Finding Errors of Hybrid Systems by Optimising an Abstraction-Based Quality Estimate

Stefan Ratschan[1] and Jan-Georg Smaus[2]

[1] Academy of Sciences of the Czech Republic, stefan.ratschan@cs.cas.cz
[2] University of Freiburg, Germany, smaus@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 main 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. In contrast to related work, we consider hybrid systems with completely deterministic evolution.

Note: This AVACS report is the extended version of a conference paper [44]. It contains some proof sketches, more background material, more details on experimental results, and more discussions of related work.

1 Introduction

A hybrid system is a dynamical system with combined discrete and continuous state and evolution. Hybrid systems are an important formalism for modelling embedded systems. An important problem is to ensure correctness, i.e., *verification* [19, 20, 43]. However, during the design (debugging) process, hybrid systems are usually not correct yet, and hence it is important to be able to automatically find errors of hybrid systems.

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 [10, 37] 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.
- In a similar way as related work in program verification [23, 36], 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 in case the system is correct, and for falsification otherwise. 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

2.1 Hybrid Systems

Hybrid systems are systems with both continuous and discrete state. In this section, 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 (e.g., the one used by the tool PHAVer [19]) are special cases of it. A hybrid system has a finite and nonempty set S of *modes*. $I_1, \ldots, I_n \subseteq \mathbb{R}$ are compact intervals over which the n continuous variables of a hybrid system range. The state space of a hybrid system is denoted by $\Phi = S \times I_1 \times \cdots \times I_n$. Note that it is not a severe practical restriction that the continuous variables have to range over compact intervals because in most applications the variable ranges are bounded and engineers use their experience to choose reasonable values for the interval bounds.

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

Informally speaking, the predicate *Init* specifies the initial states of a hybrid system and *Unsafe* the set of unsafe states that should not be reachable from an initial state. 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 $r : [0, l] \to I_1 \times \cdots \times I_n$, differentiable on $[0, l]$. A* trajectory *of H is a sequence of mode/flow pairs $(s_0, r_0), \ldots, (s_k, r_k)$ of lengths $l_0, \ldots, l_k$ such that for all $i \in \{0, \ldots, k\}$,*

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 $(s_0, r_0), \ldots, (s_k, r_k)$ H such that $(s_0, r_0(0)) \in Init$ and $(s_k, r_k(l_k)) \in Unsafe$. H is* safe *if it does not have an error trajectory.*

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

Example 1. Consider a hybrid system with $\Phi = \{m_1, m_2\} \times [0,2] \times [0,1]$ and the following constraints:

$$\begin{array}{ll}
\mathit{Init}(s,(x_1,x_2)) & = (s = m_1 \wedge x_1 = 0 \wedge x_2 = 0) \\
\mathit{Unsafe}(s,(x_1,x_2)) & = (x_2 \geq 2) \\
\mathit{Jump}(s,(x_1,x_2),s',(x_1',x_2')) & = (s = m_1 \wedge x_2 \geq 1 \rightarrow s' = m_2 \wedge x_1' = x_1 \wedge x_2' = x_2) \\
\mathit{Flow}(s,(x_1,x_2),(\dot{x}_1,\dot{x}_2)) & = (s = m_1 \rightarrow \dot{x}_1 = 1 \wedge \dot{x}_2 = 1 \wedge 0 \leq x_1 \leq 1) \wedge \\
& \quad (s = m_2 \rightarrow \dot{x}_1 = 1 \wedge \dot{x}_2 = -1 \wedge 1 \leq x_1 \leq 2)
\end{array}$$

The hybrid system can switch modes from m_1 to m_2 if $x_2 \geq 1$. The continuous behaviour is very simple: In mode m_1, the values of x_1, x_2 change with slope 1; in m_2, variable x_1 has slope 1 and x_2 has slope -1. For a flow in m_1, the *invariant* $0 \leq x_1 \leq 1$ must hold and in m_2, the invariant $1 \leq x_1 \leq 2$ must hold. Note that the constraint $0 \leq x_1 \leq 1$ in *flow* forces a jump from mode m_1 to m_2 if x_1 becomes 1. In general, an invariant that has to hold in a mode can be modelled by formulating a flow constraint that does not allow a continuous behaviour in certain regions. A trajectory of the hybrid system starting from the initial state $(m_1, (0,0))$ is r_0, r_1, where the flows $r_1, r_2 : [0,1] \rightarrow \Phi$ are given by

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

It is not hard to see that this hybrid system is safe, since x_2 changes its slope from 1 to -1 and $x_2 = 1$ when $x_1 = 1$.

2.2 Abstractions and Simulations of Hybrid Systems

We assume that we have an algorithm available for computing abstractions for hybrid systems.

Usually [15], abstractions are defined so that for every concrete behaviour, there is a corresponding abstract behaviour (*over*approximation). 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*) consist of subsets of the state-space Φ. Some nodes are marked as initial (we call them* initial abstract states*), and some as unsafe (we call them* unsafe abstract states*).*

For a mode/flow pair (s, r) of length l, an abstraction *is a path $a_1, \ldots, a_{\tilde{k}}$ in A such that there exist $0 = \ell_0 \leq \ell_1 \leq \ldots \leq \ell_{\tilde{k}} = l$ 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 $(s_0, r_0), \ldots, (s_k, r_k)$ of H with corresponding flow lengths $l_0, \ldots, l_k$, an abstract error trajectory *is a path*

$$a_{1,1}, \ldots, a_{1,\tilde{k}_1}, \ldots, a_{k,1}, \ldots, a_{k,\tilde{k}_k}$$

in A such that $a_{1,1}$ is initial, $a_{k,\tilde{k}_k}$ is unsafe, and for every $i \in \{0, \ldots, k\}$, we have that $a_{i,1}, \ldots, a_{i,\tilde{k}_i}$ 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 [3, 14]. We use a technique where each abstract state is a mode paired with a hyper-rectangle (box) $\subseteq I_1 \times \cdots \times I_n$, as implemented in the tool HSOLVER [43].

HSOLVER was designed for verification—its use for falsification is novel to this work. Given an abstraction, interval constraint propagation [8] based on the auxiliary tool RSOLVER [41, 42] is used to compute an incrementally more precise abstraction. In this paper, we run HSOLVER in what we call "mixed mode", i.e., we modify HSOLVER in such a way that the usual verification is interleaved with falsification attempts.

Usually, each abstract state only contains elements of a single mode. 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. We have developed our falsification technique for such an abstraction, but it does not seem hard to adapt it 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. This is done whenever an over-approximation of all points fulfilling a certain constraint is desired, and we want this over-approximation to be as tight as possible. We will come back to this point later.

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). Note however that such methods usually are not completely precise: all methods commonly used (both in research and in industrial practice) for simulating sufficiently general hybrid systems are prone to errors due to time-discretisation, and due to floating-point rounding. We neglect this aspect here, i.e., we pretend the use of a simulation method that models actual trajectories precisely enough for the user's purposes.

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 ($[l_1, u_1] \times \ldots \times [l_n, u_n]$). The hybrid system has a set of initial states given by a mode paired with some constraints that specify a certain continuous subset of $\mathbb{R}^n$. 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.

In fact, to understand the problem, it is helpful to give a naïve solution, obtained by running simulations exhaustively. This is shown in Fig. 1. We use $grid(\Phi, w)$ to denote

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

Fig. 1. A naïve solution

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.

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 definition of the quality estimate is rather involved and thus the entire Sec. 4 is devoted to it. However, its main features are discussed in the next subsection.

3.2 Main Features of the Quality Estimate

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. However, we do not attempt to formalise what it means to be “closer to being an error trajectory”, since this has several aspects to it, and any weighting of these aspects is necessarily arbitrary. Ultimately, what counts is that the estimate provides good guidance for the search. However, in Sec. 5.2, we will formalise some criteria for good guidance.
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, i.e., the overall cost of running each simulation should not be dominated by the cost of computing the quality estimate.

Note that our entire approach aims at systems that are well-behaved in the following sense: the state space should be partitioned into a small number of regions where in each region, small changes in the startpoint result in small changes of the overall trajectory. Within such regions, the statement "A is close to being an error trajectory" will usually imply "there is an error trajectory close to A" with high probability, which can be exploited by the search algorithm.

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.

3.3 Interaction with Verification

Recall from Sec. 2.2 that the verification algorithm maintains an abstraction of the concrete system, and 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 the time devoted to verification and falsification attempts. Secondly, speaking in the terminology of reinforcement learning [50], we have to strike the balance between *exploitation* (searching in regions that looked promising so far) and *exploration* (searching everywhere, including in less promising regions).

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). Depending on whether the resulting box(es) contain(s) a point found by previous searches, the search resumes from this point or starts from scratch.

The rationale is twofold: concerning the first balance, the idea is that refinements of the abstraction that actually affect an initial state are likely to actually affect, i.e., improve, the quality estimate for simulations starting in this initial state. Concerning the second balance, the idea 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. 3.4).

3.4 Doing the Right Simulations

We have just explained that we only start simulations in points in initial abstract states. However, we can prune the candidate startpoints much more than that.[3]

For an initial abstract state (box), we compute a sub-box by pruning away those 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.

[3] Recall that when we introduced the concept of pruning above in Section 2.2, we explained that it can occur in two forms: First, points not on an error trajectory can be removed, and second, boxes can be pruned to smaller ones over-approximating a certain constraint. We will use the second form here.

```
$A :=$ initial abstraction /*Initialisation*/
**foreach** $B \in A$
  $B.crossmid := mid(strongbox(B))$; $B.crosstips := makecross(strongbox(B))$
**while true**
  $\mathcal{B} := \emptyset$ /*Verification part*/
  **while** $\mathcal{B} = \emptyset$
    refine and prune $A$; $\mathcal{B} :=$ set of changed initial boxes of $A$
    **if** $\nexists$ errorpath in $A$ then **output** SAFE; **exit**
  $moves := 0$; $shrinks := 0$ /*Falsification part*/
  **choose** $B \in \mathcal{B}$ **with** $qual(sim(B.crossmid, A))$ **maximal**
  **if** $B.crossmid \notin strongbox(B)$ **or**
        $qual(sim(mid(strongbox(B)), A)) > qual(sim(B.crossmid, A))$
    $B.crossmid := mid(strongbox(B))$; $B.crosstips := makecross(strongbox(B))$
  **while** $moves \leq cros\_chg$ **and** $shrinks \leq cros\_chg$
    **choose** $p \in B.crosstips$ **with** $qual(sim(p, A))$ **maximal**
    **if** $qual(sim(p, A)) > qual(sim(B.crossmid, A))$
      $B.crossmid := p$
      $B.crosstips := shiftcross(B.crossmid, B.crosstips)$
      $moves := moves + 1$
    **else**
      $B.crosstips := halvecross(B.crosstips)$
      $shrinks := shrinks + 1$
```

Fig. 2. Overview of our algorithm

Merely relying on the strong initial boxes to become sufficiently small to find some startpoint of an error trajectory, as we have explained in Sec. 3.3 above, 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 (to be defined in Sec. 4). The usual numerical methods exploit derivatives. However, due to the partially discrete nature of hybrid systems, derivatives are not generally available in our context, and therefore we use so-called direct search methods [28], specifically the *compass method*. The compass method guarantees that one finds a *local* optimum of continuously differentiable functions with Lipschitz continuous gradient [28, Theorem 3.11]. However, it also works well in practice for non-differentiable or even dis-continuous functions [28, Sec. 6] which is the main reason for their usefulness in our context.

Intuitively, this method can be explained using the metaphor of searching a certain geographical landmark using maps. At the beginning, we do not yet have any idea where the landmark might be. Hence we use a very coarse map to find the approximate area where the landmark lies. After that, we switch to finer and finer maps to locate the landmark more precisely.

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 $(s_1, \ldots, s_n)$ and the size of B is $(2d_1, \ldots, 2d_n)$, then we have $1+2n$ points $(s_1, \ldots, s_n)$, $(s_1 - d_1, s_2, \ldots, s_n)$, $(s_1 + d_1, s_2, \ldots, s_n), \ldots, (s_1, \ldots, s_{n-1}, s_n - d_n)$, $(s_1, \ldots, s_{n-1}, s_n + d_n)$. For each of these points, we start a simulation and compute a quality estimate f. If f attains an optimum in some point other than $(s_1, \ldots, s_n)$, we move the cross to this point and continue. If the optimum is attained in $(s_1, \ldots, s_n)$, we halve the size of the cross and continue. Note that in contrast to the exponential growth in n needed to sample the n-dimensional real space up to a certain maximal distance between neighbouring samples, the number of simulation points of a given cross in the compass method is *linear* in the problem dimension. 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. 3.3), and its quality is still higher than the quality of a simulation at the midpoint of B, then search is continued using this cross. Otherwise, we assume that the abstraction has changed so much that the optimum is not close to the cross midpoint anymore. Hence, 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.

3.5 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 $p_0, \ldots, p_n$ is defined by

$$\mathit{ini\_wgh} * \mathit{isInit}(p_0) + \max_i \{-\mathit{scaledDist}(p_i) \cdot \frac{i}{i - \mathit{distAbstr}(p_0, \ldots, p_i)}\} \qquad (1)$$

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, depending on the hybrid system, even just finding an initial state can be non-trivial, and moreover, such a simulation can be still be close to an error trajectory, since, of course, a non-initial point can be close to an initial point.

The second aspect of a simulation being close to an error trajectory is how close the simulation *eventually* gets to an unsafe state. Here we rely on systems being well-behaved as explained in in Sec. 3.2, i.e., if one simulation gets very close to an unsafe state, then we hope that nearby there will be a trajectory that actually hits the unsafe

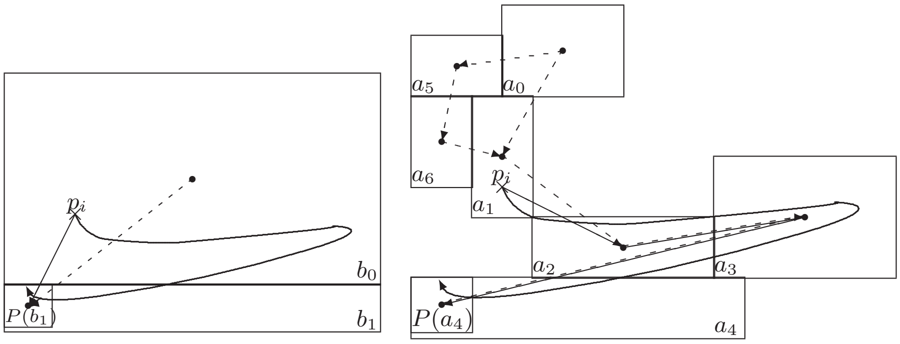

Fig. 3. Illustrating the distance estimate

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, in order to be able to compute it efficiently (and to be able to compute it at all). We do this by taking the length of a certain line segment sequence, based on information information from the abstraction.

As explained in Sec. 2.2, 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 the box belongs to is clear from the context). Therefore, we shall use a geometrical rendering of this graph as an approximation of concrete trajectories, namely, 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. 3.3. 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 the 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 $d_1, \ldots, d_n$ 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. This is indicated by the $P(a_4)$ in the figure. 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.

We replace all abstract states in the shortest path by these sets and adjust the weights accordingly, without however recomputing the shortest paths.

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 $a_1, \ldots, a_{k'}$ 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 $|p_i - M(P(a_2))| + \sum_{j=2,\ldots,k'-1} w(P(a_j), P(a_{j+1}))$.

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 $P(a_1)$. Let us call this distance 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. Given that $P(a_1)$ has sidelengths $d_1, \ldots, d_n$, we formally define δ as follows: If p_i is inside of $P(a_1)$, then $\delta := \max\{|\frac{x_1}{d_1}|, \ldots, |\frac{x_n}{d_n}|\}\frac{d_1+\cdots+d_n}{n}$, where $x_1, \ldots, x_n$ 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 $\frac{d_1+\cdots+d_n}{2n}$ The latter expression is the distance assigned to a point lying on the box boundary. Finally, we define $dist(p_i)$ as $\delta + \sum_{j=1,\ldots,k'-1} w(P(a_j), P(a_{j+1}))$. 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.

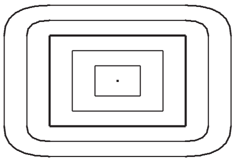

Fig. 4: Level sets

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. This is possible due to the fact that our abstraction covers, in general, a set that is smaller than the reach set (see the end of Sec. 2.2). 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 $\frac{i}{i - distAbstr(p_0,\ldots,p_i)}$ 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

In Sec. 3.2 we mentioned four desired features of our quality estimate. We now discuss these features. 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. 3.5, 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 "really" good simulations are more likely to run longer than the "really" bad ones.
2. In situations as illustrated in Fig. 3, where it requires a reasonably fine abstraction to realise that initially moving away from the unsafe states is "really" good, the effect of the refinement is that All "really" 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 eventually, ensuring that an error trajectory is not missed due to premature cancelling.

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

For the fourth feature, we now analyse the cost of computing the quality estimate. Clearly, the few basic arithmetic operations of (1) can increase the cost of running a simulation only by a fixed amount for each simulation point. However, computing the distance $dist(p)$ is costly, since it involves a shortest path computation on the abstraction graph, computation of subsets, and so on. Fortunately, the abstraction remains constant throughout a given simulation, and usually even throughout several simulations. Hence we can factor out this part and pre-compute it once and for all for a given abstraction. As a consequence, computing the quality estimate does not increase the complexity order of the simulation computation. It has to be said though that looking up, for each simulation point, the abstract state it is in, may incur a considerable overhead.

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 (as well as all related results in the literature) depend on the assumption that we do our simulations with enough precision concerning floating-point computation and time discretisation.

We rely on the assumption 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 $\Phi \to \mathbb{R}$ 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 $i \geq k$, $f_i(q) < f_i(p)$.*

Based on this, we can formally prove Item 2 of our desired features. In the following, 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.*

Proof. (*Sketch*) Let p and q be two arbitrary but fixed points on an error trajectory h, with p occurring earlier than q. We assume that for all $i \in \mathbb{N}$, $dist_{A_i}(q) \geq dist_{A_i}(p)$ and derive a contradiction.

Let $i \in \mathbb{N}$ be such that p and q are in different abstract states of A_i (this will be the case for every i that is large enough). Then $dist_{A_i}(p) \leq dist_{A_i}(q)$ implies that the shortest abstract path from p, call it $p\triangleright$, cannot lead over an abstract state containing q. However, since our hybrid systems are deterministic, there is only one unique trajectory leaving p—the error trajectory leading to q. Hence, due to convergence of the sequence of abstractions, and the definition of abstraction, there is a j such that $p\triangleright$ will not be in A_j, a contradiction. □

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.*

Proof. (*Sketch*) Our simulations are dense in the set of initial sets of error trajectories. Hence the falsification algorithm will eventually simulate in a distance smaller than ε from the robust error trajectory. Moreover, due to the convergence of *dist* the quality estimate will eventually monotonically improve on such a trajectory, and hence the simulation will not be cancelled before reaching an unsafe point. □

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 of Sec. 3.1, 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".

Example	ver.	mixed	ref.	sim.	sim. steps
`2-tanksSAFE`	5.0	45.4	63	626	130513
`2-tanksSAFE2`	10.4	115.8	111	1193	250198
`2-tanksSAFE3`	68.1	517.6	546	4815	1023456
`convoi-1SAFE`	0.1	0.1	0	0	0
`ecoSAFE`	88.6	108.3	224	511	257271
`s-focusSAFE`	3.0	4.0	9	119	21138

Table 1. Safe Systems

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 (ODEs, e.g. [47]) with only naïve handling of jumps. In practice, more sophisticated ODE-solvers and precise jump detection [35] 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.

Concerning the impact of the implementation parameters, we have no systematic analysis but the choices are partly based on our experience, and we will discuss them now.

We set $\Delta = 0.01$, assuming that the continuous behaviour within a timespan of 0.01 can be described as a straight line with reasonable accuracy. This is the case for most benchmarks we discuss below, although on rare occasions Δ is either too big (so that the simulations will be grossly inaccurate) or too small (so that simulations will take prohibitively much time). In particular, this is the case for `mutant`, see Table 2.

We set $cros\_chg = 2$, 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. The original benchmarks can be found at `http://hsolver.sourceforge.net/benchmarks/benchmarks.html`, and the modified ones at `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.

Recall that HSolver is originally [43] a *verification* tool. Our method is implemented as a special running mode of HSolver where verification is interleaved with falsification. To give an idea of the cost of falsification attempts, we ran some experiments for a few safe systems. Table 1 shows the results: it compares the HSolver runtime in seconds for pure **ver**ification mode and **mixed** mode. Moreover, it shows the number of abstraction refinements, the number of simulations, and the total number of single simulation steps. We see that HSolver can run about an order of magnitude slower in mixed mode. This indicates that on examples suspected to be safe, it might be a good idea to adjust the parameters so that less time is devoted to falsification attempts.

Table 2 shows the results for the unsafe examples. In addition to Table 1, the table shows the number of jumps of the trajectory that was found. 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.

In addition to showing the figures for our new algorithm, Table 2 also shows the figures for the the naïve algorithm of Sec. 3.1 (as will be discussed in the next section, all related work assumes systems with inputs and behaves similarly to the naïve algorithm in our case without inputs). In our implementation of the naïve algorithm, l was initially chosen to be 100 as stated in Fig. 1. However, a simulation is started only in grid points that are initial. Thus it may happen that l is increased before the start of an actual simulation, which explains, for example, why we have 327 simulation steps for `eco` in spite of having just one simulation.

The naïve algorithm performs very well on some apparently easy examples, where the method we propose here also performs well, but on numerous examples the naïve algorithm 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 the 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 2, 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. However, we have no way of knowing whether an improved method might find error trajectories with more jumps for harder examples, on which our current implementation exhausts the computational resources.

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 $\Delta = 0.5$.

	our algorithm					naïve algorithm		
Example	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 $sim\_cnc = 400$	0.1	0	1	328	2	0.1	1	327
eco	2.1	10	63	21154	2	0.1	1	313
eco2 $sim\_cnc = 400$	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 $sim\_cnc = 20$	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 $sim\_cnc = 105$	0.0	0	1	201	0	∞	∞	∞
parabola $sim\_cnc = 100$	0.3	4	43	4443	0	∞	∞	∞
parabola $sim\_cnc = 50$	1.0	35	71	4751	0	∞	∞	∞
parabola $sim\_cnc = 30$	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 2. Unsafe Systems

7 Related Work

Our work has some resemblance with heuristic search in artificial intelligence (AI), [45] 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, which move from state to state striving for improvements.

Just like for local search in the AI context, completeness of the search is a critical issue for us. The basic compass method (shrinking and moving the cross) runs the risk of getting stuck in a local optimum. We achieve completeness thanks to the fact that the abstraction will eventually become extremely fine. In a practical sense, we try to avoid getting stuck in a local optimum by frequently considering a restart (see lines 10-13 in Fig. 2).

In spite of the fact that we search for an error *trajectory* (=*path*), there is not much analogy between our approach and AI *path* search problems, because we do not use the objective function to decide "where to go next within this trajectory". There would be such an analogy if we looked at our search problem more generally, in particular, consider the problem of choosing the right jumps for non-deterministic systems. This has been done for *timed automata*, where it is an instance of *directed model checking* [48, 29]. Doing it in our context, i.e., for hybrid systems, is a topic for future work.

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 [50].

The traditional approach to falsifying safety of dynamical systems is manual or semi-automatic simulation using according tools [30, and references therein].

Work on methods that directly try to falsify hybrid systems (in contrast to work that uses simulation for verification, as discussed below) usually consider the case of 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 [10, 37] to compute an under-approximation of the set of trajectories of a given hybrid system. Another approach studies how to avoid redundant simulations as much as possible by merging similar parts of simulations [32].

Even though these methods were designed for systems with input, it is possible to apply them to systems without input (i.e., with deterministic evolution). However, the strategy of the above algorithms is to try to fill the state space as much as possible (according to some measure) with simulations. As a result, they would start a huge number of simulations in parallel—either from a grid (similar to our naïve algorithm from Sec. 3.1) or from random sample points. In the case of highly non-deterministic systems, such a strategy is promising since by filling the set of possible trajectories fast, the probability of hitting upon an error trajectory is high. However, for systems with only a small amount of non-determinism, and especially, completely deterministic evolution, this creates a huge number of useless simulations. We avoid this by guiding our search using abstractions in order to quickly arrive at a simulation close to an error trajectory, and to avoid simulating in areas of the state space for which we know that they do not lie on an error trajectory.

Tools for counterexample guided abstraction refinement [14, 2] based on flow pipe computation may terminate with a concrete counterexample. For a given counterexample, several simulation runs are tried according to a certain strategy.

Searching for counter-examples is also related to optimal control [9]. Work on optimal control for hybrid systems [12, 24, 7, 5, 49, 38] determines inputs that optimise a certain continuous objective function. In our work, instead of such an optimisation problem we consider a decision problem ("is/is not a counterexample") for systems without inputs. Moreover, work on optimal control usually does not take information from safety verification into account, while we tightly integrate verification and falsification.

Recently a new paradigm of *verification by simulation* has received attention [22, 17, 13, 31, 4]. For these methods (unlike in our case) 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 [11, 25, 34, 21, 46], where—instead of trying to verify a property fully—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 find simulation runs that optimise this test coverage function. In some of these methods [21, 46], model checking is used to guide simulation in order to increase coverage as fast as possible.

Such methods for verification by simulation or for computing test coverages have to cover the whole state space densely with simulations, where the interpretation of the word "densely" depends on the individual method. In contrast to that, we do *not* concentrate on covering the state space, but try to find *one single error trajectory* as fast as possible (similar to some methods in software model checking [23, 36], see the discussion below).

Prajna and Rantzer [40] show that—in analogy to Lyapunov functions—the existence of certain functions implies the reachability of given sets in (non-linear) ODEs. Such functions can be computed using sum-of-squares (SOS) tools [39].

For linear discrete-time systems with input, another approach [26] exhaustively simulates the system using a time and state discretisation of the input, and tries to avoid redundant computation by merging nearby trajectories.

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 [23] 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., [36]). 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 quality estimate.

8 Conclusion

We have presented a method for finding error trajectories of hybrid systems. The method combines ideas from AI heuristic search, abstraction-based verification, numerical optimisation, and reinforcement learning. We run simulations searching for one that corresponds to an error trajectory. For each simulation, the point at which it is cancelled depends on a quality estimate computed on-the-fly, using information from the HSolver abstraction. The neighbourhood in which new simulations are started is also determined using this quality estimate.

We have demonstrated on several well-known benchmarks that our method is able to find error trajectories. Various lessons learnt were explained in Sec. 6. What we consider the most important point is that the quality estimate based on the abstraction, which is a byproduct of the verification algorithm, becomes more faithful as the abstraction becomes refined. We would have liked to phrase this as “thanks to the improving quality estimate, it is a good idea to choose *sim_cnc* very small”, but based on our experiments we should say more modestly that “thanks to the improving quality estimate, we will eventually recover from a too small choice of *sim_cnc*”.

There are various directions for future work.

1. We consider the main challenge to be the exploitation of the *partially* continuous nature of hybrid systems, that is, the fact that sometimes (but not always) trajectories depend smoothly on their initial point. Numerical analysis provides a myriad of optimisation algorithm for differentiable functions. One might be able to adapt such algorithms to our case and use corresponding derivatives in the optimisation algorithm.
2. We would like to extend our approach to systems with non-deterministic evolution, in particular, non-deterministic jumps. In this case, the simulation will have to make decisions. Our idea is that these decisions should be based on the abstract error paths.
3. In analogy to heuristic global search in AI [45], e.g. greedy best-first search or A^*, we might improve our method by keeping several simulations run so far in a search queue, and selecting the most promising of those simulations for being continued for a certain while.
4. Another idea is to improve the simulations by running them backwards as well as forwards [10].
5. It would be desirable to make the method less dependent on arbitrary choices for the implementation parameters.
6. Lyapunov functions [27] decrease their value on trajectories of ordinary differential equations. We will try to improve the quality estimate by using such Lyapunov function for estimating the length of a trajectory from a given point to a box.

Acknowledgements The first author has been supported by GAČR grant 201/08/J020 and by the institutional research plan AV0Z100300504. The second author 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). We would like to thank Tomáš Dzetkulič for help with the implementation.

References

1. L. de Alfaro and J. Palsberg, editors. *EMSOFT ’08: 7th ACM International Conference on Embedded Software*, New York, NY, USA, 2008. ACM.
2. R. Alur, T. Dang, and F. Ivančić. Counter-example guided predicate abstraction of hybrid systems. In H. Garavel and J. Hatcliff, editors, *TACAS’03: 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *LNCS*, pages 208–223. Springer, 2003.
3. R. Alur, T. Dang, and F. Ivančić. Predicate abstraction for reachability analysis of hybrid systems. *Transactions on Embedded Computing Systems*, 5(1):152–199, 2006.
4. R. Alur, A. Kanade, S. Ramesh, and K. C. Shashidhar. Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. In de Alfaro and Palsberg [1], pages 89–98.
5. P. I. Barton, J. R. Banga, and S. Galan. Optimization of hybrid discrete/continuous dynamic systems. *Computers and Chemical Engineering*, 24(12):2171–2182, 2000.
6. 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.
7. A. Bemporad, F. Borrelli, and M. Morari. Piecewise linear optimal controllers for hybrid systems. In *American Control Conference*, pages 1190–1194, Chicago, USA, June 2000.
8. F. Benhamou and L. Granvilliers. Continuous and interval constraints. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 16, pages 571–603. Elsevier, Amsterdam, 2006.
9. D. P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 1995.
10. 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.
11. A. Bhatia and E. Frazzoli. Sampling-based resolution-complete algorithms for safety falsification of linear systems. In Egerstedt and Mishra [18], pages 606–609.
12. M. S. Branicky, V. S. Borkar, and S. K. Mitter. A unified framework for hybrid control: Model and optimal control theory. *IEEE Transactions on Automatic Control*, 43(1):31–45, 1998.
13. P. Cheng and V. Kuma. Sampling-based falsification and verification of controllers for continuous dynamic systems. *The International Journal of Robotics Research*, 27(11-12):1232–1245, 2008.
14. 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. *International Journal of Foundations of Computer Science*, 14(4):583–604, 2003.
15. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
16. W. Damm and H. Hermanns, editors. *CAV’07: 19th International Conference on Computer Aided Verification*, volume 4590 of *LNCS*, 2007.
17. A. Donz and O. Maler. Systematic simulation using sensitivity analysis. In Bemporad et al. [6], pages 174–189.
18. M. Egerstedt and B. Mishra, editors. *HSCC’04: 11th International Workshop on Hybrid Systems: Computation and Control*, volume 4981 of *LNCS*. Springer, 2004.
19. G. Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. In Morari and Thiele [33], pages 258–273.
20. M. Frnzle and C. Herde. Hysat: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30(3):179–198, 2007.

21. A. A. Gadkari, A. Yeolekar, J. Suresh, S. Ramesh, S. Mohalik, and K. C. Shashidhar. AutoMOTGen: Automatic model oriented test generator for embedded control systems. In A. Gupta and S. Malik, editors, *CAV'08: 20th International Conference on Computer Aided Verification*, volume 5123 of *LNCS*, pages 204–208. Springer, 2008.
22. A. Girard and G. Pappas. Verification using simulation. In J. Hespanha and A. Tiwari, editors, *HSCC'06: 9th International Workshop on Hybrid Systems: Computation and Control*, volume 3927 of *LNCS*, pages 272–286. Springer, 2006.
23. B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 117–127. ACM, 2006.
24. S. Hedlund and A. Rantzer. Optimal control of hybrid systems. In *38th IEEE Conference on Decision and Control*, Phoenix, Arizona, Dec. 1999.
25. 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. [6], pages 329–242.
26. J. Kapinski, B. H. Krogh, O. Maler, and O. Stursberg. On systematic simulation of open continuous systems. In O. Maler and A. Pnueli, editors, *HSCC'03: 6th International Workshop on Hybrid Systems: Computation and Control*, volume 2623 of *LNCS*. Springer, 2003.
27. H. K. Khalil. *Nonlinear Systems*. Prentice Hall, 3rd edition, 2002.
28. 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.
29. S. Kupferschmid, K. Drger, J. Hoffmann, B. Finkbeiner, H. Dierks, A. Podelski, and G. Behrmann. UPPAAL/DMC – Abstraction-based heuristics for directed model checking. In O. Grumberg and M. Huth, editors, *TACAS'07: 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *LNCS*, pages 679–682. Springer, 2007.
30. E. A. Lee and H. Zheng. Operational semantics of hybrid systems. In Morari and Thiele [33].
31. F. Lerda, J. Kapinski, E. M. Clarke, and B. H. Krogh. Verification of supervisory control software using state proximity and merging. In Egerstedt and Mishra [18], pages 344–357.
32. F. Lerda, J. Kapinski, H. Maka, E. Clarke, and B. Krogh. Model checking in-the-loop: Finding counterexamples by systematic simulation. In *American Control Conference*, 2008.
33. M. Morari and L. Thiele, editors. *HSCC'03: Proceedings of the 8th International Workshop on Hybrid Systems: Computation and Control*, volume 3414 of *LNCS*. Springer, 2005.
34. T. Nahhal and T. Dang. Test coverage for continuous and hybrid systems. In Damm and Hermanns [16], pages 449–462.
35. T. Park and P. I. Barton. State event location in differential-algebraic models. *ACM Transactions on Modelling and Computer Simulation*, 6(2):137–165, 1996.
36. F. M. D. Paula and A. J. Hu. An effective guidance strategy for abstraction-guided simulation. In *DAC '07: 44th Annual Conference on Design Automation*, pages 63–68. ACM, 2007.
37. E. Plaku, L. E. Kavraki, and M. Y. Vardi. Hybrid systems: From verification to falsification. In Damm and Hermanns [16], pages 463–476.
38. B. Potočnik, A. Bemporad, F. D. Torrisi, G. Mušič, and B. Zupančič. Hybrid modelling and optimal control of a multiproduct batch plant. *Control Engineering Practice*, 12:1127–1137, 2004.
39. S. Prajna, A. Papachristodoulou, and P. A. Parrilo. Introducing SOSTOOLS: A general purpose sum of squares programming solver. In *CDC'02: IEEE Conference on Decision and Control*, 2002.
40. S. Prajna and A. Rantzer. Primal-dual tests for safety and reachability. In Morari and Thiele [33].
41. S. Ratschan. RSOLVER. `http://rsolver.sourceforge.net`, 2004. Software package.
42. S. Ratschan. Efficient solving of quantified inequality constraints over the real numbers. *ACM Transactions on Computational Logic*, 7(4):723–748, 2006.
43. S. Ratschan and Z. She. Safety verification of hybrid systems by constraint propagation based abstraction refinement. *ACM Transactions in Embedded Computing Systems*, 6(1), 2007.

44. S. Ratschan and J.-G. Smaus. Finding errors of hybrid systems by optimising an abstraction-based quality estimate. In C. Dubois, editor, *3rd International Conference on Tests and Proofs*, volume 5668 of *LNCS*, pages 153–168. Springer, 2009.
45. S. J. Russell and P. Norvig. *Artificial Intelligence: a Modern Approach*. Series in Artificial Intelligence. Prentice Hall, 2003.
46. M. Satpathy, A. Yeolekar, and S. Ramesh. Randomized directed testing (REDIRECT) for Simulink/Stateflow models. In de Alfaro and Palsberg [1], pages 217–226.
47. G. Sewell. *The Numerical Solution of Ordinary and Partial Differential Equations*. Academic Press, 1988.
48. J.-G. Smaus and J. Hoffmann. Relaxation refinement: A new method to generate heuristic functions. In D. Peled and M. Wooldridge, editors, *MoChArt'08: 5th Workshop on Model Checking and Artificial Intelligence*, volume 5348 of *LNCS*, pages 147–165. Springer, 2009.
49. O. Stursberg. A graph-search algorithm for optimal control of hybrid systems. In *43rd IEEE Conference on Decision and Control*, pages 1412–1417, 2004.
50. R. S. Sutton and A. G. Barto. *Reinforcement Learning*. The MIT Press, 1998.