An Introduction to Hybrid Automata, Numerical Simulation and Reachability Analysis

Goran Frehse

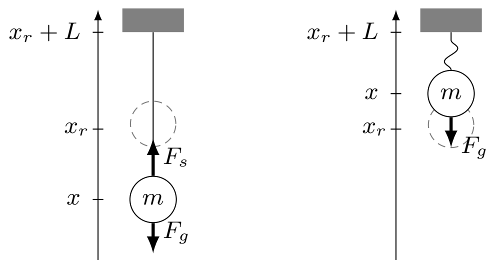
Verimag, Université Joseph Fourier - Grenoble 1,
2 avenue de Vignate, Centre Equation,
38610 Giéres, France,
frehse@imag.fr

Abstract. Hybrid automata combine finite state models with continuous variables that are governed by differential equations. Hybrid automata are used to model systems in a wide range of domains such as automotive control, robotics, electronic circuits, systems biology, and health care. Numerical simulation approximates the evolution of the variables with a sequence of points in discretized time. This highly scalable technique is widely used in engineering and design, but it is difficult to simulate all representative behaviors of a system. To ensure that no critical behaviors are missed, reachability analysis aims at accurately and quickly computing a cover of the states of the system that are reachable from a given set of initial states. Reachability can be used to formally show safety and bounded liveness properties. This chapter outlines the major concepts and discusses advantages and shortcomings of the different techniques.

1 Introduction

Hybrid automata are a modeling formalism that combines discrete states with continuously evolving, real-valued variables. The discrete states and the possible transitions from one state to another are described with a finite state-transition system. A change in discrete state can update the continuous variables and modify the set of differential equations that describes how variables evolve with time. Hybrid automata are non-deterministic, which means that different futures may be available from any given state. Rates of change or variable updates can be described by providing bounds instead of fixed numbers. Incomplete knowledge about initial conditions, perturbations, parameters, etc. can easily be captured this way. Hybrid automata capture a rich variety of behaviors, and are used in a wide range of domains such as automotive control, robotics, electronic circuits, systems biology, and health care. The hybrid automaton model is well-suited for formal analysis, in the sense that sets of behaviors are readily described by mathematical equations.

In the next section, we present the hybrid automaton formalism with an example and give a formal definition of the model and its semantics. Behaviors of hybrid automata are computed in practice using numerical simulation, which



(a) *extension* (b) *freefall*

Fig. 1. A ball suspended from a ceiling by an elastic string

is presented in Sect. 3. We discuss major aspects such as approximation errors, stability, stiffness and Zeno behavior. Simulation techniques have recently been extended to provide formal guarantees for certain classes of systems. Reachability techniques are discussed in Sect. 4. Different algorithms and data structures are suitable for reachability, depending mainly on the type of dynamics: piecewise constant, affine, and nonlinear. We present an overview of the main techniques, with particular attention to scalability.

Related work is indicated throughout the text, but given the rich literature on the topic this introduction is far from exhaustive. For further reading, see [43, 4, 50].

2 Hybrid Automata

We introduce the hybrid automaton formalism with the following example. The technical details are fleshed out in the sections that follow. Consider a ball that is suspended from a ceiling by a string, as shown in Fig. 1. We will construct a simple model that only takes into account the vertical movement of the ball.

Equations of motion. Let x be the variable representing the position of the ball measured in the upwards direction, and let x_r be the position of the ball when the string is at its natural length L. The ball has mass m and is at all times subject to a gravitational force

$$F_g = -mg.$$

The string is elastic, so when the string is extended beyond its natural length L, i.e., $x \leq x_r$, it acts as a damped spring pulling the ball upwards with a force

$$F_s = -k(x - x_r) - d\dot{x},$$

where k is the spring constant and d is a damping factor that models the loss of energy through deformation of the string. When $x \geq x_r$, the string is slack and only the gravitational force is acting on the ball.

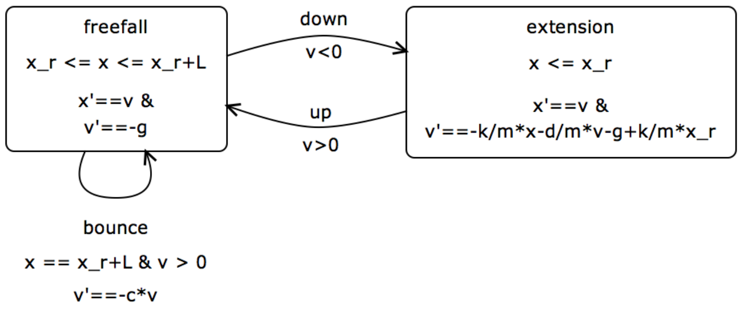


Fig. 2. A hybrid automaton model of a ball on string, constructed in the tool SpaceEx. In the flow equations, x' denotes the derivative $\dot{x}$

The case distinction for the string force leads to two *discrete states* of the system, which we shall call *freefall* and *extension*. Each state is associated with a different set of differential equations and active for different values of x. The system is in *freefall* when $x \geq x_r$ and we have

$$m\ddot{x} = F_g = -mg.$$

The system is in *extension* when $x \leq x_r$, with

$$m\ddot{x} = F_g + F_s = -mg + kx_r - kx - d\dot{x}.$$

A *discrete event*, also called *transition*, takes place when the ball hits the ceiling, i.e., when $x = x_r + L$. We assume that the collision is elastic, so the velocity of the ball changes its sign and is diminished by a factor $c \in [0, 1]$ that reflects the loss of energy during the collision. Denoting the value of x after the collision with x', we get

$$\dot{x}' = -c\dot{x}.$$

Hybrid automaton model. Figure 2 shows a hybrid automaton that models the system. Each discrete state, also called *location*, is labelled with the differential equations that govern the variables. Typically, this is a *first-order ordinary differential equation system* (ODE) of the form

$$\dot{x} = f(x).$$

We refer to the ODE as the *dynamics* of the system. To bring the laws of motion to ODE form we introduce an auxiliary variable to replace $\ddot{x}$. Let v be the velocity of the ball, i.e., $\dot{x} = v$. Then $\ddot{x} = \dot{v}$ and we can substitute $\ddot{x}$ with $\dot{v}$. Solving the equations so that all derivatives are on the left hand side leads to the dynamics shown in Fig. 2.

In addition to the ODE, the *locations* are also labelled with a staying condition, called *invariant*. It characterizes any non-differential conditions that must hold whenever the system is in the location, such as boundary conditions or algebraic equations. In location *extension*, the invariant is $x \leq x_r$ since this is when the string extends beyond its natural length. In location *freefall*, the invariant is given by two constraints: Since the ball needs to be above where it extends the string, we have $x \geq x_r$. Since the ball cannot go further up than the ceiling, which is located at $x = x_r + L$, we have $x \leq x_r + L$.

When the ball does hit the ceiling with positive velocity, it bounces off and its velocity changes sign. This is modeled by a transition with label *bounce*. It goes from location *freefall* to *freefall*, since the ball remains subject to the same differential and boundary conditions. The transition is labeled with a *guard condition*

$$x = x_r + L \wedge v > 0,$$

which must be satisfied for a state to be able to take the jump. The target state after the jump is expressed by the *assignment*

$$v' = -cv.$$

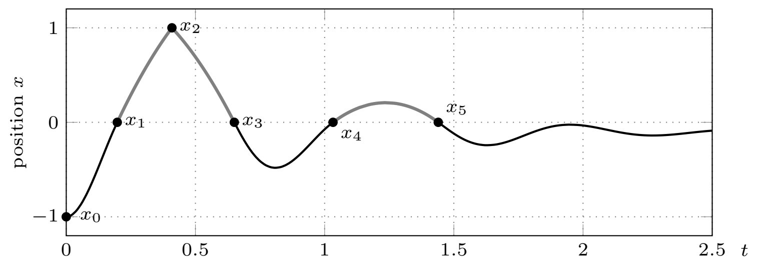
Variables not mentioned in the assignment are considered unchanged.

Transitions *up* and *down* model the switching between *freefall* and *extension*. A guard condition on the velocity of the ball is included to avoid unnecessary switches when the ball is on the switching border $x = x_r$ and $v = 0$. It also helps verification tools that are based on overapproximations, since it excludes unnecessary switches.

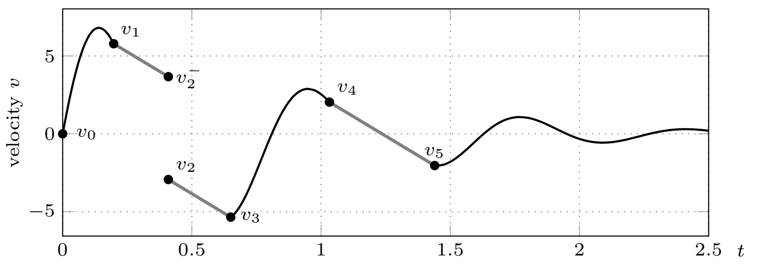
Run semantics. The behavior of a hybrid automaton is described by the evolution of its continuous variables and its location, i.e., its discrete state. Note that in general there need not be any correspondence between the values of the variables, called the *continuous state*, and the discrete state. In our example, the ball can be in position $x = x_r$ in location *freefall* and in location *extension*, independently of the value of v. The combination of the continuous state and the discrete state is called the *state* of the hybrid automaton. The state, by definition, is what determines the set of possible futures of the system. The evolution of the state over time is called a *run* of the system. In the following, we choose the parameter values $L = 1$, $x_r = 0$, $k = 100$, $d = 4$, $c = 0.8$, $m = 1$, and $g = 10$.

Figure 3 shows the evolution of the variables x, v over time, starting from $x_0 = -1, v_0 = 0$, and location *extension*. We follow this evolution location by location:

1. The ball is pulled upwards by the string until $x_1 = x_r$, at which point the automaton can no longer spend time in location *extension* since otherwise the invariant $x \leq x_r$ would be violated. Since v is positive, the transition *up* to location *freefall* is enabled, and taking it is the only possible future in which time can progress.
2. From (x_1, v_1), the ball continues its upward motion in a parabola until it reaches the ceiling at $x_2 = x_r + L$, where the invariant $x \leq x_r + L$ ensures that the only possible future is to take the transition *bounce*. The transition instantaneously changes the velocity from v_2^- to v_2, see Fig. 3(b).



(a) position x over time t



(b) velocity v over time t

Fig. 3. A run of the ball on string model consists of a sequence of trajectories. Trajectories in location *extension* are shown in black, trajectories in *freefall* in gray

3. From (x_2, v_2), the ball falls towards the ground. When $x_3 = x_r$, the transition *down*, which is enabled since $v < 0$, leads to location *extension*.
4. The ball is pulled upwards until it reaches $x = x_r$, where the transition *up* leads to location *freefall*.
5. In location *freefall* at (x_4, v_4), the ball follows a parabola until $x_5 = x_r$, where the transition *down* takes it back to location *extension*.
6. The trajectory from (x_5, v_5) remains in the invariant for all time, and none of the transitions are enabled. The ball converges towards its equilibrium point.

Each of the steps of the above sequence corresponds to one location and a differentiable function of time representing the evolution of the continuous variables. Going from one step to the next is associated with the label of corresponding transition. Such a sequence is called the *run* of a hybrid automaton, and the set of runs defines the semantics (the set of behaviors) of the system.

2.1 Preliminaries

Hybrid automata describe the evolution of a set of real-valued variables over time. We now introduce the notation for describing sets of values for these variables.

Variables. Let $X = \{x_1, \dots, x_n\}$ be a finite set of identifiers we call *variables*. Attributing a real value to each variable we get a *valuation* over X, written as $\mathbf{x} \in \mathbb{R}^X$ or $\mathbf{x} : X \to \mathbb{R}$. We will use the primed variables $X' = \{x'_1, \dots, x'_n\}$ to denote successor values and the dotted variables $\dot{X} = \{\dot{x}_1, \dots, \dot{x}_n\}$ to denote the derivatives of the variables with respect to time. Given a set of variables $Y \subseteq X$, the *projection* $\mathbf{y} = \mathbf{x}\downarrow_Y$ is a valuation over Y that maps each variable in Y to the same value that it has in x. We may simply use a vector $\mathbf{x} \in \mathbb{R}^n$ if it is clear from the context which index of the vector corresponds to which variable. We denote the i-th element of a vector $\mathbf{x}$ as $\mathbf{x}_i$ or $\mathbf{x}(i)$ if the latter is ambiguous. In the following, we use $\mathbb{R}^n$ instead of $\mathbb{R}^X$ except when the correspondance between indices and variables is not obvious, e.g., when valuations over different sets of variables are involved.

Predicates. A *predicate* over X is an expression that, given a valuation $\mathbf{x}$ over X, can be evaluated to either true or false. A *linear constraint* is a predicate

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \le b,$$

where $a_1, \dots a_n$ and b are real-valued constants, and whose sign may be strict ($<$) or nonstrict ($\le$). A linear constraint is written in vector notation as

$$\mathbf{a}^\mathsf{T}\mathbf{x} \le b,$$

with coefficient vector $\mathbf{a} \in \mathbb{R}^n$ and inhomogeneous coefficient $b \in \mathbb{R}$. A predicate over X defines a continuous set, which is the subset of $\mathbb{R}^X$ on which the predicate evaluates to true.

Polyhedra. A conjunction of finitely many linear constraints defines an $\mathcal{H}$-*polyhedron*, or polyhedron in *constraint form*,

$$\mathcal{P} = \left\{\mathbf{x} \,\middle|\, \bigwedge\nolimits_{i=1}^{m} \mathbf{a}_i^\mathsf{T}\mathbf{x} \bowtie_i b_i\right\}, \text{ with } \bowtie_i \in \{<, \le\},$$

with *facet normals* $\mathbf{a}_i \in \mathbb{R}^n$ and *inhomogeneous coefficients* $b_i \in \mathbb{R}$. In vector-matrix notation, an $\mathcal{H}$-*polyhedron* can be written as

$$\mathcal{P} = \left\{\mathbf{x} \,\middle|\, A\mathbf{x} \bowtie \mathbf{b}\right\}, \text{ with } A = \begin{pmatrix} \mathbf{a}_1^\mathsf{T} \\ \vdots \\ \mathbf{a}_m^\mathsf{T} \end{pmatrix}, \bowtie = \begin{pmatrix} \bowtie_1 \\ \vdots \\ \bowtie_m \end{pmatrix}, \mathbf{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}.$$

An $\mathcal{H}$-polyhedron is a *closed* set if it can be defined using only nonstrict constraints. A bounded polyhedron is called a *polytope*. Note that the constraints defining $\mathcal{P}$ are not necessarily unique. A closed polyhedron $\mathcal{P}$ can be represented in *generator form* by a pair (V, R), where $V \subseteq \mathbb{R}^n$ is a finite set of *vertices*, and $R \subseteq \mathbb{R}^n$ is a finite set of *rays*. They define the $\mathcal{V}$-*polyhedron*

$$P = \left\{ \sum_{\mathbf{v}_i \in V} \lambda_i \cdot \mathbf{v}_i + \sum_{\mathbf{r}_j \in R} \mu_j \cdot \mathbf{r}_j \,\middle|\, \lambda_i \ge 0, \mu_j \ge 0, \sum_i \lambda_i = 1 \right\},$$

which consists of the convex hull of the vertices, extended towards infinity along the directions of the rays. The generator representation can be extended with *closure points* to deal with non-closed polyhedra [10]. An $\mathcal{H}$-polyhedron can be converted to a $\mathcal{V}$-polyhedron and vice versa, but this may increase the complexity exponentially.

2.2 Definition and Semantics

We now give a formal definition of a hybrid automaton and its run semantics.

Definition 1 (Hybrid automaton). *[5, 35] A* hybrid automaton

$$H = (\mathsf{Loc}, \mathsf{Lab}, \mathsf{Edg}, X, \mathsf{Init}, \mathsf{Inv}, \mathsf{Flow}, \mathsf{Jump})$$

consists of

- *a finite set of* locations $\mathsf{Loc} = \{\ell_1, \ldots, \ell_m\}$ *represents the discrete states,*
- *a finite set of* synchronization labels Lab*, also called its* alphabet*, which can be used to coordinate state changes between several automata,*
- *a finite set of edges* $\mathsf{Edg} \subseteq \mathsf{Loc} \times \mathsf{Lab} \times \mathsf{Loc}$*, also called* transitions*, which determines which discrete state changes are possible using which label,*
- *a finite set of* variables $X = \{x_1, \ldots, x_n\}$*, partitioned into uncontrolled variables* U *and controlled variables* Y*;* a state *of* H *consists of a location* ℓ *and a value for each of the variables, and is denoted by* $s = (\ell, \mathbf{x})$*;*
- *a set of states* Inv *called* invariant *or* staying condition*; it restricts for each location the values that* x *can possibly take and so determines how long the system can remain in the location;*
- *a set of* initial *states* $\mathsf{Init} \subseteq \mathsf{Inv}$*; every behavior of* H *must start in one of the initial states;*
- *a* flow relation Flow*, where* $\mathsf{Flow}(\ell) \subseteq \mathbb{R}^{\dot{X}} \times \mathbb{R}^{X}$ *gives for each state* $(\ell, \mathbf{x})$ *the set of possible derivatives* $\dot{\mathbf{x}}$*, e.g., using a differential equation such as*

  $$\dot{\mathbf{x}} = f(\mathbf{x});$$

  Given a location ℓ*, a* trajectory *of duration* $\delta \geq 0$ *is a continuously differentiable function* $\xi : [0, \delta] \to \mathbb{R}^X$ *such that for all* $t \in [0, \delta]$*,* $(\dot{\xi}(t), \xi(t)) \in \mathsf{Flow}(\ell)$*. The trajectory* satisfies the invariant *if for all* $t \in [0, \delta]$*,* $\xi(t) \in \mathsf{Inv}(\ell)$*.*
- *a* jump relation Jump*, where* $\mathsf{Jump}(e) \subseteq \mathbb{R}^X \times \mathbb{R}^{X'}$ *defines for each transition* $e \in \mathsf{Edg}$ *the set of possible successors* $\mathbf{x}'$ *of* $\mathbf{x}$*; jump relations are typically given by a* guard *set* $\mathcal{G} \subseteq \mathbb{R}^X$ *and an assignment (or reset)* $\mathbf{x}' = r(\mathbf{x})$ *as*

  $$\mathsf{Jump}(e) = \{(\mathbf{x}, \mathbf{x}') \mid \mathbf{x} \in \mathcal{G} \wedge \mathbf{x}' = r(\mathbf{x})\}.$$

We define the behavior of a hybrid automaton with a *run*: starting from one of the initial states, the state evolves according to the differential equations whilst time passes, and according to the jump relations when taking an (instantaneous) transition. Special events, which we call *uncontrolled assignments*, model an environment that can make arbitrary changes to the uncontrolled variables.

Definition 2 (Run semantics). *A* run *of H is a sequence*

$$(\ell_0, \mathbf{x}_0) \xrightarrow{\delta_0, \xi_0} (\ell_0, \xi_0(\delta_0)) \xrightarrow{\alpha_0} (\ell_1, \mathbf{x}_1) \xrightarrow{\delta_1, \xi_1} (\ell_1, \xi_1(\delta_1)) \ldots \xrightarrow{\alpha_{N-1}} (\ell_N, \mathbf{x}_N),$$

with $\alpha_i \in \mathsf{Lab} \cup \{\tau\}$, satisfying for $i = 0, \ldots, N-1$:

1. *The first state is an initial state of the automaton, i.e., $(\ell_0, \mathbf{x}_0) \in \mathsf{Init}$.*
2. Trajectories: *In location ℓ_i, ξ_i is a trajectory of duration δ_i that satisfies the invariant.*
3. Jumps: *If $\alpha_i \in \mathsf{Lab}$, there exists a transition $(\ell_i, \alpha_i, \ell_{i+1}) \in \mathsf{Edg}$ with jump relation $\mathsf{Jump}(e)$ such that $(\xi_i(\delta_i), \mathbf{x}_{i+1}) \in \mathsf{Jump}(e)$ and $\mathbf{x}_{i+1} \in \mathsf{Inv}(\ell_{i+1})$.*
4. Uncontrolled assignments: *If $\alpha_i = \tau$, then $\ell_i = \ell_{i+1}$ and $\xi_i(\delta_i)\downarrow_Y = \mathbf{x}_{i+1}\downarrow_Y$. This represents arbitrary assignments that the environment might perform on the uncontrolled variables $U = X \setminus Y$.*

A state $(\ell, \mathbf{x})$ is reachable *if there exists a run with $(\ell_i, \mathbf{x}_i) = (\ell, \mathbf{x})$ for some i.*

Note that the strict alternation of trajectories and jumps in Def. 2 is of no particular importance. Two consecutive jumps can be represented by inserting a trajectory with duration zero (which always exists), and two consecutive trajectories can be represented by inserting an uncontrolled assignment jump that does not modify the variables.

Example 1 (Ball/String). The Ball on String example is modeled by the hybrid automaton shown in Fig. 2. All elements except the labels and the initial states are visible in the figure. Since we do not expect x or v to be modified by the environment, we consider both variables to be controlled (as is usually the case for variables whose derivative is given). The set of synchronization labels is $\mathsf{Lab} = \{\mathit{bounce}, \mathit{up}, \mathit{down}\}$. For now, we assume $\mathsf{Init} = \{\mathit{extension}\} \times \{x = -1, v = 0\}$.

In location *extension*, the dynamics are those of a damped oscillator. The ODE system is linear in the variables x and v, so its solution is a combination of exponential, sine and cosine functions of time. In location *freefall*, the dynamics are also linear, but of a particularly simple kind. The derivative of v is constant, $\dot{v} = -g$, so that v evolves in a straight line and x in a parabola. Figure 4 shows the trajectories of the same run as in Fig. 3, but in the state space (also called phase space), which allows one to graph over an infinite time horizon.

May and Must semantics. In Def. 2, transitions may be taken when they are enabled, but there is no obligation to do so – the system may remain in a location as long as the invariant is satisfied. These so-called *may* semantics allow one to include nondeterminism about when a transition will be taken, e.g., when it is not clear how fast a discrete controller will react to a stimulus. In the Ball/String example, this could be used if the length of the string (position of the ceiling) is not exactly known. In contrast, *must* or *ASAP* semantics dictate that the transition is taken as soon as possible. These semantics are used by simulators such as Simulink [45], Dymola [13], MapleSim [44], etc., since they require deterministic models. Some verification tools, like HyTech [32] and PHAVer [23], allow one to include both types of transitions.

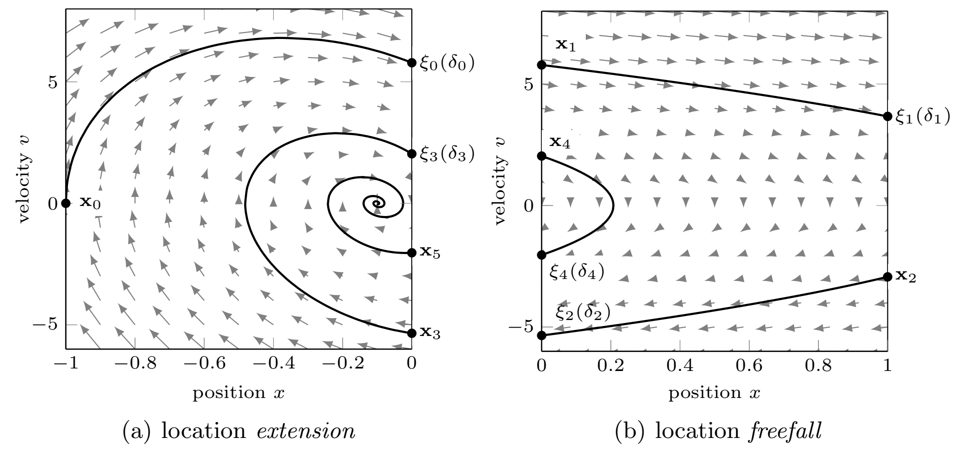


Fig. 4. A sample run of the ball/string example, with trajectories $\xi_0, \ldots, \xi_5$. The initial state $\mathbf{x}_0$ corresponds to the variable values $x = -1, v = 0$ in location *extension*. The arrows indicate the direction and magnitude of the derivative

3 Numerical Simulation

The approximate computation of a run of a hybrid automaton is called *numerical simulation*. This technique is widely applied in industrial practice and has the distinct advantage that precise and highly scalable algorithms are available. In this section, we briefly introduce the basic principles and discuss the major difficulties. Numerical simulation is related to formal verification in several ways:

- The reachability methods of Sect. 4 are built on the same principles, and similar difficulties arise, such as numerics, stability, stiffness and Zeno.
- Set-based extensions of numerical simulation algorithms are available in both approximate [12] and conservative forms [11].
- Verification-by-simulation techniques extend simulation runs to their neighborhoods such that coverage can be formally guaranteed.
- Numerical simulation is typically the validation technique used for *constructing* models. Knowing its strengths and limitations may help to avoid modeling errors and oversights.
- In *testing*, a large number of simulation runs are computed to sample as much of the state space as possible. Various guiding schemes choose these runs intelligently, aiming to achieve coverage similar (but not equal) to formal methods.

Computing a simulation run for a given maximum number of jumps and a given time horizon consists of the following steps:

1. Choose a single state from the set of initial states.
2. *Continuous step:* Compute a trajectory by solving the ODE of the location, stopping when the invariant is violated or the time horizon has been reached.

3. *Discrete step:* Detect the transitions that are enabled along the trajectory. If available, choose one of the transitions, and a time point when it is enabled. Compute one of the successor states of the jump.
4. If the maximum number of jumps or the time horizon has been reached, stop. Otherwise, continue with step 2.

In the following section, we present an overview on solving ODEs, which is the main ingredient for the continuous step. In Sect. 3.2, we discuss how to detect state-based events, such as violating the invariant or entering a guard, which is the main ingredient for the discrete step. In actual implementations, both techniques are intertwined to generate a sequence of states on the run with as little computational overhead as possible.

3.1 Solving ODEs

We consider an *ordinary differential equation* (ODE) of the form

$$\dot{\mathbf{x}} = f(\mathbf{x}),$$

where $\dot{\mathbf{x}} = d\mathbf{x}/dt$ represents the rate of change of $\mathbf{x}$ with respect to time $t \in \mathbb{R}^{\geq 0}$. Solving the ODE for a given initial state $\mathbf{x}_0$ and time horizon T means to find a function $\xi(t)$ such that $\xi(0) = \mathbf{x}_0$ and $\dot{\xi}(t) = f(\xi(t))$ for all $t \in [0, T]$. This is referred to as an *initial value problem.* We are interested in solving the ODE numerically, which means computing a sequence of states $\mathbf{x}_0, \ldots, \mathbf{x}_N$ that approximates $\xi(t)$ at time points $t_0, \ldots, t_N$. The choice of time points is either fixed with a given time step h, i.e.,

$$t_{i+1} = t_i + h,$$

or h is adapted on the fly in order to achieve a given error bound. Standard ODE solvers do not guarantee actual error bounds at the computed points, since such bounds are frequently overly conservative in practice. Instead, it is guaranteed that, at least for certain classes of problems, the approximation error vanishes as $h \to 0$. While in engineering practice the linear interpolation between these points is typically considered a good approximation of $\xi(t)$, there are no a-priori bounds on the distance between the linear interpolation and the actual solution. We now provide a brief overview over the main methods for solving ODEs, a readable introduction can be found, e.g., in [14]. In the following, we consider ODEs of a single variable x. The extension to a vector $\mathbf{x}$ is straightforward.

Euler's Method The simplest integration method is called *Euler's method.* It computes the sequence

$$x_{i+1} = x_i + f(x_i)h. \tag{1}$$

An estimation for the *local error*, i.e., the error made at each step, can be obtained by comparing the sequence to a Taylor series expansion around x_i,

$$x_{i+1} = x_i + \dot{x}_i h + \frac{\ddot{x}_i}{2!}h^2 + \ldots + \frac{\overset{(n-1)}{x_i}}{n!}h^n + \mathcal{O}(h^{n+1}), \tag{2}$$

where $\mathcal{O}(h^{n+1})$ specifies that the truncation error is proportional to h^{n+1} if h is chosen small enough. Substituting $\dot{x}_i = f(x_i)$, we can see that the first two terms of the Taylor expansion are identical to Euler's sequence. The local (one-step) approximation error is therefore $\varepsilon_a = \mathcal{O}(h^2)$.

The *global error* is the sum of the local errors at each step. Note that local errors may cancel each other out. The estimation of the global error is more complex than for the local error, but it can be shown for Euler's method that it is $\mathcal{O}(h)$. It is therefore called *first-order* method. In principle, this means that any desired accuracy can be achieved by choosing h small enough. However, we must also take the numerical roundoff error into account, which is $\mathcal{O}(1/h)$. If the time steps are too small, the roundoff error will surpass the approximation error and the accuracy will decrease. This limitation is common to all ODE solvers, and motivates the search for integration methods with a smaller error for the *same number* of function evaluations.

Example 2 (Ball/String). Figure 5 shows approximations of a trajectory in location *extension*, starting from $x = -1, v = 0$. Euler's method was applied for time steps $h = 0.05, 0.025, 0.0125, 0.00625$. For $h = 0.05$, the Euler approximation diverges towards infinity. With decreasing time steps, the approximation converges towards the exact solution, shown in black. For $h = 0.00625$, the global error at the end of the trajectory amounts to 175% of the exact value. The state space view in Fig. 5(b) superimposes the trajectory approximation with a quiver plot of the derivative. At each x_i in the sequence, Euler's method applies the local derivative $f(x_i)$ for h time units, i.e., it follows $f(x_i)$ along a straight line.

Stability and Implicit Methods If the time step is too large, the global error of Euler's method may go quickly to infinity. Consider the linear ODE

$$\dot{x} = ax, \tag{3}$$

which converges to zero for $a < 0$. Euler's method computes the sequence

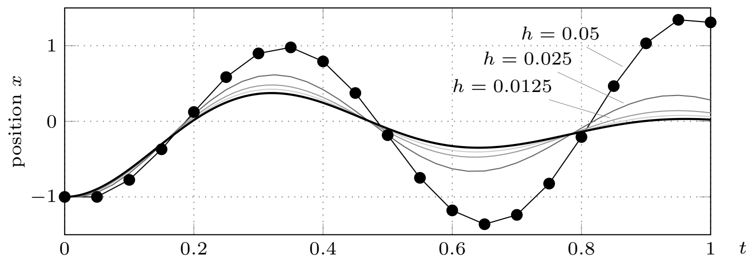
$$x_{i+1} = x_i + f(x_i)h = x_i + ax_ih = (1 + ah)x_i.$$

This sequence converges to zero iff $|1 + ah| < 1$. If $h > -2/a$, then $|x_i| \to \infty$ as $i \to \infty$. Because Euler's method converges only under certain conditions, it is called *conditionally stable*. An *unconditionally stable* method is the *backwards Euler* method, which computes the sequence

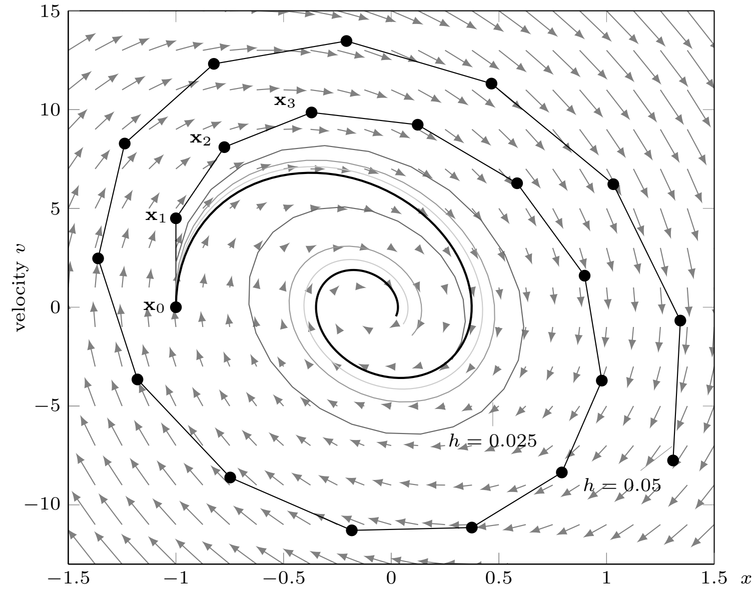
$$x_{i+1} = x_i + f(x_{i+1})h. \tag{4}$$

The backwards Euler method is an *implicit method*, since the unknown value x_{i+1} figures on both sides of the equation. It must be computed iteratively, e.g., using root-finding techniques. Like other implicit methods, the backwards Euler method thus requires more function evaluations than explicit methods. For the linear ODE (3), the backwards Euler sequence can be rearranged to

$$x_{i+1} = \frac{1}{1 - ah} x_i.$$



(a) position x in the time domain



(b) state space, with arrows indicating the derivative

Fig. 5. A trajectory of the ball/string example, approximated with Euler's method for varying time steps. The exact solution is shown in solid black

Since $\frac{1}{1-ah} < 1$ for all $a < 0$ and $h > 0$, the backwards Euler method converges whenever the ODE does, independently of the step size. It is therefore called *unconditionally stable.*

Runge-Kutta Methods Runge-Kutta (RK) methods are a family of higher-order integration methods that use intermediate evaluations of $f(x)$ to improve the precision. Explicit Runge-Kutta methods compute the sequence

$$x_{i+1} = x_i + \phi(x_i, h)h, \qquad (5)$$

where the *increment function* $\phi(x_i, h)$ can be interpreted as a representative slope over the time interval. The increment function is given as

$$\phi(x_i, h) = a_1 k_1 + a_2 k_2 + \cdots + a_n k_n, \tag{6}$$

where the a_i are constants and the k_i are obtained by evaluating the ODE at intermediate states,

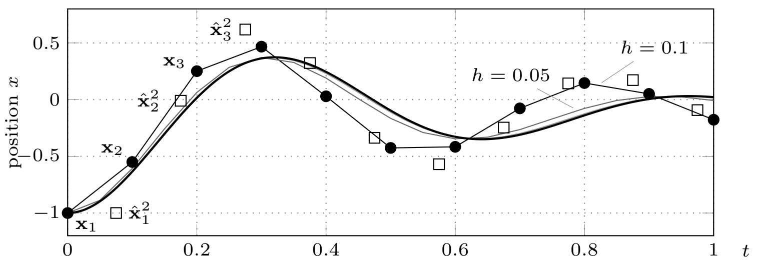
$$\begin{aligned}
k_1 &= f(\hat{x}_i^1), \;\; \hat{x}_i^1 = x_i, \\
k_2 &= f(\hat{x}_i^2), \;\; \hat{x}_i^2 = x_i + q_{11} k_1 h, \\
k_3 &= f(\hat{x}_i^3), \;\; \hat{x}_i^3 = x_i + q_{21} k_1 h + q_{21} k_2 h, \\
&\;\;\vdots \qquad\qquad \vdots \\
k_n &= f(\hat{x}_i^n), \;\; \hat{x}_i^n = x_i + q_{(n-1)1} k_1 h + q_{(n-1)2} k_2 h + \cdots + q_{(n-1)(n-1)} k_{n-1} h,
\end{aligned} \tag{7}$$

where the q_{ij} are constants. Note that k_1 is used to derive the intermediate state $\hat{x}_i^2$ that leads to k_2, etc. The parameters a_i,q_{ij} are derived by equating the terms in (5) to those of a Taylor series expansion, so the method has zero error if the solution is a nth order polynomial. There are more parameters than terms, so the remaining parameters can be chosen to optimize other properties such as the truncation error. Euler's method is a RK method with $n = 1$ and $a_1 = 1$. *Ralston's method* is given by $n = 2$, $a_1 = 1/3$, $a_2 = 2/3$, and $q_{11} = 3/4$, and is the second-order RK method with the smallest truncation error. Runge-Kutta methods for $n = 2, \ldots, 5$ are used in practice, with truncation error $\mathcal{O}(h^{n+1})$ and global error $\mathcal{O}(h^n)$.

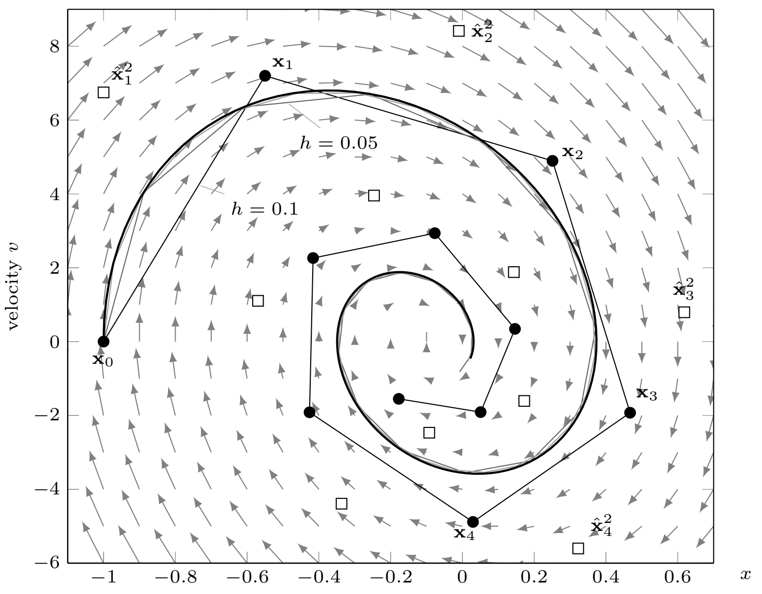
Example 3 (Ball/String). Figure 6 shows approximations of the trajectory from Ex. 2, using Ralston's method for $h = 0.1, 0.05, 0.025,\ 0.0125$. The number of evaluations for $h = 0.1$ with Ralston's method is the same as with Euler's method at $h = 0.05$. At $h = 0.025$, the global error at the end of the trajectory is 33% of the exact value, while at $h = 0.0125$ it is 7%, which is consistent with a global error of $\mathcal{O}(h^2)$. Figure 6(b) shows the approximation with a quiver plot of the derivative and illustrates that the $\mathbf{x}_{i+1}$ are closer to the real solution than the $\hat{\mathbf{x}}_i^2$ from which they are derived.

Error estimation and adaptive time steps Relatively precise error estimates can be obtained by comparing the result of two sequences with different levels of precision. One such approach is halving time steps, i.e., taking the difference between the result for time steps $h/2$ and h. Another is to take the difference between a $(n-1)$th order and a nth order solver. *Runge-Kutta-Fehlberg* (RKF) methods combine $(n-1)$th order and nth order RK methods such that the intermediate results from one sequence are used in the other, so it requires no more evaluations than an nth order RK method on its own. Popular ODE solvers are RKF 2(3) and RKF 4(5), also known as `ode23` and `ode45`.

The estimated error can be used to adapt the time steps. Let ε_a be the current estimate of the truncation error, and ϵ_d be the desired error. The time step can be adapted, e.g., using $h \leftarrow h\left|\epsilon_d/\varepsilon_a\right|^{\alpha}$, where $\alpha = 0.25$ if $\varepsilon_a < \epsilon_d$, and $\alpha = 0.2$ otherwise [48].



(a) position x in the time domain



(b) state space, with arrows indicating the derivative

Fig. 6. A trajectory of the ball/string example, approximated with Ralston's method for different time steps h. Intermediate states $\hat{\mathbf{x}}_i^2$ are shown as squares for $h = 0.1$, the exact solution is shown in solid black

Stiff Systems A system of ODEs is called *stiff* if it involves rapidly changing components together with slowly changing ones, typically with time constants differing by a factor of 1000 or more. On the scale of the slow time constant, the rapid changes thus seem to take place nearly instantaneously, and have little effect once they have died down. Nonetheless, solvers are forced to take tiny time steps throughout the entire time horizon due to stability problems caused by the fast time constants: the approximation error in each step moves the rapid component away from its equilibrium, which is followed by a rapid move back to the equilibrium that must be taken into account by taking small steps. Special

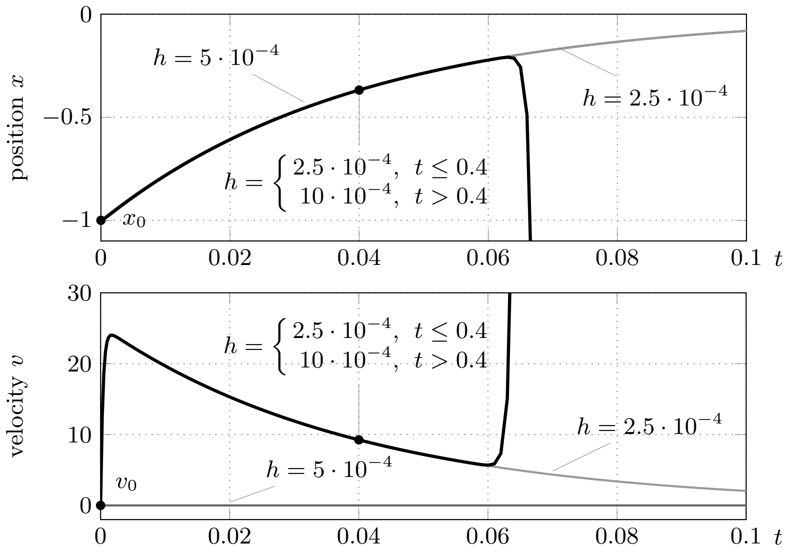


Fig. 7. When applied to stiff systems, Runge Kutta methods can be very sensitive to the time step and require a small time step throughout. Increasing the time step h even during slow phases can make the approximation diverge

solvers are available for stiff ODEs, using implicit methods to achieve stability at larger time steps.

Example 4 (Ball/String). The Ball/String system is stiff for sufficiently small values of the mass m. Figure 7 shows several approximations of the velocity trajectory from Ex. 2, obtained using Ralston's method for $m = 1/1000$. To obtain a stable result, the time step h must be about a factor 100 smaller than in Ex. 3. For $h = 0.5 \cdot 10^{-4}$, the solution is qualitatively false: v remains close to zero (x is approximated curiously well). For $h = 0.25 \cdot 10^{-4}$, the solution is approximated well, which indicates how sensitive the error is to the time step. The bold line depicts the trajectory obtained by using a time step $h = 0.25 \cdot 10^{-4}$ up to $t = 0.04$, and then switching to $h = 10 \cdot 10^{-4}$. Twenty steps after the switch, the sequence suddenly diverges to infinity.

3.2 Computing Trajectories and Jumps

Using an appropriate ODE solver from the previous section, we can approximate points on a trajectory in a given location up to arbitrary precision. However, we need to detect when the trajectory leaves the invariant, since this poses a hard limit on how long the system can stay in the location. This is related to the problem of detecting jumps, i.e., finding out when any of the outgoing transitions are enabled. Both types of events can be detected as a *zero crossing* of suitable functions that are zero on the border of the invariant and guard sets. Typically, a vector of such functions is passed to the ODE solver, which stops whenever one of the functions changes sign from integration step to another. The solver then uses a root-finding algorithm to approximate the exact time of the crossing and returns the time and corresponding state at the crossing.

Shortcomings. Several difficulties arise in the above procedure, for a detailed discussion see [51]:

- Missed events: It is hard to ensure that no roots are missed. This means that violations of the invariants or states in the guard could go undetected.
- Increased computational cost: Using the ODE solver over a strictly increasing sequence of time points allows it to reuse certain intermediate states. This advantage is lost through the back-and-forth of the root-finding algorithm.
- Relaxed constraints: Since the ODE can only be solved approximately, the crossing state may lie slightly outside the guard or invariant. Relaxing the constraints to account for numerical errors may generate spurious behavior.

Zeno Behavior The switching times in a hybrid system may get closer and closer together, to the point that the sequence of switching times converges. This means that an infinite amount of events take place in a finite amount of time. Such behavior is called *Zeno*, and poses a particular problem for numerical simulation, since the simulator seems to get "stuck" as switching times converge.

Example 5 (Ball/String). Consider the Ball/String system turned upside down by reversing the gravitational acceleration to $g = -10$. In this form it becomes a variation of the well-known *bouncing ball*, a standard example for Zeno behavior. Figure 8 shows a run, starting from $x = -1, v = 0$, in which the switching times converge at around $t = 5.5$.

3.3 Accounting for Nondeterminism

The biggest challenge for numerical simulation is nondeterminism, such as

- sets of states: a state must be selected from the set of initial states and from the successor states in the jump relation;
- jumps times: the jump time of a transition with may semantics must be chosen from an interval of time;
- dynamics: a *differential inclusion*, such as $\dot{x} \in [-1, 1]$, leaves a choice for the derivative at each time step;
- discrete successors: if several guards are enabled simultaneously, one must choose between transitions;

A numerical simulation needs to pick one value from the set of possible choices in order to compute the next state. The set of possible runs from a single initial state grows exponentially with each choice. This is particularly detrimental if the dynamics are nondeterministic, where the choice is made at every time step.

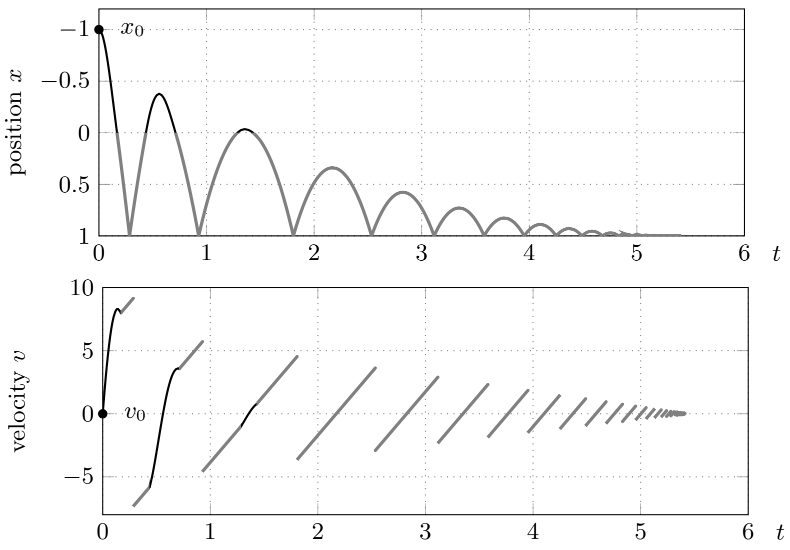


Fig. 8. Zeno behavior in the Ball/String example for $g = -10$. Trajectories in location *extension* are shown in black, trajectories in *freefall* in gray

Typical simulation environments such as Simulink [45], Modelica [46], or Ptolemy [21] use purely deterministic models with must semantics (ref. Sect 2.2), which make jump times deterministic. One way to deal with the complexity resulting from nondeterminism is to associate probabilities to the choices. This leads to *stochastic hybrid systems*, for which techniques such as Monte Carlo simulation can produce a sampling of the possible runs that is associated with probabilities [15].

3.4 Verification by simulation

Verification by simulation techniques aim at formally proving bounded-horizon properties by computing a finite number of trajectories [22, 37]. A so-called bisimulation metric is established to identify a neighborhood around each state such that the states inside remain sufficiently close together for all time. Under suitable assumptions, this makes it possible to identify a finite subset of initial states whose trajectories are sufficient to show certain properties of the system [28]. Similar techniques can be applied for parameter synthesis [20].

4 Reachability Analysis

Reachability analysis extends the concept of numerical simulation from numbers to sets. By computing with sets of states, nondeterminism in the model can be fully taken into account, and the analysis can be exhaustive, even up to an infinite time horizon. Furthermore, successor computations can be conservative in the sense that the computed sets are sure to cover all solutions. Just like numerical simulation, reachability has to resort to approximations if the dynamics of the system are complex. On the downside, the cost of set-based computations generally increases sharply with the number of continuous variables, so scalability is critical. Scalable approximations are available for certain types of dynamics, as discussed later in this section, but this performance comes at a price in accuracy. The trade-off between runtime and accuracy remains a central problem in reachability analysis.

Decidability. The problem whether a given state is reachable from the initial states is generally undecidable for hybrid automata, which means that no algorithm exists that eventually terminates with the right answer [35]. The main subclass for which the problem is decidable are *timed automata*, where all derivatives have the value 1, guards are given by constraints with only one variable each (this may be extended to include the difference between two variables), and jumps either leave a variable unchanged or reset it to a constant value.

4.1 Reachability Algorithm

The standard method to compute the reachable states is to iterate the following *one-step successor* operators for discrete and continuous transitions. Given a set of states S, let $\mathsf{Post}_C(S)$ be the set of states reachable by letting time elapse from any state in S,

$$\mathsf{Post}_C(S) = \{(\ell, \xi(\delta)) \mid \exists(\ell, x) \in S : (\ell, \mathbf{x}) \xrightarrow{\delta,\xi} (\ell, \xi(\delta))\}.$$

Let $\mathsf{Post}_D(S)$ be the set of states resulting from a jump from any state in S,

$$\mathsf{Post}_D(S) = \{(\ell', \mathbf{x}') \mid \exists(\ell', \mathbf{x}') \in S, \exists\alpha \in \mathsf{Lab} \cup \{\tau\} : (\ell, \mathbf{x}) \xrightarrow{\alpha} (\ell', \mathbf{x}')\}.$$

Starting from the initial states, $\mathsf{Post}_C(S)$ and $\mathsf{Post}_D(S)$ are computed in alternation and all states that are obtained are recorded, as in the following sequence:

$$R_0 = \mathsf{Post}_C(\mathsf{Init}), \tag{8}$$

$$R_{i+1} = R_i \cup \mathsf{Post}_C(\mathsf{Post}_D(R_i)). \tag{9}$$

If the sequence reaches a fixed-point, i.e., when $R_{i+1} = R_i$, then R_i is the set of reachable states. Note that simply computing the sequence and testing for a fixed-point may not lead to termination, even for systems where reachability is decidable. E.g., a system with an (unbounded) counter would enter a new state at each iteration such that the fixed-point is never reached.

In tools such as HyTech [33], PHAVer [23] and SpaceEx [24], the sequence (8) is computed using *symbolic states* $s = (\ell, \mathcal{P})$, where $\ell \in \mathsf{Loc}$ and $\mathcal{P}$ is a continuous set, e.g., a polyhedron. Computing the timed successors Post_C of a symbolic state $s = (\ell, \mathcal{P})$ produces a new symbolic state $s' = (\ell, \mathcal{P}')$. Computing the jump successors Post_D of $s = (\ell, \mathcal{P})$ involves iterating over all outgoing

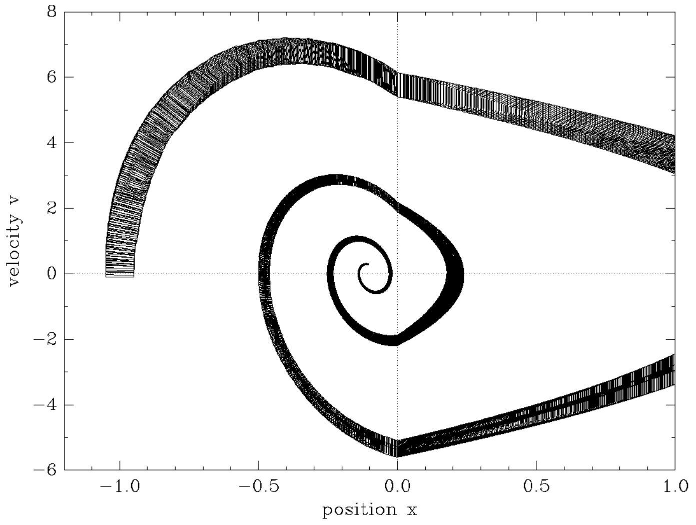


Fig. 9. Reachable states of the ball/string example, computed using SpaceEx

transitions of ℓ, and produces a set of symbolic states $\{s'_1, \dots, s'_N\}$, each in one of the target locations. A *waiting list* contains the symbolic states whose successors still need to be explored, and a *passed list* contains all symbolic states computed so far. The fixed-point computation proceeds as follows:

1. Initialization: Compute the continuous successors of the initial states and put them on the waiting list.
2. Pop a symbolic state s from the waiting list and compute its one-step successors $\{s'_1, \dots, s'_N\} = \mathsf{Post}_C(\mathsf{Post}_D(s))$.
3. Containment checking: Discard the s'_i that have previously been encountered, i.e., those contained in any symbolic state on the passed list. Add the remaining symbolic states to the passed and waiting list.
4. If the waiting list is empty, terminate and return the passed list as the reachable states. Otherwise, continue with step 2.

Different approaches are taken for computing the one-step successors, depending on the type of dynamics. In the following sections, we present the major methods.

Example 6. Figure 9 shows the reachable states of the ball/string example, starting from an initial set of $-1.05 \leq x \leq -0.95$, $-0.1 \leq v \leq 0.1$ in location *extension*. Initializing the waiting list with the continuous successors of the intial states, the fixed point is reached on the 6th iteration. Each symbolic state corresponds to a segment of the run from Ex. 1, and contains all of the corresponding trajectories, from any of the initial states.

4.2 Piecewise Constant Dynamics

Hybrid automata with piecewise constant dynamics (PCDA), also called *linear hybrid automata* (LHA), have

- initial states and invariants given by conjunctions of linear constraints,
- flows given by conjunctions of linear constraints over the derivatives $\dot{X}$, and
- jumps given by linear constraints over $X \cup X'$, where X' denote the variables after the jump.

The one-step successors of PCDA can be computed exactly, which is not the case for the more complex dynamics discussed in later sections. For simplicity we will assume that flow constraints are closed and bounded. Examples for flow constraints of a PCDA include differential inclusions such as $\dot{x} \in [1, 2]$, and conservation laws such as $\dot{x} + \dot{y} = 0$. The jump constraints of a PCDA can generate complex behavior, and even chaos [16]. For example, PCDA can model *discrete-time affine systems*, a widely used class of control systems, with jump constraints of the form $\mathbf{x}' = A\mathbf{x} + \mathbf{b}$.

Continuous successors. In the following, we discuss how to compute the states reachable by time elapse in a given location ℓ. Since ℓ is clear from the context we call $\mathbf{x}$ a (continuous) state. By definition, a trajectory can be an arbitrarily curved function as long as it is differentiable and satisfies the constraints of flow and invariant. For PCDA, it suffices to consider straight-line trajectories:

Lemma 1. *[36] In any given location of a PCDA, there is a trajectory $\xi(t)$ from $\mathbf{x} = \xi(0)$ to $\mathbf{x}' = \xi(\delta)$ for some $\delta > 0$ iff $\eta(t) = \mathbf{x} + \mathbf{q}t$ with $\mathbf{q} = (\mathbf{x}' - \mathbf{x})/\delta$ is a trajectory from $\mathbf{x}$ to $\mathbf{x}'$.*

Consider polyhedra $\mathcal{P}$ and $\mathcal{Q}$. The states on straight line trajectories starting in $\mathcal{P}$ with constant derivative $\dot{x} = \mathbf{q}$ for any $\mathbf{q} \in \mathcal{Q}$ are the *time successors* [6].

$$\mathcal{P} \nearrow \mathcal{Q} = \{\mathbf{x}' \mid \mathbf{x} \in \mathcal{P}, \mathbf{q} \in \mathcal{Q}, t \in \mathbb{R}^{\geq 0}, \mathbf{x}' = \mathbf{x} + \mathbf{q}t\}. \tag{10}$$

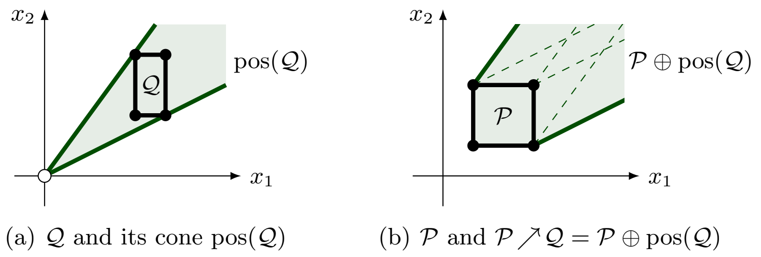
We now transform the right-hand term of (10) into a linear constraint. Let $\mathcal{P}$ and $\mathcal{Q}$ be polyhedra given in vector-matrix form as $\mathcal{P} = \{\mathbf{x} \mid A\mathbf{x} \leq \mathbf{b}\}$, $\mathcal{Q} = \{\mathbf{q} \mid \bar{A}\mathbf{q} \leq \bar{\mathbf{b}}\}$. Eliminating $\mathbf{q} = \frac{\mathbf{x}' - \mathbf{x}}{t}$ for $t > 0$ and multiplying with t yields

$$\mathcal{P} \nearrow \mathcal{Q} = \left\{\mathbf{x}' \;\middle|\; A\mathbf{x} \leq \mathbf{b} \;\wedge\; \bar{A}(\mathbf{x}' - \mathbf{x}) \leq \bar{\mathbf{b}} \cdot t \;\wedge\; t \geq 0\right\}. \tag{11}$$

The above set is a polyhedron that can be computed by quantifier elimination over $X \cup \{t\}$ using, e.g., Fourier-Motzkin elimination. The time successors can also be obtained using geometrical operations, as illustrated in Fig. 10. The *cone* of $\mathcal{Q}$ is the polyhedron $\text{pos}(\mathcal{Q}) = \{\mathbf{q} \cdot t \mid \mathbf{q} \in \mathcal{Q}, t \geq 0\}$. The *Minkowski sum* is defined as $\mathcal{P} \oplus \mathcal{Q} = \{\mathbf{p} + \mathbf{q} \mid \mathbf{p} \in \mathcal{P}, \mathbf{q} \in \mathcal{Q}\}$. The time successors are [31]

$$\mathcal{P} \nearrow \mathcal{Q} = \mathcal{P} \oplus \text{pos}(\mathcal{Q}). \tag{12}$$

If $\mathcal{P}$ and $\mathcal{Q}$ are closed with generator representation (V, R) and $(\bar{V}, \bar{R})$, respectively, then a generator representation of $\mathcal{P} \nearrow \mathcal{Q}$ is $(V, R \cup \bar{V} \cup \bar{R})$. It remains to ensure that the time successors satisfy the invariant $\mathsf{Inv}(\ell)$, which leads to the following continuous successor operator for PCDA.



(a) $\mathcal{Q}$ and its cone pos($\mathcal{Q}$) (b) $\mathcal{P}$ and $\mathcal{P} \nearrow \mathcal{Q} = \mathcal{P} \oplus \text{pos}(\mathcal{Q})$

Fig. 10. The time successors $\mathcal{P} \nearrow \mathcal{Q}$, obtained using geometric operations on $\mathcal{P}$ and $\mathcal{Q}$

Lemma 2. *[6] The* continuous successors *of a polyhedron $\mathcal{P}$ in location ℓ are*

$$\mathsf{post}_\ell(P) = \big(P \nearrow \mathsf{Flow}(\ell)\big) \cap \mathsf{Inv}(\ell).$$

The computation of the time successors is of exponential complexity. In the form of (11), it requires quantifier elimination, while in the form of (12) it requires switching representations, since intersection computed on $\mathcal{H}$-polyhedra and Minkowski sum on $\mathcal{V}$-polyhedra.

Discrete successors. The *discrete successors* of a polyhedron $\mathcal{P}$ for an edge $e = (\ell, \alpha, k)$ is the polyhedron:

$$\mathsf{post}_e(P)\big\{\mathbf{x}' \bigm| \exists \mathbf{x} \in \mathcal{P} : (\mathbf{x}, \mathbf{x}') \in \mathsf{Jump}(\epsilon) \wedge \mathbf{x}' \in \mathsf{Inv}(k)\big\}.$$

This set is defined using existential quantification, and computing it may require costly quantifier elimination. Frequently occurring special cases can be computed more efficiently. Consider $\mathsf{Jump}(e)$ given by a guard $\mathbf{x} \in \mathcal{G}$ and an assignment $\mathbf{x}' = C\mathbf{x} + \mathbf{d}$, with a constant matrix C and a vector $\mathbf{d}$ of appropriate dimensions. The discrete successors are

$$\mathsf{post}_e(P) = \big(C(\mathcal{P} \cap \mathcal{G}) \oplus \{\mathbf{d}\}\big) \cap \mathsf{Inv}(k). \tag{13}$$

If C is invertible and $\mathcal{P}, \mathcal{G}$ are $\mathcal{H}$-polyhedra, the computation is straightforward since intersection corresponds to concatenation of constraints, and for any polyhedron $\mathcal{Q} = \{\mathbf{x} \mid A\mathbf{x} \leq \mathbf{b}\}$,

$$C\mathcal{Q} \oplus \{\mathbf{d}\} = \{\mathbf{x} \mid AC^{-1}\mathbf{x} \leq b + C^{-1}\mathbf{d}\}.$$

4.3 Piecewise Affine Dynamics

Hybrid automata with piecewise affine dynamics (PWA) have

- initial states and invariants given by conjunctions of linear constraints,
- flows given by affine ODEs, and
- jumps given by a guard set and linear assignments.

We divide the continuous variables into *state variables* $X = \{x_1, \ldots, x_n\}$, whose derivative is explicitly defined, and *input variables* $U = \{u_1, \ldots, u_m\}$, whose derivative is unconstrained. The input variables can be used to model nondeterminism such as open inputs to the system, approximation errors, disturbances, etc. In each location of a PWA, the continuous dynamics are given by *affine ODEs* of the form

$$\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u}, \qquad \mathbf{u} \in \mathcal{U}, \tag{14}$$

where A and B are matrices of appropriate dimension and the *input set* $\mathcal{U}$ is compact and convex. Note that $\mathcal{U}$ may be specified in the invariant. To simplify notation in this section, we assume that constants are modeled with $\mathcal{U}$, e.g., $\dot{\mathbf{x}} = A\mathbf{x}+\mathbf{b}$ with $B = I$ and $\mathcal{U} = \{b\}$. Some differential inclusions can be brought to the form of (14) by introducing auxiliary variables. The jump constraints of an edge e are defined by a *guard* set $\mathcal{G}$ and an assignment of the form

$$\mathbf{x}' = C\mathbf{x} + D\mathbf{u}, \tag{15}$$

where $\mathbf{x}'$ denotes the value of $\mathbf{x}$ after the jump, u is defined as above and C and D are matrices of appropriate dimension.

Continuous successors. We start with the basic construction, which ignores the invariant. The evolution of the input variables is described by an *input signal* $\zeta : \mathbb{R}^{\geq 0} \to \mathcal{U}$ that attributes to each point in time a value of the input $\mathbf{u}$. The input signal does not need to be continuous. A trajectory $\xi(t)$ from a state $\mathbf{x}_0$ is the solution of the differential equation (14) for initial condition $\xi(0) = \mathbf{x}_0$ and a given input signal ζ. It has the form

$$\xi_{\mathbf{x}_0,\zeta}(t) = e^{At}\mathbf{x}_0 + \int_0^t e^{A(t-s)} B\zeta(s) ds. \tag{16}$$

It consists of the superposition of the solution of the *autonomous* system, obtained for $\zeta(t) = 0$, and the *input integral* obtained for $x_0 = 0$. Let $\mathcal{X}_t$ be the states reachable in time t from any state in $\mathcal{X}_0$ and let $\mathcal{Y}_t$ be the states reachable from $\mathcal{X}_0 = \{0\}$, then (16) can be written as

$$\mathcal{X}_t = e^{At}\mathcal{X}_0 \oplus \mathcal{Y}_t. \tag{17}$$

The goal is to compute a finite sequence of sets $\Omega_0, \Omega_1, \ldots$ such that

$$\bigcup_{0 \leq t \leq T} \mathcal{X}_t \subseteq \Omega_0 \cup \Omega_1 \cup \ldots. \tag{18}$$

We present the construction of the sequence Ω_k for a fixed *time step* $\delta > 0$ such that Ω_k covers $\mathcal{X}_t$ for $t \in [k\delta, (k+1)\delta]$, as illustrated in Fig. 11. The so-called *semi-group* property of reachability says that, starting from $\mathcal{X}_s$, for any $s \geq 0$, and then waiting r time units leads to the same states as starting from $\mathcal{X}_0$ and waiting $r + s$ time units. Applying this to (17), we obtain that for any $r, s \geq 0$,

$$\mathcal{X}_{r+s} = e^{Ar}\mathcal{X}_s \oplus \mathcal{Y}_r. \tag{19}$$

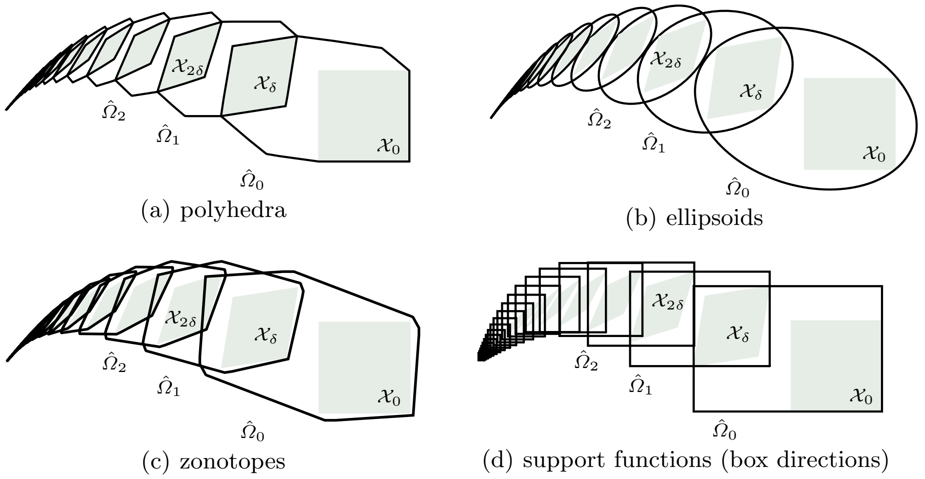


Fig. 11. A sequence of sets $\Omega_0, \Omega_1, \dots$ that covers $\mathcal{X}_t$ over a finite time horizon T. The choice of set representation for Ω_k has a substantial impact on accuracy and computational complexity

Substituting $r \leftarrow \delta$, $s \leftarrow k\delta$, we get a time discretization

$$\mathcal{X}_{(k+1)\delta} = e^{A\delta}\mathcal{X}_{k\delta} \oplus \mathcal{Y}_\delta.$$

It follows that if we have initial approximations Ω_0 and Ψ_δ such that

$$\bigcup_{0\le t\le\delta} \mathcal{X}_t \subseteq \Omega_0, \qquad \mathcal{Y}_\delta \subseteq \Psi_\delta, \tag{20}$$

then the sequence

$$\Omega_{k+1} = e^{A\delta}\Omega_k \oplus \Psi_\delta. \tag{21}$$

satisfies (18). Note that Ω_0 covers the reachable set over an interval of time $[0,\delta]$, while Ψ_δ covers the values of the input integral at a single time instant δ.

Computing initial approximations Ω_0 and Ψ_δ . The set Ω_0 needs to cover $\mathcal{X}_t$ from $t=0$ to $t=\delta$. A good starting point for such a cover is the convex hull of $\mathcal{X}_0$ and $\mathcal{X}_\delta$. One approach, shown in Fig. 12(a), is to compute the convex hull in constraint representation, and push the facets out far enough to be conservative [29]. The required values can be computed from a Taylor approximation of (16) [8], or by solving an optimization problem [18]. Note that the cost of computing the exact constraints of the convex hull can be exponential in the number of variables, which limits the scalability of this approach.

A scalable way to obtain Ω_0 is to bloat $\mathcal{X}_0$ and $\mathcal{X}_\delta$ enough to compensate for the curvature of trajectories, as illustrated in Fig. 12(b). The approach from [26] uses uniform bloating and whose approximation error is asymptotically linear in the time step δ as $\delta \to 0$. This is asymptotically optimal for any approximation

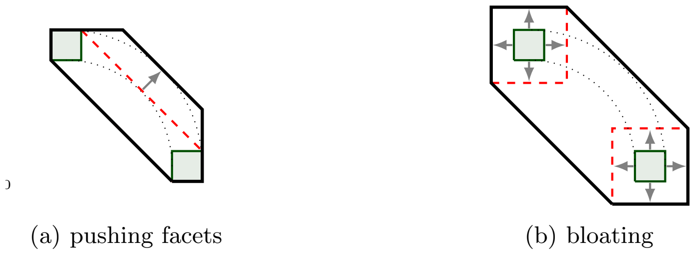


Fig. 12. An approximation Ω_0 that covers $\mathcal{X}_t$ for $t \in [0, \delta]$ can be obtained from the convex hull of $\mathcal{X}_0$ and $\mathcal{X}_\delta$ and compensating for the curvature of trajectories

containing the convex hull of $\mathcal{X}_0$ and $\mathcal{X}_\delta$ [41]. The bloating factor is derived from a Taylor approximation of (16), whose remainder is bounded using norms. To formalize the above statements, we use the following notation. Let $\|\cdot\|$ be a vector norm and let $\|A\|$ be its induced matrix norm.[1] Let $\mu(\mathcal{X}) = \max_{x\in\mathcal{X}} \|x\|$ and let $\mathcal{B}$ be the unit *ball* of the norm, i.e., the largest set $\mathcal{B}$ such that $\mu(\mathcal{B}) = 1$. For a scalar c, let $c\mathcal{X} = \{cx \mid x \in \mathcal{X}\}$.

Lemma 3. *[26] Given a set of initial states $\mathcal{X}_0$ and affine dynamics (14), let*

$$\begin{aligned}\alpha_\delta &= \mu(\mathcal{X}_0)\cdot(e^{\|A\|\delta} - 1 - \|A\|\delta),\\ \beta_\delta &= \tfrac{1}{\|A\|}\mu(B\mathcal{U})\cdot(e^{\|A\|\delta} - 1),\\ \Omega_0 &= \mathrm{chull}(\mathcal{X}_0 \cup e^{A\delta}\mathcal{X}_0) \oplus (\alpha_\delta + \beta_\delta)\mathcal{B},\\ \Psi_\delta &= \beta_\delta\mathcal{B}.\end{aligned}$$

Then $\bigcup_{0\le t\le\delta} \mathcal{X}_t \subseteq \Omega_0$ and $\mathcal{Y}_\delta \subseteq \Psi_\delta$.

Approximations and the wrapping effect. The sequence in (21) can be problematic to compute since the complexity of Ω_k may increase sharply with k. To avoid this increase in complexity, we approximate each Ω_k by a simplified set. Let Appr be an *approximation function* such that for any set $\mathcal{P}$, $\mathcal{P} \subseteq \mathrm{Appr}(\mathcal{P})$. The sequence (21) then becomes

$$\hat{\Omega}_{k+1} = \mathrm{Appr}(e^{A\delta}\hat{\Omega}_k \oplus \Psi_\delta). \tag{22}$$

The recursive application of the approximation function can lead to an exponential increase in the approximation error. This phenomenon is known in numerical analysis as the *wrapping effect* [38] and is illustrated in Fig. 13.

For affine dynamics, the wrapping effect can be avoided by combining two techniques [27]. First, the alternation of the map $e^{Ak\delta}$ with the Minkowski sum in (21) is avoided by splitting it into two sequences

$$\begin{aligned}\hat{\Psi}_{k+1} &= \mathrm{Appr}(e^{Ak\delta}\Psi_\delta) \oplus \hat{\Psi}_k, \qquad \text{with } \hat{\Psi}_0 = \{0\},\\ \hat{\Omega}_k &= \mathrm{Appr}(e^{Ak\delta}\Omega_0) \oplus \hat{\Psi}_k.\end{aligned} \tag{23}$$

[1] For example, the *infinity norm* $\|x\|_\infty = \max\{|x_1|, \dots, |x_n|\}$ induces the matrix norm $\|A\| = \max_{1\le i\le n} \sum_{j=1}^{m} |a_{ij}|$, where A is of dimension $n \times m$. Its ball $\mathcal{B}_\infty$ is a cube of side length 2.

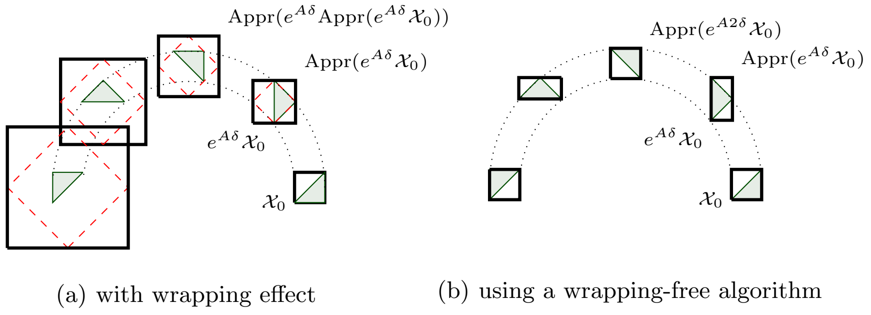

$\mathrm{Appr}(e^{A\delta}\mathrm{Appr}(e^{A\delta}\mathcal{X}_0))$
$\mathrm{Appr}(e^{A\delta}\mathcal{X}_0)$
$e^{A\delta}\mathcal{X}_0$
$\mathcal{X}_0$
(a) with wrapping effect
$\mathrm{Appr}(e^{A2\delta}\mathcal{X}_0)$
$\mathrm{Appr}(e^{A\delta}\mathcal{X}_0)$
$e^{A\delta}\mathcal{X}_0$
$\mathcal{X}_0$
(b) using a wrapping-free algorithm

Fig. 13. An example for the wrapping effect, with $e^{A\delta}$ performing a rotation of 45 degrees around the origin. The exact solution is $e^{Ak\delta}\mathcal{X}_0$ (shaded). Mapping (dashed) and then applying the approximation operator (thick) at each step leads to the wrapping effect. For visual clarity, $\mathcal{X}_0$ is used here instead of Ω_0

Second, the approximation operator is chosen such that

$$\mathrm{Appr}(\mathcal{P} \oplus \mathcal{Q}) = \mathrm{Appr}(\mathcal{P}) \oplus \mathrm{Appr}(\mathcal{Q}),$$

which is the case, e.g., for the interval hull (bounding box). Under this assumption it holds that $\hat{\Omega}_k = \mathrm{Appr}(\Omega_k)$, which means the resulting approximation is free of the wrapping effect.

Invariants. A simple but frequently sufficient heuristic to account for the invariant is to stop computing the sequence Ω_k as soon as Ω_k lies completely outside of the invariant. The computed $\Omega_0, \ldots, \Omega_{k-1}$ are then intersected with $\mathsf{Inv}(\ell)$, which produces an overapproximation of the exact solution. A more precise solution can be obtained by intersecting at each step with the set of states reachable from the invariant itself [30].

Discrete successors. Consider an edge $e = (\ell, \sigma, k)$ of a PWA, with guard set $\mathcal{G}$ and assignment

$$\mathbf{x}' = C\mathbf{x} + D\mathbf{u}.$$

Recall that $u \in \mathcal{U}$, where $\mathcal{U}$ is compact, convex and given by constraints in $Inv(\ell)$. The discrete successors of a set $\mathcal{P}$ is

$$\mathsf{post}_e(\mathcal{P}) = \big(C(\mathcal{P} \cap \mathcal{G}) \oplus D\mathcal{U}\big) \cap \mathsf{Inv}(k).$$

Set Representations Whether the presented successor operators $\mathsf{post}_\ell(\mathcal{P})$ and $\mathsf{post}_e(\mathcal{P})$ are efficient to compute, depends on the type of set used for $\mathcal{P}$ and how it is represented. We summarize some of the set representations proposed in literature. Scalable implementations and approximations need to be available for the operators in the algorithm. Using the initial approximation from Lemma 3 and the recurrence equation (23), the operators are linear map, Minkowski sum, convex hull and intersection.

Polyhedra. Figure 11(a) shows a reach set approximation computed using polyhedra. The class of polyhedra is closed under all required operations, i.e., linear map, Minkowski sum, convex hull, and intersection. However, not all of them scale well. As mentioned in Sect. 4.2, intersection is computed on $\mathcal{H}$-polyhedra and Minkowski sum on $\mathcal{V}$-polyhedra, and the result can be of exponential complexity in both forms. A polyhedral approximation for the non-scalable operations can be efficiently computed by a-priori fixing the facet normals of the result, which leads to so-called *template polyhedra.* The accuracy of the approximation can be increased by including additional directions, leading to a scalable approach [9].

Ellipsoids. A scalable reachability algorithm for affine dynamics is obtained for ellipsoids [39], see Fig. 11(b). An *ellipsoid* $\mathcal{E}(\mathbf{c}, Q) \subseteq \mathbb{R}^n$ is represented by a center $\mathbf{c} \in \mathbb{R}^n$ and a positive definite[2] matrix $Q \in \mathbb{R}^{n\times n}$,

$$\mathcal{E}(\mathbf{c}, Q) = \big\{\mathbf{x} \bigm| (\mathbf{x}-\mathbf{c})^{\mathsf{T}} Q^{-1}(\mathbf{x}-\mathbf{c}) \leq 1\big\}.$$

Deterministic affine transforms can be computed efficiently for ellipsoids. However, ellipsoids are not closed under Minkowski sum, convex hull, nor intersection. One therefore suffers from the wrapping effect unless $B\mathcal{U}$ is a singleton. Efficient approximations are available for Minkowski sum, convex hull, and special cases of intersection, but the computation of discrete successors can be problematic in terms of accuracy. For an implementation, see [40].

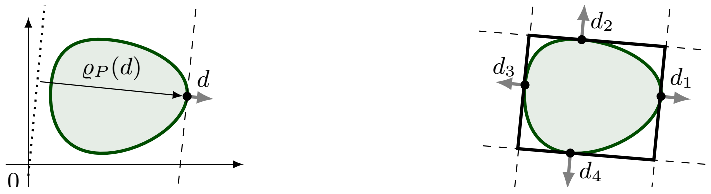
Zonotopes. Zonotopes are a subclass of central-symmetric polytopes that has been used successfully for reachability analysis [26, 3], see Fig. 11(c). A *zonotope* $\mathcal{P} \subseteq \mathbb{R}^n$ is defined by a center $\mathbf{c} \in \mathbb{R}^n$ and generators $\mathbf{v}_1, \ldots, \mathbf{v}_k \in \mathbb{R}^n$ as

$$\mathcal{P} = \big\{\mathbf{c} + \sum\nolimits_{i=1}^{k} \alpha_i \mathbf{v}_i \bigm| \alpha_i \in [-1, 1]\big\}.$$

Affine transformations and Minkowski sum can be computed efficiently for zonotopes. Since zonotopes are closed under Minkowski sum, it is straightforward to devise an approximation operator Appr that distributes over Minkowski sum and use the wrapping-free sequence (23). Zonotopes are neither closed under convex hull, nor under intersection. But efficient approximations exists, and the accuracy of approximating the convex hull in the above reachabililty algorithm can be improved by taking smaller time steps. However, the lack of accuracy in intersections can make the computation of discrete successors with zonotopes problematic. In special cases it can be advantageous to use an approach called *continuization* to avoid the intersection operation, see [2].

Support functions. The support function of a convex set represents the set exactly. Support functions lead to very scalable algorithms since linear map,

[2] A matrix Q is positive definite iff it is symmetric and $\mathbf{x}^{\mathsf{T}} Q\mathbf{x} > 0$ for all $\mathbf{x} \neq 0$.



(a) support function in direction d (b) outer approximation

Fig. 14. Evaluating the support function in a set of directions gives a polyhedral outer approximations that can be computed very efficiently

Minkowski sum, and convex hull correspond to simple operations on vectors and scalars [42, 30].

The *support function* $\varrho_{\mathcal{P}} : \mathbb{R}^n \to \mathbb{R}$ of a nonempty, closed, bounded, and convex set $\mathcal{P}$ is $\varrho_{\mathcal{P}}(\mathbf{d}) = \max\{\mathbf{d}^{\mathsf{T}}\mathbf{x} \mid \mathbf{x} \in \mathcal{P}\}$. It attributes to every *direction* $d \in \mathbb{R}^n$ the position of the tangent halfspace in that direction, see Fig. 14(a). The values of the support function over a set of directions $\mathcal{D} \subseteq \mathbb{R}^n$ define an *outer approximation*

$$\lceil \mathcal{P} \rceil_{\mathcal{D}} = \bigcap_{\mathbf{d} \in \mathcal{D}} \left\{ \mathbf{d}^{\mathsf{T}} x \leq \varrho_{\mathcal{P}}(\mathbf{d}) \right\}.$$

If $\mathcal{D} = \mathbb{R}^n$ or $\mathcal{D}$ is the ball of a norm, then $\lceil \mathcal{P} \rceil_{\mathcal{D}} = \mathcal{P}$. If $\mathcal{D}$ is a finite set of directions, the outer approximation is a polyhedron, see Fig. 14(b). If $\mathcal{D}$ consists of the positive and negative axis directions, the result is an interval hull (bounding box), see Fig. 11(d). If the goal is to compute an outer approximation of a given accuracy, one does not escape the curse of dimensionality: an outer approximation with an error of ε in n dimensions requires $\mathcal{O}(1/\varepsilon^{n-1})$ directions. However, even a small number of directions can lead to reachability results with an acceptable approximation error [24].

Computing the support function of the sequence (23) for a given direction can be done very efficiently even without the approximation operator Appr [30]. Linear map, Minkowski sum, and convex hull are easily computed with support functions. The intersection operation is more complex, and can be formulated as an optimization problem [30]. Switching to $\mathcal{H}$-polyhedra (the outer approximation) before intersection operations can avoid scalability problems, but leads to an overapproximation error [24].

Clustering The accuracy of the approximation in Lemma 3 depends on the size of the time step. This property, common to all approaches cited in Sect. 4.3, points to a potential bottleneck: To achieve a desired accuracy, one may end up with a large number of sets to cover the required time horizon. In the next iteration of the fixed point computation, each one of these sets may become the initial set of yet another sequence, easily leading to an exponential increase in the number of sets. If only very few of these sets intersect with the guard sets, the discrete successor computation acts as a filter that might just keep the number of sets manageable. But this is not the case in general; note that these sets necessarily overlap. To prevent an explosion in the number of sets, a common approach is to cluster together all sets that intersect with the same guard [30]. The clustering operation, e.g., taking the convex hull, can itself be costly and adds to the approximation error in a way that is not easy to quantify. An approach to obtain a suboptimal number of clusters for a given error bound is presented in [25].

4.4 Nonlinear Dynamics

We give only a very brief overview of techniques that deal with nonlinear dynamics

$$\dot{\mathbf{x}} = f(\mathbf{x}),$$

where f is usually assumed to be globally Lipschitz continuous.

Linearization. One way to deal with nonlinear dynamics is to approximate them with affine dynamics $\dot{\mathbf{x}} = A\mathbf{x} + \mathbf{u}, \mathbf{u} \in \mathcal{U}$ and then use reachability algorithms for affine dynamics. First, the states are confined to a bounded domain $\mathcal{S}$. This could be the invariant in a location, or $\mathcal{S}$ can be derived from suitable bounds around a given set of initial states. Then, a suitable matrix A and vector $\mathbf{b}$ are chosen. For example, linearizing $f(x)$ around a point $\mathbf{x}_0 \in \mathcal{S}$ gives matrix elements $a_{ij} = \frac{\partial f_i}{\partial x_j}\big|_{\mathbf{x}=\mathbf{x}_0}$ and $\mathbf{b} = f(\mathbf{x}_0) - A\mathbf{x}_0$. Finally, one derives a set $\mathcal{U}_\epsilon$ that bounds the error such that for all $x \in \mathcal{S}$, $f(x) - (Ax + b) \in \mathcal{U}_\epsilon$. Such bounds can be obtained using, e.g., interval arithmetic or optimization techniques. The states reachable using the affine dynamics $\dot{x} = Ax + u,\ u \in \mathcal{U}_\epsilon \oplus \{b\}$ cover those of the original nonlinear dynamics.

The accuracy of the linearization depends on the size of the domain $\mathcal{S}$ and can be increased by partitioning $\mathcal{S}$ into smaller parts. This process is known as *phase portrait approximation* [34]. It can be of use even when dealing with purely continuous dynamical systems, and is also referred to as *hybridization* [7].

Polynomial Approximations. If the dynamics are polynomial, bringing them to Bernstein form allows one to compute conservative approximations of successors sets in polynomial form [19, 47]. Another approach is to use *Taylor models*, which are polynomial approximations of a functions that are derived from a higher-order Taylor expansion and an interval bound on the remainder. The resulting ODE can be solved by iterative approximations using the Picard operator. The reachable states are approximated by sets that are polyhedra [49] or polynomial images of intervals [17]. A similar approach uses polynomial images of zonotopes, which are themselves images of intervals [1]. Since polynomial images of intervals are generally not closed under intersection, the accuracy may be diminished when computing discrete successors.

5 Conclusions

Systems with mixed continuous-discrete dynamics can exhibit complex behaviors that are difficult to analyze and predict, even for small systems with only a handful of variables. If safety or performance is critical, one would like to verify that the systems behaves according to the specification. Hybrid automata provide a rigorous mathematical formalism for describing and verifying such systems. Certain types of specifications can also be described in this form, as *monitor automata* that are run in parallel with the rest of the system, with an error location that is reachable if the specification is violated.

The most basic way to analyze the behavior of a hybrid automaton is to pick an initial state and numerically compute an approximation of one of its runs. By computing a sufficiently large number of such runs, sampling a variety of initial states and other sources of nondeterminism, one hopes to get a fairly good idea about the system. But this is not exhaustive and critical behavior may be missed. Set-based reachability, in the form presented in this chapter, is an extension of numerical simulation that establishes a conservative cover of all possible runs. If a safety specification is satisfied by the cover, one can be sure that none of the runs violates the specification. If the cover violates the specification, one either needs to tune the analysis parameters in order to obtain a closer approximation, try to confirm the violation with numerical simulation, or resort to alternative techniques.

The biggest drawback of set-based reachability is the computational cost, which depends on the number of variables in the system and the complexity of the dynamics. Scalable algorithms are known for systems with piecewise affine dynamics, but the trade-off between approximation accuracy and computational cost remains a challenge. Recent progress for systems with nonlinear dynamics has also lead to encouraging results for more and more complex systems. As reachability techniques mature, it remains to figure out how to best integrate them in the design and engineering process, connect them to existing models, and establish suitable specifications.

References

1. M. Althoff. Reachability analysis of nonlinear systems using conservative polynomialization and non-convex sets. In *Hybrid systems: computation and control (HSCC'13)*, pages 173–182. ACM, 2013.
2. M. Althoff and B. H. Krogh. Avoiding geometric intersection operations in reachability analysis of hybrid systems. In *Hybrid Systems: Computation and Control (HSCC'12)*, pages 45–54. ACM, 2012.
3. M. Althoff, B. H. Krogh, and O. Stursberg. Analyzing reachability of linear dynamic systems with parametric uncertainties. In A. Rauh and E. Auer, editors, *Modeling, Design, and Simulation of Systems with Uncertainties.* Springer, 2011.
4. R. Alur. Formal verification of hybrid systems. In S. Chakraborty, A. Jerraya, S. K. Baruah, and S. Fischmeister, editors, *EMSOFT*, pages 273–278. ACM, 2011.

5. R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.
6. R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, LNCS 736, pages 209–229. Springer, 1993.
7. E. Asarin, T. Dang, and A. Girard. Hybridization methods for the analysis of nonlinear systems. *Acta Inf.*, 43(7):451–476, 2007.
8. E. Asarin, T. Dang, O. Maler, and O. Bournez. Approximate reachability analysis of piecewise-linear dynamical systems. In *Hybrid Systems: Computation and Control (HSCC'00)*, volume 1790 of *LNCS*, pages 20–31. Springer, 2000.
9. E. Asarin, T. Dang, O. Maler, and R. Testylier. Using redundant constraints for refinement. In *Automated Technology for Verification and Analysis*, pages 37–51. Springer, 2010.
10. R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.
11. O. Bouissou, A. Chapoutot, S. Mimram, and B. Strazzulla. Set-based simulation for design and verification of simulink models. In *Embedded Real Time Software and Systems (ERTS'14)*, 2014.
12. O. Bouissou, S. Mimram, and A. Chapoutot. Hyson: Set-based simulation of hybrid systems. In *RSP*, pages 79–85. IEEE, October 2012.
13. D. Brück, H. Elmqvist, S. E. Mattsson, and H. Olsson. Dymola for multi-engineering modeling and simulation. In *Proceedings of Modelica*, 2002.
14. R. P. Canale and S. C. Chapra. Numerical methods for engineers. *Mc Graw Hill, New York*, 1998.
15. C. G. Cassandras and J. Lygeros. *Stochastic hybrid systems*. CRC Press, 2006.
16. C. Chase, J. Serrano, and P. J. Ramadge. Periodicity and chaos from switched flow systems: contrasting examples of discretely controlled continuous systems. *Automatic Control, IEEE Transactions on*, 38(1):70–83, 1993.
17. X. Chen, E. Ábrahám, and S. Sankaranarayanan. Taylor model flowpipe construction for non-linear hybrid systems. In *RTSS*, pages 183–192. IEEE Computer Society, 2012.
18. A. Chutinan and B. H. Krogh. Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In F. W. Vaandrager and J. H. van Schuppen, editors, *HSCC*, volume 1569 of *LNCS*, pages 76–90. Springer, 1999.
19. T. Dang and R. Testylier. Reachability analysis for polynomial dynamical systems using the bernstein expansion. *Reliable Computing*, 17(2):128–152, 2012.
20. A. Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *Computer Aided Verification*, pages 167–170. Springer, 2010.
21. J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity-the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.
22. G. E. Fainekos, A. Girard, and G. J. Pappas. Temporal logic verification using simulation. In *Formal Modeling and Analysis of Timed Systems*, pages 171–186. Springer, 2006.
23. G. Frehse. PHAVer: algorithmic verification of hybrid systems past HyTech. *STTT*, 10(3):263–279, 2008.
24. G. Frehse, C. L. Guernic, A. Donzé, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable verification of hybrid systems. In G. Gopalakrishnan and S. Qadeer, editors, *CAV*, LNCS. Springer, 2011.

25. G. Frehse, R. Kateja, and C. Le Guernic. Flowpipe approximation and clustering in space-time. In *Hybrid systems: computation and control (HSCC'13)*, pages 203–212. ACM, 2013.
26. A. Girard. Reachability of uncertain linear systems using zonotopes. In M. Morari and L. Thiele, editors, *HSCC*, volume 3414 of *LNCS*, pages 291–305. Springer, 2005.
27. A. Girard, C. L. Guernic, and O. Maler. Efficient computation of reachable sets of linear time-invariant systems with inputs. In J. P. Hespanha and A. Tiwari, editors, *HSCC*, volume 3927 of *LNCS*, pages 257–271. Springer, 2006.
28. A. Girard and G. Zheng. Verification of safety and liveness properties of metric transition systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 11(S2):54, 2012.
29. M. R. Greenstreet. Verifying safety properties of differential equations. In *Computer Aided Verification*, pages 277–287. Springer, 1996.
30. C. L. Guernic and A. Girard. Reachability analysis of hybrid systems using support functions. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *LNCS*, pages 540–554. Springer, 2009.
31. N. Halbwachs, Y.-E. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In *International Static Analysis Symposium, SAS'94*, Namur (Belgium), September 1994.
32. T. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, pages 110–122, 1997.
33. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. In O. Grumberg, editor, *CAV*, volume 1254 of *LNCS*, pages 460–463. Springer, 1997.
34. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *IEEE Transactions on Automatic Control*, 43:540–554, 1998.
35. T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57:94–124, 1998.
36. P.-H. Ho. *Automatic Analysis of Hybrid Systems*. PhD thesis, Cornell University, Aug. 1995. Technical Report CSD-TR95-1536.
37. A. A. Julius, G. E. Fainekos, M. Anand, I. Lee, and G. J. Pappas. Robust test generation and coverage for hybrid systems. In *Hybrid Systems: Computation and Control*, pages 329–342. Springer, 2007.
38. W. Kühn. Rigorously computed orbits of dynamical systems without the wrapping effect. *Computing*, 61(1):47–67, 1998.
39. A. B. Kurzhanski and P. Varaiya. *Dynamics and Control of Trajectory Tubes*. Springer, 2014.
40. A. A. Kurzhanskiy and P. Varaiya. Ellipsoidal toolbox (et). In *Decision and Control, 2006 45th IEEE Conference on*, pages 1498–1503. IEEE, 2006.
41. C. Le Guernic. *Reachability analysis of hybrid systems with linear continuous dynamics*. PhD thesis, Université Grenoble 1 - Joseph Fourier, 2009.
42. A. V. Lotov, V. A. Bushenkov, and G. K. Kamenev. *Interactive Decision Maps*, volume 89 of *Applied Optimization*. Kluwer, 2004.
43. O. Maler. Algorithmic verification of continuous and hybrid systems. In *Int. Workshop on Verification of Infinite-State System (Infinity)*, 2013.
44. MapleSoft. Maplesim 7: Advanced system-level modeling. `http://www.maplesoft.com/products/maplesim`, 2015.
45. MathWorks. Mathworks simulink: Simulation et model-based design, Mar. 2014. `www.mathworks.fr/products/simulink`.

46. S. E. Mattsson, H. Elmqvist, and M. Otter. Physical system modeling with modelica. *Control Engineering Practice*, 6(4):501–510, 1998.
47. P. Prabhakar and M. Viswanathan. A dynamic algorithm for approximate flow computations. In E. Frazzoli and R. Grosu, editors, *HSCC*, pages 133–142. ACM, 2011.
48. W. H. Press. *Numerical recipes 3rd edition: The art of scientific computing.* Cambridge University Press, 2007.
49. S. Sankaranarayanan, T. Dang, and F. Ivančić. Symbolic model checking of hybrid systems using template polyhedra. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 188–202. Springer, 2008.
50. P. Tabuada. *Verification and Control of Hybrid Systems: A Symbolic Approach.* Springer, 2009.
51. F. Zhang, M. Yeddanapudi, and P. Mosterman. Zero-crossing location and detection algorithms for hybrid system simulation. In *IFAC World Congress*, pages 7967–7972, 2008.