|4)	TO	19.16 (5|4|4)
	TO	**3.48** (5)	**3.48** (5)	18.9 (5)	TO	18.8 (5)
	TO	18.9 (5)	TO	18.9 (5)	TO	18.8 (5)

Observations. The results in Table 2 show that our method in general is competitive to classical approaches, as the running times are in the same orders of magnitude as the fastest setting when using dynamic refinement and in some cases our method is even faster. From the results we can infer manifold:

– Our implementation currently supports re-using information of guard intersection timings (see Sect. 3.1) while other information such as time intervals where a state set is fully contained in the set defined by the invariant of a location are not used. Keeping track of this reduced information already noticeably influences the running times as costly intersection operations for transition guards can be avoided for most computed segments and the running times can compete with the optimal setting. This shows that the additional cost of pre-computing parts of the search tree can be compensated in terms of running time when information is properly re-used.
– The length of the counterexample plays a significant role — in the bouncing ball benchmark the set of bad states is reachable after one discrete transition and from then on never again while in the 5-D switching system the set of bad states is reached in the last reachable location which causes a refinement of the whole tree and a recovery to a lower refinement level is not possible. In the platoon benchmark, stepping back to a lower refinement level does not provide any advantages, as an intersection with the set of bad states occurs before transition timings can be recorded (see Fig. 6(a)). To overcome this problem a future implementation should allow for additional entry points for refinement in order to reduce the length of the refinement path (see Sect. 5).
– The shape of the search tree influences the effectiveness of our approach. As the navigation benchmark is the only benchmark in our set where the resulting search tree naturally branches due to multiple outgoing transitions



(a) Platoon benchmark (variables t and e_1) for strategy s_3. Refinements (blue) increase in saturation. Discrete jumps occur at multiples of $5t$, Bad states are $e_1 \leq 42$ (bottom, red).



(b) Navigation benchmark (instance 9) with strategy s_1. The set of bad states (left box, red); the set of good states (bottom box, green); Refinements of strategy s_1 (blue, orange).

Fig. 6. Result plots for the platoon and the navigation benchmarks with refinement. (Color figure online)

per location, the effect of partial refinement can especially be observed for this benchmark. Whole subtrees can be cut off and are shown to be unreachable on higher refinement levels such that the number of nodes is reduced. The presented method renders most effectively for systems exhibiting non-determinism, which is reflected in a strongly branching search tree.

– Coarse analysis allows for fast discovery of the search tree, possibly requiring more nodes to be computed. We can observe that for models with non-determinism the number of nodes at the highest required level is lower than when using the classical approach. Together with the running times this confirms our assumption that putting effort in selective, partial refinement of single branches pays off in terms of computational effort.

In conclusion we expect a strategy where a coarse analysis precedes a fine-grained setting (e.g. strategy s_3) which allows to detect enabled transitions quickly and to recover fast after the removal of a spurious counterexample shows good results on average.

5 Conclusion

We presented a reachability analysis algorithm with dynamic configuration adjustment, which allows to refine search configurations to obtain conclusive results, but exploits as much information as possible from previous computations in order to keep the computational effort as low as possible. We plan to continue our work in several directions:

Incrementality. Our current implementation re-uses information from previous refinement levels about the time intervals of jump enabledness. We will implement also the re-usage of information when an invariant is definitely true or definitely violated (when the flowpipe segment for a time interval was fully contained or fully outside the invariant set).
Additional parameters. The current implementation supports 3 parameters in search strategies: time-step size, state set representation, aggregation and clustering settings. We aim at extend our search strategies with the adjustment of further parameters.
Dynamic strategy synthesis. Using information about a counterexample, e.g. the Hausdorff distance between the set of bad states and the state set intersecting it, automatically deriving strategies for partial path refinement could be further investigated.
Parameter synthesis. With little modification we can use our approach also to synthesize the coarsest parameter setting which still allows to verify safety. This can be achieved by strategies, where the parameter settings decrease in precision and the analysis stops when a bad state is potentially reachable.
Partial path refinement. Partial refinement of counterexamples, for example restricted to a suffix, could possibly improve the effectiveness of the approach (if the refinement of the suffix renders a bad state unreachable).
Conditional strategies. We defined search strategies to be ordered sequences of parameter configurations, which are used one after the other for refinements. Introducing *trees* of configurations with conditional branching would allow even more powerful strategies where the characteristics of the system or runtime information (like previous refinement times, state set sizes, number of sets aggregated etc.) can be used to determine which branch to take for the next refinement.

References

1. Althoff, M., Bak, S., Cattaruzza, D., Chen, X., Frehse, G., Ray, R., Schupp, S.: ARCH-COMP17 category report: continuous and hybrid systems with linear continuous dynamics. In: Proceedings of ARCH 2017, pp. 143–159 (2017)
2. Althoff, M., Dolan, J.M.: Online verification of automated road vehicles using reachability analysis. IEEE Trans. Robot. **30**(4), 903–918 (2014)
3. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoret. Comput. Sci. **138**(1), 3–34 (1995)
4. Ben Makhlouf, I., Kowalewski, S., Chávez Grunewald, M., Abel, D.: Safety assessment of networked vehicle platoon controllers- practical experiences with available tools. In: Proceedings of ADHS 2009 (2009)
5. Bogomolov, S., Donzé, A., Frehse, G., Grosu, R., Johnson, T.T., Ladan, H., Podelski, A., Wehrle, M.: Guided search for hybrid systems based on coarse-grained space abstractions. STTT **18**(4), 449–467 (2016)
6. Bogomolov, S., Frehse, G., Giacobbe, M., Henzinger, T.A.: Counterexample-guided refinement of template polyhedra. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 589–606. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_34

7. Bouissou, O., Chapoutot, A., Mimram, S.: Computing flowpipe of nonlinear hybrid systems with numerical methods. CoRR abs/1306.2305 (2013)
8. Chen, X.: Reachability Analysis of Non-Linear Hybrid Systems Using Taylor Models. Ph.D. thesis, RWTH Aachen University, Germany (2015)
9. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: an analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 258–263. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_18
10. Chen, X., Schupp, S., Makhlouf, I.B., Ábrahám, E., Frehse, G., Kowalewski, S.: A benchmark suite for hybrid systems reachability analysis. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 408–414. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_29
11. Collins, P., Bresolin, D., Geretti, L., Villa, T.: Computing the evolution of hybrid systems using rigorous function calculus. In: Proceedings of ADHS 2012, pp. 284–290. IFAC-PapersOnLine (2012)
12. Duggirala, P.S., Mitra, S., Viswanathan, M., Potok, M.: C2E2: a verification tool for stateflow models. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 68–82. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_5
13. Eggers, A.: Direct handling of ordinary differential equations in constraint-solving-based analysis of hybrid systems. Ph.D. thesis, Universität Oldenburg, Germany (2014)
14. Fehnker, A., Ivančić, F.: Benchmarks for hybrid systems verification. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 326–341. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24743-2_22
15. Fränzle, M., Herde, C., Ratschan, S., Schubert, T., Teige, T.: Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. J. Satisf. Boolean Model. Comput. **1**, 209–236 (2007)
16. Frehse, G., Kateja, R., Le Guernic, C.: Flowpipe approximation and clustering in space-time. In: Proceedings of HSCC 2013, pp. 203–212. ACM (2013)
17. Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_30
18. Hagemann, W., Möhlmann, E., Rakow, A.: Verifying a PI controller using SoapBox and Stabhyli: experiences on establishing properties for a steering controller. In: Proceedings of ARCH 2014. EPiC Series in Computer Science, vol. 34, pp. 115–125. EasyChair (2014)
19. HyCreate. http://stanleybak.com/projects/hycreate/hycreate.html
20. HyReach. https://embedded.rwth-aachen.de/doku.php?id=en:tools:hyreach
21. Immler, F.: Tool presentation: Isabelle/hol for reachability analysis of continuous systems. In: Frehse, G., Althoff, M. (eds.) ARCH14-15. 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems. EPiC Series in Computer Science, vol. 34, pp. 180–187. EasyChair (2015)
22. Kong, S., Gao, S., Chen, W., Clarke, E.: dReach: δ-reachability analysis for hybrid systems. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 200–205. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_15
23. Le Guernic, C.: Reachability analysis of hybrid systems with linear continuous dynamics. Ph.D. thesis, Université Joseph-Fourier-Grenoble I, France (2009)

24. Nellen, J., Driessen, K., Neuhäußer, M., Ábrahám, E., Wolters, B.: Two CEGAR-based approaches for the safety verification of PLC-controlled plants. Inf. Syst. Front. **18**(5), 927–952 (2016)
25. Platzer, A., Quesel, J.-D.: KeYmaera: a hybrid theorem prover for hybrid systems (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 171–178. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_15
26. Ratschan, S., She, Z.: Safety verification of hybrid systems by constraint propagation based abstraction refinement. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 573–589. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31954-2_37
27. Schupp, S., Ábrahám, E., Makhlouf, I.B., Kowalewski, S.: HyPro: A C++ library of state set representations for hybrid systems reachability analysis. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 288–294. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_20
28. Taha, W., et al.: Acumen: an open-source testbed for cyber-physical systems research. In: Mandler, B., et al. (eds.) IoT360 2015. LNICST, vol. 169, pp. 118–130. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47063-4_11