International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-023-00707-0

GENERAL

Regular

On the applicability of hybrid systems safety verification tools from the automotive perspective

Stefan Schupp[1] · Erika Ábrahám[1] · Md Tawhid Bin Waez[2] · Thomas Rambow[2] · Zeng Qiu[2]

Accepted: 30 May 2023

Abstract

Traditionally, extensive vehicle testing is applied to assure the robustness and safety of automotive systems. This approach is highly challenged by increasing system complexity. Formal verification lends a powerful framework for model-based safety assurance, but due to the mixed discrete–continuous behavior of automotive systems, traditional tools for discrete program verification are helpful but not sufficient.

In academia, during the last two decades new approaches arose for the formal verification of such mixed discrete-continuous systems. However, the industry is not fully aware of this development, the tools are seldom tried and their applicability is not well examined. In a Ford–RWTH research alliance project, we aimed at evaluating the potential of knowledge and technology transfer in this area.

This paper has two main objectives. Firstly, we want to report on the state-of-the-art in the above-mentioned academic development in a generally understandable form, targeted to interested potential users. Secondly, we want to share our observations after testing different available tools for their applicability and usability in the automotive sector and as a conclusion devise some recommendations.

Keywords Hybrid systems · Reachability analysis · Formal methods · Safety verification

1 Introduction

Safety is critical for automotive industry. However, the complexity of automotive systems puts serious challenges on their safety assurance. The increasing usage of digital controllers and software for driving assistance and autonomous functionalities poses high demands on safety verification. While testing-based approaches may provide intuition on the system behavior and allow deducing system properties, these methods cannot provide guarantees as rigorous *safety verification* can provide.

Formal methods for providing mathematically rigorous guarantees of certain system properties have a long history in computer science, mostly in academic contexts. Over the recent years, one can observe an increased usage of formal methods also in the automotive context [33, 35, 36].

A group of researchers from Ford Motor Company and RWTH Aachen University demonstrated (by applying on an open-loop discrete next-generation control software) that formal verification for discrete systems is capable of finding issues that have not been detected in standard automotive model-in-the-loop (MIL) testing and software-in-the-loop (SIL) testing, and thus that formal specification and verification has the potential to greatly improve the quality of products [15, 16, 37].

That success motivated us to investigate further into the verification of closed-loop requirements for automotive control software. The usage of digital controllers in a continuous environment has resulted in the demand for the verification of so-called *hybrid systems*, where the word *hybrid* refers to their mixed discrete–continuous nature. Different automated formal verification techniques for safety properties ("something bad never happens") of hybrid systems have been proposed in academia, based on different frameworks. These techniques can be used to analyze different types of systems, using fully automated or interactive methods, considering executions of arbitrary length or executions restricted to a bounded horizon, computing exact or overapproximative results.

Unlike any other industry, both safety assurance and extreme cost-effectiveness are key requirements for business survivability in automotive product development. However,

✉ S. Schupp
stefan.schupp@cs.rwth-aachen.de

[1] RWTH Aachen University, Aachen, Germany

[2] Ford Motor Company, Dearborn, MI, USA

Published online: 30 June 2023

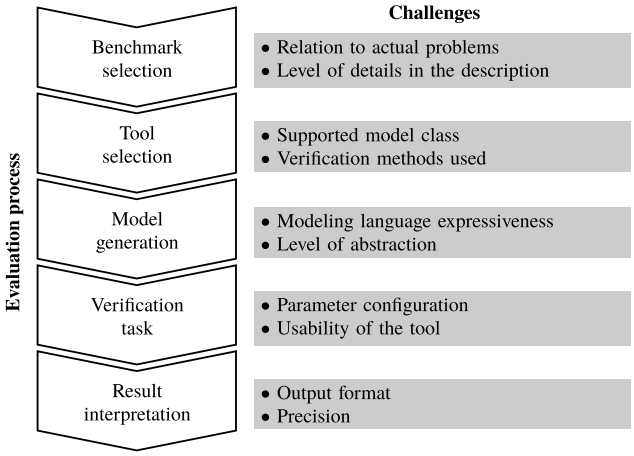


Fig. 1 Evaluation process stages and challenges

due to the size and complexity of automotive systems, formal methods for the analysis of even only discrete systems are typically applied at unit level but not yet at system level, where hybrid behavior is present. Stronger than in academic research, in automotive product development the criterion of usability is almost as critical as analytical capabilities. In order to understand whether the academic methods and tools for the formal analysis of hybrid systems could be of use also in an industrial automotive context, we need to know their capabilities and evaluate them with respect to a variety of aspects.

This work has two main objectives. Firstly, we aim at providing an *overview* over existing tools and their analytical functionalities, relate the methods behind them and indicate what a user can expect from using a certain approach. Secondly, we want to give insights into their *usage* including the task of modeling, the execution of the verification process and the interpretation of the results. We aim at providing general observations regarding the applicability of these tools in an industrial context, and highlight the challenges one has to face during this process.

Outline. The paper has two parts. In the first part we describe the used technologies, starting with an overview on existing modeling paradigms and formal verification methods for hybrid systems in Sect. 2, followed by a description of our tool selection in Sect. 3.

The second part is devoted to evaluation. In the Sects. 4–7 we discuss the different stages of our evaluation, shown in Fig. 1: (i) the challenges that we faced, which can be generalized to challenges engineersface when analyzing a hybrid system, and (ii) general observations on the corresponding processes, which can guide users towards successful verification. The stages we distinguish are the benchmark selection in Sect. 4, the model generation in Sect. 5, the execution of the verification tasks in Sect. 6, and the observed results in Sect. 7. In Sect. 8 we present our experiences on applying some of the tools on a more realistic case study from automotive industry. Finally, in Sect. 9 we conclude the paper and make some remarks for future research and development.

Part 1: Technology

2 Hybrid systems and their formal modeling

Discrete systems change their state during system execution in no time, *dynamical* systems evolve continuously over time, and *hybrid* systems have both discrete and continuous components. A typical hybrid system example is a digitally controlled physical system, where the digital controller can discretely change the system state, and where between two discrete changes the values of some physical quantities like speed, temperature, or pressure evolve continuously over time.

The first step to enable computer support for the analysis of hybrid systems is *modeling*, i.e., the encoding of the system behavior in a given formalism. A popular modeling formalism for hybrid systems are *hybrid automata* [30, 31]. Hybrid automata extend discrete transition systems, defined over a set of *locations* (also called *control modes*) and a set of variables. Only continuous variables are supported; discrete variables can be modeled as continuous variables with zero derivatives. The system state is fixed by the current location and the current variable values, and can change by taking a *discrete transition* (*jump*), leading from the current location to another one, potentially guarded by some enabling condition, and potentially changing the system state also by modifying the values of certain variables. Hybrid automata extend such classical discrete transition systems with a dynamic (continuous) behavior: while the control stays in a location, *time evolution* (*flows*) let the values of the variables evolve continuously according to some dynamics, which is specified by *ordinary differential equations* (*ODEs*). *Invariants* can be attached to the locations to restrict the maximal duration of time elapse, modeling the fact that some discrete events will occur within certain deadlines. Semantically, time can pass only as long as the current location's invariant holds.

Before its formal definition, we first illustrate hybrid automata on an example.

Example 1 (Bouncing ball)
In the classical *bouncing ball* example, a ball is dropped from some initial height with zero initial velocity. Gravity accelerates the ball towards the earth and it falls until it hits the ground. Then it bounces back into the air, rises until its velocity becomes zero, and starts to fall again. Upon bouncing, the ball loses a fraction of its kinetic energy.

A hybrid automaton model of the bouncing ball is illustrated graphically in Fig. 2. The model has a single location

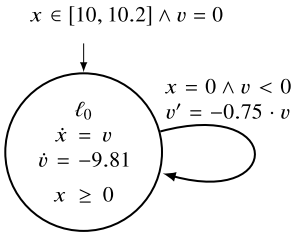


Fig. 2 A graphical representation of a hybrid automaton model for the bouncing ball

ℓ_0 and two variables: x with any initial value between 10 and 10.2 models the vertical position (height) and v with initial value 0 the vertical velocity of the ball.

The flow in ℓ_0 is specified by the differential equations $\dot{x} = v$ and $\dot{v} = -9.81$, with the gravitational force as the only influence on the speed of the ball. This bouncing is represented by the only jump with guard $x = 0 \wedge v < 0$ (that means bouncing only occurs when the ball falls from above and reaches the ground) and reset $v' = -0.75 \cdot v$ (i.e., the sign of the velocity is inverted and the velocity is dampened by a constant factor 0.75). The invariant $x \geq 0$ in ℓ_0 models that the bouncing indeed happens when the ball falls ($\dot{x} < 0$) and reaches the ground ($x = 0$). At this point, further time elapse would violate the invariant, therefore a jump needs to be taken. The only available jump is enabled, i.e., its guard is true, therefore it can be taken. Taking it will change the sign of the derivative of x by the reset $v' = -0.75 \cdot v$, such that time can pass further and the ball can raise and fall again. Note that directly after bouncing only time can pass, because $v > 0$ holds and therefore the jump's guard is not satisfied.

While the above example model is deterministic, in general, hybrid automata might be nondeterministic. When several jumps are enabled simultaneously, any of them can be chosen *nondeterministically* for execution. Per default, enabled jumps may but do not have to be executed, potentially leading to continuous nondeterminism between flows and jumps. Some formalisms support also *urgent* jumps, whose enabledness forces a discrete step.

Next we formalize the above notions. The following simplified syntactical definition of hybrid automata is sufficient for our purposes; more complex formalisms exist to allow compositional or hierarchical modeling with different communication mechanisms [25].

Definition 1 (Hybrid automata: Syntax [30])
A *hybrid automaton* is a tuple $\mathcal{H} = (Loc, Var, Flow, Inv, Edge, Init)$ consisting of the following components:

- Loc is a finite set of *locations* or *control modes*.
- $Var = \{x_1, \ldots, x_d\}$ is a finite ordered set of real-valued *variables*; we also use vector notation $x = (x_1, \ldots, x_d)$ and call d the *dimension* of $\mathcal{H}$. We define $\dot{Var} = \{\dot{x}_1, \ldots, \dot{x}_d\}$ (to represent first derivatives) and $Var' = \{x'_1, \ldots, x'_d\}$ (to describe discrete successors). Let $Pred_X$ denote the set of all predicates with free variables from a set X.
- $Flow : Loc \to Pred_{Var \cup \dot{Var}}$ specifies for each location its *flow* or *dynamics*.
- $Inv : Loc \to Pred_{Var}$ specifies location *invariants*.
- $Edge \subseteq Loc \times Pred_{Var} \times Pred_{Var \cup Var'} \times Loc$ is a finite set of *discrete transitions* or *jumps*. For a jump $(\ell_1, g, r, \ell_2) \in Edge$, ℓ_1 is its *source* location, ℓ_2 is its *target* location, g specifies the jump's *guard* and r its *reset* predicate, where primed variables represent the state after the step.
- $Init : Loc \to Pred_{Var}$ defines *initial* predicates.

Next we define the formal semantics of hybrid automata, i.e., its execution. Let $\mathbb{R}$ denote the set of all real numbers, $\mathbb{R}_{\geq 0}$ the nonnegative reals, $\mathbb{N}$ the natural numbers (including zero) and $\mathbb{Z}$ the integers. *States* of a d-dimensional hybrid automaton with variables $Var = \{x_1, \ldots, x_d\}$ are pairs (ℓ, ν), where $\ell \in Loc$ is the current location and $\nu = (\nu_1, \ldots, \nu_d) \in \mathbb{R}^d$ specifies the current values of the variables (ν_i is the value of x_i). For $g \in Pred_{Var}$ and $\nu = (\nu_1, \ldots, \nu_d) \in \mathbb{R}^d$, we write $\nu \models g$ to denote that substituting ν_i for each x_i evaluates g to true; $\nu, \nu' \models r$ for $g \in Pred_{Var \cup Var'}$ is defined similarly, substituting x_i as ν_i and x'_i as ν'_i; the case $\nu, \dot{\nu} \models f$ for $f \in Pred_{Var \cup \dot{Var}}$ is analogous.

A state (ℓ, ν) is *initial* if it satisfies both the initial condition and the invariant of location ℓ, i.e., $\nu \models Init(\ell) \wedge Inv(\ell)$. State changes are due to time steps or discrete steps.

Time steps (*flows*) model the passage of time: while staying in a location, the values of the variables evolve continuously according to a function which satisfies the flow condition of the current location. Furthermore, the invariant of the location must not be violated during the whole time step. Given a set of states, the states which can be visited from it via time evolution according to the flow in the given location form a *flowpipe*. When flows define constant derivatives for all variables then we talk about *linear behavior*. When flows are described by linear predicates (i.e., linear differential equations) we talk about *linear dynamics*, and in the case of more expressive predicates (involving, e.g., polynomials or trigonometric functions) about *nonlinear dynamics*.

Discrete steps (*jumps*) follow a discrete transition, moving the control from one location to another, given that the jump's guard is satisfied in the predecessor state; the successor state, resulting from variable resets according to the reset predicate, must satisfy the invariant of the target location.

The *transition relation* in the form of time and discrete steps is formalized by two operational semantics rules. A set of assumptions needs to be satisfied to execute a step; notationally, these are listed above a horizontal line. Below the line the transition itself is specified, i.e., how the step changes the state if it is executed. The name of the rule is written on the right of the rule.

Definition 2 (Hybrid automata: Semantics)
The *one-step operational semantics* of a hybrid automaton $\mathcal{H} = (Loc, Var, Flow, Inv, Edge, Init)$ of dimension d is speci-

Table 1 Decidability results for subclasses of hybrid automata; *rectangular conditions* are conjunctions of inequalities $x_i \sim c$ with $x_i \in Var$, $c \in \mathbb{Z}$ and $\sim \in \{>, \geq, =, \leq, <\}$; *rectangular derivatives and resets* are similar but use dotted ($\dot{x}_i \sim c$) resp. primed ($x_i' \sim c$) variables; *linear derivatives* are conjunctions of equalities $\dot{x}_i = e$ with $\dot{x}_i \in \dot{Var}$ and e a linear expression over Var; *linear conditions* are conjunctions of linear (in)equalities over Var; *linear resets* are conjunctions of (in)equalities $x_i' \sim e$ with $x_i' \in Var'$, $\sim \in \{>, \geq, =, \leq, <\}$ and e a linear expression over Var; *arbitrary* components may use any arithmetic formulas using addition, multiplication, trigonometric, exponential and logarithmic functions, etc.

Hybrid automata model subclass	Derivatives	Conditions	Resets	Decidability of bounded reachability	Decidability of unbounded reachability
Timed automata	constant 1	rectangular	resets to 0	✓	✓
Initialized rectangular automata	rectangular	rectangular	rectangular	✓	✓
	reset forced when derivative changes				
Rectangular automata	rectangular	rectangular	rectangular	✓	×
Linear hybrid automata I	rectangular	linear	linear	✓	×
Linear hybrid automata II	linear	linear	linear	×	×
Nonlinear hybrid automata	arbitrary	arbitrary	arbitrary	×	×

fied by the following rules:

$$\frac{\begin{array}{c}\ell \in Loc \quad \nu, \nu' \in \mathbb{R}^d \quad t \in \mathbb{R}_{\geq 0} \quad f : [0,t] \to \mathbb{R}^d \\ df/dt = \dot{f} : (0,t) \to \mathbb{R}^d \quad f(0) = \nu \quad f(t) = \nu' \\ \forall t' \in (0,t).\ f(t'), \dot{f}(t') \models Flow(\ell) \\ \forall t' \in [0,t].\ f(t') \models Inv(\ell)\end{array}}{(\ell, \nu) \xrightarrow{t} (\ell, \nu')}\ \texttt{Flow}$$

$$\frac{\begin{array}{c}e = (\ell, g, r, \ell') \in Edge \quad \nu, \nu' \in \mathbb{R}^d \\ \nu \models g \quad \nu, \nu' \models r \quad \nu' \models Inv(\ell')\end{array}}{(\ell, \nu) \xrightarrow{e} (\ell', \nu')}\ \texttt{Jump}$$

A *path* of $\mathcal{H}$ is a (finite or infinite) sequence $(\ell_0, \nu_0) \to (\ell_1, \nu_1) \to (\ell_2, \nu_2) \ldots$ of $\mathcal{H}$-states $(\ell_i, \nu_i) \in Loc \times \mathbb{R}^d$ connected by time or discrete steps $\to = (\bigcup_{t \in \mathbb{R}_{\geq 0}} \xrightarrow{t}) \cup (\bigcup_{e \in Edge} \xrightarrow{e})$ such that $\nu_0 \models Inv(\ell_0)$; we call a path *initial* if additionally $\nu_0 \models Init(\ell_0)$. A state (ℓ, ν) of $\mathcal{H}$ is *reachable* (in $\mathcal{H}$) if there is an initial finite path $(\ell_0, \nu_0) \to \cdots \to (\ell, \nu)$ of $\mathcal{H}$.

Given a hybrid automaton $\mathcal{H}$ and a subset S of its states, the *reachability problem* is the problem to decide whether there exists a state $s \in S$ that is reachable in $\mathcal{H}$. According to the shapes of the flow, invariant, guard and reset conditions, different subclasses differ in their expressivity (i.e., the type of system that can be modeled) and the decidability of the reachability problem (see Table 1). In this work we focus on linear hybrid automata II and nonlinear hybrid automata allowing linear and nonlinear ODEs in the models, and do not cover algorithms and tools for more restricted classes like timed automata.

If a state is reachable in a model then a path leading to it is a *witness* (of reachability); if S represents unsafe states then such a path is called a *counterexample* (to safety). As the reachability problem is undecidable for hybrid automata, most methods *overapproximate* reachability. Overapproximative analysis might detect *spurious* counterexamples in the overapproximation for which there is no corresponding path in the model.

At this place we do not discuss further modeling formalisms, as most of the tools we consider in this paper use an input language based on hybrid automata; some others use similar notions but assume known solutions to the flow conditions in form of *difference equations*; also in use are axiomatic definitions for theorem proving and *hybrid Petri nets* [5] that extend Petri nets by *continuous places* to model dynamic behavior.

3 Tool selection

In this section we describe four safety verification techniques for hybrid systems and our selection of tools implementing them.

3.1 Challenges

The research field of hybrid systems safety verification is relatively young. There exist different algorithms and academic tools but their number is (yet) much lower than for the analysis of discrete systems. Since 2017 a friendly competition [1] for hybrid systems safety verification tools is part of the Applied Verification for Continuous and Hybrid Systems (ARCH) workshop series. This competition offers different tracks according to model categories, for instance, *piecewise linear hybrid systems* or *nonlinear continuous systems*. The

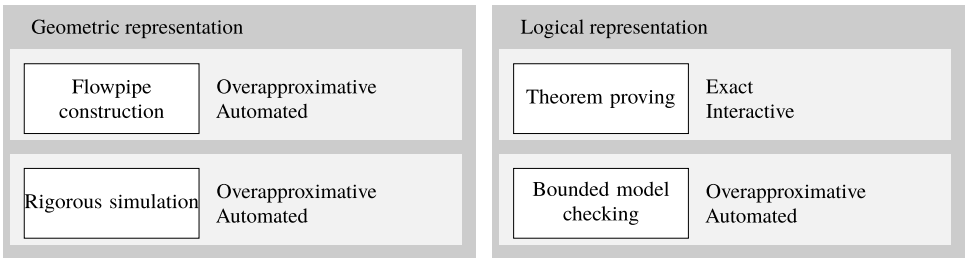


Fig. 3 Characteristics of different hybrid system safety verification approaches

results from this competition help to identify actively developed tools, determine the model types they support, and get an impression on efficiency. Different tools use different methodologies, one being more appropriate for certain purposes than others, but the choice of methodology might be challenging for users who are not familiar with the underlying verification techniques. Furthermore, even tools using similar ideas have different strengths and weaknesses, and their efficiency on concrete problems cannot be predicted.

Since exact computations are computationally expensive, most tools sidestep to floating-point computations, which might introduce numerical imprecision due to rounding. Theoretically, rounding errors can be quantified, which allows numeric approaches to provide provably correct results at the cost of additional overapproximation. One such approach was implemented in the bounded model checking tool HSolver [42]. However, most tools apply inexact computations without costly error quantification, thus without any guarantee for correct results. One exception is theorem proving, which is less automated but provides certificates in terms of proofs; also the core of most theorem provers is verified, such that bugs in the tools are unlikely. As a consequence, theorem proving-based approaches require the user to provide overapproximations for undecidable dynamics such as dynamics described by transcendental functions (e.g., $\dot{x} = \cos(y)$ where x, y are variables).

In general, *documentation* and *usability* also pose a challenge, especially for prototype tools for proof of concept. For academic tools (which are all tools in this field), usability is not the foremost criterion as tools usually are written to show the general feasibility of a certain approach or idea rather than providing a user-friendly tool.

With the increasing appearance of so-called *repeatability evaluations* for academic conferences, in which peer-reviewers try to use a certain tool to replicate the achieved results, we expect that the overall quality of tools with respect to usability and documentation will increase; as at least the latter usually is a strict requirement to pass these evaluations.

3.2 The selected methodologies and tools

We consider four methodologically different classes of algorithms using *flowpipe construction*, *theorem proving*, *bounded model checking* (*BMC*), or *rigorous simulation*. As depicted in Fig. 3, flowpipe construction and rigorous simulation are both fully automated and over-approximate reachability without user interaction. Typically, they use some geometric shapes (boxes, polytopes, etc.) for the overapproximative representation of state sets. Furthermore, both approaches use time segmentation and analyze reachability within the time segments separately. In contrast, approaches based on theorem proving and bounded model checking use logical formulas as representations, and time is not necessarily segmented. Theorem proving is very powerful and provides exact results but it is interactive, whereas bounded model checking might be overapproximative but does not require any user interaction.

To illustrate the verification process, we made a selection of tools implementing the above algorithms. Our selection was driven by popularity and the aim to cover each of the above methodology categories. Thereby we focused on tools under active development, providing a sufficient level of usability and maintenance. We have chosen *seven tools*: CoRA [6], Flow* [22], SpaceEx [28] implementing reachability analysis based on flowpipe construction, Acumen [46] and HyLaa [11] implementing simulation-based verification, dReach [32] implementing bounded model checking, and KeYmaera [29] which is based on theorem proving. Their high-level properties are summarized in Table 2. All tools are free and open-source as to our knowledge no commercialization has happened in this field yet. In the following four subsections, we present each algorithmic approach and the selected tools, together with estimations on what a user can expect, if the respective approach is used.

The list of our selected tools is by far not complete. The tools we selected are the most established ones in the different categories, for instance, they take part in the annual competitions, are used in case studies, and are often cited and compared against. For the interested reader, some further tools under active development are: Ariadne [14], C2E2 [26], HyCreate [10], NLtoolbox [47] for models with nonlinear ODEs, and HyPro [43] and JuliaReach [21] for models with linear ODEs.

The programming library HyPro and a specific verification tool[1] using the HyPro library called HyDRA are

[1] In most of our publications, both are referred to as HyPro.

Table 2 High-level properties of the evaluated tools ordered by approach (Flowpipe construction (FPC), rigorous simulation (RIS), bounded model checking (BMC), and theorem proving (THM))

		Dynamics	Tool interface	Modeling language	# Param.	Reachable-set visualization
FPC	CoRA	Nonlinear	MATLAB-toolbox	MATLAB code	Large	Configurable plotting
	Flow*	Nonlinear	Command-line tool	Own automata-based language	Few	2-D Plotting
	SpaceEx	Linear	Web-GUI	XML format for hybrid automata	Med.	2-D Plotting
RIS	Acumen	Nonlinear	Java-GUI	Own programmatic language	Med.	All variables over time
	HyLaa	Linear	Python framework	Python code	Few	Configurable plotting
BMC	dReach	Nonlinear	Command-line tool	Own automata-based language	Few	All variables over time, only counterexample candidates
THM	KeYmaeraX	Nonlinear	Web-GUI	Own logical language	0	None

developed in our group. Since we know their internals, and thus we can tune them much better than the other tools, in order to avoid an unfair advantage, we did not include HyPro in our selection. In another context, we already compared HyPro to some other tools [45]. This comparison indicates that HyPro is competitive, but it is not superior to the other tools that we selected. We note furthermore that also Flow* has been created in our group, but its maintenance has been moved in 2015 first to the University of Colorado and later to the University of Dayton.

3.2.1 Flowpipe-construction-based reachability analysis

Methods from this class iteratively compute the set of all states that are reachable from a given set of initial states. To overapproximate flowpipes, they divide a given time horizon into smaller segments and overapproximate reachability within the time segments individually by separate *flowpipe segments*; the union of all computed flowpipe segments overapproximates what can be reached via any time step within the time horizon (see Fig. 11). State sets (like initial states or flowpipe segments) are typically overapproximated by sets with certain geometric shapes (e.g., boxes, convex polytopes or zonotopes). Symbolic representations (e.g., support functions or Taylor models) are also used (see Fig. 13). Various algorithmic improvements and different state set representations have been developed in the past resulting in a diverse group of tools for this approach.

Exploiting the geometric nature of state sets, it is common to provide visualizations of the overapproximated sets of reachable states via plots of the computed flowpipe segments. While these plots are not required for the verification, they certainly help the user to understand the system behavior on a higher level.

A central aspect for reachability analysis via flowpipe construction is which datatypes are used to represent state sets. Together with some other parameters like the length of the time segments, the chosen state set representation has a strong influence on computational effort and precision. Once a parameter configuration is selected, flowpipe-construction-based tools are fully automated. Results are typically not guaranteed to be correct for most of the tools, due to the usage of inexact numerical computations.

CoRA The CoRA tool [6] implements a flowpipe-construction-based reachability analysis engine using zonotopes and zonotope-affiliated state set representations such as polynomial zonotopes. For probabilistic hybrid systems, probabilistic zonotopes are provided.

CoRA is a MATLAB-toolbox designed as a framework for the verification of linear and nonlinear hybrid systems. When using the CoRA MATLAB framework, models, parameter configurations and calls to the verification engine need to be embedded into MATLAB-scripts by the user. Similarly to SpaceEx (see below), CoRA supports compositionally specified models. The tool offers the possibility to read SpaceEx-files and create a description of the underlying hybrid automaton; compositional models are preprocessed and the product automaton is built. After the analysis, plotting (via built-in MATLAB-functions) needs to be done by the user, and the computed overapproximation of reachability has to be checked against the specification by the user. In our evaluation we used the release from 2018.

Flow* The Flow* tool [22] also implements flowpipe-construction-based reachability analysis especially for nonlinear hybrid systems based on Taylor models. The tool is implemented in C++ and once compiled from source serves as a command-line-tool without graphical user interface (GUI).

The tool takes exactly one hybrid automaton as input, for compositional systems the product automaton has to be built by the user. Using a FLOW*-specific syntax, models are specified in a single file which contains two parts: the first part specifies the system variables as well as the settings used for the analysis, while the second part contains the description of the system as a hybrid automaton as well as the set of initial states and the set of unsafe states. A hybrid automaton is described by a set of locations with (potentially nonlinear) dynamics and invariant conditions. In a second step, jumps are described by means of edges connecting the locations, together with their guards and reset relations. The tool returns the result of the safety verification via command line as well as a file for plotting (via `gnuplot`) of projections of the sets of reachable states on a previously selected pair of state space dimensions.

FLOW* was among the first tools for the analysis of nonlinear hybrid systems and has been extended with several features in later works. In this work we use version 2.0.0.

SPACEEX The SPACEEX tool [28] implements flowpipe-construction-based reachability analysis for hybrid automata with linear ODEs.

A `Java`-based graphical user interface (GUI) offers a convenient way for modeling (see Fig. 4). The specified models are stored using an XML-based syntax, which is human readable, even though reading and editing in the XML-format is not intended. When loading verification problems, besides the XML-based model specification, a second file needs to specify values for the analysis parameters as well as the set of initial states and the set of unsafe states. SPACEEX supports model specification by parallel composition.

The tool is implemented in `C++` and can be compiled from source. Additionally, a server for SPACEEX runs on a virtual machine and allows to use it through a browser-based GUI. The results of the verification in the GUI can be visualized and saved as `png` files for multiple pairs of variables. In case a specification is given, only the intersection of the set of reachable states with the set defining the specification is shown (the plot is empty, if safety could be shown).

For our evaluation we use the most recent command-line version 0.9.8f; the screenshots of the GUI are made with the most recent virtual machine image version 0.9.8d of SPACEEX.

3.2.2 Bounded model checking

Tools based on the idea of bounded model checking encode state sets and the semantics of hybrid automata as logical formulas. Intuitively, a formula is built whose solutions encode all paths of a hybrid automaton with a fixed length (fixed number of jumps and potentially upper bounds on the duration of time steps). The conjunction of this formula with the negation of the safety property encodes unsafe paths of the given length. Satisfiability modulo theories (SMT) solvers can be used to check whether the passed formula has a solution; in the case of satisfiability the method reports unsafety along with a counterexample, otherwise the specification cannot be violated and thus the system is safe for the current unrolling. Starting with paths without any jumps, the analysis tool will check the existence of counterexamples and if no counterexample is found then increase the number of considered jumps by one and continue the verification until a given upper bound of jumps is reached or a counterexample is found.

For the analysis of hybrid systems the number of tools implementing bounded model checking is relatively small, as the logical formulation of a problem in the hybrid domain requires a theory for differential equations and SMT solvers that support this theory. Most implementations are based on interval arithmetic, which scales well with increasing dimension but may lead to strong error accumulation (wrapping effects) during computation. In contrast to flowpipe-construction-based approaches, the visualization of the results is not inherent in the method. The set of parameters for these approaches is usually small, the number of steps which are unrolled as well as the time horizon are the central parameters. Once the upper bound (and potentially also a lower bound) of steps for the unrolling is set, tools implementing bounded model checking operate autonomously.

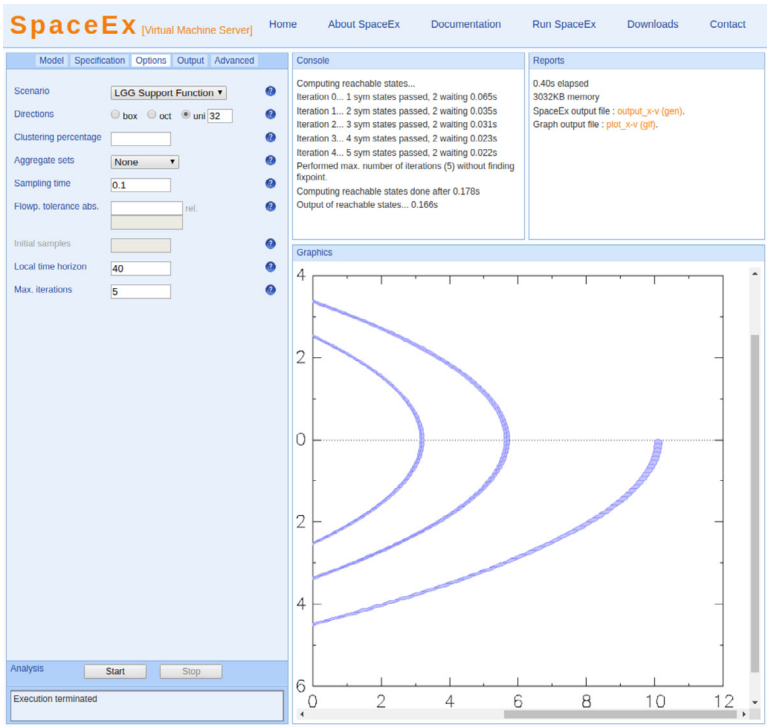
DREACH The DREACH tool [32] implements a bounded-model-checking-based approach on top of the SMT solver DREAL, developed by the same group. The main decision procedure used is interval constraint propagation which does not necessarily terminate, therefore the developers implement δ-satisfiability in DREAL, which assures correctness of the results up to a predefined accuracy δ.

DREACH will return "SAT" if the specification is violated and "UNSAT" otherwise, as the answer to the satisfiability of the encoding of unsafe paths.

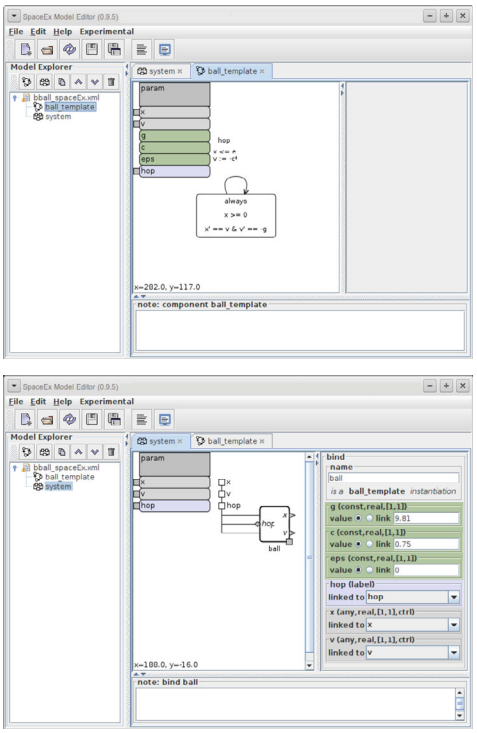
While the tool itself comes without a GUI, visualization of the results is possible via a web-browser, which features zooming and panning of the obtained plots. The visualization shows the sets of reachable states for each variable plotted over time. However, the plotting of reachable states is only possible when the specification is violated as only the sets of reachable states which contain a potential counterexample trajectory are output. For our evaluation, we use the latest version 3.16.06.02 of DREACH which is available online.

3.2.3 Rigorous simulation

Tools based on this approach usually compute "simulation-equivalent" reachability in which an overapproximation of the set of reachable states can be obtained from a finite set of



(a) The panel on the left can be used to load model files and configurations as well as to modify parameters, initial sets of states or the specification. The central panel outputs plots for the computed sets of reachable states, the upper pane shows status information.



(b) The SpaceEx model editor allows graphical modeling. Systems are built from components (single hybrid automata) which are connected and configured via bindings (lower image).

Fig. 4 Screenshot of the SpaceEx GUI (left) and the model editor (right)

(rigorous) simulation traces. Rigorous simulation uses validated numeric such that numerical errors can be quantified and bounded. Some of the tools conservatively overapproximate the system behavior over the dense *continuous* time domain, but some others consider the system's behavior at *discrete* points in time only. Discrete-time approaches are naturally faster and therefore able to provide an intuition on the behavior of systems with more expressive dynamics and higher-dimensional state spaces. However, time discretization sacrifices soundness: even when using exact arithmetic computations, the approach might not detect all (continuous-time) counterexamples.

Acumen The Acumen tool [46] implements various simulation and a few verification approaches based on interval arithmetic for linear and nonlinear hybrid systems over continuous and discrete time.

The tool is shipped as a platform-independent `Java` binary with a `Java`-GUI (see Fig. 5). Acumen's description language is different from the previously discussed hybrid-automata-like description languages: for Acumen, a hybrid system is specified as a hybrid program where continuous (e.g., `x' = ...`) and discrete (e.g., `x += ...`) assignments are executed. Acumen does not explicitly support the definition of locations. Instead, one can encode the locations of a hybrid automaton by an additional variable and specify different dynamics in different locations via a continuous assignment with a case-distinction (similar to a switch-statement). Discrete switches are encoded as conditional discrete updates of the mode-variables in one location block. The reason behind this modeling approach is that for Acumen, in the block where dynamics behavior is specified, all lines of code are executed in parallel unless they are guarded by conditions.

Acumen provides plots for all variables over time via its GUI. We used the version from December 2016.

HyLaa Also the HyLaa tool [11] implements simulation-based reachability analysis. It supports hybrid automata with linear dynamics only. This approach utilizes discrete-time numerical simulations of a given hybrid system with fixed step-size. Consequently, in contrast to all other tools, HyLaa computes the set of reachable states of a hybrid system at fixed points in time instead of overapproximating the set of all reachable states within a given time interval. A conservative overapproximation enables verification, as it covers the complete reachability (within certain time

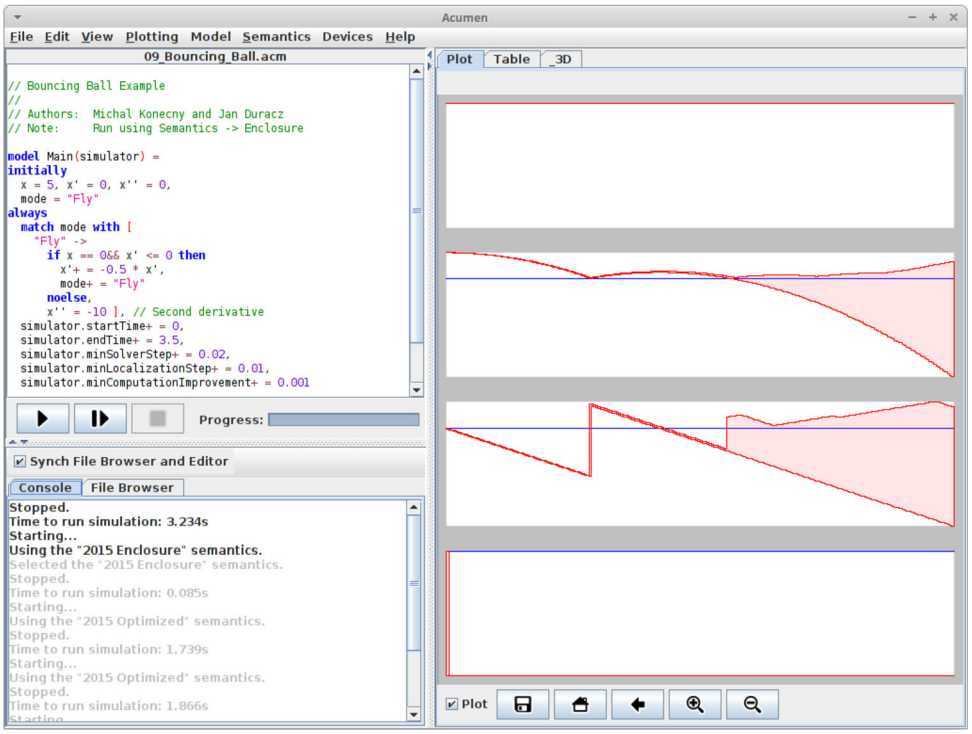


Fig. 5 Screenshot of the ACUMEN GUI. The upper left frame serves as an editor, the panel on the right shows plot results after computing the reachability or simulation of a system. Status information is printed in the lower left panel

bounds) but it comes with high computational costs; computing reachability at discrete time points is more precise and computationally much cheaper but in general it does not cover all reachable states. The objective of this implementation lies on improving scalability towards high-dimensional state spaces.

HYLAA is implemented in `Python`, i.e., the tool is platform-independent. Models for HYLAA are provided by means of a `Python`-script, in which analysis parameters and a safety specification need to be specified. The analysis core is called from within the script to perform reachability analysis, safety check and optionally plotting of the results. As the developer of the tool is also part of the development team of the model transformation tool HYST [12] (see Sect. 5.2), transformation from SPACEEX-files to HYLAA scripts via HYST works out-of-the-box and usually does not require much adaptions afterwards.

Results obtained by HYLAA can be visualized as `png` files for chosen pairs of variables. For our evaluation, we use version 2 from July 2019 of HYLAA.

3.2.4 Theorem proving

Theorem provers like COQ [20], ISABELLE/HOL [38], KEY [4], or PVS [39] provide in their core an axiomatization of first-order-logic and some widely used theories (e.g., arithmetic). To apply theorem proving for the verification of hybrid systems, approaches in this class define, on top of that core, an axiomatization for the theory of differential equations and dynamic and hybrid systems. The safety (or other properties) of these systems can be proven by defining proofs in those axiomatic systems, either automatically or semiautomatically based on required user interaction to guide the proof.

Theorem-proving-based approaches do not provide any visualization of the outcome and often require user-interaction. While this can be seen as a drawback, results obtained with this approach are guaranteed to be sound.

KEYMAERAX A tool that implements hybrid systems verification via theorem proving is KEYMAERAX [29] which utilizes the theorem prover KEY.

In our evaluation we used the web-based GUI of KEYMAERAX which serves as a front end for the underlying `Java`-implementation and allows to create and modify models as well as to interactively guide the proof for safety. Models are provided by the user in differential dynamic logic, which extends propositional logic by theories required for the analysis of dynamic and hybrid systems. The general structure first defines problem variables and constants, and then a single formula which encodes the hybrid system. Note that KEYMAERAX is restricted to decidable theories, which excludes direct usage of transcendental functions such as sin and cos in the model description. However, the developers have described how a user may circumvent this problem in certain cases via differential axiomatization [41] in which transcendental functions can be avoided in the description of the systems' dynamics. As KEYMAERAX uses theorem

proving, so-called tactics (which are rules for the semiautomated prover on which technique to use for the provided formula and its sub formulas) can be specified in the model file as well.

Local instances of KeYmaeraX can be equipped with several solving backends. The user manual states that using Mathematica as a backend works best; as we did not have a license for Mathematica we used the suggested freely available WolframEngine [2]. For our evaluation we used version 4.7.4 of KeYmaeraX.

3.3 Installing the tools

In general, we did not observe any major difficulties in setting up the tools using the manuals.

For Acumen the requirement of `Java-8` was not made public but was resolved quickly thanks to the active developers. As mentioned in Sect. 3, KeYmaeraX supports several backends for arithmetic, with a preference for Mathematica. As we did not have access to a Mathematica license we used the freely available WolframEngine as a backend, therefore we emphasize that the evaluation of KeYmaeraX needs to be treated with care, due to its clearly nonoptimal setup.

Part 2: Evaluation

4 Benchmark selection criteria

In Sect. 2 we have presented hybrid automata as a model for hybrid systems. In the following we use the term (*formal*) *model* when we talk about a (*formal*) abstraction of a concrete system. The term *benchmark* refers to the combination of a model and a safety specification (usually given as a set of states which should be avoided). *Families of benchmarks* refer to a group of benchmarks where either the model is fixed and there are different specifications given or to a group of different instantiations of parameterized models. For the latter, usually the general behavior of the single models in the family is related and only differs in complexity, e.g., the number of variables.

4.1 Challenges

The first task in our evaluation process was to select a set of benchmarks, which is well suited to draw conclusions regarding the applicability and usefulness of hybrid systems analysis tools for industrial applications.

Unfortunately, the number of available benchmarks is quite limited. The largest collection of about 60 benchmarks is presented on the ARCH platform [3]. The annual ARCH workshop allows researchers to present and add new benchmarks to the collection or to give updates on the solutions of existing benchmarks. Examples of further academic benchmark collections are [27] and [23]. Most of these benchmarks are academic and represent simplified systems.

Given a pool of benchmarks, setting up meaningful selection criteria is also far from being trivial, as typical criteria are either not measurable or they do not reliably assure expected properties. We discuss our criteria in Sect. 4.2, and present the selection in Sect. 4.3.

4.2 Benchmarks properties

Below we specify our benchmark selection criteria, which are partly quantitative (i.e., measurable) and partly qualitative.

Relevance We consider the connection of the benchmark to automotive systems as well as the model's *level of abstraction*, i.e., how realistic the model is in comparison to the original system. Too little abstraction usually renders the model difficult or impossible to analyze, while too much abstraction potentially removes relevant behavior due to oversimplification. However, these properties are hard to quantify.

Formal definition Some of the ARCH-benchmarks come with model files for certain tools while others are informally specified. We restricted our selection to formalized benchmarks.

State space dimension (dim) The number of variables in a model, called its *dimension*, gives a first estimation on the complexity of the related verification task, even though it needs to be handled with care: though the running times typically increase exponentially with increasing dimension, there are also high-dimensional models that are easy to analyze and low-dimensional models whose analysis is hard. Furthermore, different models of the same system might have different dimensionality. Besides the inherent complexity of the modeled system, another source of high dimensionality can be a conversion of higher-order differential equations to systems of first-order differential equations (as in the building benchmark `bld_48` below), as the former are not supported in hybrid automata.

Number of locations (loc) and jumps (jmp) The size of the discrete part of a hybrid automaton is another indicator of problem complexity, even though also here no reliable predictions can be made as, e.g., not all locations might be reachable from the initial states.

Models from industrial applications are often composed from several communicating and/or synchronizing subsystems. To our knowledge, no compositional reachability analysis approach exists at the moment, i.e., verification requires

the syntactic composition of the subsystems, building a single large hybrid automaton, whose size is exponential in the number of subsystems. A few tools support compositional input and build only the necessary parts of the composition on-the-fly.

Nondeterminism Discrete and continuous nondeterminism harden the verification task, because checking all possible choices causes branchings in the analysis process. Even deterministic models might lead to such branchings due to overapproximative computations, when additional steps get enabled from the overapproximative part. The level of nondeterminism is not easily measurable: though there might be syntactical hints for nondeterminism, determining whether this causes branchings in the analysis cannot be reliably detected by syntactical checks.

External inputs A specific type of continuous nondeterminism are uncertainties in a system, such as the environment temperature or human input for car steering, that influence the dynamics. These so-called external inputs, which require computationally involved methods during the analysis, affect running times negatively. Tools differentiate between *time-varying* and *fixed* external inputs, and usually require a bounded domain for them.

Hyperplanar guards They model discrete deterministic switching that happens *exactly* at a certain system state, for instance when a timer has reached a certain threshold-value. For this reason, in the literature they are sometimes also referred to as *switching planes* and the corresponding systems as *switching systems*. A two-dimensional illustration is shown in Fig. 6, where only the green, one-dimensional line segment within the orange state set satisfies the guard condition (g, blue). Hyperplanar guards can be defined explicitly by linear equations or implicitly by combinations of inequalities (as in the bouncing ball example, see Fig. 2). In all selected benchmarks, guard sets which are hyperplanar are defined explicitly.

For some approaches that use floating-point arithmetic as, for instance, defined in IEEE 754, switching planes might cause numerical instabilities due to implicit rounding: lesser-dimensional state sets may become empty due to rounding. This can be overcome in some cases by *enlargement* (see Fig. 7) by giving the hyperplane a certain "thickness" in the direction of its normal vector. Technically, for some linear term t over the variables and a rational constant b, we replace the hyperplanar guard $t = b$ by $b - \epsilon \leq t \leq b + \epsilon$ for some $\epsilon > 0$.

Zeno behavior This phenomenon occurs when infinitely many jumps are taken in finite time. Zeno behavior causes *time-convergence*, meaning that time will never pass beyond

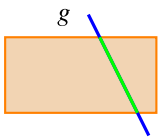


Fig. 6 Hyperplanar guard example (Color figure online)

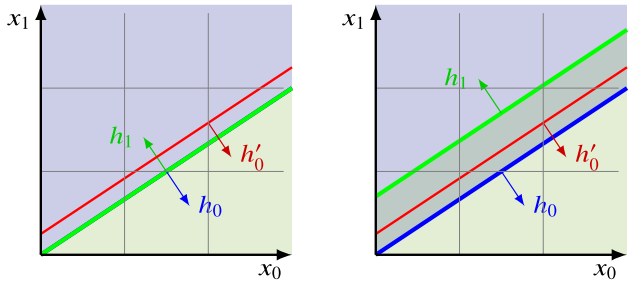


Fig. 7 (Left) A guard set (green, normal vector h_1) and a state set (blue, normal vector h_0) intersect in a line segment (thick blue), similarly to what happens for hyperplanar guards; rounding of the state set (red, normal vector h'_0) may render this intersection empty. (Right) *Enlargement* artificially enlarges guard sets and leads to increased numerical stability (now the intersection is two-dimensional) at the cost of an overapproximation error (Color figure online)

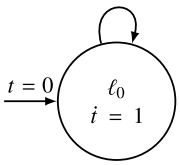


Fig. 8 Hybrid automaton with Zeno behavior

a given finite bound. For example, the model in Fig. 8 allows taking the jump infinitely often without letting any time elapse, as the jump is always enabled. Another example is the bouncing ball (see Fig. 2), where the ball bounces more and more frequently and time converges to a certain finite value but never reaches it.

Obviously, real-life applications are free from Zeno behavior. Nonetheless, certain kinds of abstraction might add Zeno paths to models. For instance, the model of the bouncing ball neglects certain energy losses upon bouncing. Zeno paths are problematic for reachability analysis methods that use iterative successor computations to overapproximate reachability within a given time horizon: if the total duration of a Zeno path falls within this time horizon, then infinitely many time successor computations would be scheduled by the method and it would not terminate. The general problem to decide whether a model has any Zeno path is difficult (actually as hard as the reachability problem itself).

4.3 The selected benchmarks

For our selection we focused on a set of quantitative criteria and on relevance as a qualitative criterion. We aimed at benchmarks with varying state space dimension to evaluate scalability. Additionally, we considered benchmarks with

Table 3 The selected benchmark instances and their high-level characteristics. The columns from left to right specify: the benchmark instance (Name), whether the system is hybrid (hyb), number of variables (dim), number of locations (loc), number of jumps (jmp), whether there is external input (Ext in), whether all guard sets are hyperplanar (Hypl grd), and whether Zeno paths exist (Zeno)

	Name	hyb	dim	loc	jmp	Ext in	Hypl grd	Zeno
Linear	bld_48	×	48	1	0	✓	N/A	N/A
	plt_42	✓	9	2	2	✓	✓	×
	plt_30	✓	9	2	2	✓	✓	×
	flt_04	✓	6	4	4	×	✓	×
	flt_08	✓	10	4	4	×	✓	×
	flt_16	✓	18	4	4	×	✓	×
	flt_32	✓	34	4	4	×	✓	×
Nonlinear	ste_05	✓	3	7	12	×	×	✓
	ste_10	✓	3	7	12	×	×	✓
	ste_RV	✓	3	7	12	×	×	✓
	gly_01	✓	5	3	4	×	✓	×
	gly_02	✓	5	3	4	×	✓	×
	gly_03	✓	4	2	1	×	✓	×
	pnd_01	×	4	1	0	×	N/A	N/A

varying numbers of discrete locations as well as different types of dynamics. Furthermore, we have chosen benchmarks for which preliminary results have shown that most of the tools could solve them.

Based on the above criteria, we have chosen *six benchmark families*, three with linear differential equations and three with nonlinear ones, four hybrid and two purely continuous systems. Considering different variants, parameters, initial sets and safety specifications, we will use in total *14 benchmarks* within these six families. Table 3 shows the benchmarks and their properties. The names of the benchmarks are composed from the name of the benchmark family followed by a unique instance id.

A detailed description of the selected benchmarks can be found in Appendix A; the model files and description are available online.[2]

5 Model generation

Formal verification requires formal models, which we generated for all selected benchmarks and tools.

Summarized key observations:

– *Model construction could be done for all the tools with a reasonable amount of effort.*
– *Models can be defined conveniently with the SpaceEx modeling GUI.*
– *The tool Hyst converts SpaceEx-models to certain other input languages and CoRA offers an internal SpaceEx-to-CoRA converter.*
– *Some approaches are distributed as executable binaries while others are programming libraries.*
– *Not all input languages have a formal semantics. Therefore, they can only be compared semi- or informally.*

5.1 Challenges

It is not clear upfront which tool is able to solve a given verification problem. We can identify those tools that support the required expressivity and functionality, but it is hard to predict which tools might succeed and which not. Thus typically one would try several tools.

Unfortunately, there is no widely accepted standard input language for the formal specification of hybrid systems. Therefore, the user has to model each benchmark for each tool in its specific input language. This step is not only tedious but also challenging, as for different modeling languages semantics-preserving transformations have to be tailor-made due to the varying expressiveness of the description languages.

As mentioned in Sect. 4.2, the presence or absence of certain model behavior like Zeno paths or switching planes might have a major impact on the running times and the outcome of the verification process. However, inexperienced users might not be able to recognize the existence of Zeno paths or other, for the analysis disadvantageous properties.

5.2 The model generation process

We started investigating the tools and input languages by first reading the provided user manuals. Hello world examples for hybrid system modeling, such as the bouncing ball, are available for all tools. These modeling examples (which are usually included in the code distribution, in the users' manual or on the tool's web page) are mostly well documented and have been very helpful to understand the basic concepts.

The SpaceEx language has become quite common and is supported also by some other tools. We could use available SpaceEx models for all of the selected benchmarks. An example SpaceEx model for the bouncing ball is shown in Algorithm 1, screenshots for the modeling GUI are shown in Fig. 4b.

Also some other tools use hybrid-automata-style input models but the description languages vary in syntax as well as expressiveness. As a second example, Algorithm 2 shows a Flow* model of the bouncing ball.

The Java-based conversion tool Hyst [12] provides transformations from SpaceEx-format into the input languages of the tools dReach, Flow*, and HyLaa (see

[2] https://ths.rwth-aachen.de/research/projects/ford-aachen-hybrid/.

Algorithm 1 Code example for SpaceEx for the bouncing ball. Note that SpaceEx comes with a GUI (see Fig. 4a) to create models, such that manual editing of model files is rarely required

```
<?xml version="1.0" encoding="UTF-8"?>
<sspaceex xmlns=
  "http://www-verimag.imag.fr/xml-namespaces/
      sspaceex"
  version="0.2" math="SpaceEx">
<component id="ball_template">
  <param name="x" type="real" local="false"
    d1="1" d2="1" dynamics="any" />
  <param name="v" type="real" local="false"
    d1="1" d2="1" dynamics="any" />
  <param name="hop" type="label" local="false"
       />
  <location id="1" name="always" x="174.5" y
      ="225.5"
    width="135.0" height="73.0">
    <invariant>x &gt;= 0</invariant>
    <flow>x' == v &amp; v' == -9.81</flow>
  </location>
  <transition source="1" target="1">
    <guard>x &lt;= 0 &amp; v &lt; 0</guard>
    <assignment>v := -0.75*v</assignment>
    <labelposition x="-41.0" y="-69.0" />
  </transition>
</component>
<component id="system">
  <param name="x" type="real" local="false"
    d1="1" d2="1" dynamics="any" controlled="
        true" />
  <param name="v" type="real" local="false" d1
      ="1" d2="1"
    dynamics="any" controlled="true" />
  <bind component="ball_template" as="ball" x
      ="2.0" y="1.0">
    <map key="x">x</map>
    <map key="v">v</map>
  </bind>
</component>
</sspaceex>
```

Algorithm 2 Code example for Flow* for the bouncing ball

```
hybrid reachability {
  state var x, v

  [ ... settings ... ]

  modes {
    l0 { lti ode { x' = v  v' = -9.81 } inv {
        x >= 0.0 } }
  }
  jumps {
    l0 -> l0 guard { x <= 0.0  v <= 0.0 }
        reset { v' := -0.75 * v }
    parallelotope aggregation {}
  }
  init {
    l0 { v in [0, 0] x in [10, 10.2] }
  }
}
```

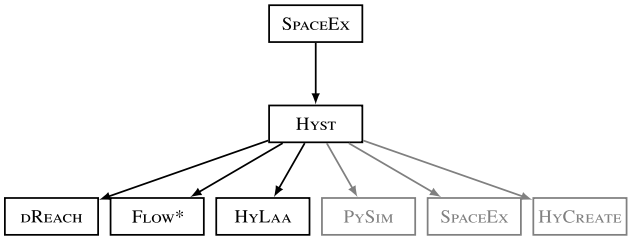


Fig. 9 Model transformation possibilities via Hyst. Input is always provided in the SpaceEx language. The output languages supported but not used in this work are depicted in gray

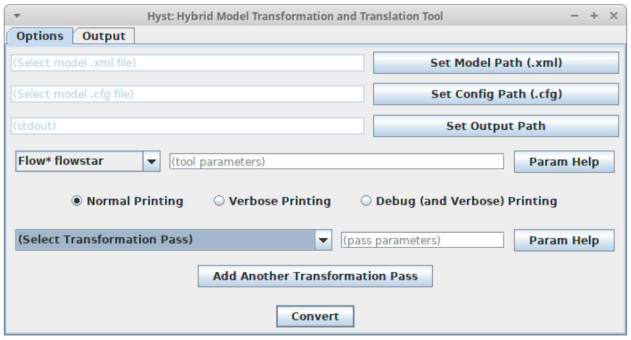


Fig. 10 GUI of the transformation tool Hyst. Apart from transformation, the tool allows including preprocessing (option: transformation pass) to simplify systems according to predefined rules

Fig. 9). Hyst comes with a GUI (see Fig. 10) and allows including the so-called transformation-passes, which modify the parsed automaton syntactically or semantically before transformation. Syntactic transformation passes may for instance add explicit identity resets, simplify arithmetic expressions, or substitute parameters by values. Semantic transformation passes among others allow scaling time or reducing the order of the ordinary differential equation systems which describe the dynamics. Several transformation passes may be combined sequentially (to be executed one after the other). All these transformations preserve some kind of semantic equivalence, even though not always at the level of concrete trajectories. Such transformations might be necessary to enable model transformation from one input language to another but they might also be used to ease the analysis task.

For CoRA, the model script can be obtained via an internal SpaceEx-to-CoRA converter. The models for KeYmaeraX and Acumen had to be specified manually, without tool support.

5.3 Observations

Automated model transformation Though we could obtain initial models for dReach, Flow* and HyLaa using Hyst, not all SpaceEx language features are translatable

to other tools (for instance, differential algebraic variables, which are auxiliary variables whose values are defined, e.g., by equations). Furthermore, specifications that can be parsed by SpaceEx are sometimes rejected by Hyst, because Hyst requires explicit initializations for all parameters, whereas SpaceEx is more generous using default values for missing parameter values. Due to these issues, the generated models needed some manual adaption, but in general, the transformation worked well.

As mentioned before, the automated model transformation does not cover all tools. The major difficulty for model transformation is to identify compatible fragments of the input languages for which there exists a semantics-preserving transformation. For example, it is challenging to transform automata-based models into semantically equivalent differential dynamic logic programs for KeYmaeraX, or into programming languages required by CoRA and HyLaa.

Programming libraries CoRA and HyLaa do not provide executables but require the user to write a program, which calls a reachability analysis method. For CoRA, we had to write most parts of the instructions (apart from the hybrid automaton definition, see above) manually, whereas for HyLaa we could obtain a program via transformation with Hyst. During modeling (and later execution) we could observe that programming libraries allow for more freedom in the design of the analysis but are also more error-prone.

Syntax and readability In general, the syntax of the used languages is well understandable apart from a few peculiarities.

The readability of models differs between the tools: many tools (CoRA, dReach, Flow*, HyLaa) aim at using automata-based descriptions via dedicated languages such that they can be modified by hand, while for others compactness (KeYmaeraX) or simplified parsing for a machine (SpaceEx) lies in the focus. For programming libraries, the model definitions are lengthy and might be harder to parse by humans.

We noted a few issues, which might seem unusual to new users of the respective tool. We found that for Flow*, whitespace within a single line in the input has a semantics, which is unusual and may lead to confusion (see, e.g., specification of guards in Algorithm 2 where several constraints are put in a single line). Furthermore, division by a constant in Flow* has to be encoded via multiplication. For Acumen, its input language includes the declaration of a special block in the model (labeled by `always`), in which every statement is executed as if it was run in parallel—this feature is documented but requires a bit more cognitive capacity to be fully comprehended by the user.

We expect that readability is only a relevant criterion when models need to be adjusted by hand, e.g., during prototypical development.

Soundness Even if most input languages are intuitive, not all have a formal semantics, and we are not aware of any formal correctness proof for the Hyst transformation tool. For our benchmarks, we did not prove semantic equivalence formally, but applied code review and compared the computed reachability results to validate equivalence. During these checks, we did not detect any inconsistencies.

6 Verification task

In this section we chronologically report our observations throughout the different phases of the verification task.

6.1 Challenges

An important issue is the identification of suitable *analysis parameter* values (see Sect. 6.2). Correct parameter setting is a major obstacle for users. Even though parameters are usually documented, the documentations are not always complete. To be able to estimate the effects of certain parameters, the user needs to be aware of the internals of the tools. Each tool implements its own algorithm with own parameters. The parameter sets of different tools are nearly incomparable, thus parameter configurations are in general not transferable.

After the identification of the parameter values, the verification step can be run (Sect. 6.3). Besides detecting syntactic problems in the models we also discovered a few bugs in the tools as well as cases of numeric instabilities, but all in all, running the verification process did not cause any major problems.

6.2 Parameter specification

Summarized key observations:

- *Different tools come with different, often incomparable parameter sets.*
- *Tool parameters, their meaning and interplay are often poorly documented.*
- *Finding suitable parameter configurations requires an understanding of the implemented method.*
- *Only a subset of the tools offers a GUI (SpaceEx, KeYmaera, Acumen).*
- *Smaller parameter sets simplify the verification process but restrict optimality; adaptive parameters may help the user without losing optimality.*

Especially approaches implementing flowpipe-construction-based reachability analysis come with larger sets of parameters to tune; theorem proving and bounded model checking require less tunings. For each selected benchmark, the parameter settings we used for each of the selected tools can be found in Appendix B.

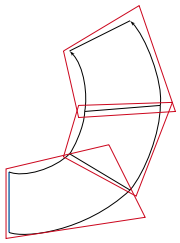


Fig. 11 Reachability within time segments (lines, blue) and their overapproximating flowpipe segments (polygons, red). A color figure is available online

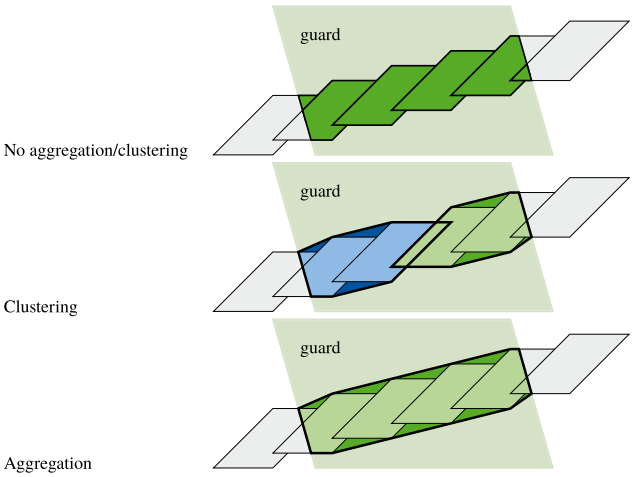


Fig. 12 Clustering groups state sets and overapproximates the union of each group by a single set; aggregation overapproximates the union of all sets by a single set

Documentation Most tools document only their most basic parameters, and often without explaining their effects and interplay in detail. This is probably partly due to the academic nature of the tools, but also because it is difficult to make general statements about the parameters' meaning and influence on the computations.

In some cases, missing documentation could be compensated by consulting available examples. For instance, ACUMEN documents its simulation parameters but not those for verification. However, it offers an exceptionally large set of examples, which helped to find out which analysis parameters are available and how they affect the analysis.

Finding suitable parameter values The parameter values affect precision and running times and have therefore a strong impact on the verification success. However, for some of the available parameters it is hard to find suitable values because this process requires a deeper understanding of the implemented methods.

For instance, the flowpipe construction approach overapproximates reachability according to the continuous dynamics within a given time horizon by overapproximating time segments of a given *time step* length separately (see Fig. 11). From each of these flowpipe segments, jump successors can be either processed separately or, in order to reduce the computational effort, these can be *clustered* or *aggregated* (see Fig. 12). Both parameters (time step size and aggregation) can have significant influence on the running times. For instance, turning off aggregation may result in an exponential increase in the number of segments to compute. Furthermore, each segment is overapproximated using special state *set representations*, some of them illustrated in Fig. 13. The intuitive effect of these parameters is illustrated in Fig. 14.

One would expect as it is the case in Fig. 14 that decreasing the time step size leads to higher precision on the cost of longer running times, and conversely that increasing the time step size has the opposite effect. While the effects on precision *usually* behave as expected, in some cases increasing the time step size leads to increased overapproximation from which more states are reachable, what might enable further jumps, leading to increased computational effort and longer running times. In rare cases, depending on the implemented approach and the system which is analyzed, it may also happen that a smaller time step size *reduces* precision, as errors during the computation accumulate, which has a stronger effect if more segments are computed. We observed this for the FLOW* tool on certain benchmarks.

In addition, parameters do not always behave linearly. For instance, FLOW* allows to set the *precision* of floating point numbers by fixing the number of bits for the mantissa. Processors are optimized for the default value of 53 bits for the mantissa, which corresponds to double-precision. In our experiments, increasing the precision caused longer running times but reducing it did not have any observable effect.

In general, we observe that while a large set of available configuration parameters may enable very precise customization, often the required effort to correctly adjust these parameters and to estimate their impact outweighs the gained advantage.

To ease the parameter selection, some of the tools implement *adaptive* analysis methods. The STC algorithm implemented in SPACEEX, for example, allows giving a relative error bound on the computation and adapts its parameters to achieve this bound. Similarly, FLOW* provides the feature of adaptive Taylor model orders during computation. The tool HYLAA implements counterexample-guided refinement of the computation by switching off the aggregation. In our execution we did observe the positive effects of adaption: adaptive parameters ease the usage of a tool, as parameters can be chosen from a set and thus do not require precise setting but instead the tool will adjust those parameters on demand during execution.

Dependencies and incompatibilities Some parameters influence each other, but there is very limited documentation on how parameters interact. Some parameters only have measurable effects in combination with other parameter configu-

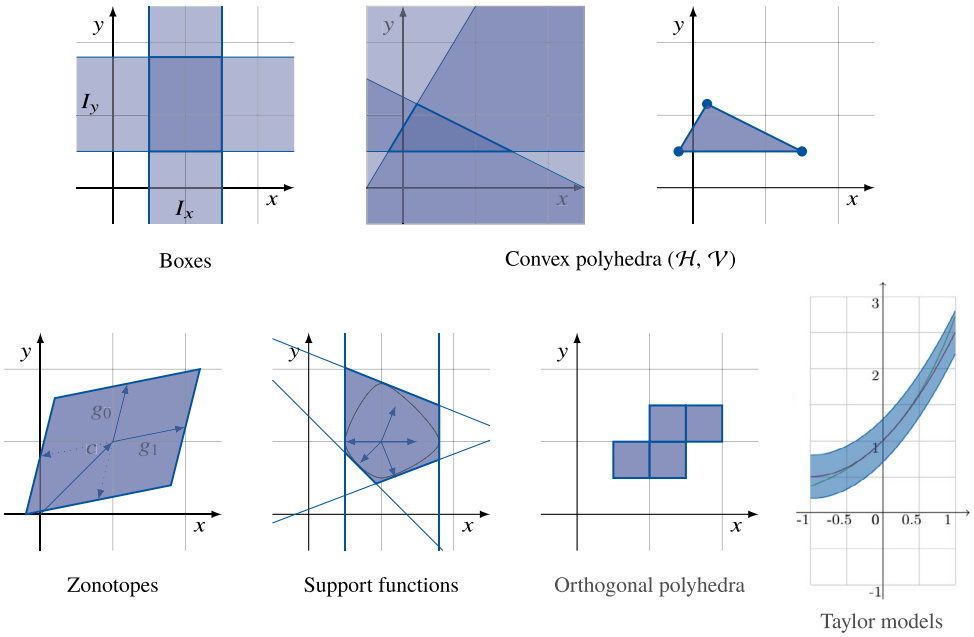
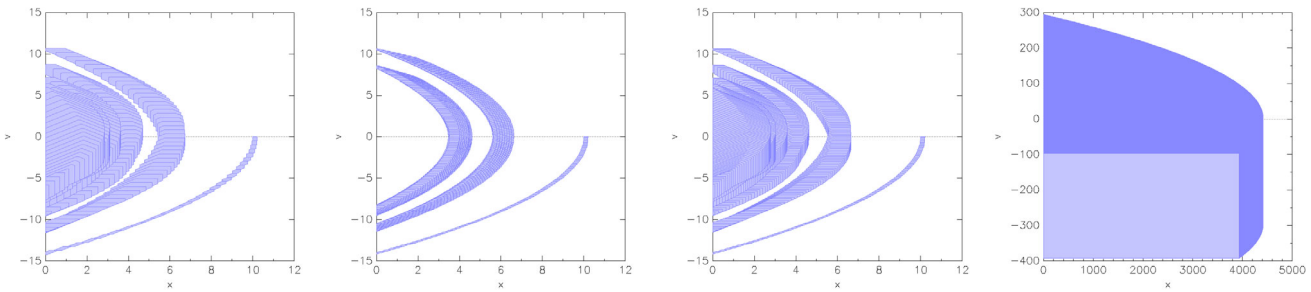


Fig. 13 State set representations in flowpipe-construction-based reachability analysis



(a) Support functions, $\delta = 0.1$, aggregation.

(b) Support functions, $\delta = 0.01$, no aggregation.

(c) Support functions, $\delta = 0.01$, aggregation.

(d) Boxes, $\delta = 0.01$, aggregation.

Fig. 14 Influence of different analysis parameters on the verification result illustrated using SpaceEx on the bouncing ball example. We vary the state set representation (support functions vs. boxes), the time step size δ, and whether successors of discrete jumps are aggregated or not

rations which may lead to misinterpretation of a parameter's effects and influences.

Most of the tools have *incompatible* parameter value combinations, but also this information is not well documented. For instance, some parameter configurations related to guard intersection operations in the tool CoRA are not compatible—the tool will throw errors during computation, which requires manual effort to relate those errors to a wrong parameter configuration.

A GUI may help to identify incompatible parameter value combinations; for example, SpaceEx prevents the user from choosing certain parameters when not applicable by greying the respective fields in the GUI depending on prior choices of other parameters. On the other hand, we found that for the selection of clustering/aggregation, which are different methods on how to treat state sets during discrete jumps, SpaceEx offers four options: `none` (which can be seen as default), `clustering`, `aggregation`, and an empty field, which caused confusion, as it is unclear what its semantics is.

KeYmaeraX also offers a GUI for modeling and proof creation and provides extensive tooltips along with links to guides to aid the user in creating proofs. We feel that similar assistance could also improve the usability of other tools significantly.

For tools without a GUI (CoRA, dReach, Flow*, HyLaa), where parameters are provided via the terminal or within a file and the option of excluding certain combinations of parameters is limited, it may improve usability by mentioning parameter interaction insights including infeasible and zero-impact parameter configurations in the user manual.

Comparability As the parameter sets of the tools are not comparable (one of the few commonly shared parameters is the time-step size for flowpipe-construction-based tools), parameter configurations need to be identified for each tool separately. In this evaluation, being able to *relate parameters* from different tools to each other played an important role for comparability. One example for this is how to bound the analysis. Most tools bound the *jump depth* to a certain (user-defined) value d, i.e., each analyzed execution trace exhibits at most d discrete jumps. This is different in SpaceEx, where the parameter `iterations` bounds the *total* number of considered discrete jumps over all analyzed traces in the analysis. While this is a perfectly reasonable way to effectively bound the execution, without knowing the search heuristics (e.g., whether depth-first or breadth-first search is executed) it is hard to estimate the effect of the `iteration` parameter on the analysis and thus it is hard to obtain parameter values for the parameter `iteration` that are in effect comparable to a certain jump depth. Also CoRA handles bounded analysis differently—here, only the execution time is bounded, i.e., a maximal number of jumps cannot be provided. In our evaluation we resolved this issue by setting the jump depth (respectively the number of iterations) to a reasonably high value and bounded the global time horizon of the benchmark via an additional clock variable (that means the time duration of the model execution is bounded similar to the approach that CoRA implements). Furthermore, we aimed to set parameters for each tool in such a way that the resulting precision is similar.

Initial sets The shape and size of the initial set might have a strong impact on the verification outcome. When verification does not succeed, dividing the initial set into subsets and checking them individually often helps to verify a model. However, this is a trial and error process, and is not automated.

6.3 Running the verification process

Summarized key observations:

- *Most tools run fully automated, only KeYmaeraX required user-interaction during the analysis.*
- *Syntactic problems in the models are detected at runtime.*
- *We observed numerical instabilities in certain cases.*

Verification process Most verification processes could be run without any major problems, we encountered just a few issues.

As modeling and verification are separate processes, syntactic problems are typically detected at the attempt of execution, for instance not being able to divide by a constant c but instead using a factor $1/c$.

In rare cases we encountered bugs in the tools but it is not always fully clear whether something is caused by a bug or by the fact that we cannot truly interpret the results. As an example, dReach implements δ-reachability, therefore counterexamples might be spurious; however, for certain precisions it happened, that a specification composed as a half-space was declared as not being reachable while its inverse was also declared as being not reachable, which seems to be contradicting. When we reported bugs, most tool developers reacted promptly and provided fixes that indicates the active development in this community; some other bugs are still under investigation.

CoRA tends to require more memory than other tools. Additionally, CoRA caches parts of its computations, which results in different running times for repeated analyses.

Verification with KeYmaeraX is interactive. Despite lots of tooltips and hints on the rules which can be used, the usage of this tool often requires in-depth knowledge of the model under verification as well as the proof rules. This understanding was lacking in our process and we failed to verify the selected benchmarks.

Numerical instabilities Since most tools use floating point arithmetic, numerical issues may arise, especially during computations with point-sets, hyperplanar guards or non-full-dimensional sets (see Sect. 4). Whenever numerical instabilities became an issue, we have relaxed the model by *enlarging* the respective constraints.

7 Results and observations

In this section we discuss the obtained results and condense some recommendations.

7.1 Challenges

The *running times* of the verification, provided in Sect. 7.2, strongly depend on the parameter configuration (see also Sect. 6.2). Especially for large systems, adjusting parameters becomes a time-consuming process as the user needs to wait for the analysis to finish and may require several runs to find a suitable parameter configuration. Tools which allow prematurely terminating computation once the safety specification is violated ease this process.

Another important evaluation criterion is the reachability result, i.e., the computed set of all reachable states, or more importantly the size of the overapproximation. However, due to different representations and a lack of a mathematical basis for measuring set sizes and distances, there is no easy way to compare the computed sets of reachable states. To illustrate this, Figs. 15 and 16 show exemplary plots which highlight the differences between different tool outputs.

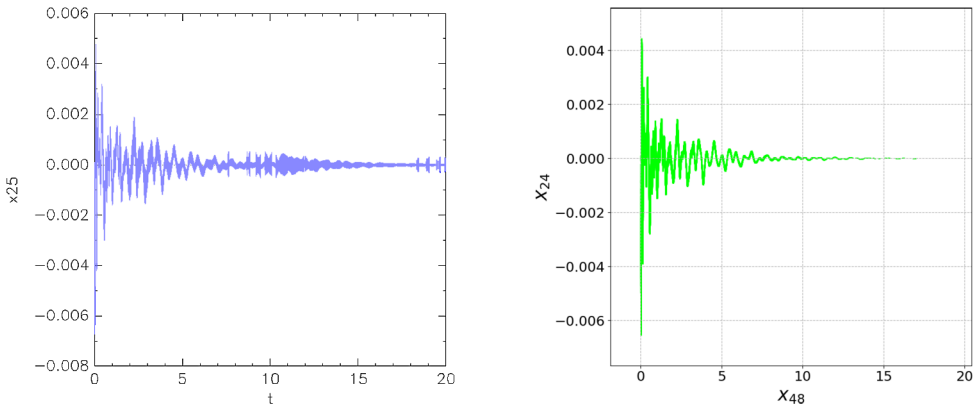


Fig. 15 Plots for the building system (`bld_48`) computed with SpaceEx (left) and HyLaa (right). Both plots show the displacement of the 8th floor of the building over time

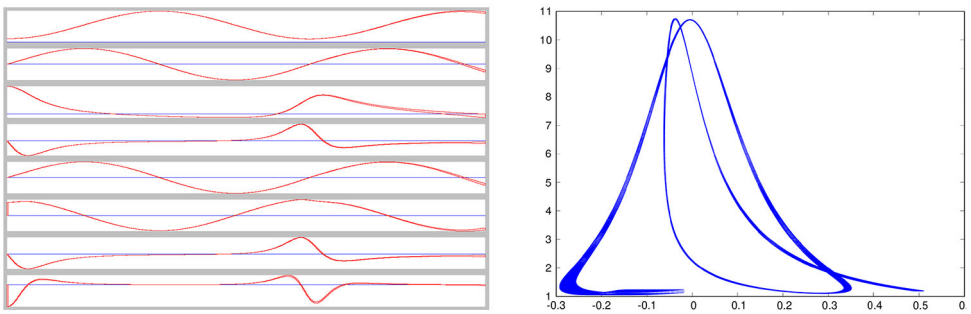


Fig. 16 Plot for the spring pendulum system (`pnd_01`) computed with Acumen (left) and Flow* (right). While Acumen plots each variable over time, Flow* (like many tools implementing flowpipe-construction-based reachability analysis) plots two user-selected variables (here: r (spring length on the vertical axis) over θ (angle on the horizontal axis), which corresponds to the 3rd and 1st variable in the plot from Acumen)

Note that while the general intention of safety verification tools is to provide an answer to the question, whether a given model of a system conforms to a given specification, a *visualization* of the computed sets of reachable states may help to understand the obtained results, estimate the correctness of the model, the size of the overapproximation or the spuriousness of a counterexample. Conversely, not being able to get visual feedback of the computed sets of reachable states of a given system makes debugging and parameter adaption a challenging task. Especially in case of complex systems, if the verification process detects potentially unsafe behavior, without a visual feedback it is hard to estimate whether the model is correct, and whether it is really unsafe. A discussion on the interpretation of verification outcomes can be found in Sect. 7.3.

We conclude this section with some recommendations in Sect. 7.4.

7.2 Running times

In Table 4 we present the running times on a machine with an Intel Core i7 (4×4 GHz) CPU and 16 GB RAM. We do not list running times for the interactive KeYmaeraX tool as they are not comparable to fully automated computations. We emphasize furthermore that the analysis approaches and the parameter sets of the tools are so different, that the running times are far from being comparable and do not provide an optimal measure for tool quality.

We could observe that for some tools the running times differ from the running times reported in the ARCH-competition [7–9]. The reason for this is that in the competition the tool developers were able to adjust the models and tools according to the respective benchmark instance. The building benchmark (`bld_48`), which has the highest dimension among the selected benchmarks could be verified by most tools but the running times were sometimes longer than other reported running times. A similar effect can be observed for the platoon benchmark instances (`plt_30`, `plt_42`). This observation hints that expert knowledge and experience with a specific tool may significantly improve analysis results. On the one hand, this raises the demand for more automated approaches to lower this bar of required expertise and on the other hand shows that a close collaboration between industry and tool developers for technology transfer may improve the general quality of results obtained by inexperienced users as well.

Table 4 Running times in sec for our experiments (⊥: unsafe states reached in the overapproximation; to: timeout after 20 min; *: error; N/A: tool is not suitable for the instance)

	Instance	Flowpipe construction			BMC	Rig. simulation	
		CoRA	Flow*	SpaceEx	dReach	Acumen	HyLaa
Linear	bld_48	89.23	to	48.00	43.55	*	40.41
	plt_42	*	⊥	0.74	to	to	25.11
	plt_30	*	21.63	12.56	to	to	26.97
	flt_04	*	1.34	0.07	1.31	*	0.07
	flt_08	*	3.57	0.32	4.80	*	0.09
	flt_16	*	21.34	0.26	8.12	*	0.21
	flt_32	*	131.28	3.29	38.94	*	0.72
Nonlinear	ste_05	1.49	0.39	N/A	to	10.00	N/A
	ste_10	*	3.05	N/A	to	*	N/A
	ste_RV	*	474.79	N/A	to	*	N/A
	gly_01	*	19.79	N/A	5.45	to	N/A
	gly_02	119.40	27.76	N/A	5.88	to	N/A
	gly_03	*	15.95	N/A	0.21	to	N/A
	pnd_01	9.25	919.32	N/A	0.26	*	N/A

Timeouts For the evaluation, we use a timeout of 20 minutes ("to" in Table 4). This timeout threshold is chosen based on experience and to keep the duration of the evaluation process within reasonable time bounds. While larger values for the timeout threshold theoretically yield less timeouts in the results, experience has shown that on benchmarks of this size tools tend to run out of resources, e.g., memory, instead of being successful on longer runs.

When searching for unsafe paths, uncertainty via external input (bld, plt) or a higher number of locations and outgoing jumps per location (ste) increase the problem complexity, such that we can observe more frequent timeout for such benchmarks. The increased complexity seems to have the strongest impact on the bounded model checking approach, where executions are encoded as logical formulas; we can observe that dReach struggles with such benchmarks but works very well on others. For the steering controller benchmark family (ste_05, ste_10, ste_RV), dReach was able to successfully verify paths with up to 16 jumps within the timeout but could not cover all paths up to the global time horizon of 30 time units. The platoon benchmarks (plt_42, plt_30) do not exhibit strong branching; perhaps the external input made the analysis hard for bounded model checking as the tool dReach could not complete the computations for the first time-transition within the provided time limit.

Internal problems The entries "*" in the table denote internal problems, such as numerical issues or segmentation faults. As pointed out in Sect. 6.3 most of the observed errors were allottable to numerical issues as far as the error message could reveal. CoRA works well on continuous systems but suffers from numerical instabilities when jumps are involved: computing guard intersections, especially for switching planes (hyperplanar guards), cause problems for the used library for representing convex polytopes. Unfortunately for CoRA, most benchmarks are hybrid; another benchmark collection could have been more appropriate to demonstrate the strengths of the tool.

Inconclusive results Entries marked with "⊥" denote cases in which we were not able to verify the model due to overapproximation errors being too large, neither with the parameter configuration listed in Appendix B nor with several other configurations we tried.

Impact of methodology The implemented approach for safety verification has a huge effect on the analysis results, which also depends on the type of application and analyzed system. For instance, the results for the tool Acumen seem not to be convincing at first sight. The tool itself is powerful. We can highly recommend it for simulation, for which it was developed; verification is a recent functionality developed on top of simulation but it uses interval arithmetic that might lead to intensive case splitting, and often causes large error-accumulation.

For the linear benchmarks, both tools SpaceEx and HyLaa provide results on all selected systems. For fairness, we mention that both tools focus on linear hybrid systems only and thus are able to provide a tailored approach for verification. Furthermore, HyLaa works with discrete-time numerical simulation: HyLaa computes the set of reachable states of a hybrid system only at fixed points in time, instead of over-approximating the set of all reachable states in continuous, dense time.

The tool Flow*, which is dedicated to the analysis of nonlinear hybrid systems, produced in general longer run-

ning times for linear benchmarks, which is not surprising as the algorithm for nonlinear hybrid systems is computationally more involved; on nonlinear benchmarks Flow* yielded the best performance. The tool CoRA implements both approaches, one for linear hybrid systems and one for nonlinear hybrid systems; depending on the dynamics, the tool selects the appropriate approach.

For all tools, we could observe that the analysis of purely continuous systems is more developed than the analysis of systems with mixed discrete–continuous behavior. This result is expected (and now confirmed by our observations), as the extension towards hybrid systems includes more challenges not being present in a purely continuous systems.

7.3 Result visualization

Most tools based on rigorous simulation or flowpipe construction provide some visualization of the computed sets of reachable states; usually projections of the sets of reachable states can be plotted to a file while some of the tested tools plot variable valuations over time. We could observe, that for tools with a GUI, zooming and panning features were helpful.

Note that all tools which allow selecting a set of state space dimensions for plotting (CoRA, Flow*, HyLaa, SpaceEx) require recomputing the set of reachable states, i.e., the full analysis, once the selected variables for plotting are changed. However, some of the tools allow selecting multiple pairs of variables for plotting to create a set of plots at once; two example plots computed with SpaceEx and HyLaa are shown in Fig. 15. In contrast to this, the tools Acumen and dReach provide plots of each variable over time (see, e.g., Fig. 16).

The KeYmaeraX tool does not provide any visualization of results.

7.4 Recommendations

A wide range of academic tools exists for hybrid systems verification—in this work we have only covered some of them. Depending on the problem at hand and based on our observations during the experimental evaluation, the decision of a suitable tool may profit from the following high-level observations which summarize our results:

- *Linear continuous systems.* For linear continuous systems, the tools SpaceEx, CoRA, dReach, and HyLaa produced promising results (see Table 4).
- *Linear hybrid systems.* Although many tools claim to cover this class of systems, we feel that SpaceEx provides the most mature implementation. If the user can afford to check reachability at discrete points in time only, HyLaa may be an option as well. Both tools HyLaa and SpaceEx were able to successfully verify all provided benchmarks in this class; during the evaluation both required at most few adaptions in the parameters.
- *Nonlinear continuous systems.* For nonlinear continuous systems, CoRA and dReach show promising results with respect to running times and capabilities (see Table 4), furthermore Flow* allows analyzing this class of systems.
- *Nonlinear hybrid systems.* For this class of systems, the tool Flow* seems to provide the most mature implementation as it was the only tool which could provide results for all benchmarks in this category with the chosen settings.

Note that these observations were made based on the results obtained during our experimental evaluation on a restricted set of benchmarks.

8 A more realistic case study

To go beyond academic examples and test the tools' usage on a real-world application, we modeled a fuel cell thermal management (FCTM) system [40, 50] developed at Ford Motor Company in collaboration with the University of Michigan.

8.1 The system

In the FCTM system, the electric engine of a car is powered by a battery, which can be charged using a stack of fuel cells. The cells operate most efficiently at a certain temperature, thus the ambient temperature as well as the cell usage (which warms them up) influence their efficiency. Furthermore, for a long battery lifetime, the battery should not be operated at extremal state-of-charge values. Therefore, a controller should keep the cells' temperature as well as the battery's state of charge within certain bounds.

The components of the FCTM system are depicted in Fig. 17. A liquid can be pumped through the fuel cell stack. The temperature of this liquid affects (increases or decreases) the fuel cells' temperature. The controller operates in a feedback-loop periodically with a cycle time δ. In each control cycle, the controller senses the plant's state and decides whether or not to pump the liquid. When pumping, the controller can either utilize a battery-powered heating device to increase the temperature of the coolant liquid (and therewith also decrease the battery's state of charge), or let the coolant liquid pass though the car's radiator to reduce its temperature, where the cooling-effect of the radiator depends on the car's velocity and the ambient temperature.

The controller uses a look-up table that has been synthesized using model-predictive control [50]. The state space of the plant is partitioned into 21 regions and each region is mapped to a controller output, which affects the dynamics

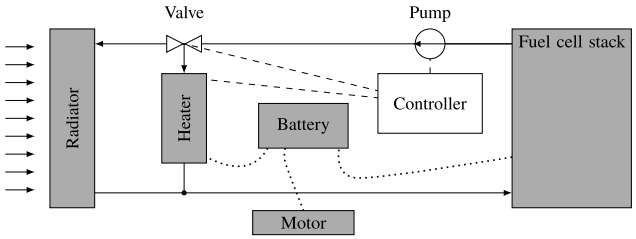


Fig. 17 Structure of the fuel cell thermal management (FCTM) system. The controller may influence (dashed) the valve to control the coolant flow, the pump to adjust the flow rate and the power used by the heater. Coolant flows are depicted as arrows, wires are depicted as dotted lines

of the plant for the next control cycle. The controller outputs are four-dimensional, specifying a Boolean value for the directing of the coolant towards the heater or the radiator, as well as numeric values for the coolant flow rate, the current taken from the fuel cell and the power directed to the heating device.

8.2 The SIMULINK model

The dynamics of the car is given in [40]. Since it is a proprietary model, confidential details are not publicly available, but for the modeling we were provided with SIMULINK-files of the complete system with fully specified dynamics and a fully specified controller.

In the SIMULINK model, the car's state is described by five continuously evolving variables and piecewise constant controller outputs. The continuous dynamics is specified by nonlinear differential equations which also contain implicit switching between different dynamics. The switching is a result of min/max operations comparing variables, for instance boundaries of the state space where the dynamic changes (e.g., when the ambient temperature falls below certain boundaries). That means that while the expressions suggest a purely dynamical system, i.e., a system with one mode of operation, due to these conditions the system is implicitly hybrid with discrete switches between different modes. Apart from implicit switches, the dynamics also contains rational terms, transcendental functions and nonlinear polynomials, which poses special requirements on the tools available for verification.

Additionally, the usage of the motor and the environmental conditions need to be specified as input. The energy drained by the motor is modeled via standard driving cycles, other operating conditions are given as intervals (e.g., ambient temperature) or provided as constants (e.g., oxygen partial pressure).

8.3 The hybrid automaton model

First we needed to specify a hybrid automaton model for the FCTM system. Our goal was to model the system as accurately as possible within the expressivity bounds of the verification tools, i.e., while ensuring that at least one tool can (in theory) compute reachability for the modeled system.

We have created the hybrid automaton model, visualized in Fig. 18, as follows:

– The SIMULINK dynamics uses four Boolean switches, resulting in 16 possible continuous dynamics. For each of these dynamics under the different switch evaluations, we define a control location – which we call *switch modes* – in the hybrid automaton. For instance, if the original dynamics contains a max operation on two variables a, b, e.g., $\max(a,b)$, we introduce locations where $a \geq b$ holds and locations where $a \leq b$ holds. Each of these 16 switch modes is connected via guarded transitions to four other switch modes, where each of these transitions models the truth value change for one of the Boolean switches. Dynamic behavior for each of the switch modes is bounded by invariant constraints, assuring that the switch mode is changed when the switching conditions change.
– The above plant model is still parametric in the controller output. To model the controller's effect, we have created 21 instances of the plant model (with 16 switch modes each), each of them instantiated with one of the 21 possible controller outputs as provided by the lookup table.
– Next, we model the controller's execution, alternating between continuous evolution and the discrete controller execution steps.

  The controller does not modify the continuous variables' values but rather affects the dynamics they are subject to. Furthermore, the switching conditions do not refer to any control parameters. Thus, the controller execution cannot change the switch modes, only the plant instance may potentially induce switching. For example, as depicted in Fig. 18, if the system was in switch mode ℓ_{15} in plant instance 20 and the controller is invoked, the switch mode after controller invocation will again be ℓ_{15}, but possibly in a different plant instance with different dynamics.

  Thus, in order to model the control, we could add a discrete transition from each switch mode ℓ_i in a plant instance to all switch modes ℓ_i in all the plant instances. However, we can achieve a major reduction of the number of transitions needed to model the control step (from $21 \cdot 16 \cdot 21$ to $21 \cdot 16 \cdot 2$) by introducing one controller location for each switch mode ($controller_i$ for switch mode ℓ_i in Fig. 18). Controller execution is then modeled by moving from any switch mode ℓ_i to controller location $controller_i$ and from there to a switch mode ℓ_i in any of the plant instances.
– Finally, we need to model the periodic execution of the controller with cycle time δ. We do so by defining a clock t with derivative 1 and initial value 0. We guard each jump from a switch to a control mode by $t = \delta$ and add a reset

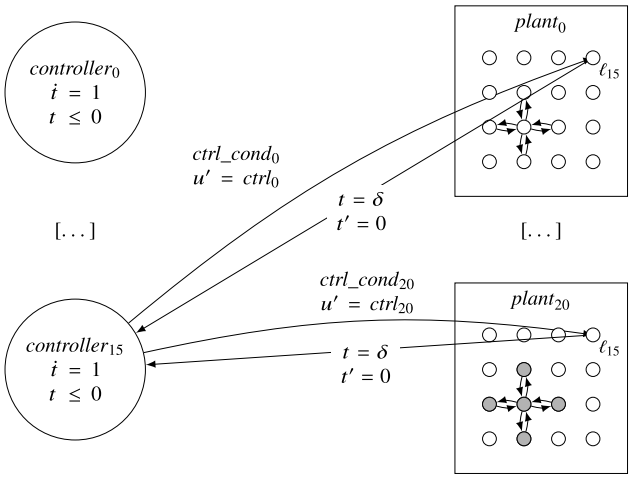


Fig. 18 Visualization of the FCTM system model with plant instances $plant_i$ $(0 \leq i < 21)$, switch modes ℓ_j and controller locations $controller_j$ $(0 \leq j < 16)$; jumps between switch modes are indicated only for one switch mode per plant instance. The jumps to and from controller locations refer to an additional clock t to model the cycle time δ, and the output $ctrl_k$ of the controller under condition $ctrl\_cond_k$. In our evaluation we considered the gray-shaded part of the model first (`fctm_I`) and later a single plant instance (`fctm_II`)

$t := 0$ to it. Additionally, invariant constraints $t \leq \delta$ in each switch mode and $t \leq 0$ in each controller mode ensure that the controller execution is urgent and instantaneous.

The resulting model has 10 continuous variables, $16 + 21 \cdot 16 = 352$ locations and $(21 \cdot 16 \cdot 4) + (21 \cdot 16 \cdot 2) = 672$ transitions. The model size could be reduced, e.g., by removing discrete transitions that can never be enabled (e.g., whose guard condition contradicts to the source location's invariant) However, as we explain below, observed problems during the execution did not occur due to the complexity of the discrete structure of the model but were observable even within a small part of one plant instance.

8.4 Verification

As the dynamics are nonlinear, we have selected the tools CoRA, dReach, and Flow* for the analysis of the model. As a first test, our goal was to compute the sets of all reachable states (within some time and jump bounds) to see whether the tools in general can produce results.

For a first verification attempt we used from a single plant instance with a fixed controller input, a submodel with one location and its four neighboring locations of a specific plant instance with 8 connecting jumps (indicated by the shaded locations in Fig. 18). The characteristics of this subsystem (`fctm_I`) are given in Table 5.

In a first attempt we tried the tool Flow*, however, the tool did not manage to parse the dynamics completely but terminated with an error during parsing. Further attempts in reducing the complexity of the dynamics by replacing single terms by simpler expressions or even constants were unsuccessful. Our final attempt contained only dynamics for one variable (all other variables were treated as constants), the dynamics was specified as a polynomial expression with rational coefficients and degree at most two as well as a square-root expression. Neither this model could be analyzed.

Table 5 High-level characteristics of the fuel cell model (full model: `fctm_F`, subsystem I: `fctm_I`, subsystem II: `fctm_II`) similar to Table 3

Name	hyb	dim	loc	jmp	Ext in	Hypl grd	Zeno
`fctm_I`	✓	10	5	8	×	✓	✓
`fctm_II`	✓	10	16	64	×	✓	✓
`fctm_F`	✓	10	352	672	×	✓	✓

Originally, the system was provided in Simulink. Therefore, using the Matlab-based tool CoRA seemed reasonable. During the evaluation of a single plant instance (16 locations), the tool successfully parsed the dynamics and was able to compute sets of reachable states (instance `fctm_I`) within single locations. However, processing discrete jumps between the switch modes did not succeed: those jumps are guarded by hyperplanes (see Sect. 4.2), for which CoRA's discrete event detection fails.

To overcome this, we tried to enlarge the guards. As an example, a guard which previously was expressed as $x = c$ now was changed to $c - \epsilon \leq x \wedge x \leq c + \epsilon$ for a sufficiently large $\epsilon > 0$. Nonetheless, even with enlargement $\epsilon = 1$ we still could not achieve any conclusive results.

In a last attempt we tried the tool dReach on a single plant instance (`fctm_II`). The tool seemed to compute sets of reachable states but we were not able to estimate their validity, as it was not possible to obtain any visualization of the computed sets of reachable states, and other output was too complex to be understandable by humans.

Developers in industry often work with simulations, whose traces over time can be visualized and may be used to estimate the correctness of a model and the correctness of the computed sets of reachable states. The tool CoRA allows to simulate, but due to the aforementioned issues we were not able to get results and consequently no further insights.

Given the special model structure, probably dedicated methods adapted to this problem structure could be more successful. For example, one could separate discrete variables (i.e., variables whose values do not change during continuous evolution) and handle them with more light-weight computations as in [44]. Unfortunately, the approach proposed in [44] was only implemented for hybrid automata with linear ODEs and thus cannot be applied to our model.

9 Conclusion

The aim of this work was to analyze the applicability and usability of state-of-the-art hybrid systems safety verification tools for industrial applications with a focus on application in the automotive sector. To do so, we have chosen six academic benchmark families (altogether 14 benchmarks) with respect to different criteria that are relevant for the application in the automotive industry. The benchmarks include systems with linear and nonlinear dynamics, contain both hybrid and purely continuous systems. Additionally, benchmarks with variations of crucial parameters such as the state space dimension or the number of discrete locations are included. We have selected six representative tools from the four major different approaches for hybrid systems safety verification and evaluated those tools against the chosen benchmarks. Additionally, we have taken a system from the automotive industry as a more pragmatic benchmark to evaluate the selected tools.

Challenges and observations from the model transformation, tool execution and benchmarking are presented in this work and allow for various conclusions. The key observations made in this work can be summarized as follows:

- *Analytical capabilities.* The results on the academic benchmarks in the first part of the evaluation differ strongly from the results observed on the industrial benchmark. In academia, developers tend to address one challenge at a time and provide an in-depth analysis and solution which is implemented in a tool. In contrast to this, industrial benchmarks exhibit a combination of challenges which explains the observed results.
- *Reliability.* Most tools in this field of research are less than five years old and developed from academic prototypes. Those tools often involve numerical approaches and rely on third-party libraries for their implementation.

  While most tools and the approaches they implement are theoretically capable of handling a certain system, we faced a few errors due to technical problems (erroneous parsing, problems in some third-party libraries, etc.).
- *Scalability.* A central aspect for industrial applications is scalability. Academic research aims to provide solutions to scientifically interesting problems, which might be different to the problems industrial engineers and researchers have to face. Furthermore, available benchmark collections serve the testing, evaluation and comparison of different approaches, therefore they often focus on problems that are hard enough to be challenging but also easy enough to be solved by at least some available methods. Systems from industry and thus also the corresponding models thereof usually are significantly larger in size and require higher computational effort than the academic benchmarks. Furthermore, the composition of models remains challenging for most approaches which is not addressed by most of the tools.
- *Automation.* Different tools and approaches come with different, often incomparable parameter sets of different sizes. Smaller parameter sets simplify the verification process but restrict optimality; adaptive parameters may help the user without loosing optimality. Furthermore, tool parameters are often poorly documented, their meaning and interplay is often unclear. Thus, finding suitable parameter configurations is difficult and requires an understanding of the implemented method.
- *Modeling.* Model construction could be done for all the tools with a reasonable amount of effort. Models can be defined conveniently with the SpaceEx modeling GUI, while the tool Hyst and a SpaceEx-to-CoRA transformator convert SpaceEx-models to certain other input languages. Modeling in programming frameworks (e.g., CoRA, HyLaa) is more flexible but might be also more effortful than invoking binaries.

  An option for larger companies such as Ford Motor Company can be to use their own internal description language of hybrid systems and a conversion tool for the analysis tools in product development as for instance described in [16] for C-code verification.
- *Usability.* All tools evaluated were of academic nature, as a commercialization in this field has not happened, yet.

  Consequently, academic tools usually come with less features, less documentation and support in comparison to commercial tools. The increased usage of artifact evaluations by external reviewers for conference submissions are part of a recent development. Apart from the repeatability of the published experiments, a side-effect is the improvement of the usability and robustness of tools, as external reviewers need to be able to run and modify examples during those evaluations.

  From our observations, documentation and usability of the functionalities, especially parameter descriptions, can be improved for most tools. Additionally, in-depth knowledge of hybrid automata as well as the implemented approach is required for most tools. Continuous interactions with industrial partners might increase sensibility of tool developers for the work flow of industrial engineers and increase the usability. Well-designed GUIs which aid the selection of analysis parameters and combinations thereof strongly improve usability.

Increasing scalability is already a natural academic objective, focusing also on the reliability of academic tools and their capability to analyze systems that combine multiple challenges is crucial to enable their usage also in industrial context. More industrial benchmarks could be an important driving force in the research community, guiding development towards tools that can handle these systems exhibiting multiple challenges at once.

Even if an academic tool is powerful and easy to use, academia cannot assure reliable support by contracts, and cannot replace commercial development. As an example, the tool developed by the company BTC is built from several existing tools originating from academic research as a backend for a commercial C-code verification tool capable of handling industrial applications [15] even though most of those academic tools themselves are not capable of handling industrial applications [49]. This indicates that while results of individual tools might be less positive, a composition of approaches may lead to success.

Technologies are continuously improving and technology transfer to industry is starting, driven most importantly by the urgent need of automated approaches for the safety analysis of complex industrial systems. Though the above observations show that the current academic tools are not ready yet to be embedded as push-button components in industrial processes, impressive academic developments on several keypoints have been achieved in the last decades as the reports on the annually ARCH-competition indicate [7–9].

These developments let us hope that these trends will enable the future industrial usage and contribute to the development of safe and reliable systems.

Appendix A: Detailed description of the selected benchmarks

Building (linear, continuous) The building benchmark (`bld_48`) was proposed to ARCH within a collection of large-scale, continuous systems [48]. This benchmark considers a simplified model of the Los Angeles Hospital, a building with eight floors modeled by a continuous system comprising 48 variables. The displacement and dynamics of each floor is modeled by a beam-model. The goal is to validate, whether an initial displacement converges to a stable equilibrium within reasonable time and with limited overshoot. In the instance considered, the displacement modeled by variable x_{25} should never exceed a bound of 0.006 within the first 20 time units (time horizon). Similarly to the vehicle platoon benchmark (see below), this benchmark is a constant part of the ARCH Friendly Competition.

Vehicle platoon (linear, hybrid) In this work we consider two instances (`plt_30` and `plt_42`) of the vehicle platoon benchmark [13]. This benchmark is one of the first benchmarks proposed to ARCH and has been used in various evaluations, among them it has been a constant part of the ARCH Friendly Competition. The system models a platoon of three vehicles, where the first one is human-driven and the two following ones are autonomous. The vehicle distance, speed and acceleration are modeled via 9 state variables. The following vehicles are equipped with a controller to keep the distance at a certain level. In the model, the vehicles communicate the current distances via radio; the model contains a location for normal operation of the radio and a second location representing disturbed communication in which the dynamics changes as the controller does not get proper input readings. While in the original version of the benchmark communication breakdowns could happen at arbitrary points in time, simplified versions have been used in the ARCH Friendly Competition where the communication breaks down deterministically as most tools cannot handle arbitrary switching.

The safe region in the two models bounds the distance between the vehicles to at least 30 m and 42 m, respectively. The time horizon is set to 20 time units, due to the deterministic switching every 5 time units at most 3 discrete jumps are analyzed.

Filtered oscillator (linear, hybrid) Being designed as a scalable model, the filtered oscillator benchmark comes with four instances (`flt_04`, `flt_08`, `flt_16`, and `flt_32`). This model has been presented in the context of the publication of the tool SpaceEx as one of the benchmarks [28] and been reused in later publications to show performance improvements. The system consists of a switched oscillator system modeled via four locations, where the output of the oscillator is smoothed by a series of first-order filters. In this work, the number next to each instance denotes the number of applied filters. The state space is built from two variables which are used to model the oscillator and additional k variables for a series of k filters. This means that additional filtering is realized by additional variables in the dynamics, i.e., an increased state space dimension.

The specification aims at keeping one of the variables y used to describe the oscillating behavior below a fixed threshold value ($y \leq 0.5$). In our evaluation at most 10 discrete jumps are analyzed with a time horizon of 10 time units.

Steering controller (nonlinear, hybrid) The simplified model of a steering controller was first proposed as a modular system [24]. The system models a steering controller, which aims at keeping a car close to the center of the lane. The lane is divided into seven regions, one for the center and symmetrically three regions left and right of the center depending on the distance to the lane center with different steering dynamics. As only the distance to the center of the lane is considered, the system dynamics in each of the seven locations can be described by two variables. In the original version, a controller for the driving speed was considered as well; we omit this controller and assume constant speed or arbitrary speed from an interval and instead focus on the lane-keeping property of the system. In our work we consider three variations (`ste_05`, `ste_10`, and `ste_RV`) of the benchmark

in which the speed of the vehicle is set to 5 $\mathrm{m\,s^{-1}}$, 10 $\mathrm{m\,s^{-1}}$, or may vary within the interval $[5,10]$ $\mathrm{m\,s^{-1}}$, respectively. Though all instances contain potential Zeno behavior, trajectories (for instance, `ste_05`) should not leave the center region and thus Zeno behavior should not be present.

The specification used in the evaluation is similar to the original one in which the lane should never be left, i.e., the distance to the center of the lane should never exceed a certain boundary (here 10 m). We use a time horizon of 30 time units and a maximum of 10 jumps.

Glycemic control (nonlinear, hybrid) Coming from biology, this family of systems models the amount of blood-sugar over time under different models for insulin discharge [17–19]. The model was extracted by Xin Chen and presented as a part of a benchmark collection [23]. Three different instances have been identified (`gly_01`, `gly_02`, and `gly_03`) which differ in their dynamics and number of locations (two, respectively three). This family of benchmarks was chosen, as it exhibits a larger state space dimension (four and five variables) than the other benchmarks for nonlinear hybrid systems.

The goal in this benchmark is to show that the difference of the blood-sugar concentration to the base-value never exceeds a certain threshold. In the analysis we use a time horizon of 720 time units and a maximal jump depth of 10.

Spring–pendulum (nonlinear, continuous) The model of a mass attached to a spring–pendulum is well known in mechanics [34]. Being a continuous, nonlinear system (`pnd_01`), this model uses transcendental functions to describe the dynamics of the system. In its original version, the dynamics are described as second-order differential equations over two variables—in the used version, the dynamics are converted to a system of first-order differential equations over four variables by adding the intermediate variables.

In our evaluation we used a time horizon of 10 time units. As there is no specification given our aim was to compute reachability only.

Appendix B: Settings for the evaluation

The settings for each tool used for the evaluation are given here. We use the time horizon and jump depths as listed for the benchmarks in Appendix A. As SpaceEx counts the number of executed jumps differently, jump depth and time horizon have been integrated into the formal models by adding auxiliary clocks.

Some of the tools provide default values for parameters in case these are not set by the user. For those tools, if a parameter is not listed below, the default values have been used in the evaluation.

	Building
Acumen	Semantics: 2015 Enclosure, maxTimeStep = 0.015625, endTime = 1
CoRA	step size = 0.005, zonotopeOrder = 50, polytopeOrder = 1, taylorTerms = 4, reduction = girard, guardIntersect = zonoGirard, enclosureEnables = 3;5
dReach	k = 0, l = 0
Flow*	fixed steps = 0.001, remainder estimation = 1e-4, identity precondition, adaptive orders = 3-8, cutoff 1e-15, precision = 53
HyLaa	step size = 0.001
SpaceEx	scenario = supp, directions = box, time step = 0.001, iter-max = 10, rel-err = 1.0e-8, abs-err = 1.0e-12

	Platoon, BND30
Acumen	Semantics: 2015 Enclosure, maxTimeStep = 0.001, endTime = 4
CoRA	step size = 0.02, zonotopeOrder = 200, polytopeOrder = 3, taylorTerms = 4, reduction = girard
dReach	k = 10, l = 0
Flow*	fixed steps = 0.0005, remainder estimation = 1e-4, identity precondition, adaptive orders = 3-8, cutoff 1e-15, precision = 53, max jumps = 3
HyLaa	step size = 0.1
SpaceEx	scenario = supp, directions = oct, time step = 1, fp.-tol = 3, fp.-tol-rel = 0, iter-max = 200, rel-err = 1.0e-9, abs-err = 1.0e-12, set aggregation = none

	Platoon, BND42
Acumen	Semantics: 2015 Enclosure, maxTimeStep = 0.001, endTime = 2.5
CoRA	step size = 0.02, taylorTerms = 4, zonotopeOrder = 200, polytopeOrder = 3, reduction = girard
dReach	k = 10, l = 0
Flow*	fixed steps = 0.005, remainder estimation = 1e-4, identity precondition, adaptive orders = 3-8, cutoff 1e-15, precision = 53, max jumps = 3
HyLaa	step size = 0.1
SpaceEx	scenario = supp, directions = box, time step = 1, fp.-tol = 3, fp.-tol-rel = 0, iter-max = 200, rel-err = 1.0e-9, abs-err = 1.0e-12, set aggregation = none

	Filtered Oscillator
Acumen	Semantics: 2015 Enclosure, maxTimeStep = 0.001, endTime = 4
CoRA	step size = 0.01, zonotopeOrder = 100, polytopeOrder = 1, taylorTerms = 4, reduction = girard, guardIntersect = polytope, enclosureEnables = 3;5
dReach	k = 10, l = 0
Flow*	fixed steps = 0.05, remainder estimation = 1e-5, identity precondition, fixed orders = 8, cutoff 1e-15, precision = 128, max jumps = 10
HyLaa	step size = 0.1
SpaceEx	scenario = supp, directions = box, time step = 0.1, fp.-tol = 1, fp.-tol-rel = 0, iter-max = 10, rel-err = 1.0e-12, abs-err = 1.0e-15, set aggregation = none

Steering Controller	
ACUMEN	Semantics: 2015 Enclosure, maxTimeStep = 0.015625
CORA	step size = 0.1, zonotopeOrder = 100, polytopeOrder = 20, taylorTerms = 10, reduction = girard, guardIntersect = polytope, enclosureEnables = 3;5
DREACH	k = 99999999, l = 4
FLOW*	fixed steps = 0.05, remainder estimation = 1e-4, identity precondition, adaptive orders = 3-8, cutoff 1e-15, precision = 53, max jumps = 9999999

For the steering controller the lower bound on jumps for DREACH is not zero, as the bad states are at earliest reachable (if at all) after four jumps.

Glycemic Control	
ACUMEN	Semantics: 2015 Enclosure, maxTimeStep = 0.015625,
CORA	step size = 0.01/ 0.05 / 0.05, zonotopeOrder = 20, polytopeOrder = 10, taylorTerms = 10, reduction = girard, guardIntersect = polytope, enclosureEnables = 5
DREACH	k = 10, l = 0
FLOW*	fixed steps = 0.05, remainder estimation = 1e-2, identity precondition, adaptive orders = 2-5, cutoff 1e-12, precision = 53, max jumps = 10

Pendulum	
ACUMEN	Semantics: 2015 Enclosure, maxTimeStep = 0.01, endTime = 7
CORA	step size = 0.01, zonotopeOrder = 100, polytopeOrder = 1, taylorTerms = 4, reduction = girard, guardIntersect = polytope, enclosureEnables = 3;5
DREACH	k = 1, l = 1
FLOW*	fixed steps = 0.003, remainder estimation = 1e-10, identity precondition, adaptive orders = 6-8, cutoff 1e-12, precision = 53

Note that for the model of the pendulum system for DREACH an extra location indicating the violation has been introduced which is why the bounds for the unrolling are chosen equal to one instead of being zero as, for instance, in the building benchmark.

Appendix C: Code examples

We give code examples (Algorithms 3–7) for the bouncing ball for all tools but SPACEEX and FLOW*, which are shown in the Figures Algorithm 1 resp. Algorithm 2.

Algorithm 3 Code example for ACUMEN for the bouncing ball system

```
model Main(simulator) =
initially
  x = [10 .. 10.2],
  x' = 0, x'' = 0
always
  claim x >= 0,
  if x == 0 && x' < 0 then
  x'+ = -0.75*x'
  else
  x'' = -9.81,

hypothesis "x less than initial state" x <=
    10.21,
hypothesis "x larger than zero" x >= 0,

simulator.endTime += 5
```

Algorithm 4 Code example for CoRA for the bouncing ball system

```
function res = bouncing_ball()

  options.x0= [10.1; 0];
  options.R0 = zonotope([options.x0, 0.1 * [1,
       0]]);
  options.startLoc = 1;

  [ ... settings ... ]

  A = [0 1; 0 0];
  B = [0;0];
  c = [0; -9.81];
  dynamics = linearSys('linearSys', A, B, c);

  invariant = mptPolytope(struct('A', [-1 0],
      'b', 0));

  reset1.A = [1,0;0,-0.75];
  reset1.b = zeros(2,1);

  guard1 = mptPolytope(struct('A',
      [-1,0;1,0;0,1],'b', [0;0;0]));
  transitions{1} = transition(guard, reset, 1,
       'ball','ball');

  options.uLoc{1} = 0;
  options.uLocTrans{1} = 0;
  options.Uloc{1} = zonotope(0);

  loc{1} = location('ball', 1, invariant,
      transitions , dynamics);

  HA = hybridAutomaton(loc);

  [HA] = reach(HA,options);
  figure
  hold on
  options.projectedDimensions = [1,2];
  options.plotType = 'b';
  plot(HA,'reachableSet',options); %plot
      reachable set
  res = 1;
```

Algorithm 5 Code example for dReach for the bouncing ball system

```
[0, 15] x;
[-18, 18] v;
[0, 5] time;

{ mode 1;
  invt:
    (x >= 0);
  flow:
    d/dt[x] = v;
    d/dt[v] = -9.81;
  jump:
    (and (x = 0) (v < 0)) ==> @1 (and (x' = x)
         (v' = -0.75*v));
}
init:
@1 (and (x >= 10) (x <= 10.2) (v = 0));

goal:
@1 (and (x >= 10.2));
```

Algorithm 6 Code example for HyLaa for the bouncing ball system

```
[ ... imports ...]

def define_ha():
  ha = HybridAutomaton()
  # dynamics variable order: [x, v, affine]

  ball = ha.new_mode('ball')
  a_matrix = [ \
    [0, 1, 0], \
    [0, 0, -9.81], \
    [0, 0, 0], \
    ]
  ball.set_dynamics(a_matrix)
  ball.set_invariant([[-1, 0, 0]], [0])

  _error = ha.new_mode('_error')

  # transitions
  jump = ha.new_transition(ball, ball)
  jump.set_guard([
      [-1, 0, 0], \
      [1, 0, 0], \
      [0, 1, 0], \
      ], [0, 0, 0])
  reset_csr = [[1, 0, 0], \
               [0, -0.75, 0], \
               [0, 0, 1], \
               ]
  jump.set_reset(reset_csr)

  ha.new_transition(ball,_error).set_guard
      ([[1,0,0],], [0,])
  ha.new_transition(ball,_error).set_guard
      ([[-1,0,0],], [-10.2,])
  return ha

def define_init_states(ha):
  init_lpi = lputil.from_box([(10, 10.2), (0,
      0), (1, 1)], ball)
  init_list = [StateSet(init_lpi, ball)]
  return init_list

def define_settings(image_path):
  [ ... ]
  return settings

def run_hylaa(image_path):
  ha = define_ha()
  init = define_init_states(ha)
  settings = define_settings(image_path)
  result = Core(ha, settings).run(init)
  return result

if __name__ == '__main__':
  image_path = 'out.png'
  run_hylaa(image_path)
```

Algorithm 7 Code example for KeYmaeraX for the bouncing ball system

```
ArchiveEntry "Bouncing Ball"

Definitions
  Real H;
End.

ProgramVariables
  Real x, v;
End.

Problem
  (x=H & H >= 10 & H<=10.2 & v=0)
  ->
  [
  {
    {x'=v,v'=-9.81 & x>=0}
    {?x=0 & v < 0; v:=-0.75*v;  ++  ?x!=0;}
  }* @invariant(2*9.81*x=2*9.81*H-v^2 & x>=0)
  ] (x>=0 & x<=H)
End.

End.
```

Acknowledgements We are grateful to Amey Karnik and his Ford team for sharing the model of the fuel cell system with us. We thank Liren Yang and his University of Michigan team for providing and patiently explaining the controller of this fuel cell system. We appreciate the help of Tristan Ebert, Marta Grobelna, Sergej Neuberger, and Tom Schäfers in the evaluation. We also thank our anonymous reviewers for their detailed feedback.

Funding Open Access funding enabled and organized by Projekt DEAL. We are thankful for the funding of this work by Ford Motor Company in the course of the project "Safety Verification for Mixed Discrete–Continuous Automotive Systems".

References

1. https://cps-vo.org/group/ARCH/FriendlyCompetition
2. https://www.wolfram.com/engine/
3. https://cps-vo.org/group/ARCH/benchmarks
4. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification – The KeY Book – From Theory to Practice. LNCS, vol. 10001. Springer, Berlin (2016)

5. Alla, H., David, R.: Continuous and hybrid Petri nets. J. Circuits Syst. Comput. **8**(01), 159–188 (1998)
6. Althoff, M.: An introduction to CORA 2015 (tool presentation). In: Proc. of ARCH'15. EPiC Series in Computing, vol. 34, pp. 120–151. EasyChair (2015)
7. Althoff, M., Bak, S., Cattaruzza, D., Chen, X., Frehse, G., Ray, R., Schupp, S.: ARCH-COMP17 category report: continuous and hybrid systems with linear continuous dynamics. In: Proc. of ARCH'17. EPiC Series in Computing, vol. 48, pp. 143–159. EasyChair (2017)
8. Althoff, M., Bak, S., Chen, X., Fan, C., Forets, M., Frehse, G., Kochdumper, N., Li, Y., Mitra, S., Ray, R., Schilling, C., Schupp, S.: ARCH-COMP18 category report: continuous and hybrid systems with linear continuous dynamics. In: Proc. of ARCH'18. EPiC Series in Computing, vol. 54, pp. 23–52. EasyChair (2018)
9. Althoff, M., Bak, S., Forets, M., Frehse, G., Kochdumper, N., Ray, R., Schilling, C., Schupp, S.: ARCH-COMP19 category report: continuous and hybrid systems with linear continuous dynamics. In: Proc. of ARCH'19. EPiC Series in Computing, vol. 61, pp. 14–40. EasyChair (2019)
10. Bak, S., Caccamo, M.: Computing reachability for nonlinear systems with HyCreate (2013). Poster at HSCC'13
11. Bak, S., Duggirala, P.S.: Hylaa: a tool for computing simulation-equivalent reachability for linear systems. In: Proc. of HSCC'17, pp. 173–178. ACM, New York (2017)
12. Bak, S., Bogomolov, S., Johnson, T.T.: Hyst: a source transformation and translation tool for hybrid automaton models. In: Proc. of HSCC'15, pp. 128–133. ACM, New York (2015)
13. Ben Makhlouf, I., Kowalewski, S.: Networked cooperative platoon of vehicles for testing methods and verification tools. In: Proc. of ARCH'14. EPiC Series in Computing, vol. 34, pp. 37–42. EasyChair (2014)
14. Benvenuti, L., Bresolin, D., Casagrande, A., Collins, P., Ferrari, A., Mazzi, E., Sangiovanni-Vincentelli, A., Villa, T.: Reachability computation for hybrid systems with ariadne. IFAC Proc. Vol. **41**(2), 8960–8965 (2008)
15. Berger, P., Katoen, J.P., Ábrahám, E., Waez, M.T.B., Rambow, T.: Verifying auto-generated C code from Simulink. In: Proc. of FM'18, pp. 312–328. Springer, Berlin (2018)
16. Berger, P., Nellen, J., Katoen, J.P., Ábrahám, E., Waez, M.T.B., Rambow, T.: Multiple analyses, requirements once: simplifying testing and verification in automotive model-based development. In: Proc. of FMICS'19. LNCS, vol. 11687, pp. 59–75. Springer, Berlin (2019)
17. Bergman, R.N., Ider, Y.Z., Bowden, C.R., Cobelli, C.: Quantitative estimation of insulin sensitivity. Am. J. Physiol: Endocrinol. Metab. **236**(6), E667 (1979)
18. Bergman, R.N., Phillips, L.S., Cobelli, C.: Physiologic evaluation of factors controlling glucose tolerance in man: measurement of insulin sensitivity and beta-cell glucose sensitivity from the response to intravenous glucose. J. Clin. Invest. **68**(6), 1456–1467 (1981)
19. Bergman, R.N., Finegood, D.T., Ader, M.: Assessment of insulin sensitivity in vivo. Endocr. Rev. **6**(1), 45–86 (1985)
20. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Springer, Berlin (2013)
21. Bogomolov, S., Forets, M., Frehse, G., Potomkin, K., Schilling, C.: JuliaReach: a toolbox for set-based reachability. In: Proc. of HSCC'19, pp. 39–44. ACM, New York (2019)
22. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: an analyzer for non-linear hybrid systems. In: Proc. of CAV'13, pp. 258–263. Springer, Berlin (2013)
23. Chen, X., Schupp, S., Ben Makhlouf, I., Ábrahám, E., Frehse, G., Kowalewski, S.: A benchmark suite for hybrid systems reachability analysis. In: Proc. of NFM'15, pp. 408–414. Springer, Berlin (2015)
24. Damm, W., Möhlmann, E., Rakow, A.: Component based design of hybrid systems: a case study on concurrency and coupling. In: Proc. of HSCC'14, pp. 145–150. ACM, New York (2014)
25. Donzé, A., Frehse, G.: Modular, hierarchical models of control systems in SpaceEx. In: Proc. of ECC'13, pp. 4244–4251. IEEE, New York (2013)
26. Fan, C., Qi, B., Mitra, S., Viswanathan, M., Duggirala, P.S.: Automatic reachability analysis for nonlinear hybrid models with C2E2. In: Proc. of CAV'16, pp. 531–538. Springer, Berlin (2016)
27. Fehnker, A., Ivančić, F.: Benchmarks for hybrid systems verification. In: Proc. of HSCC'04, pp. 326–341. Springer, Berlin (2004)
28. Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: scalable verification of hybrid systems. In: Proc. of CAV'11, pp. 379–395. Springer, Berlin (2011)
29. Fulton, N., Mitsch, S., Quesel, J.D., Völp, M., Platzer, A.: KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. In: Proc. of CADE'15, pp. 527–538. Springer, Berlin (2015)
30. Henzinger, T.A.: The theory of hybrid automata. In: Verification of Digital and Hybrid Systems, pp. 265–292. Springer, Berlin (2000)
31. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? J. Comput. Syst. Sci. **57**(1), 94–124 (1998)
32. Kong, S., Gao, S., Chen, W., Clarke, E.: dReach: δ-reachability analysis for hybrid systems. In: Proc. of TACAS'15, pp. 200–205. Springer, Berlin (2015)
33. Masood, J., Philippsen, R., Duracz, J., Taha, W., Eriksson, H., Grante, C.: Domain analysis for standardised functional safety: a case study on design-time verification of automatic emergency braking. In: Proc. of FISITA'14, pp. 2–6. KIVI (2014)
34. Meiss, J.D.: Differential Dynamical Systems, vol. 14. SIAM, Philadelphia (2007)
35. Mishra, A., Roy, S.K.: Towards formal verification of adaptive cruise controller using SpaceEx. In: Proc. of VLSI-SATA'16, pp. 1–6. IEEE, New York (2016)
36. Müller, A., Mitsch, S., Platzer, A.: Verified traffic networks: component-based verification of cyber-physical flow systems. In: Proc. of ITSC'15, pp. 757–764. IEEE, New York (2015)
37. Nellen, J., Rambow, T., Waez, M.T.B., Ábrahám, E., Katoen, J.P.: Formal verification of automotive Simulink controller models: empirical technical challenges, evaluation and recommendations. In: Proc. of FM'18, pp. 382–398. Springer, Berlin (2018)
38. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, vol. 2283. Springer, Berlin (2002)
39. Owre, S., Rushby, J.M., Shankar, N.: Pvs: a prototype verification system. In: Proc. of CADE-11, pp. 748–752. Springer, Berlin (1992)
40. Pence, B.L., Chen, J.: A framework for control oriented modeling of Pem fuel cells. In: Proc. of DSCC'15, vol. 57250, p. V002T26A002. American Society of Mechanical Engineers, New York (2015)
41. Platzer, A.: Differential-algebraic dynamic logic for differential-algebraic programs. J. Log. Comput. **20**(1), 309–352 (2010)
42. Ratschan, S., She, Z.: Safety verification of hybrid systems by constraint propagation-based abstraction refinement. ACM Trans. Embed. Comput. Syst. **6**(1), 8–31 (2007)
43. Schupp, S., Ábrahám, E., Ben Makhlouf, I., Kowalewski, S.: HyPro: a C++ library for state set representations for hybrid systems reachability analysis. In: Proc. of NFM'17. LNCS, vol. 10227, pp. 288–294. Springer, Berlin (2017)
44. Schupp, S., Nellen, J., Ábrahám, E.: Divide and conquer: variable set separation in hybrid systems reachability analysis. In: Proc. of QAPL'17, EPTCS, vol. 250, pp. 1–14. Open Publishing Association (2017)
45. Schupp, S., Ábrahám, E., Ebert, T.: Recent developments in theory and tool support for hybrid systems verification with hypro. Inf. Comput. **289**, 104945 (2022)

46. Taha, W., Duracz, A., Zeng, Y., Atkinson, K., Bartha, F.A., Brauner, P., Duracz, J., Xu, F., Cartwright, R., Konečnỳ, M., et al.: Acumen: an open-source testbed for cyber-physical systems research. In: Proc. of IIoTS'15, pp. 118–130. Springer, Berlin (2015)
47. Testylier, R., Dang, T.: NLTOOLBOX: a library for reachability computation of nonlinear dynamical systems. In: Proc. of ATVA'13, pp. 469–473. Springer, Berlin (2013)
48. Tran, H.D., Nguyen, L.V., Johnson, T.T.: Large-scale linear systems from order-reduction (benchmark proposal). In: Proc. of ARCH'16. EPiC Series in Computing, vol. 43, pp. 60–67. EasyChair (2016)
49. Westhofen, L., Berger, P., Katoen, J.P.: Benchmarking software model checkers on automotive code. Preprint, CoRR (2020). arXiv:2003.11689
50. Yang, L., Karnik, A., Pence, B., Waez, M.T.B., Ozay, N.: Fuel cell thermal management: modeling, specifications, and correct-by-construction control synthesis. IEEE Trans. Control Syst. Technol. **28**, 1638–1651 (2020)