

Finding Errors of Hybrid Systems by Optimising an Abstraction-Based Quality Estimate^{*}

Stefan Ratschan¹ ^{**} and Jan-Georg Smaus² ^{***}

¹ Academy of Sciences of the Czech Republic, stefan.ratschan@cs.cas.cz

² University of Freiburg, Germany, smaas@informatik.uni-freiburg.de

Abstract. We present an algorithm for falsifying safety properties of hybrid systems, i.e., for finding a trajectory to an unsafe state. The approach is to approximate how close a point is to being an initial point of an error trajectory using a real-valued quality function, and then to use numerical optimisation to search for an optimum of this function. The function is computed by running simulations, where information coming from abstractions computed by a verification algorithm is exploited to determine whether a simulation looks promising and should be continued or cancelled. This information becomes more reliable as the abstraction becomes more refined. We thus interleave falsification and verification attempts.

1 Introduction

A hybrid system is a dynamical system with combined discrete and continuous state and evolution. An important problem is to ensure correctness, i.e., *verification*. However, during the design (debugging) process, hybrid systems are usually not correct yet, and hence *error detection* is equally important.

We address here the problem of automatically finding error trajectories that lead the system from an initial to an unsafe state. We distinguish ourselves from other recent works [3, 18] by two main aspects:

- The above methods aim at systems with a high amount of non-determinism (e.g., in the form of inputs), and do a broad search in the statespace spanned by the non-deterministic choices. For systems with completely deterministic evolution however, it is important to distinguish and prefer those regions of the search space that are most promising, which is the aim of this work.

^{*} An extended version of this paper has appeared as a technical report [20]. We acknowledge the help of Tomáš Dzetkulič with the implementation.

^{**} The work of Stefan Ratschan has been supported by GAČR grant 201/08/J020 and by the institutional research plan AV0Z100300504.

^{***} The work of Jan-Georg Smaus has been supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center “Automatic Verification and Analysis of Complex Systems” (SFB TR14/AVACS).

- In a similar way as related work in program verification [11, 17], we do not assume a-priori that our system is incorrect, but rather, we interleave verification, using abstractions of the system, and falsification attempts. The information contained in abstractions is valuable both for verification and falsification. More specifically, the abstraction allows for estimating whether a simulation approaches an unsafe state or not and is thus a promising candidate for an error trajectory.

The main idea of our algorithm is the following: We define a real-valued function (the *quality estimate*) onto the state space that approximates the notion of a given point being close to an initial point of an error trajectory. Then we use numerical optimisation techniques to search for an optimum of this quality estimate. The quality estimate is computed using information from the abstraction, and its accuracy improves as the abstraction is refined, hereby improving the chances of numerical search finding an actual error trajectory.

The rest of this paper is organised as follows: In the next section we define hybrid systems and abstractions thereof. In Sec. 3 we explain our search algorithm. In Sec. 4 we define the quality estimate. In Sec. 5 we discuss and analyse our method. Section 6 explains the implementation and reports on experiments. Section 7 is on related work, and Sec. 8 concludes.

2 Hybrid Systems and Abstractions

We briefly recall our formalism for modelling hybrid systems. It captures many relevant classes of hybrid systems, and many other formalisms for hybrid systems in the literature are special cases of it.

A hybrid system has a finite and nonempty set S of *modes*. $I_1, \dots, I_n \subseteq \mathbb{R}$ are compact intervals over which the n continuous variables of a hybrid system range. The state space of a hybrid system is denoted by $\Phi = S \times I_1 \times \dots \times I_n$.

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

The set $Init$ specifies the initial states of a hybrid system and $Unsafe$ the set of unsafe states. The relation $Flow$ specifies the possible continuous flows of the system by relating each state to corresponding derivatives, and $Jump$ specifies the possible discontinuous jumps by relating each state to a successor state. Formally, the behaviour of H is defined as follows:

Definition 2. A flow of length $l \geq 0$ is a function $r : [0, l] \rightarrow I_1 \times \dots \times I_n$, differentiable on $[0, l]$. A trajectory of H is a sequence of mode/flow pairs $(s_0, r_0), \dots, (s_k, r_k)$ of lengths l_0, \dots, l_k such that for all $i \in \{0, \dots, k\}$,

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

An error trajectory of a hybrid system H is a trajectory $(s_0, r_0), \dots, (s_k, r_k)$ of H such that $(s_0, r_0(0)) \in \text{Init}$ and $(s_k, r_k(l_k)) \in \text{Unsafe}$. H is safe if it does not have an error trajectory.

We use the following constraint language to specify hybrid systems: the variable s and the tuple of variables $\mathbf{x} = (x_1, \dots, x_n)$ range over S and $I_1 \times \dots \times I_n$, respectively. The tuple $\dot{\mathbf{x}} = (\dot{x}_1, \dots, \dot{x}_n)$, ranging over \mathbb{R}^n , denotes the derivatives of x_1, \dots, x_n . The variable s' and the tuple $\mathbf{x}' = (x'_1, \dots, x'_n)$, ranging over S and $I_1 \times \dots \times I_n$, respectively, denote the targets of jumps. Constraints are arbitrary Boolean combinations of equalities and inequalities over terms that may contain function symbols, like $+$, \times , \exp , \sin , and \cos . Based on this, the flows, jumps, initial and unsafe states of a hybrid system are given by constraints $\text{Flow}(s, \mathbf{x}, \dot{\mathbf{x}})$, $\text{Jump}(s, \mathbf{x}, s', \mathbf{x}')$, $\text{Init}(s, \mathbf{x})$ and $\text{Unsafe}(s, \mathbf{x})$, respectively.

Usually [7], abstractions are defined so that for every concrete behaviour, there is a corresponding abstract behaviour. The rationale is that if the abstract system is error-free, then so is the concrete system. However, for this very rationale, all that matters is that each *error* behaviour is mapped to some abstract error behaviour, while not all *correct* behaviours need to be captured.

Definition 3. Given a hybrid system H , let A be a directed graph whose nodes (the abstract states) are subsets of the state-space Φ . Some nodes are marked as initial, and some as unsafe.

For a mode/flow pair (s, r) of length l , an abstraction is a path $a_1, \dots, a_{\bar{k}}$ in A such that there exist $0 = \ell_0 \leq \ell_1 \leq \dots \leq \ell_{\bar{k}} = l$ such that for every $t \in [0, l]$ where $\ell_{j-1} \leq t \leq \ell_j$, it holds that $(s, r(t)) \in a_j$.

For an error trajectory $(s_0, r_0), \dots, (s_k, r_k)$ with flow lengths l_0, \dots, l_k , an abstract error trajectory is a path $a_{1,1}, \dots, a_{1,\bar{k}_1}, \dots, a_{k,1}, \dots, a_{k,\bar{k}_k}$ in A such that $a_{1,1}$ is initial, a_{k,\bar{k}_k} is unsafe, and for every $i \in \{0, \dots, k\}$, we have that $a_{i,1}, \dots, a_{i,\bar{k}_i}$ is an abstraction of (s_i, r_i) .

We call the directed graph A an abstraction of H iff, for each concrete error trajectory, there is an abstract error trajectory.

Abstractions can be useful for falsification because the abstract error trajectories narrow down the search space for concrete error trajectories. There are several methods available for computing such abstractions [1, 6]. We use a technique where each abstract state is a mode paired with a hyper-rectangle (*box*) $\subseteq I_1 \times \dots \times I_n$, as implemented in the tool HSOLVER [19]. In HSOLVER, an abstraction that is not fine enough yet to verify the desired property is refined by *splitting* a box (usually the biggest) in half. It does not seem hard to adapt our method to other kinds of abstraction.

Note that we do *not* assume that abstractions cover the whole state space (or reach set) with abstract states, but they do cover the set of all points lying on an error trajectory. In fact, one of the main features of HSOLVER is that it removes points from the abstraction for which it can prove that they cannot lie on an error trajectory. We call this *pruning*. Another kind of pruning is to use the underlying constraint solver to remove points from abstract states that do not fulfil a given (e.g., initial) constraint.

A *simulation* is an explicitly constructed sequence of points in Φ corresponding to the points of a trajectory at discrete moments in time. The distance between these moments is called *step size* (Δ). We do not give a precise definition here, as our search algorithm is independent of the concrete method for doing simulations (see Sec. 6). In this paper, we neglect aspects of imprecision of simulation methods.

Unlike actual error trajectories, a simulation might a-priori leave the abstraction, due to the fact that our abstraction covers, in general, a set that is smaller than the reach set, and also because the simulation might even leave the statespace, in which case it a-fortiori leaves the abstraction.

3 The Search Algorithm

3.1 The Problem and a Naïve Solution

We have a hybrid system with possibly several modes, and for each mode, a bounded statespace ($[l_1, u_1] \times \dots \times [l_n, u_n]$). We want to find an error trajectory, i.e., a trajectory leading from an initial state to an unsafe state.

We focus on systems that are deterministic in two senses: in the *continuous* sense (the flow is described by differential equations, not inequalities) and in the *discrete* sense (the jumps occur deterministically). Hence the problem reduces to determining the startpoint of an error trajectory among the initial states.

In practice, trajectories are *constructed* by running a *simulation*. Since our hybrid systems are deterministic, the only decision to take about a running simulation is when to cancel it.

To understand the problem, it is helpful to give a naïve solution, obtained by running simulations exhaustively. We use $grid(\Phi, w)$ to denote a set of grids of width w (one for each mode) consisting of points in the statespace, and $simulate(p, l)$ to denote a procedure which starts a simulation in p for l steps and returns *true* iff this simulation is an error trajectory, i.e., it starts in an initial state, reaches an unsafe state, and never leaves the statespace in between.

```

procedure find_startpoint
   $w := 1.0; l := 100$  /*ad-hoc*/
  while true
    foreach  $p \in grid(\Phi, w)$ 
      if  $simulate(p, l)$  return  $p$ 
     $w := w/2; l := l * 2$ 

```

Fig. 1: Naïve solution

From this naïve solution, it is clear that we have a search problem whose search space consists of two components. On the one hand, we search in Φ for a startpoint, on the other hand, we search in \mathbb{N} for determining a sufficient simulation length that will actually produce an error trajectory.

Unfortunately, running simulations is expensive, and hence we should try to avoid unnecessary simulation steps. The naïve procedure simulates unnecessarily on three different levels of granularity, leading to three aims of our work:

- If the system is safe, the procedure will run forever, although one might be able to prove safety quickly—our aim is thus to interleave verification with falsification attempts so that we can prove safety or unsafety, as applies.

- The procedure will run simulations evenly distributed on the whole statespace, even if some parts look more promising than others—our aim is thus to give preference to the more promising simulations.
- Each individual simulation will run for a pre-determined amount of time, ignoring the information gained during the simulation run—our aim is thus to cancel simulations when they do not look promising enough anymore.

To address these three aims we view the falsification problem as the problem of searching for an error trajectory, where the search procedure tries to exploit the information available from verification. The search procedure uses a quality estimate for simulations in order to determine which startpoints are the most promising, and when to cancel a simulation. The main features of the quality estimate are the following:

1. The estimate should measure the *relative* closeness of a simulation to representing an error trajectory, i.e., if simulation A gets a better estimate than simulation B , then A should be closer to being an error trajectory than B .
2. The faithfulness of the estimate should improve as the abstraction is refined.
3. Computation of the quality estimate should be on-the-fly, i.e., for each simulation step, the quality estimate of the simulation up to that point should be available (this is important for deciding when to cancel a simulation).
4. The overhead of computing the quality estimate should be low.

Our approach can be understood without knowing the precise definition of the quality estimate, and thus we will have three subsections addressing the above aims in turn. The corresponding algorithm is summarised in Fig. 2.

3.2 Interaction with Verification

Recall from Def. 3 that an error trajectory can only start within an *initial* abstract state. Hence we only search for error trajectories in these abstract states.

Now we must decide when to start such a search and for how long to run it, i.e., we have to strike a balance between verification and falsification attempts. Secondly, we have to strike the balance between *exploitation* (searching in regions that looked promising so far) and *exploration* (searching everywhere) [22].

Our design decision for striking those balances is to call the falsification algorithm after a refinement whenever an initial abstract state has been split or pruned, i.e., to keep running the verification while this is not the case (line 6). For the first balance, the rationale is that refinements of the abstraction that affect an initial state are likely to actually affect, i.e., improve, the quality estimate for simulations starting in this initial state. For the second balance, the rationale is that every initial state will have its turn to be affected by an abstraction refinement, so that that part of the search space will be explored.

Note that as the boxes converge towards size 0, we ensure completeness of our search procedure just like using the naïve procedure of Sec. 3.1. Why a refined abstraction improves the quality estimate will be explained in Sec. 5.1.

Just like the verification procedure “decides” when to pass the baton to falsification, the falsification procedure reciprocates (see Sec. 3.3).

```

1:  $A :=$  initial abstraction /*Initialisation*/
2: foreach  $B \in A$ 
3:    $B.crossmid := mid(strongbox(B)); B.crosstips := makecross(strongbox(B))$ 
4: while true
5:    $\mathcal{B} := \emptyset$  /*Verification part*/
6:   while  $\mathcal{B} = \emptyset$ 
7:     refine and prune  $A$ ;  $\mathcal{B} :=$  set of changed initial boxes of  $A$ 
8:     if  $\exists$  errorpath in  $A$  then output SAFE; exit
9:      $moves := 0; shrinks := 0$  /*Falsification part*/
10:    choose  $B \in \mathcal{B}$  with  $qual(sim(B.crossmid, A))$  maximal
11:    if  $B.crossmid \notin strongbox(B)$  or
12:       $qual(sim(mid(strongbox(B)), A)) > qual(sim(B.crossmid, A))$ 
13:       $B.crossmid := mid(strongbox(B)); B.crosstips := makecross(strongbox(B))$ 
14:    while  $moves \leq cros\_chg$  and  $shrinks \leq cros\_chg$ 
15:      choose  $p \in B.crosstips$  with  $qual(sim(p, A))$  maximal
16:      if  $qual(sim(p, A)) > qual(sim(B.crossmid, A))$ 
17:         $B.crossmid := p$ 
18:         $B.crosstips := shiftcross(B.crossmid, B.crosstips)$ 
19:         $moves := moves + 1$ 
20:      else
21:         $B.crosstips := halvecross(B.crosstips)$ 
22:         $shrinks := shrinks + 1$ 

```

Fig. 2. Overview of our algorithm

3.3 Doing the Right Simulations

We only start simulations in points in initial abstract states. But we can prune the candidate startpoints even more. For an initial abstract state (box), we compute a sub-box by pruning the parts for which we can show that they contain no initial points. We call this sub-box *strong* initial box. It can be much smaller than the entire initial box, leading to a vast reduction of the search space.

Merely relying on the strong initial boxes to become small enough to find a startpoint of an error trajectory is likely to be extremely inefficient—it is crucial to attempt to find a good simulation within such a box *quickly*.

Essentially, we understand the search problem of doing the right simulations as a numerical optimisation problem, where the objective function to be optimised is the quality estimate. We use so-called direct search methods [13], specifically the *compass method*.

The compass method guarantees that one finds a *local* optimum of continuously differentiable functions with Lipschitz continuous gradient [13, Theorem 3.11]. However, it also works well in practice for non-differentiable or even discontinuous functions [13, Section 6] which is the main reason for its usefulness in our context.

The method can be explained using the metaphor of searching a geographical landmark using maps that are initially coarse and then successively become finer.

We do this by taking a strong initial box B and considering an n -dimensional cross that fits exactly into B . That is, if the midpoint of B is (s_1, \dots, s_n) and

the size of B is $(2d_1, \dots, 2d_n)$, then we have $1 + 2n$ points $(s_1, \dots, s_n), (s_1 - d_1, s_2, \dots, s_n), \dots, (s_1, \dots, s_{n-1}, s_n + d_n)$. For each point, we start a simulation and compute a quality estimate f . If f has an optimum in some point p other than (s_1, \dots, s_n) , we move the cross to p and continue. If the optimum is in (s_1, \dots, s_n) , we halve the size of the cross and continue. Note that the number of simulation points of a given cross in the compass method is *linear* in the dimension n . The actual compass method is shown in lines 14-22.

However, we do not run the compass method for each modified initial box, but rather, we only consider the most promising box (line 10).

The compass method terminates when either the number of cross shrinkings or of cross moves has exceeded the threshold *cros_chg* (see Sec. 6). The current cross midpoint and cross size are remembered. When the falsification is later resumed, if the cross midpoint is still contained in the *modified* strong initial box B (see Sec. 3.2), and its quality is still higher than the quality of a simulation at the midpoint of B , then the search is continued using this cross. Otherwise, we assume that the quality estimate has changed considerably, and hence the search is restarted with a cross that fits exactly into B (lines 11-13). Note that it is assumed that the function *sim* will output an error trajectory if it finds one and exit the entire computation.

3.4 Doing Simulations Right

Since—apart from the set of initial states—our hybrid systems are completely deterministic, the only choice to be taken during a simulation is when to cancel it. Intuitively, we cancel simulations that are not improving sufficiently quickly. In detail, we cancel a simulation if one of the following situations occur:

- an unsafe state is hit, or
- the simulation has run outside of the abstraction for more than *sim_cnc* (a constant) steps, or
- the global quality estimate has not improved during the last *sim_cnc* steps, the local quality estimate has not improved in the very last step, and the simulation is currently within the abstraction. The notions “global” and “local” will become clearer in Sec. 4.

Note that any cancelling incurs the risk that a simulation might not run long enough to prove that it could actually be a good simulation. This risk is countered by the fact that our abstraction is refined over time, as explained in Sec. 5.1.

4 The Definition of the Quality Estimate

Slightly simplifying, the quality of a simulation consisting of points p_0, \dots, p_q is defined by

$$ini_wgh * isInit(p_0) + \max_i \left\{ -scaledDist(p_i) \cdot \frac{i}{i - distAbstr(p_0, \dots, p_i)} \right\} \quad (1)$$

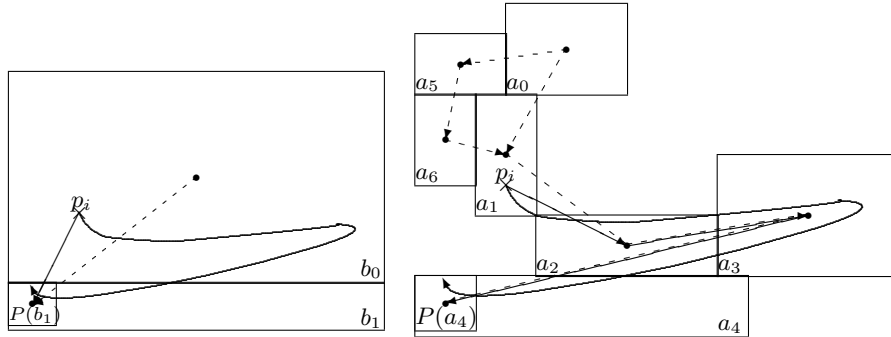


Fig. 3. Illustrating the distance estimate

In the rest of this section we explain this formula.

The most basic aspect of a simulation being close to an error trajectory is whether it actually starts in an initial state. We reward a simulation that does so with a constant *ini_wgh*. One might argue that starting a simulation in a non-initial state is a waste since the simulation will definitely not be an error trajectory. However, such a simulation can still be close to an error trajectory, since, of course, a non-initial point can be close to an initial point.

The second aspect is how close the simulation *eventually* gets to an unsafe state. We compute the closeness of all individual simulation points to an unsafe state, and take the optimum of these (see the max in (1)). Note that this optimum can be easily computed on-the-fly.

We now turn to the individual points, i.e., the expression inside of the maximisation, which we may refer to as *local* quality of point p_i , whereas the overall formula defines the *global* quality. The ideal measure for the local quality of p_i would be the negation of the length of the trajectory from p_i to some unsafe state, $-\infty$ if the trajectory from this point never reaches an unsafe state. This is illustrated in Fig. 3, r.h.s. The curve shows the trajectory starting from p_i , and we assume that it ends in an unsafe state. However, it is the very effort of computing this curve that we want to avoid. Therefore, we approximate this ideal measure by taking the length of a certain line segment sequence, based on information from the abstraction.

As explained in Sec. 2, an abstraction is a directed graph, and in our particular case the nodes of this graph are mode/box pairs (in the sequel, we speak of boxes and assume that the mode is clear from the context). Therefore, we shall use a geometrical rendering of this graph as an approximation of concrete trajectories, by taking the line segments between the midpoints of boxes within the same mode, for any abstract states that are connected in the graph. This is again illustrated in Fig. 3, r.h.s. Here a_0 is an initial abstract state and a_4 is an unsafe abstract state, and $P(a_4)$ is the *strong* unsafe box corresponding to a_4 , defined in analogy to the strong initial boxes explained in Sec. 3.2. The dashed lines are the line segments between connected abstract states. For the point p_i , the

estimated distance is the length of the solid line segment sequence, which partly coincides with the dashed line segments, namely from a_2 to $P(a_4)$. Note that the sequence resembles the actual trajectory, the curve. For a coarser abstraction, there will be no or little such resemblance, see l.h.s. figure and Sec. 5.1.

We will now explain this formally. For any box a , we denote by $M(a)$ the midpoint of a , and by J_a the maximal distance between any two points in a , i.e., $\sqrt{\sum_{i=1}^n d_i^2}$, where d_1, \dots, d_n are the sidelengths of a . For two points p, p' , we denote by $|p - p'|$ the Euclidean distance between p and p' .

For a moment, let us leave aside the fact that we are looking at a particular point p_i , and just consider the abstraction. Using a graph algorithm, we compute the shortest abstract error trajectories using the edge weights $w(a, a') =$

$$\begin{cases} |M(a) - M(a')| & \text{if } a \text{ and } a' \text{ lie in the same mode and are connected} \\ & \text{by an abstract transition;} \\ J_{a'} & \text{if } a \text{ and } a' \text{ are connected by a jump;} \\ \infty & \text{otherwise.} \end{cases}$$

Stated briefly, the rationale for choosing $J_{a'}$ as edge weight above is that $J_{a'}$ estimates the length of a trajectory segment within a' , making the ‘‘pessimistic’’ assumption that the trajectory goes from one corner to the opposite corner.

Above, we have said that we are interested in the distance of p_i to ‘‘some’’ unsafe state. In order to use an approximation of the set of unsafe states that is as tight as possible, we use the *strong* unsafe box of each unsafe state here. In analogy, for abstract states for which the next element in the shortest path has a different mode, the trajectory has to do a jump, and so we compute a subset containing all the points from which a jump might start. For any abstract state a as just said, we denote this (possibly non-proper) subset by $P(a)$. For other abstract states, P is simply the identity, to simplify the notation.

Now we reconsider the point p_i . We determine the abstract state a_1 that contains the point p_i , provided such an abstract state exists (the case that it does not exist will be considered later). Since our abstraction only contains states that lie on an abstract error trajectory (see Def. 3) there must be an abstract trajectory from a_1 to an unsafe abstract state. Letting $a_1, \dots, a_{k'}$ be the shortest one, we define the distance $dist(p_i)$ as follows:

If $k' \geq 2$ and a_1, a_2 have the same mode, then we define $dist(p_i)$ as $|p_i - M(P(a_2))| + \sum_{j=2, \dots, k'-1} w(P(a_j), P(a_{j+1}))$.

Otherwise, we either have $k' = 1$ (i.e., a_1 is an unsafe box), or a_1 is a jump source box. In this case, we would like to compute the distance of p_i to $P(a_1)$, call it δ . But what exactly do we mean by the distance from a point to a box? The answer is illustrated by the figure to the right: the boundary of $P(a_1)$ is drawn with thick lines; for the midpoint we have $\delta = 0$, and each rectangle (possibly with rounded corners) contains points with identical δ .

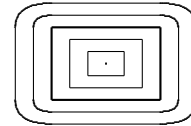


Fig. 4: Level sets

Given that $P(a_1)$ has sidelengths d_1, \dots, d_n , we formally define δ as follows: If p_i is inside of $P(a_1)$, then $\delta := \max\{|\frac{x_1}{d_1}|, \dots, |\frac{x_n}{d_n}|\} \frac{d_1 + \dots + d_n}{n}$, where x_1, \dots, x_n is the distance of p_i to the individual components of $M(P(a_1))$.

If p_i is outside of $P(a_1)$, then δ is defined as the Euclidean distance to the nearest point on the box boundary plus $\frac{d_1+\dots+d_n}{2n}$. The latter expression is the distance assigned to a point lying on the box boundary. Finally, we define $dist(p_i)$ as $\delta + \sum_{j=1, \dots, k'-1} w(P(a_j), P(a_{j+1}))$. Observe that the summation of $w(P(a_j), P(a_{j+1}))$ starts with $j = 1$, unlike in the previous paragraph, because δ only “covers” the distance to the jump source point within a_1 , whereas the expression $|p_i - M(P(a_2))|$ “covers” some of the way within a_2 .

In order to make the quality measure independent of the actual size of the state space, all distances are scaled to the interval $[0, 1]$ by dividing them by the length of the diagonal of the statespace in the corresponding mode. The result is denoted by $scaledDist(p_i)$, see (1).

We now consider the situation that a simulation contains points outside of the abstraction, which is possible, as explained in Sec. 2. A simulation that leaves the abstraction, or even the statespace, cannot be an error trajectory; but in analogy to simulations not starting in an initial state, it can still be *close* to an error trajectory. Therefore, we penalise such simulations but we do not reject them altogether. We do this by weighting the quality estimate for each simulation point according to the proportion of simulation points having lied outside of the abstraction up to that point (see the term $\frac{i}{i - distAbstr(p_0, \dots, p_i)}$ in (1)). Simulation points that lie outside of the abstraction themselves receive distance ∞ , so they do not have any influence on the overall quality of the simulation. The degree to which a simulation leaves the abstraction is thus a third aspect of a simulation being close to an error trajectory.

Why do we use the *shortest* abstract error path to estimate how far a point is from an error state? In fact, it might happen that some or *the* actual error trajectory follows some longer abstract error path. However, the probability that we are able to find an error trajectory in short time is highest in the case where this error trajectory is *short*. Hence we try to aim our search at areas likely to contain such a short error trajectory.

5 Analysis of our Method

5.1 Discussion of the Quality Estimate

We now discuss the four features listed in Sec. 3.1. Concerning the first two, we also have some formal results, see Sec. 5.2.

Concerning the first feature, the evidence, besides the fact that the quality measure was designed with this feature as foremost feature in mind, is in the successful experiments in Sec. 6.

Now consider the second feature. As explained in Sec. 3.4, any cancelling incurs the risk that a simulation might not run long enough to prove that it could actually be a good simulation. In fact, if the flow is such that from a point p it first moves away from the unsafe states and then approaches them, then simply using Euclidean distance for quality measurement would wrongly suggest that the simulation starting in point p is deteriorating at the beginning.

This is illustrated in Fig. 3, r.h.s.: if the simulation from point p_i stays very close to the solid line in its first steps, it actually moves away from $P(a_4)$. However, we also see that the abstraction shown is fine enough, so that the quality will increase during the first steps, i.e., the quality function is sufficiently faithful to recognise that the simulation is really improving. This is in contrast to the l.h.s. figure, where the abstraction is coarse. Note that the refinement has two effects:

1. The good simulations are more likely to run longer than the bad ones.
2. All good simulations will run longer than on previous tries.

The first effect will help the first component of the search (see Sec. 3.1): finding the right startpoint. The second effect will help the second component of the search: making simulations run long enough.

Concerning the third feature, the fact that the measure is computed on-the-fly is clear from the construction.

For the fourth feature, it turns out that although computing $dist(p)$ is costly (it involves a shortest path computation on the abstraction graph etc.), this cost is amortised because the abstraction remains constant at least throughout the current simulation. Thus, computing the quality estimate does not increase the complexity order of the simulation computation.

5.2 Formal Analysis

In this section, we will first formally prove that our definition of quality estimate fulfils a formalisation of the first two desired features of Section 5.1. Based on this, we will then prove that our algorithm finds all error trajectories that are robust in a certain, yet to be defined sense. All formal results in this section depend on the assumption that we do our simulations with enough precision concerning floating-point computation and time discretisation. The actual proofs can be found in the technical report [20].

We assume that we can compute arbitrarily precise abstractions:

Definition 4. *A sequence of abstractions A_1, A_2, \dots is convergent iff for every trajectory that is not an error trajectory there is a k such that for all $i \geq k$ there is no corresponding trajectory in A_i .*

Now we formalise what it means for a quality estimate to become arbitrarily precise:

Definition 5. *A sequence of functions f_1, f_2, \dots in $\Phi \rightarrow \mathbb{R}$ is convergent iff for two points p and q on the same error trajectory h such that p occurs earlier than q on h , there is a k such that for all $i \geq k$, $f_i(q) < f_i(p)$.*

We now prove Item 2 of our desired features. We denote by $dist_A$ the distance function (see Sec. 4) based on abstraction A .

Theorem 1. *Let A_1, A_2, \dots be a convergent sequence of abstractions. Then the sequence $dist_{A_1}, dist_{A_2}, \dots$, is convergent.*

Now call an error trajectory h *robust* iff there is an $\varepsilon > 0$ such all trajectories starting with a distance smaller than ε from h is also an error trajectory. We call a hybrid system that has a robust error trajectory *robustly unsafe*.

Theorem 2. *Our falsification algorithm finds an error trajectory for every robustly unsafe hybrid system H .*

Note that the above are theoretical completeness results: we will eventually find every error trajectory thanks to the fact that our abstractions will eventually be extremely precise. In practice, relying on this alone is extremely inefficient, just like the naïve algorithm, for which the same completeness result also holds. Hence, the theorems should be interpreted in the sense of: “Although our method cancels simulations whenever the abstraction suggests no further improvement, the method is still complete”.

6 Implementation and Experiments

We implemented our method and tested it on some well-known benchmarks.

In our prototype we use a simple Euler method for solving ordinary differential equations (e.g. [21]) with only naïve handling of jumps. In practice, more sophisticated ODE-solvers and precise jump detection [16] could be used. Due to re-use of HSOLVER (i.e., verification) code, this prototypical implementation runs quite slowly (3 orders of magnitude slower than hard-coded C simulation) but serves as an experimentation platform.

We now discuss the choices of the implementation parameters. The optimal choice of Δ depends on the speed of the behaviour of each individual example. For simplicity, we took $\Delta = 0.01$ which worked well for most examples.

We set *cross_chg* = 2, which is much smaller than what we intuitively expected to be reasonable, but we found that for bigger values, the compass method will get trapped in local minima of a poor quality estimate.

We set *sim_cnc* = 200, which seems rather small to us, and yet, to demonstrate that simulations eventually “survive” thanks to the faithfulness of the quality estimate, rather than a generously chosen value of *sim_cnc*, we set *sim_cnc* much smaller for some experiments reported below.

We set *ini_wgh* = 0.5, which roughly means that whether a simulation starts in an initial state is as important as the other aspects mentioned in Sec. 4.

For the experiments, we used a machine with two Intel Xeon processors running at 3.02 GHz with 6 GB RAM.

Our benchmarks were obtained by modifications of various well-known benchmarks from the literature, see <http://hsolver.sourceforge.net/benchmarks/falsification>. The modifications were necessary because the benchmarks were mostly safe, and so we injected an error into those systems by relaxing some constraints describing the initial or unsafe states or the jump guards.

We have also run some experiments on safe systems, to evaluate the cost of falsification attempts. HSOLVER ran between comparably fast and up to an order

of magnitude slower when run in the mode where falsification and verification attempts are interleaved. For space reasons we do not give any figures [20].

Table 1 shows the results for the unsafe examples: the runtime in seconds, the number of abstraction refinements, simulations, the total number of single simulation steps, and the number of jumps of the trajectory that was found. We give figures for our algorithm and the naïve algorithm of Sec. 3.1 (as will be discussed in the next section, all related work assumes systems with inputs and behaves similar to the naïve algorithm in our case without inputs). We consider the main figure for evaluating efficiency to be the number of simulation steps, since this number is independent of the actual implementation of the method.

The naïve algorithm performs very well on some apparently easy examples, where the method we propose here also performs well, but on numerous examples it does not terminate within several hours, indicated by ∞ . For hard examples, using a more sophisticated method such as ours is absolutely crucial, while for easy examples, one might easily hit an error trajectory by chance.

One observation when doing the modifications was that for some benchmarks, relaxing these constraints to some extent still resulted in a safe system. In fact, ideally what happens when one gradually relaxes a safe system is that it gradually transcends from “easy to prove safe” to “hard to prove safe” to “impossible to prove either way” to “hard to prove unsafe” to “easy to prove unsafe”. This is the case e.g. for **2-tanks**, and *partly* for **real-eigen** (see Table 1, where **real-eigen5** is the hardest and **real-eigen** is the easiest). However, we found numerous exceptions from this ideal, where some of the changes are very abrupt or not monotonic: **clock**, **convoi**, **real-eigen**, **van-der-pole2**.

Note that we have several examples where an error trajectory containing one or two jumps is found. For **eco**, we verified that these jumps are necessary, i.e., when we remove the jumps, the system becomes safe. This indicates that our quality estimate works reasonably well even for simulations that contain a jump.

We did an experiment with **focus** showing that even for a too small value of *sim_cnc*, simulations will eventually “survive” long enough thanks to the refinement of the quality function. The example is extremely easy for HSOLVER, provided *sim_cnc* is not too small. For *sim_cnc* = 20, an error trajectory is found but after 434 refinements. In this experiment, the startpoint found eventually is tried dozens of times before, but each time the simulation is cancelled prematurely. The same effect occurred for **eco** and **eco2**.

We have also created an example where we isolate the aspect just mentioned: **parabola**. In this example, the flow is $y = 20x^2$, and the initial and unsafe states are small boxes around the points $(-1, 20)$ and $(1, 20)$, respectively. That is, the error trajectory looked for is an extremely tight parabola. The search for the right startpoint is trivial; the problem is though that if *sim_cnc* is too small and the quality function is not faithful enough yet, then the simulations will be cancelled prematurely. This can be seen in the table where we tried values for *sim_cnc* ranging from 30 to 105.

For **mutant**, choosing $\Delta = 0.01$ is inappropriate, because 0.01 is minute relative to the state space size. We therefore chose $\Delta = 0.5$.

Example	our algorithm					naïve algorithm		
	time	ref.	sim.	sim. steps	jumps	time	sim.	sim. steps
2-tanks	11.5	7	130	23943	0	230.7	2372	554303
car	0.5	0	6	1033	1	0.6	3	272
clock	0.3	0	21	4387	0	4.3	175	59264
convoi	0.04	0	1	7	0	∞	∞	∞
eco <i>sim_cnc</i> =400	0.1	0	1	328	2	0.1	1	327
eco	2.1	10	63	21154	2	0.1	1	313
eco2 <i>sim_cnc</i> =400	0.1	0	1	328	2	0.1	1	327
eco2	45.3	152	422	118862	2	0.1	1	313
focus	0.1	0	10	2626	0	0.04	1	131
focus <i>sim_cnc</i> =20	29.7	434	288	13218	0	0.04	1	131
mutant $\Delta=0.5$	196.7	6	150	1421803	0	∞	∞	∞
navigation	1.6	0	22	5454	1	2.9	3	241
navigation2	1937.7	14	506	138206	1	∞	∞	∞
parabola <i>sim_cnc</i> =105	0.0	0	1	201	0	∞	∞	∞
parabola <i>sim_cnc</i> =100	0.3	4	43	4443	0	∞	∞	∞
parabola <i>sim_cnc</i> =50	1.0	35	71	4751	0	∞	∞	∞
parabola <i>sim_cnc</i> =30	18.0	353	113	7495	0	∞	∞	∞
real-eigen	0.7	1	44	8523	0	∞	∞	∞
real-eigen2	2.5	4	126	24165	0	∞	∞	∞
real-eigen3	4.5	10	214	41853	0	∞	∞	∞
real-eigen4	58.1	87	816	166450	0	∞	∞	∞
real-eigen5	250.8	314	1521	312567	0	∞	∞	∞
van-der-pole	0.4	1	36	3725	1	0.2	1	35
van-der-pole2	1.7	3	88	14546	1	∞	∞	∞

Table 1. Unsafe Systems

7 Related Work

Our work has some resemblance with heuristic search in artificial intelligence (AI), namely with *pure optimisation* problems, where the aim is to find a node in a graph which is good or optimal according to some *objective function*. One may also introduce such an objective function just for providing guidance in search algorithms. This is similar to our approach. It is distinctive of our work that the objective *function* itself improves over time. Our search method, the compass method, is similar to *local search* methods in AI.

In contrast to heuristic search in AI, we do not decide *whether* to do a simulation depending on the cheaply pre-computed quality of that simulation, but rather, we compute the quality as we do the simulation, and depending on this quality we will do other simulations in the neighbourhood. This is similar to reinforcement learning [22].

Methods that directly try to falsify hybrid systems (in contrast to using simulation for verification, as discussed below) usually consider hybrid systems with inputs, searching for inputs that drive the system from an initial to an unsafe state. One major approach in this direction is to adapt techniques from robotic motion planning [3, 18] to compute an under-approximation of the set of trajecto-

ries of a given hybrid system. Another approach studies how to avoid redundant simulations as much as possible by merging similar parts of simulations [14].

Although these methods were designed for systems with input, it is possible to apply them to systems without input (i.e., with deterministic evolution). However, their strategy is to try to fill the state space as much as possible with simulations. As a result, they would start a huge number of simulations in parallel (similar to our naïve algorithm from Sec. 3.1). In the case of highly non-deterministic systems, such a strategy is promising since the probability of hitting upon an error trajectory is high. However, for systems with little or no non-determinism, this creates many useless simulations. We avoid this by guiding our search using abstractions in order to quickly arrive at a simulation close to an error trajectory.

In software model checking, the synergy between verification and falsification (i.e., testing, debugging) is the subject of a lot of recent research, see for example, Gulavani [11] and the references therein. Also, the idea to use abstraction to define a heuristic function for local search has been studied in software model checking (e.g., [17]). In contrast to that, hybrid systems have a partially continuous state space with corresponding geometrical properties which we exploited in our search algorithm and our definition of the quality estimate.

Recently a new paradigm of *verification by simulation* has received attention [10, 9, 5]. The main goal is verification of a correct input system. Error trajectories may be computed as a by-product.

An alternative approach to the verification/falsification paradigm is to use test coverages [4, 12, 15], where—instead of trying to fully verify a property—one defines a function that measures how large a part of the hybrid system is covered by a given set of simulation runs. Then one tries to cover the state space with simulations in such way that this test coverage function is optimised which again contrasts our strategy of trying to find *one single error trajectory*.

8 Conclusion

We have presented a method for finding error trajectories of hybrid systems. We consider the main challenge for future work to be the exploitation of the *partially* continuous nature of hybrid systems and the fact that numerical analysis provides a myriad of optimisation algorithm for continuous functions.

References

1. R. Alur, T. Dang, and F. Ivančić. Predicate abstraction for reachability analysis of hybrid systems. *Trans. on Embedded Computing Sys.*, 5(1):152–199, 2006.
2. A. Bemporad, A. Bicchi, and G. Buttazzo, editors. *HSCC'07: 10th Int. Workshop on Hybrid Systems: Computation and Control*, volume 4416 of *LNCS*. Springer, 2007.
3. A. Bhatia and E. Frazzoli. Incremental search methods for reachability analysis of continuous and hybrid systems. In R. Alur and G. J. Pappas, editors, *HSCC'04*:

- 7th Int. Workshop on Hybrid Systems: Computation and Control*, volume 2993 of *LNCS*, pages 142–156. Springer, 2004.
4. A. Bhatia and E. Frazzoli. Sampling-based resolution-complete algorithms for safety falsification of linear systems. In M. Egerstedt and B. Mishra, editors, *HSCC*, volume 4981 of *LNCS*, pages 606–609. Springer, 2008.
 5. P. Cheng and V. Kuma. Sampling-based falsification and verification of controllers for continuous dynamic systems. *Int. J. of Robotics Research*, 27(11-12):1232–1245, 2008.
 6. E. Clarke, A. Fehnker, Z. Han, B. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald. Abstraction and counterexample-guided refinement in model checking of hybrid systems. *Int. J. of Foundations of Comp. Sci.*, 14(4):583–604, 2003.
 7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
 8. W. Damm and H. Hermanns, editors. *CAV'07: 19th Int. Conf. on Computer Aided Verification*, volume 4590 of *LNCS*, 2007.
 9. A. Donzé and O. Maler. Systematic simulation using sensitivity analysis. In Bemporad et al. [2], pages 174–189.
 10. A. Girard and G. Pappas. Verification using simulation. In J. Hespanha and A. Tiwari, editors, *HSCC'06: 9th Int. Workshop on Hybrid Systems: Computation and Control*, volume 3927 of *LNCS*, pages 272–286. Springer, 2006.
 11. B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYN-ERGY: a new algorithm for property checking. In M. Young and P. T. Devanbu, editors, *SIGSOFT '06/FSE-14: Proc. of the 14th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, pages 117–127. ACM, 2006.
 12. A. A. Julius, G. E. Fainekos, M. Anand, I. Lee, and G. J. Pappas. Robust test generation and coverage for hybrid systems. In Bemporad et al. [2], pages 329–242.
 13. T. G. Kolda, R. M. Lewis, and V. Torczon. Optimization by direct search: New perspectives on some classical and modern methods. *SIAM Review*, 45(3):385–482, 2003.
 14. F. Lerdia, J. Kapinski, H. Maka, E. Clarke, and B. Krogh. Model checking in-the-loop: Finding counterexamples by systematic simulation. In *American Control Conf.*, 2008.
 15. T. Nahhal and T. Dang. Test coverage for continuous and hybrid systems. In Damm and Hermanns [8], pages 449–462.
 16. T. Park and P. I. Barton. State event location in differential-algebraic models. *ACM Trans. Model. Comput. Simul.*, 6(2):137–165, 1996.
 17. F. M. D. Paula and A. J. Hu. An effective guidance strategy for abstraction-guided simulation. In *DAC '07: 44th Annual Conf. on Design Automation*, pages 63–68. ACM, 2007.
 18. E. Plaku, L. E. Kavvaki, and M. Y. Vardi. Hybrid systems: From verification to falsification. In Damm and Hermanns [8], pages 463–476.
 19. S. Ratschan and Z. She. Safety verification of hybrid systems by constraint propagation based abstraction refinement. *ACM Trans. in Embedded Computing Systems*, 6(1), 2007.
 20. S. Ratschan and J.-G. Smaus. Finding errors of hybrid systems by optimising an abstraction-based quality estimate. Technical Report 51, AVACS, <http://www.avacs.org>, 2009.
 21. G. Sewell. *The Numerical Solution of Ordinary and Partial Differential Equations*. Academic Press, 1988.
 22. R. S. Sutton and A. G. Barto. *Reinforcement Learning*. The MIT Press, 1998.