Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

How to Model and Prove Hybrid Systems with KeYmaera: A Tutorial on Safety*

Jan-David Quesel, Stefan Mitsch, Sarah Loos, Nikos Aréchiga, André Platzer

Carnegie Mellon University, Pittsburgh, PA, USA e-mail: {jquesel|smitsch|sloos|arechiga|aplatzer}@cmu.edu

The date of receipt and acceptance will be inserted by the editor

Abstract. This paper is a tutorial on how to model hybrid systems as hybrid programs in differential dynamic logic and how to prove complex properties about these complex hybrid systems in KeYmaera, an automatic and interactive formal verification tool for hybrid systems. Hybrid systems can model highly nontrivial controllers of physical plants, whose behaviors are often safety critical such as trains, cars, airplanes, or medical devices. Formal methods can help design systems that work correctly. This paper illustrates how KeYmaera can be used to systematically model, validate, and verify hybrid systems. We develop tutorial examples that illustrate challenges arising in many real-world systems. In the context of this tutorial, we identify the impact that modeling decisions have on the suitability of the model for verification purposes. We show how the interactive features of KeYmaera can help users understand their system designs better and prove complex properties for which the automatic prover of KeYmaera still takes an impractical amount of time. We hope this paper is a helpful resource for designers of embedded and cyber-physical systems and that it illustrates how to master common practical challenges in hybrid systems verification.

1 Introduction

Hybrid systems [3, 17, 26] feature both discrete and continuous dynamics, which is important for modeling and understanding systems with computerized or embedded controllers for physical systems. Prime examples of hybrid systems include cars [18, 34], aircraft [56, 63, 64], trains [58], robots [43], and even audio protocols [27]. The design of any controller for these systems is critical, because malfunctions may have detrimental consequences to the system operation. A number of formal verification techniques have been developed for hybrid systems, but verification is still challenging for complex applications [2]. Experience can make a big difference when making trade-offs to decide on a modeling style, on the most suitable properties to consider, and on the best way to approach the verification task.

This article introduces hybrid system modeling with differential dynamic logic [44, 45, 47, 50, 51]. Furthermore, we explain how to prove complex properties of hybrid systems with our theorem prover KeYmaera [57]. We intend this paper to be a valuable resource for system designers and researchers who face design challenges in hybrid systems and want to learn how they can successfully approach their verification task. Formal verification is a challenging task, but we argue that it is of utmost importance for safety-critical designs, and the coverage benefits compared to traditional incomplete system testing far outweigh the cost. Especially, the possibility of checking and dismissing designs early in the development cycle reduces the risk of design flaws causing costly downstream effects.

Even though some of our findings apply to other verification tools, we focus on KeYmaera [57] in this paper. KeYmaera implements differential dynamic logic [44, 45, 47, 50, 51], which is a specification and verification logic for hybrid systems. KeYmaera is based on KeY [8], and is presently the premier theorem prover for hybrid systems. For formal details and more background on the approach behind KeYmaera, we refer to the literature [45–47]. KeYmaera has now matured to a powerful verification tool that has been used successfully to verify cars [34, 37], aircraft [29, 56], trains [58], robots [36], and

* This material is based upon work supported by the National Science Foundation under NSF CAREER Award CNS-1054246, NSF EXPEDITION CNS-0926181, NSF CNS-0931985, and CNS-1035800. This research was partially supported by the German Research Council (DFG) in SFB/TR 14 AVACS. The second author is supported by the ERC under grant PIOF-GA-2012-328378. The third author is supported by DOE CSGF.

surgical robots [32], and to verify practical, real-world control schemes such as PID [6, 58]. Similar to all other verification tools, some decisions in the modeling, specification, and proof approach make verification unnecessarily tedious, while others are computationally more effective. Relative completeness results [45, 47] identify exactly which decisions are critical, but even the decisions that are not can have a dramatic impact on the effectiveness of the verification process in practice [47].

We identify best practices for hybrid systems verification that help practitioners and researchers verify hybrid systems with KeYmaera more effectively. We develop a series of tutorial examples that illustrate how to master increasingly complicated challenges in hybrid systems design and verification. These examples are carefully chosen to illustrate common phenomena that occur in practice, while being easier to understand than the full details of our specific case studies[1]: here, we illustrate hybrid systems and KeYmaera by considering motion in a series of car models. We emphasize that KeYmaera is in no way restricted to car dynamics but has been shown to work for more general dynamics, including hybrid systems with nonlinear differential equations, differential inequalities, and differential-algebraic constraints.

2 Introduction to Hybrid Systems Modeling

In this section we exemplify the main concepts of hybrid systems, before we introduce hybrid programs, a program notation for hybrid systems.

2.1 Hybrid Systems by Example

Hybrid systems, as already mentioned, comprise continuous and discrete dynamics. The movement of cars (i. e., their continuous dynamics) can be described by differential equations. Kinematic models based on Newton's laws of mechanics are sufficient for basic car interactions where p is the position of the car, v its velocity and a its acceleration. All these state variables are functions in time t. They observe the following ordinary differential equation (ODE):

$$\left(\frac{\mathsf{d}p}{\mathsf{d}t} = v,\ \frac{\mathsf{d}v}{\mathsf{d}t} = a\right) \equiv (p' = v,\ v' = a) \tag{1}$$

This ODE models that the position p of the car changes over time with velocity v, and that the velocity v changes with acceleration a. As time domain we use the non-negative real numbers, denoted by $\mathbb{R}_{\geq 0}$, and instead of $\frac{\mathsf{d}p}{\mathsf{d}t}$ we write p' for the time-derivative of p, so that the right side of (1) is the notation in KeYmaera. It is equivalent to the ODE on the left side of (1).

[1] Specific case studies include cars [34, 37], aircraft [29, 56], trains [58], robots [36], and surgical robots [32]. The models of these case studies are included in KeYmaera.

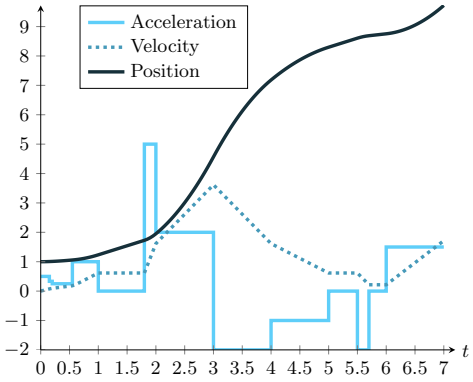


Fig. 1: One example trace of a hybrid system for car dynamics (1) with the acceleration signal changing as indicated over time and the velocity and position following according to (1)

Notice that equation (1) does not explicitly specify whether the acceleration a evolves over time. KeYmaera follows the *explicit change principle*. That is, no variable changes unless the model explicitly specifies how it changes. In particular, the absence of the derivative of a in (1) indicates a is constant during this continuous evolution. If we want acceleration a to evolve, then we need to specify how, for example, by adding another differential equation $a' = j$ where j is the jerk, and then j is implicitly constant during the continuous evolution.

If we want to model an *analog controller* for a, we can replace a in (1) by a term that describes how the analog controller sets the acceleration a, depending on the current position p and velocity v. For example, if v_s is the set-value for the velocity, we could describe a simple proportional controller with gain K_p that moves v toward the intended v_s by the differential equation $p' = v, v' = K_p(v - v_s)$.

A common alternative is to use a *discrete controller*, which turns the purely continuous dynamical system into a *hybrid system* that exhibits both discrete and continuous dynamics. A discrete controller instantaneously sets values at particular points in time. An example trajectory is shown in Fig. 1 for the car dynamics (1), controlled by a discrete controller for the acceleration a that changes its values at various instants in time as indicated. The figure traces the values of the system state variables p, v, and a over (real-valued) time t. The acceleration a changes its value instantaneously according to some discrete controller (not specified in (1)) and this effect propagates to the velocity and position according to the relations given by the differential equation (1).

Given a target speed v_s suppose we want to build a discrete controller for the acceleration a that chooses a constant positive acceleration of A if the current speed is too low and a constant deceleration of $-B$ if it is too high. Formula (2) shows a hybrid system that includes

such a controller.

$$\big(\text{if } v \leq v_s \text{ then } a := A \text{ else } a := -B \text{ fi};\ (p' = v, v' = a)\big)^* \quad (2)$$

The first statement here is a case distinction started with if and ended with fi. It first checks whether the current velocity v is less than or equal to the desired velocity v_s (i. e., whether $v \leq v_s$ holds). If that is the case then the car chooses to accelerate by executing $a := A$. This means that the value of a gets updated to the value of A. In the following, we assume A is a symbolic constant denoting the maximal acceleration. Otherwise, i. e., if $v > v_s$ then the assignment $a := -B$ gets executed, assigning the maximal deceleration of $-B$ to a. The operator ; is for sequential composition. That is, after the first statement finishes (here, the if statement) the next statement is executed, here the differential equation system $(p' = v, v' = a)$ from (1). Hence after the controller chooses an acceleration, the variables evolve according to the solution of this differential equation system for some time. During this evolution the acceleration a is constant. The operator * at the end of (2) denotes nondeterministic repetition like in a regular expression. That is, the sequence of the discrete controller and the differential equation system are repeated arbitrarily often. In this example, the loop enables the discrete controller to update the acceleration repeatedly any number of times.

A common and useful assumption when working with hybrid systems is that discrete actions do not consume time (whenever they do consume time, it is easy to transform the model to reflect this just by adding explicit extra delays). Because discrete actions are assumed not to consume time, multiple discrete actions can occur while the continuous dynamics do not evolve.

The model (2) does not specify when the continuous evolution stops to give the discrete controller another chance to react. This is because the number of loop iterations as well as the durations of the respective continuous evolutions are chosen nondeterministically (even no repetition and evolution for zero duration are allowed). Evolution domain constraints in differential equations can introduce bounds on the continuous dynamics. We model an upper bound on time in (3) with a clock variable c. That is, we ensure that at least every ε time units the discrete controller can take action.

$$\big(\text{if } v \leq v_s \text{ then } a := A \text{ else } a := -B \text{ fi};\ c := 0;\ (p' = v, v' = a, c' = 1 \ \&\ c \leq \varepsilon)\big)^* \quad (3)$$

The clock c is reset to zero by the discrete assignment $c := 0$ before every continuous evolution and then evolves with a constant rate of $c' = 1$. The formula $c \leq \varepsilon$ that is separated from the differential equation by & is an *evolution domain constraint*. Evolution domain constraints are formulas that restrict the continuous evolution of the system to stay within that domain. This means the continuous evolution starts within the specified domain and must stop before it leaves this region. Therefore, the continuous evolution in (3) evolves for at most ε time units. The evolution can still take any amount of time, nondeterministically, just not longer than ε. As a result, the discrete controller is invoked at least every ε time units because any continuous evolution for more than ε time units violates the evolution domain constraint $c \leq \varepsilon$. This modeling paradigm ensures that if the discrete control happens to react faster than within ε time (e. g., if new sensor data is returned early), then it will still satisfy the same safety properties.

Note that the model (3) only puts an upper bound on the duration of a continuous evolution, not a lower bound. The discrete controller can react faster than ε and, in fact, in Fig. 1, it does react more often. However, if a lower bound is desired for a given example, it can easily be included by using a test that allows the controller to execute only after a given time has elapsed. Tests are discussed below, but we will not discuss lower bounds on sampling in our examples, because they are usually not needed for the safety argument.

The next extension to our model adds nondeterministic choice of the acceleration. If, as in (4), we replace the assignment $a := A$ by $a := A \cup a := 0$ (read "a becomes A *or* a becomes 0"), then the controller can always choose to keep its current velocity instead of accelerating further. We use $\cup$ to denote nondeterministic choice, meaning the program can follow either side, in this case setting a to 0 or setting a to A.

$$\big(\text{if } v \leq v_s \text{ then } a := A \cup a := 0 \text{ else } a := -B \text{ fi};\ c := 0; (p' = v, v' = a, c' = 1 \ \&\ c \leq \varepsilon)\big)^* \quad (4)$$

In summary, nondeterministic choice $\cup$, repetition *, differential equations, and assignment are important modeling constructs for safety verification purposes. They allow us to capture the safety-critical aspects of many different controllers all within a single model. Their nondeterministic nature helps us not to take an overly narrow view of the behavior that we want to cover because it might occur during some system runs.

2.2 Hybrid Programs

The program model for hybrid systems that we have illustrated by example is called *hybrid programs* (HP) [45–47, 50, 51]. The syntax of hybrid programs is shown together with an informal semantics in Table 1. KeYmaera also supports an ASCII variation of the notation in Table 1. The basic terms (called θ in the table) are either rational number constants, real-valued variables or (possibly nonlinear) polynomial or rational arithmetic expressions built from those.

The effect of $x := \theta$ is an instantaneous discrete jump assigning the value of θ to the variable x. For example in

Table 1: Statements of hybrid programs (F is a first-order formula, α, β are hybrid programs)

Statement	Effect
$\alpha;\ \beta$	sequential composition where β starts after α finishes
$\alpha\ \cup\ \beta$	nondeterministic choice, following either alternative α or β
α^*	nondeterministic repetition, repeating α n times for any $n \in \mathbb{N}$
$x := \theta$	discrete assignment of the value of term θ to variable x (jump)
$x := *$	nondeterministic assignment of an arbitrary real number to x
$(x_1' = \theta_1, \ldots,$ $x_n' = \theta_n \& F)$	continuous evolution of x_i along the differential equation system $x_i' = \theta_i$ restricted to evolution domain F
$?F$	test if formula F holds at current state, abort otherwise
if F then α fi	perform α if F is true at current state, do nothing otherwise
if F then α else β fi	perform α if F is true at current state, perform β otherwise

Fig. 1, the acceleration a changes instantaneously at time 1.8 from 0 to 5, by the discrete jump $a := A$ when A has value 5. The term θ can be an arbitrary polynomial. For a car with current velocity v the deceleration necessary to come to a stop within distance m is given by $-\frac{v^2}{2m}$. The controller could assign this value to the acceleration by the assignment $a := -\frac{v^2}{2m}$, provided $m \neq 0$.

The effect of $x' = \theta \,\&\, F$ is an ongoing continuous evolution controlled by the differential equation $x' = \theta$ that is restricted to remain within the evolution domain F, which is a formula of real arithmetic over unprimed variables. The evolution is allowed to stop at any point in F but it must not leave F. Systems of differential equations and higher-order derivatives are defined accordingly: $p' = v, v' = -B \,\&\, v \geq 0$, for instance, characterizes the braking mode of a car with braking force B that holds within $v \geq 0$ and stops any time before $v < 0$. The extension to systems of differential equations is straightforward, see [45–47].

For discrete control, the test action $?F$ (read as "*assume* F") is used as a condition statement. It succeeds without changing the state if F is true in the current state, otherwise it aborts all further evolution. For example, a car controller can check whether the chosen acceleration is within physical limits by $?(-B \leq a \leq A)$. If a computation branch does not satisfy this condition, that branch is discontinued and aborts. From a modeling perspective, tests should only fail if a branch is not possible in the original system, as it will no longer be possible in the model of the system. Therefore, during verification we consider only those branches of a system where all tests succeed.

From these basic constructs, more complex hybrid programs can be built in KeYmaera similar to regular expressions. The *sequential composition* $\alpha;\beta$ expresses that hybrid program β starts after hybrid program α finishes, as in expression (2). The nondeterministic choice $\alpha \cup \beta$ expresses alternatives in the behavior of the hybrid system that are selected nondeterministically. Nondeterministic repetition α^* says that the hybrid program α repeats an arbitrary number of times, including zero. These operations can be combined to form any other control structure.

For instance, $(?v \leq v_s;\ a := A) \cup (?v \geq v_s;\ a := -B)$ says that, depending on the relation of the current speed v of some car and a given target speed v_s, a is chosen to be the maximum acceleration A if $v \leq v_s$ or maximum deceleration $-B$ if $v \geq v_s$. If both conditions are true (hence, $v = v_s$) the system chooses either way. Note that the choice between the two branches is made nondeterministically. However, the test statements abort the program execution if the left branch was chosen in a state where $v \leq v_s$ does not hold, or the right branch was chosen in a state where $v \geq v_s$ was not satisfied. In other words, only one choice works out unless $v = v_s$ in which case either $a := A$ or $a := -B$ can run. As abbreviations, KeYmaera supports *if-statements* with the usual meaning from programming languages. The if-statement can be expressed using the test action, sequential composition and the choice operator.

$$\text{if } F \text{ then } \alpha \text{ fi } \equiv (?F;\alpha) \cup (?\neg F)$$
$$\text{if } F \text{ then } \alpha \text{ else } \beta \text{ fi } \equiv (?F;\alpha) \cup (?\neg F;\beta)$$

Its semantics is that if condition F is true, the then-part α is executed, otherwise the else-part β is performed, if there is one, otherwise the statement is just skipped. Note that even though we use nondeterministic choice in the encoding, the choice becomes deterministic as the conditions in the test actions are complementary, so exactly one of the two tests $?F$ and $?\neg F$ fails in any state.

The *nondeterministic assignment* $x := *$ assigns any real value to x. That is, every time $x := *$ is run, an arbitrary real number will be put into x, possibly a different one every time. Thereby, $x := *$ expresses unbounded nondeterminism that can be used, for example, for modeling choices for controller reactions. For instance, the program $a := *; ?a > 0$ nondeterministically assigns any positive value to the acceleration a, because only positive choices for the value of a will pass the subsequent test $?a > 0$. Any nonpositive assignments will be discarded.

2.3 Differential Dynamic Logic

KeYmaera implements *differential dynamic logic* dL [44, 45, 47, 50, 51] as a specification and verification language for hybrid systems. The formulas of dL can be used to specify the properties of the hybrid systems of interest. The logic dL also comes with a proof calculus [44, 45, 47, 50, 51] that has been implemented in KeYmaera and can be used to prove these properties and, thus, verify their correctness.

Within a single specification and verification language, dL combines operational system models with means to talk about the states that are reachable by system transitions. The dL formulas are built using the operators in Table 2 where $\sim \in \{>, \geq, =, \neq, \leq, <\}$ is a comparison operator and θ_1, θ_2 are arithmetic expressions in $+, -, \cdot, /$ over the reals. The logic dL provides parametrized modal operators $[\alpha]$ and $\langle\alpha\rangle$ that refer to the states reachable by hybrid program α and can be placed in front of any formula. The formula $[\alpha]\phi$ expresses that all states reachable by hybrid program α satisfy formula ϕ. So $[\alpha]\phi$ is true in exactly those states from which running α only leads to states that satisfy ϕ. Likewise, $\langle\alpha\rangle\phi$ expresses that there is at least one state reachable by α for which ϕ holds. These modalities can be used to express necessary or possible properties of the transition behavior of α in a natural way. They can be nested or combined propositionally. For example $[\alpha]\phi \wedge [\beta]\psi$ is true in those states where all executions of α lead to states satisfying ϕ and executing β only reaches states satisfying ψ. Using modalities and propositional connectives, we can express Hoare triples $\{\phi\}\alpha\{\psi\}$ for hybrid systems by $\phi \rightarrow [\alpha]\psi$. Here the formula ϕ serves as a precondition. This means that the system must satisfy the postcondition ψ after all ways of running α only if the initial state satisfied ϕ. The logic dL supports quantifiers like in $\exists p\,[\alpha]\langle\beta\rangle\phi$, which says that there is a choice of parameter p (expressed by $\exists p$) such that for all possible behaviors of hybrid program α (expressed by $[\alpha]$) there is a reaction of hybrid program β (expressed by $\langle\beta\rangle$) that ensures ϕ. Likewise, $\exists p\,([\alpha]\phi \wedge [\beta]\psi)$ says that there is a choice of parameter p that makes both $[\alpha]\phi$ and $[\beta]\psi$ true, simultaneously. This is, the choice makes $[\alpha]\phi \wedge [\beta]\psi$ true, i. e., the formula ϕ holds for all states reachable by α executions and, independently, ψ holds after all β executions. This gives a flexible logic for specifying and verifying even sophisticated properties of hybrid systems, including the ability to refer to multiple hybrid systems at once. The variables quantified over can occur in any hybrid program or formula, so quantifiers can be used to quantify over system parameters as well as parameters in pre- and postconditions.

Note that differential equations of dL [45] constitute a crucial generalization compared to discrete dynamic logic [25, 59]. Another important change is that dL is defined over the domain $\mathbb{R}$, not natural numbers. The formal semantics of differential dynamic logic and more details about it can be found in [44, 45, 47, 50, 51].

3 Proving with KeYmaera

KeYmaera [57] is an interactive theorem prover. Its input is a single formula of differential dynamic logic combining both the system description and the property under consideration. To prove this formula, it is safely decomposed into several subtasks according to the proof rules of dL [44, 45, 47, 50, 51]. The boolean structure of the input formula is successively transformed into a proof tree (where applicable). Programs are handled by symbolic execution. That is for each program construct there is a proof rule that calculates its effect. For instance, assignments $x := \theta$ can be handled by replacing every occurrence of x by its new value θ in the postcondition. Choices in the program flow are explored separately. For example, $[\alpha \cup \beta]\phi$ is true if and only if $[\alpha]\phi$ and $[\beta]\phi$, because both paths are possible; so the system $\alpha \cup \beta$ will only be safe (satisfy ϕ) if all its α executions are safe ($[\alpha]\phi$) and all its β executions are safe ($[\beta]\phi$). For loops, KeYmaera uses (inductive) invariants. An inductive invariant for proving $\phi \rightarrow [\alpha^*]\psi$ is a formula J that is satisfied in the current state ($\phi \rightarrow J$) and, starting from any state satisfying the invariant J, executing the loop body leads into a state also satisfying the invariant J ($J \rightarrow [\alpha]J$). Hence induction shows that starting from the state just reached the program will end up in states satisfying the invariant ($A \rightarrow [\alpha^*]J$). In order to use this pattern for reasoning about formulas that are not inductive invariants themselves we add a third task: we have to show that the property we want to show is a consequence of the invariant ($J \rightarrow \psi$).

For differential equations (ODEs) there are two possible routes. If the ODE has a polynomial solution, we can replace it by a discrete assignment at each point in time t. In this case we have a polynomial for each variable that symbolically describes the value of this variable over time. Thus, we can assign this polynomial with a symbolic parameter t to the variable as long as we quantify over t, since the ODE could have evolved for any amount of time. However, if there is no polynomial solution available this would yield formulas in an undecidable theory. Instead, in those cases, we apply differential induction [46, 47, 51, 52], which is induction for differential equations showing that the derivative of the evolution domain candidate points inwards w.r.t. the region it characterizes. When proving inductive properties of loops, the loop body can be seen as direction in which the system will evolve. That is, the loop body describes one atomic evolution step. Similarly, the direction for the variables are given in the differential equation system. A set of points is invariant, if following along the given direction we stay within this set. The basic idea can be seen on the following example. Assume two func-

Table 2: Operators and (informal) meaning in differential dynamic logic ($\mathsf{d}\mathcal{L}$)

$\mathsf{d}\mathcal{L}$	Operator	Meaning
$\theta_1 \sim \theta_2$	comparison	true iff $\theta_1 \sim \theta_2$ with $\sim \in \{>, \geq, =, \neq, \leq, <\}$
$\neg\phi$	negation / not	true if ϕ is false
$\phi \wedge \psi$	conjunction / and	true if both ϕ and ψ are true
$\phi \vee \psi$	disjunction / or	true if ϕ is true or if ψ is true
$\phi \rightarrow \psi$	implication / implies	true if ϕ is false or ψ is true
$\phi \leftrightarrow \psi$	bi-implication / equivalent	true if ϕ and ψ are both true or both false
$\forall x\, \phi$	universal quantifier	true if ϕ is true for all values of variable x in $\mathbb{R}$
$\exists x\, \phi$	existential quantifier	true if ϕ is true for some values of variable x
$[\alpha]\phi$	$[\cdot]$ modality / box	true if ϕ is true after all runs of hybrid program α
$\langle\alpha\rangle\phi$	$\langle\cdot\rangle$ modality / diamond	true if ϕ is true after at least one run of hybrid program α

tions $f : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}$ and $g : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}$. If we can show that $f(0) > g(0)$ and $f'(t) \geq g'(t)$ for all $t \in \mathbb{R}_{\geq 0}$ then we can conclude that $f(t) > g(t)$ for all $t \in \mathbb{R}_{\geq 0}$. For loops we have to follow the direction for exactly one execution of the loop body. In the case of differential induction, we check for an infinitesimal step instead. That is we show that for a given candidate set, for each point that satisfies the original evolution domain constraint the gradient of the system points inwards w.r.t. this set. Unlike induction for loops, we might have to repeat this step several times, thereby strengthening the evolution domain within each step by a differential cut [46, 47, 51, 52]. Ultimately, the goal is to reach an evolution domain constraint which is a subset of the postcondition we are trying to prove.

Once we have dealt with all the modalities in the formulas, we end up with a first-order formula over the reals. Validity of those can be decided by quantifier elimination [62]. The original method proposed by Tarski however has non-elementary complexity. Davenport and Heintz have shown that the worst-case complexity of such a procedure will always be doubly exponential [16]. Still, we can use quantifier elimination in many practical examples. KeYmaera interfaces with a number of tools (e. g., Mathematica [65], Redlog [19], Z3 [41], and QEPCAD B [11]) and implements algorithms to deal with special cases more efficiently. Note that the user only interacts with KeYmaera and these tools are invoked transparently in the background.

4 Related Tools

There has been significant research on hybrid system verification and related approaches include a number of hybrid system verification tools.

The ultimate goal of these approaches is to provide fully automated verification tools for hybrid systems. Unfortunately, this is provably impossible, because the problem is not even semi-decidable [4, 45]. Therefore, different compromises have been made. Fully automated tools compromise by restricting the classes of hybrid systems they can handle and additionally they still do not guarantee termination. Semi-automated tools can automatically explore the state space, but must fall back to the user where the automated search fails. For the latter, the user can then use domain knowledge to steer the tool into a promising direction for the verification.

For the user, the decision which tool to use depends on the system characteristics that appear in the models.

Tools for Real-time Systems. Real-time systems [42] is the class that provides the most automatic tools. Therefore, models that comply to the following restrictions are best tackled with tools like Uppaal [33]. The main advantage is that reachability is decidable for real-time systems, whereas it is undecidable for hybrid systems. This is achieved by restricting models such that all continuous variables represent clocks instead of physical motion in space, i. e., the derivatives of all variables are constantly 1. Furthermore, computations are limited to resetting variables to 0 instead of arbitrary values. Uppaal uses these properties to its advantage and can check a rich set of properties by smart, exhaustive, set-valued simulation of the system.

Uppaal has been extended to a verification tool for priced timed automata [5, 10]. Priced timed automata extend real-time systems with variables that can have constant but arbitrary and changing slopes. However, they cannot be used in any way that influences the reachability relation. That is, they can neither restrict discrete computations nor continuous evolutions. However, cost optimal reachability is decidable and implemented by Uppaal CORA [9].

Tools for Linear or Affine Hybrid Systems. If the system model falls into the class of linear or affine hybrid systems, model checking [14] tools are applicable. These systems allow a much richer class of dynamics for both continuous evolutions (constant differential inclusions or linear ordinary differential equations) as well as discrete transitions (linear assignments). That is, for affine hybrid systems, all assignments have the form $x := \theta$ and

differential equations have the form $x' = \theta \,\&\, F$ where θ is restricted to a multi-variate polynomial of at most degree 1. For linear hybrid systems, the evolutions have to have the form $(A \leq x' \leq B) \,\&\, F$, where A and B are constant numbers. These inclusions are often used as overapproximations of the actual system dynamics. In both cases, F can only be a conjunction of linear constraints. Among the first tools following this line are HyTech [27], d/dt [7], and PHAVer [22]. PHAVer, which superseded HyTech, was recently superseded itself by SpaceEx [23]. SpaceEx is a fully automated tool for verification of hybrid systems with linear, affine dynamics. FOMC [15] provides methods tailored to similar systems with large discrete state spaces. Much like Uppaal, all these tools perform an exhaustive search of the state space fully automatically. If they succeed, no user interaction is necessary. However, unlike Uppaal, termination is no longer guaranteed, since reachability for linear hybrid systems is an undecidable question. Model checking is a versatile approach, which is good in finding counterexamples. Note, however, that those counterexamples need manual inspection, because the employed overapproximations may result in spurious counterexamples. Thus, it can also be used as a complementary approach to techniques showing the absence of errors, such as KeYmaera.

Tools for Nonlinear Hybrid Systems. Nonlinear hybrid systems feature even richer dynamics, i. e., the restrictions defined above for affine hybrid systems no longer apply. If specific bounds or ranges for the variables are known, numerical methods provide means to tackle the verification task. Flow* [13] can be used for bounded simulation-based model checking of nonlinear hybrid systems up to a given time horizon and bound on the number of jumps. The iSAT algorithm [21, 35] couples interval constraint propagation with Cylindrical Algebraic Decomposition and methods from SAT solving, thereby providing a solver for boolean combinations of nonlinear constraints (including transcendental functions) over discrete and continuous variables with bounded ranges. iSAT-ODE [20] extends iSAT with validated enclosure methods for the solution sets of nonlinear ordinary differential equations (ODEs), allowing Bounded Model Checking of hybrid systems on overapproximations or exact solutions of the ODEs. For low-dimensional systems, HSolver [61] offers methods for unbounded horizon reachability analysis of hybrid systems. It implements abstraction refinement based on interval constraint propagation.

Systems with rich dynamics or symbolic parameters can be verified in KeYmaera. KeYmaera is a semi-automated tool for unbounded horizon, purely symbolic and sound verification of hybrid systems. It performs automated proof search and allows the user to interact and steer the prover in cases where the automated proof search procedures fail. A strong point of KeYmaera is the automated decomposition of the original verification problem into smaller subtasks while retaining a clear connection to the original problem. This allows the user to focus on the difficult cases, where interaction is necessary and let the prover take care of those cases where the necessary steps can be performed automatically. Still, in contrast to fully automated tools, some knowledge about the core ideas behind KeYmaera is necessary to apply it successfully to complex systems, even though some systems can be verified fully automatically, such as the European Train Control System [58] and aircraft roundabout maneuvers [56]. The following sections are meant to provide an easy-to-follow introduction into these ideas based on a running example from the automotive domain.

5 KeYmaera Tutorial

Starting from a simple example, we develop a series of increasingly more complex systems which illustrate how modeling and verification challenges can be handled in KeYmaera. We highlight incremental changes between models using **boldface** symbols. The example files in this paper can be found in the project *KeYmaera Tutorial* of KeYmaera, which can be downloaded from

`http://symbolaris.com/info/KeYmaera.html` .

A series of video tutorials which complement the examples presented in this paper can be found at

`http://video.symbolaris.com` .

5.1 Example 1: Uncontrolled Continuous Car Model

First we will look at a simple system in which a car starts at some nonnegative velocity and accelerates at a constant rate along a straight lane. The requirement we want to prove is that the car never travels backward in space. Example 1 captures the setup of this scenario: when starting at the initial conditions init, all executions of the car [*plant*] must ensure the requirements req. The scenario setup is expressed using the $\mathtt{d}\mathcal{L}$ formula init $\rightarrow$ [*plant*](req): the initial conditions are on the left-hand side of a logical implication ($\rightarrow$); the (hybrid) program and the requirement form its right-hand side. We used the box modality [*plant*] to express that *all* states reachable by the continuous model of the system *plant* satisfy our requirements req.

The initial conditions are formally specified in formula (6): the velocity *and* the acceleration must both be positive initially ($v \geq 0 \wedge A > 0$). In this example, the *plant* is very simple, cf. formula (7): the derivative of position is velocity ($p' = v$) and the derivative of velocity is maximum acceleration ($v' = A$). Fig. 2 shows a sample trace of this system, where acceleration $A = 1$ is constant while velocity v and position p change according to the differential equations. Finally, formula (8) states that the velocity of the car is positive $v \geq 0$ and,

Example 1 Safety property of an uncontrolled continuous car model

$$\text{init} \rightarrow [\mathit{plant}]\ (\text{req}) \tag{5}$$
$$\text{init} \equiv v \geq 0 \land A > 0 \tag{6}$$
$$\mathit{plant} \equiv p' = v, v' = A \tag{7}$$
$$\text{req} \equiv v \geq 0 \tag{8}$$

alternative

$$\text{init} \equiv v \geq 0 \land A > 0 \land p_0 = p \tag{9}$$
$$\text{req} \equiv p \geq p_0 \tag{10}$$

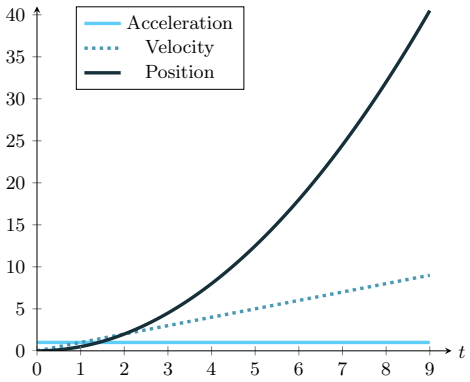


Fig. 2: Simulated trace of Example 1

thus, captures our requirement that the car never travels backward in space. Note, that many different ways exist to model even such a simple system and express correctness properties about them: in formulas (9)–(10) we use an additional variable to remember the initial position of the car and require that the car will never be before p_0 (10). Being forced to find and formalize suitable safety properties is one of the major benefits of using formal verification. For example, if our car interacts with other autonomous or malicious vehicles in the environment, safety suddenly becomes a nontrivial question of who is to blame if an accident occurs [36, 39].

KeYmaera proves Example 1 automatically. In Example 1 we modeled only continuous components in the *plant*. In the next example we will allow a discrete controller to interact with the system.

5.2 Example 2: Safety Property of Hybrid Systems

Example 1 had a plant but no controller. This means that, once started, the car would drive for an arbitrary amount of time without any option to ever change its initial decision of full acceleration. In Example 2, we introduce a discrete controller, *ctrl*, into the model of the system. The task of the controller in this example is to adjust the velocity by accelerating or braking, and still never drive backward.

The example follows closed-loop feedback control, which is a typical control system principle, as depicted in Fig. 3: a controller tries to minimize the error e (difference between a desired output response r and sensed

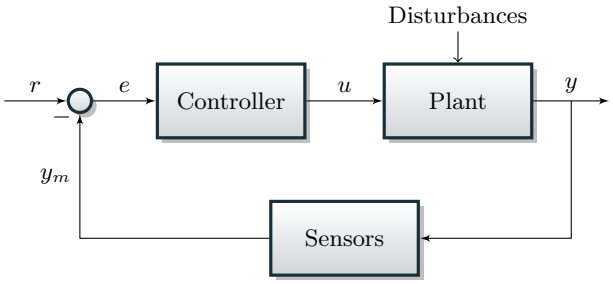


Fig. 3: Closed-loop feedback control system principle

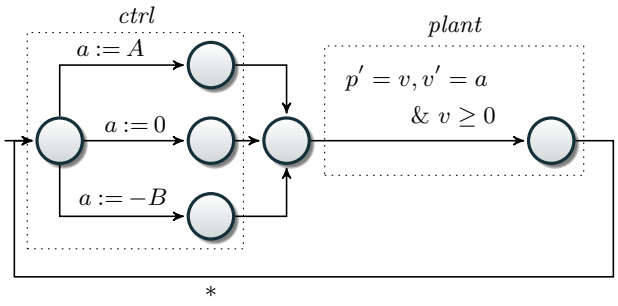
output measurements y_m) by computing set values u as input for a plant. The plant, possibly influenced by some disturbances, produces an output response y, which is fed back into the controller through sensors as measurements.

Example 2 shows the model, which was extended from Example 1. The essential difference is the hybrid program in formula (11), whose controller *ctrl* is repeated together with the *plant* nondeterministically in a loop, as indicated by the repetition star at the end of the hybrid program in (11). The state transition system of this hybrid program is depicted in Fig. 4a. The initial conditions in formula (12) now contain a specification for braking force $B > 0$. The controller has three simple options as stated in formula (13): it may cause the car to accelerate with rate $A > 0$, maintain velocity by choosing acceleration 0, or brake with rate $-B < 0$. We model the control options as a nondeterministic choice ($\cup$) in order to verify multiple concrete controllers at once. Because the hybrid program is within a $[\cdot]$ modality in (11), whether the controller chooses to accelerate, maintain velocity, or brake, the postcondition req must always hold for (11) to be true.

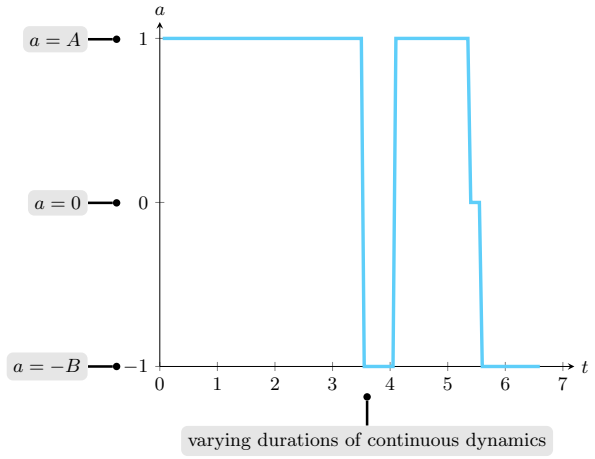
When a real car brakes, it decelerates to a complete stop—it is not possible to drive a car backward by braking. In order to model this, in formula (14) we extended the plant from the previous example and prevent the continuous dynamics from evolving beyond what is possible in the real world. So, even though evolving over time with $p' = v,\ v' = -B$ would eventually cause the car to drive backward, we disallow these traces by adding an evolution domain constraint of $v \geq 0$ in the plant (separated by &), which restricts the model of the car to realistic movement.

Example 2 Safety property of a hybrid car model

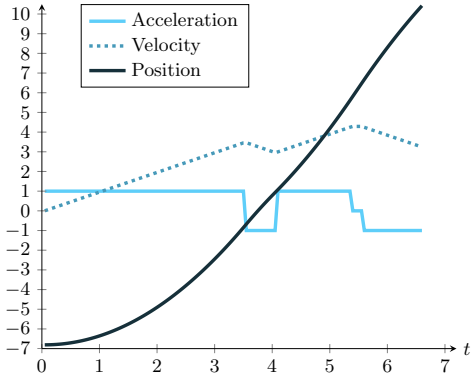
$$\text{init} \rightarrow [(\boldsymbol{\mathit{ctrl}; \mathit{plant}})^*]\ (\text{req}) \tag{11}$$
$$\text{init} \equiv v \geq 0 \land A > 0 \land \boldsymbol{B > 0} \tag{12}$$
$$\boldsymbol{\mathit{ctrl} \equiv a := A \cup a := 0 \cup a := -B} \tag{13}$$
$$\mathit{plant} \equiv p' = v, v' = a\ \boldsymbol{\&}\ \boldsymbol{v \geq 0} \tag{14}$$
$$\text{req} \equiv v \geq 0 \tag{15}$$



(a) State transition system



(b) Control decisions: accelerate, coast, and brake; nondeterministic duration of continuous dynamics



(c) Sample trace: velocity and position per acceleration choice

Fig. 4: A hybrid car controller (Example 2)

We also want the discrete controller to be able to change the acceleration of the vehicle at any time. Like in a regular expression, the nondeterministic repetition * creates a loop over the *ctrl* and *plant*. The *plant* evolves for an arbitrary amount of time (it may even evolve for zero time) as long as it satisfies the evolution domain. When the *plant* completes, the program can loop back to the *ctrl* which is again allowed to choose between accelerating, maintaining velocity, or braking. All the states that are reachable by this program must satisfy the postcondition req of formula (15), which is the same as in the previous example.

Fig. 4b shows a sequence of control choices that govern the plant for varying plant execution duration. The resulting sample trace of the continuous change of the car's velocity v and position p, which follows from these control decisions, is shown in Fig. 4c.

In order to prove properties of a loop, we need to identify an invariant, which is a formula that is true whenever the loop repeats. That is, a formula inv that is initially valid (init $\rightarrow$ inv), that implies the postcondition (use case inv $\rightarrow$ req), and where the loop body preserves the invariant (here inv $\rightarrow$ $[ctrl; plant]$inv). Invariants are critical parts of the system design. As such, they should always be communicated as part of the system, for which KeYmaera provides annotations:

$$\text{init} \rightarrow [(ctrl; plant)^*@\text{invariant}(v \geq 0)](\text{req})$$

The @invariant(inv) annotation for a loop indicates that inv is a loop invariant. KeYmaera uses this annotation to find proofs more efficiently. If absent, KeYmaera tries to compute an invariant automatically [55] as it does in Example 2. Here, the invariant $v \geq 0$ is trivial and will follow from Guideline 1. It is good style to annotate invariants when known, because they communicate and document important information about the behavior of a system.

For a step-by-step tutorial on using loop invariants in KeYmaera, watch the *Loop Invariant* video [1].

5.3 Example 3: Event-Triggered Hybrid Systems

Now we will add some complexity to the system and the controller. We want to model a stop sign assistant: while the car is driving down the lane, the controller must choose when to begin decelerating so that it stops at or before a stop sign. This means that it is no longer sufficient to let the controller run at arbitrary points in time as in Example 2, since the controller now must

brake when it approaches a stop sign. Thus, we have to change our model to prevent the plant from running an infinite amount of time. We can do this by adding an additional constraint to the evolution domain of the plant. Depending on the nature of this additional constraint we either speak of an *event-triggered system* or a *time-triggered* system. The former interrupts the plant when a particular event in the environment occurs (e. g., when the car is too close to a stop sign), while the latter interrupts the plant at periodic times (e. g., every 50 ms).

We will start with an event-triggered system, since those are often easier to prove than time-triggered systems. A time-triggered model will be discussed in Example 5.

Example 3a Stop sign controller (event-triggered)

$$\text{init} \rightarrow [(\mathit{ctrl}; \mathit{plant})^*](\text{req}) \tag{16}$$

$$\text{init} \equiv v \geq 0 \wedge A > 0 \wedge B > 0 \wedge \mathbf{safe} \tag{17}$$

$$\mathbf{safe} \equiv \boldsymbol{p + \frac{v^2}{2B} < S} \tag{18}$$

$$\mathit{ctrl} \equiv (?\mathbf{safe}; a := A) \cup (\boldsymbol{?v = 0}; a := 0) \cup (a := -B) \tag{19}$$

$$\mathit{plant} \equiv \left(p' = v, v' = a \;\&\; v \geq 0 \wedge \boldsymbol{p + \frac{v^2}{2B} \leq S}\right) \tag{20}$$

$$\cup \left(\boldsymbol{p' = v, v' = a \;\&\; v \geq 0 \wedge p + \frac{v^2}{2B} \geq S}\right) \tag{21}$$

$$\text{req} \equiv p \leq S \tag{22}$$

The stop sign assistant is modeled in Example 3a and depicted in Fig. 5a. The basic setup of the model in formula (16) is the same as in Example 2. However, we have to adapt the initial condition in (17) such that the car starts at a position that is still sufficiently distant from the stop sign. Intuitively, the car is at a safe position if it can still stop before it reaches the position S of the stop sign. Using kinematic equations, we derive that the stopping distance of the car when decelerating at rate $-B$ is $\frac{v^2}{2B}$ (see Guideline 1 for step-by-step instructions on how to derive such constraints); thus, the proposition safe is true when the current position plus the stopping distance of the car is less than the position S of the stop sign, as specified in formula (18).

Guideline 1 (Evolution domains and invariants) *In order to derive evolution domain constraints for event-triggered control, and to find an inductive invariant for the system, we analyze the model of Example 3a. We start at the safety condition and the kinematic equations of the car, as summarized in the formula below.*

$$[p' = v, v' = a \;\&\; v \geq 0]\,(p \leq S) \tag{23}$$

If the evolution domain $v \geq 0$ is sufficiently strong already (meaning $v \geq 0 \rightarrow p \leq S$), we are done. Since $v \geq 0 \rightarrow p \leq S$ is not valid here, we analyze further. Formula (23) *does not mention specific choices of acceleration. However, the controller ctrl includes one unconditional choice (braking by $a := -B$), so we have to interrupt the continuous dynamics at the latest when the car can still stop in the remaining distance to the stop sign with braking power $-B$. We, therefore, replace acceleration a with the maximum braking power $-B$.*

$$[p' = v, v' = -B \;\&\; v \geq 0]\,(p \leq S)$$

Now we analyze the differential equations step by step, starting with $p' = v$. This means, the position of the car will no longer change when its velocity becomes 0. Hence, we need to find out where the car will stop, or in other words, how far the car will be driving until it is stopped. Such questions can be answered from differential equations through integration*, so we get one indefinite integral from $p' = v$, and a nested indefinite integral because $v' = -B$.*

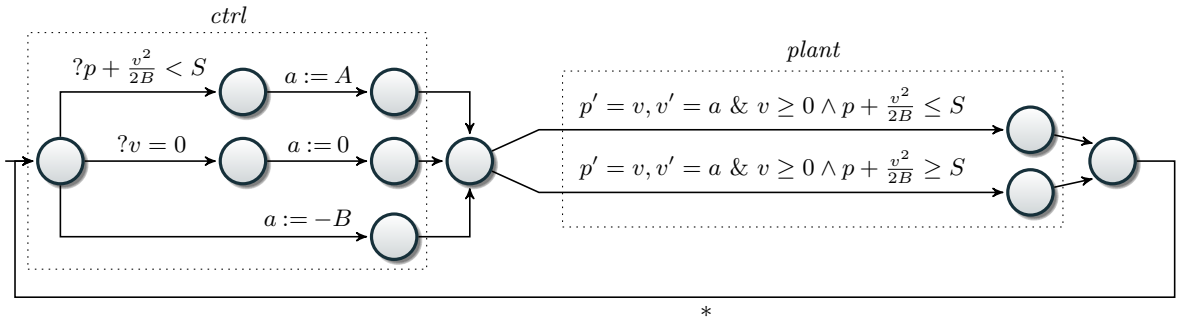
$$\int \left(v + \int (-B)\,dt\right) dt = vt - \frac{Bt^2}{2}$$

This way, we can compute the distance for specific choices of t. Most interesting to us is a choice of t when the car is stopped. We determine how long it will take the car to stop, i. e., we compute the time until $v = 0$. Since we are now analyzing velocity, we only need to integrate once: $v + \int (-B)\,dt = 0$, which gives us $v - Bt = 0$. We solve for t to get $t = \frac{v}{B}$. Now that we know the stopping time, we can determine the stopping distance easily by computing the definite integral:

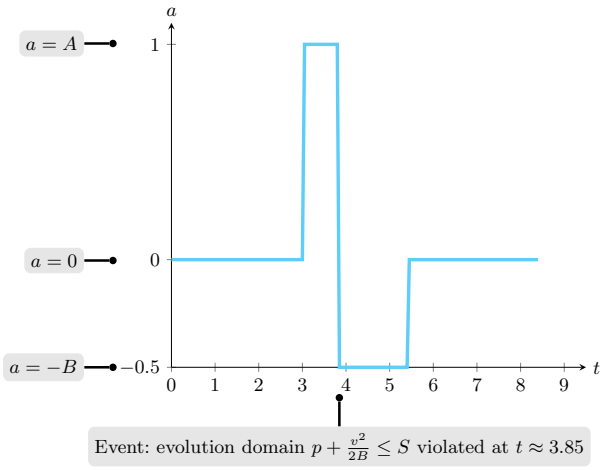
$$\int_0^{\frac{v}{B}} \left(v + \int (-B)\,dt\right) dt = vt - \frac{Bt^2}{2}\Bigg|_0^{\frac{v}{B}} = \frac{v^2}{2B}$$

In summary, the evolution domain is found by linking the stopping distance with the safety constraint as follows $p + \frac{v^2}{2B} \leq S$.

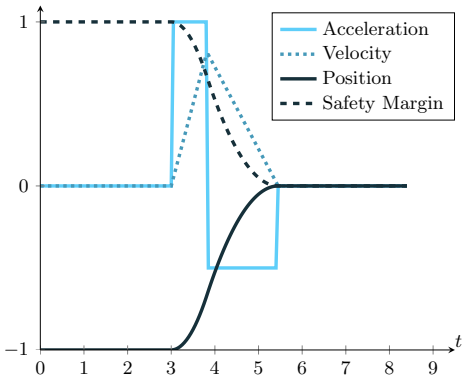
The controller in formula (19) still chooses between accelerating, maintaining velocity, and braking, but the first two options are not allowed if the car is too close to the stop sign. We restrict the choice of accelerating by adding the test ?safe, so that the car may only accelerate if it is still sufficiently distant from the stop sign. Here we see why the inequality in safe (18) must be strict; if we are stopped at the stop sign, it would not be safe to accelerate. For now, we only allow the car to maintain velocity ($a := 0$) when it is already stopped, since otherwise the car could coast through the intersection. Later, we are going to relax this unrealistic restriction. The proposition derived in Guideline 1 is added as an event-trigger to the evolution domain, cf. (20). This ensures that the controller executes if the car comes within the minimum stopping distance of the stop sign; however, the controller is free to execute at any time before this point is reached to adapt acceleration as needed, because the duration of an ODE is nondeterministic. Notice that



(a) State transition system



(b) Control decisions: first coast, then accelerate until evolution domain constraint is violated, brake, and finally stay stopped



(c) Sample trace: velocity and position per acceleration choice; distance to stop sign $|p - S|$ gives the remaining safety margin

Fig. 5: Event-triggered stop sign controller (Example 3a)

in (21) we have to include a nondeterministic choice such that the physics of the system may still evolve even when the event trigger is no longer satisfied. A full discussion of why this is important is deferred to Section 5.4. Finally, the postcondition req in formula (22) defines that in all states that are reachable by the event-triggered hybrid program, the position of the car must not exceed the position of the stop sign $p \leq S$.

An example for event-triggered control and its effects is shown in Fig. 5b and 5c: the car accelerates until the evolution domain constraint triggers braking, which causes the car to stop smoothly at the stop sign.

Using KeYmaera to Discover Constraints In this model, it was easy to use Guideline 1 to derive the stopping distance of the car, the evolution domain, and the conditions for *ctrl*; however, for more complex models, the solution may not be so apparent. We may get hints about what *ctrl* and the evolution domain should be by first trying to prove safety in a system that is obviously unsafe or that we merely suspect to be safe in some scenarios.

To illustrate this method, in Example 3b we start with a simpler version of Example 3a.

We remove the nondeterministic repetition (24), so that we do not yet have to worry about loop invariants. For lack of a better understanding of safe, in (26) we define it to be true unconditionally, which means in init (25) we just strive to not violate our requirement $p \leq S$ and place the car somewhere in front of the stop sign. It also means that *ctrl* allows the car to choose acceleration without any restrictions, which cannot always be correct, as illustrated in Fig. 6. The plant (28) and the requirement (29) remain the same as in Example 3a, but with the event-trigger removed from the evolution domain. Attempting to prove property (24) of Example 3b results in two open goals in which there are formulas which, had they been included in safe, the property req would have held (apply local quantifier elimination

Example 3b Unsafe stop sign controller design to discover safety constraints

$$\text{init} \rightarrow [\mathit{ctrl}; \mathit{plant}](\text{req}) \tag{24}$$
$$\text{init} \equiv v \geq 0 \land A > 0 \land B > 0 \land \text{safe} \land \boldsymbol{p \leq S} \tag{25}$$
$$\text{safe} \equiv \textbf{true} \tag{26}$$
$$\mathit{ctrl} \equiv (?\text{safe}; a := A) \cup (?v = 0; a := 0) \cup (a := -B) \tag{27}$$
$$\mathit{plant} \equiv p' = v, v' = a \;\&\; v \geq 0 \tag{28}$$
$$\text{req} \equiv p \leq S \tag{29}$$

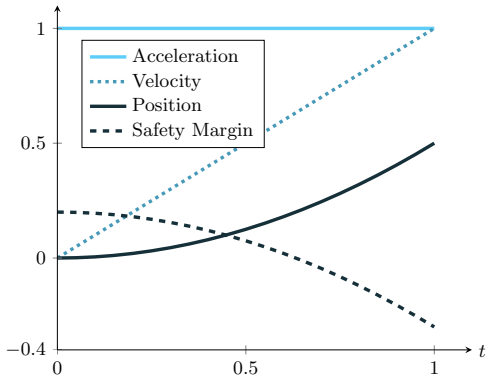


Fig. 6: Sample trace of Example 3b; safety margin becomes negative immediately due to unsafe controller

to the consequent to remove quantifiers from these formulas). Some of these formulas contradict our assumptions in init (25), so we ignore them. However, there is one remaining formula which does not contradict any assumption: $B \geq v^2(2S+-2p)^{-1}$. The failed proof attempt indicates that we should change our design to obey this constraint. With some algebraic manipulation, we see that this constraint is almost identical to the restriction we added to the *ctrl* in Example 3a.

To see an example of how KeYmaera can discover a braking constraint, watch the video *Discover Constraints Using KeYmaera* [1].

It is not uncommon for the first attempt at proving the safety of a system to be unsuccessful because the model is in fact unsafe. KeYmaera allows the user to examine a trace of the hybrid program which obeys the initial conditions, and follows the execution of the hybrid program, but violates the given safety requirement. In Example 3b, there are infinitely many such counterexamples that could be generated; however, one counterexample (which KeYmaera automatically generates) sets the position of the stop sign to be $S = 0$, the initial position and velocity of the car to be $p = -23$ and $v = 986$, and maximum acceleration $A = 38$. These assignments of values to the symbolic parameters are all permissible by the initial conditions. The transition then has the car accelerate at rate A and allows the system to evolve for 0.1 time steps, at which point the position of the car is $p = 75.79$, so the car has run past the stop sign and the requirement $p \leq S$ has been violated, showing that the system is unsafe. This behavior is similar (albeit with more extreme values) to the behavior depicted in Fig. 6.

For a video tutorial on how to generate counterexamples in KeYmaera, watch the *Find Counterexample* video [1].

5.4 Example 4: Pitfalls when Modeling Event-Triggered Hybrid Systems

When modeling event-triggered systems, we have to make sure that our event model does not restrict the physical behavior in order to react to certain events [54]. Consider a cruise control with the goal of reaching and maintaining a certain velocity, say v_s. A simple event-triggered model for this is shown in Example 4. Certainly, we can prove that this controller ensures that the velocity never exceeds the set-value v_s as every time the car reaches velocity v_s it will set the acceleration to 0.

Example 4 Event-triggered cruise control

$$\text{init} \rightarrow [(\mathit{ctrl}; \mathit{plant})^*](\text{req}) \tag{30}$$
$$\text{init} \equiv v \leq v_s \wedge A > 0 \tag{31}$$
$$\mathit{ctrl} \equiv \text{if } v = v_s \text{ then } a := 0 \text{ else } a := A \text{ fi} \tag{32}$$
$$\mathit{plant} \equiv p' = v, v' = a \;\&\; v \leq v_s \tag{33}$$
$$\text{req} \equiv v \leq v_s \tag{34}$$

$$\text{init} \rightarrow [(\mathit{ctrl}_f; \mathit{plant})^*](\text{req}) \tag{35}$$
$$\mathit{ctrl}_f \equiv a := A \tag{36}$$

$$\text{init} \rightarrow [(\mathit{ctrl}; \mathit{plant}_r)^*](\text{req}) \tag{37}$$
$$\mathit{plant}_r \equiv (p' = v, v' = a \;\&\; v \leq v_s) \cup (p' = v, v' = a \;\&\; v \geq v_s) \tag{38}$$

$$\text{init} \rightarrow [(\mathit{ctrl}_f; \mathit{plant}_r)^*](\text{req}) \tag{39}$$

Unfortunately, this is not the reason we can prove this property. Replacing the controller by one that always chooses to accelerate reveals that the validity of the formula does not depend on our control choices, i.e., formula (35) is valid as well. This stems from the fact that any continuous evolution along (33) is already restricted to the domain we consider critical. Thus, there is no transition leaving this domain once we reach its border which makes the property trivially true. However, safety in real-world systems crucially relies on correct functioning of our controllers. Thus we have to adapt the model to reflect the fact that the car could in some scenarios exceed the velocity we want to maintain and then show that our controller makes sure that it does not do so.

Consider the plant model given in (38). Here we modify the plant in such a way that time may evolve regardless of the relation of v and v_s. The controller will still be invoked and able to update the acceleration before (or at the very latest when) the velocity reaches v_s. However, now since there are transitions that might invalidate our requirement to never exceed the velocity v_s we can observe a difference between our original controller (32) and the faulty one (36). That is, the formula (37) with the original controller (32) is valid, whereas the formula (39), which depends on the faulty controller (36), is not. More generally, it is important

not to restrict physics when adding events, but only to split it into regions. These regions, formed by evolution domain constraints, must overlap: for example, $v \leq v_s$ and $v \geq v_s$ in (38) share a boundary. Otherwise, e. g., with $v < v_s$ and $v \geq v_s$, the program could not switch between the differential equation systems in (38), see Section 6.3 for details.

5.5 Example 5: Time-Triggered Hybrid Systems

Event-triggered systems like the one in Example 3a make proving easier, but they are difficult (if not impossible) to implement. In order to implement Example 3a faithfully, it would require a sensor which would sense position and velocity data continuously, so that it could notify the controller instantaneously when the car crosses the final braking point. A more realistic system is one in which the sensors take periodic measurements of position and velocity and the controller executes each time those sensor updates are taken. However, if we do not restrict the amount of time between updates, then there is no way to control the car safely, since it would essentially be driving blind. Instead, we require that the longest time between sensor updates is bounded by a symbolic $\varepsilon > 0$. To account for imperfect timing, the controller can also handle updates that come in before the ε deadline. In this section, we implement this system and prove it is safe.

Fig. 7b shows control decisions that follow this principle. At the latest every ε time units the controller senses the velocity and position of the car and makes a new decision to accelerate, stay stopped, or brake. A sample trace of the continuous dynamics resulting from these control decisions is sketched in Fig. 7c.

With this change, we must create a more intelligent controller. The controller must respect its own reaction delays to make sure to take action if it might be unsafe to defer braking until the next control cycle. There are two essential differences between Example 3a and Example 5 with its transition system depicted in Fig. 7a: Example 5 introduces a clock into the plant (44) that stops continuous dynamics before $c \leq \varepsilon$ becomes false, and the acceleration branch can only be taken if it is safe to accelerate for up to ε time.

Condition (43) uses the upper bound ε in the safety condition that allows the car to accelerate. In Example 3a we used the formula safe from (18) to determine whether it was safe for the car to accelerate at the present moment. Now, we must have a controller which not only checks that it is safe to accelerate at present, but also that doing so for up to ε time will still be safe. We use the formula safe_ε (43) in Example 5, which checks that while accelerating for ε time, the car will always be able to come to a complete stop before the stop sign. Guideline 2 describes how we can derive such safe_ε analytically from the model.

Example 5 Stop sign controller (time-triggered)

$$\text{init} \rightarrow [(\mathit{ctrl}; \mathit{plant})^*](\text{req}) \tag{40}$$

$$\text{init} \equiv v \geq 0 \land A > 0 \land B > 0 \land \boldsymbol{p + \frac{v^2}{2B} \leq S \land \varepsilon > 0} \tag{41}$$

$$\mathit{ctrl} \equiv (?\text{safe}_\varepsilon; a := A) \cup (?v = 0; a := 0) \cup (a := -B) \tag{42}$$

$$\text{safe}_\varepsilon \equiv \boldsymbol{p + \frac{v^2}{2B} + \left(\frac{A}{B} + 1\right)\left(\frac{A}{2}\varepsilon^2 + \varepsilon v\right) \leq S} \tag{43}$$

$$\mathit{plant} \equiv \boldsymbol{c := 0};\ p' = v, v' = a, \boldsymbol{c' = 1}\ \&\ v \geq 0\ \land\ \boldsymbol{c \leq \varepsilon} \tag{44}$$

$$\text{req} \equiv p \leq S \tag{45}$$

Guideline 2 (safe_ε for time-triggered control) *In the case of time-triggered control, a decision (e. g., accelerating with A) is safe when, after ε time, braking is still safe, which is captured in the following formula.*

$$\begin{aligned}&[c := 0]\\&[p' = v, v' = A, c' = 1\ \&\ v \geq 0 \land c \leq \varepsilon]\\&[p' = v, v' = -B\ \&\ v \geq 0]\,(p \leq S)\end{aligned}$$

Now, we have an explicit upper bound ε on time, so we can determine the distance traveled while accelerating with A using the following definite integral derived from the differential equations:

$$\int_0^\varepsilon \left(v + \int A dt\right) dt = vt + \frac{At^2}{2}\bigg|_0^\varepsilon = v\varepsilon + \tfrac{A\varepsilon^2}{2}\ .$$

We can also compute the new velocity after ε time using $v + \int_0^\varepsilon A dt$. This yields an equivalent formula

$$\begin{aligned}&\left[p := p + v\varepsilon + \tfrac{A\varepsilon^2}{2}; v := v + A\varepsilon\right]\\&[p' = v, v' = -B\ \&\ v \geq 0]\,(p \leq S)\end{aligned}$$

As a next step, we follow the approach from Guideline 1 to determine the distance for braking to a full stop from the increased velocity $v + A\varepsilon$:

$$\int_0^{(v+A\varepsilon)/B} (v + A\varepsilon - Bt)dt = \tfrac{v^2}{2B} + \tfrac{A}{B}\left(\tfrac{A}{2}\varepsilon^2 + \varepsilon v\right)\ ,$$

with braking time following from $v + A\varepsilon - Bt = 0$.

Since we already know the distance for braking to a full stop from Guideline 1 ($\frac{v^2}{2B}$), we could alternatively find the distance needed to compensate the increased velocity:

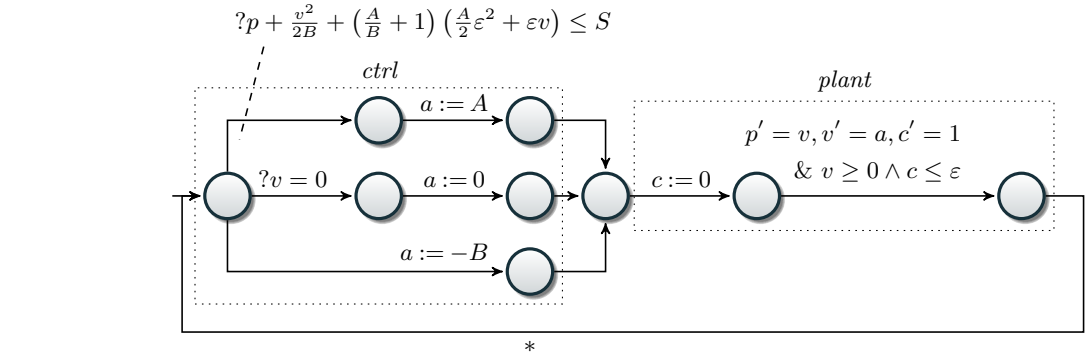
$$\int_0^{A\varepsilon/B} (v + A\varepsilon - Bt)dt = \tfrac{A}{B}\left(\tfrac{A}{2}\varepsilon^2 + \varepsilon v\right)$$

with braking time following from $A\varepsilon - Bt = 0$.

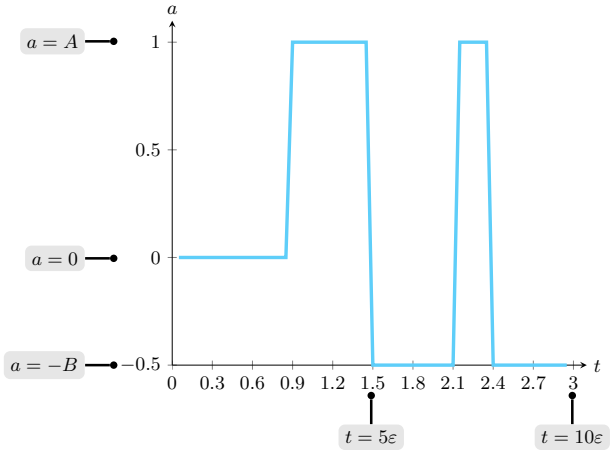
When we add the distance traveled while accelerating with the distance needed to stop afterwards, we get

$$p + \tfrac{v^2}{2B} + \left(\tfrac{A}{B} + 1\right)\left(\tfrac{A}{2}\varepsilon^2 + \varepsilon v\right) \leq S$$

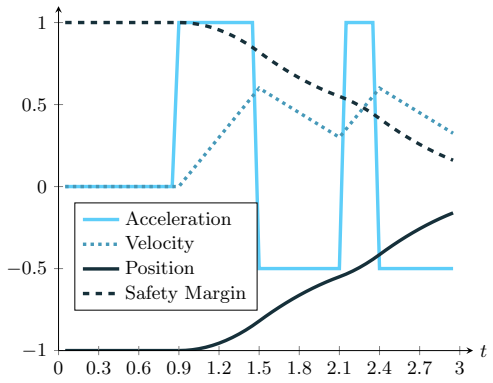
as definition for safe_ε, matching (43).



(a) State transition system



(b) Control decisions: despite nondeterministic loop duration $c_i \leq \varepsilon$, the system was simulated with $c = \varepsilon$



(c) Sample trace: velocity and position per acceleration choice; distance to stop sign $|p - S|$ gives the remaining safety margin

Fig. 7: Time-triggered stop sign controller (Example 5)

Because we have already proven a very similar system in Example 3a, it may be tempting to simply add a safety margin for how much the position of the car can change in time ε. However, the proof holds for the general, symbolic case (that is arbitrarily large values), so there is no constant error margin large enough that can be safe for all controllers.

5.6 *Example 6: Guarded Nondeterministic Assignment*

In previous examples, we have only represented controllers which can choose from a discrete choice of accelerations (either A, 0, or $-B$).

A more realistic controller would be able to choose any acceleration within a range of values representing the physical limits of the system. In Example 6 and Fig. 8a we introduce guarded nondeterministic assignment to represent an arbitrary choice of a real value within a given range.

In this example, we only need to change the *ctrl* to introduce nondeterministic assignment, while the rest of Example 6 is identical with Example 5: Line (48) of Example 6 assigns an arbitrary real value to a ($a := *$). The subsequent test checks that the value of a is in the interval $[-B, A]$ of physically possible accelerations. This operation eliminates all traces which do not satisfy the test, so only traces in which a is in $[-B, A]$ are considered. As a result, when we prove the property in Example 6, we are proving safety for all values of a within $[-B, A]$. The value assigned to a nondeterministically can be different on every loop execution. Fig. 8b shows an example sequence of the control choices made by such a controller.

Example 6 Stop sign controller (guarded nondeterministic assignment)

$$\text{init} \rightarrow [(\mathit{ctrl}; \mathit{plant})^*](\text{req}) \tag{46}$$

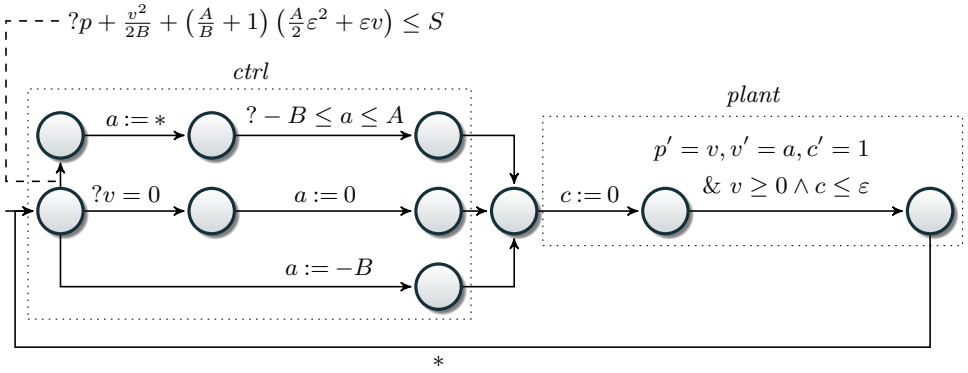
$$\text{init} \equiv v \geq 0 \wedge A > 0 \wedge B > 0 \wedge p + \frac{v^2}{2B} \leq S \wedge \varepsilon > 0 \tag{47}$$

$$\begin{aligned} \mathit{ctrl} \equiv (?\text{safe}_\varepsilon;\ \boldsymbol{a := *};\ \boldsymbol{? -B \leq a \leq A}) \\ \cup (?v = 0; a := 0) \cup (a := -B) \end{aligned} \tag{48}$$

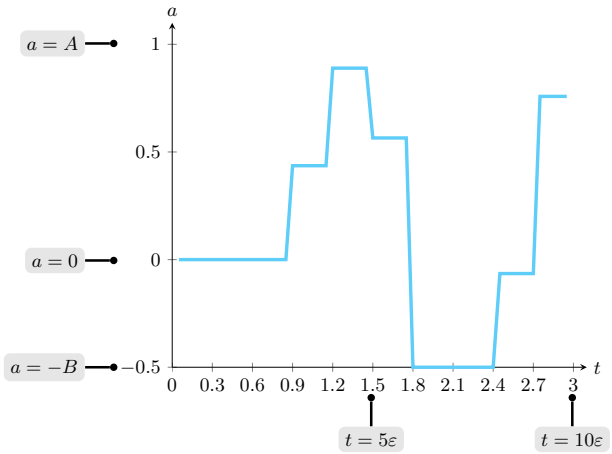
$$\text{safe}_\varepsilon \equiv p + \frac{v^2}{2B} + \left(\frac{A}{B}+1\right)\left(\frac{A}{2}\varepsilon^2 + \varepsilon v\right) \leq S \tag{49}$$

$$\mathit{plant} \equiv c := 0;\ p' = v, v' = a, c' = 1 \ \&\ v \geq 0 \wedge c \leq \varepsilon \tag{50}$$

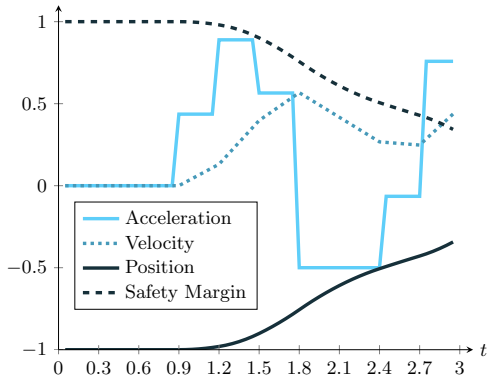
$$\text{req} \equiv p \leq S \tag{51}$$



(a) State transition system



(b) Control decisions: nondeterministically chosen acceleration constrained with guard $-B \leq a \leq A$



(c) Sample trace: velocity and position per acceleration choice; distance to stop sign $|p - S|$ gives the remaining safety margin

Fig. 8: Time-triggered controller with nondeterministic assignment (Example 6)

The resulting trace of the car's velocity and position is depicted in Fig. 8c.

When using guarded nondeterministic assignment, it is important to keep in mind some of the perils of using tests. Because tests eliminate traces completely, we no longer prove safety at any point in time for traces that fail the test. So, if we mistakenly use a guard that is impossible to satisfy, such as $?(a^2 < 0)$, our safety property will hold vacuously. Good practice is to use simple bounds that are obviously satisfiable, such as a range between two symbolic variables as shown in this example. Additionally, we can include a branch on the controller with no tests; in this example, the controller may always choose to brake.

5.7 Example 7: Nondeterministic Overapproximation

A good technique to prove properties that involve complicated formulas is to use nondeterministic overapproximation. If the value of a variable a_x is given by a complicated function $a_x = f(x)$, but the value of $f(x)$ is contained entirely in some interval $[f_1, f_2]$, the proof can often be greatly simplified by omitting the exact expression for $f(x)$ and simply allowing a_x to nondeterministically take any value in $[f_1, f_2]$ by using guarded nondeterministic assignment as discussed in Section 5.6.

For instance, in Example 6 the car's braking mechanism is modeled simply as choosing a negative acceleration, and is always fixed. In a more realistic braking model, like the one outlined in [28], braking would be modeled as

$$v' = \frac{1}{M}\left(-c_1 T_b - f_0 - c_2 v - c_3 v^2\right)$$

where T_b is the braking torque, $c_1 T_b$ is the braking force, M is the mass of the car, f_0 is the static friction force, $c_2 v$ is the rolling friction force, and $c_3 v^2$ is aerodynamic drag. This model has five more variables than the previous one. KeYmaera uses quantifier elimination as a decision procedure for first order real arithmetic, which is doubly exponential in the number of variables [16]. Thus, it helps to avoid unnecessary variables. In Example 7, we use a simpler model in which only one new variable is added.

In this model, the car's maximum total braking capability is between some symbolic parameters b and B, as modeled in (54). This means that we can guarantee

Example 7 Stop sign controller with nondeterministic braking

$$\text{init} \rightarrow [(\mathit{ctrl}; \mathit{plant})^*](\text{req}) \tag{52}$$

$$\text{init} \equiv v \geq 0 \wedge A > 0 \wedge \boldsymbol{B \geq b > 0} \wedge p + \frac{v^2}{\boldsymbol{2b}} \leq S \wedge \varepsilon > 0 \tag{53}$$

$$\begin{aligned} \mathit{ctrl} \equiv\ & (?\text{safe}_\varepsilon; a := *; ? -B \leq a \leq A) \\ & \cup (?v = 0; a := 0) \\ & \cup (\boldsymbol{a := *; ? -B \leq a \leq -b}) \end{aligned} \tag{54}$$

$$\text{safe}_\varepsilon \equiv p + \frac{v^2}{\boldsymbol{2b}} + \left(\frac{A}{\boldsymbol{b}} + 1\right)\left(\frac{A}{2}\varepsilon^2 + \varepsilon v\right) \leq S \tag{55}$$

$$\mathit{plant} \equiv c := 0; p' = v, v' = a, c' = 1 \ \&\ v \geq 0 \wedge c \leq \varepsilon \tag{56}$$

$$\text{req} \equiv p \leq S \tag{57}$$

at least b as the car's braking capability, but the brakes might be as strong as any value in the interval $[b, B]$. In comparison to Example 6, thus, we have to adapt init (53) and safe_ε (55) so that both consider the new minimum braking capability b instead of B.

Since the controls of real systems are usually deterministic and often complex, it can be useful to prove that the implemented controller is a deterministic refinement of the proved nondeterministic controller. This area is rich with possibilities for future research, but for preliminary methods on refinement, see [6].

5.8 Example 8: Differential Inequality Models of Disturbance

In this section we introduce differential inequality models as a technique to consider external disturbance [46, 47], such as the influence of road conditions on braking. If the value of a variable v changes nondeterministically according to acceleration a, as in the previous examples, and some disturbance d, we can use differential inequality models of disturbance. For example, the differential inequality $v' \leq ad$ models the effect of disturbance d on the acceleration a of our car, i. e., in the worst case the effective braking force may be reduced and the acceleration increased depending on a maximum disturbance factor d. Here, we use multiplicative disturbance; additive disturbance of the form $v' \leq a + d$ is possible in $\mathsf{d}\mathcal{L}$ as well. Observe that inequalities can also be used to bound the derivative from below, e. g. $ad \leq v'$, or both bounds at once, e. g., $a - d \leq v' \leq a + d$. However, for safety properties in many cases only one bound, upper or lower, is relevant and thus the other bound might be omitted from the models.

Example 8 introduces such a differential inequality model of disturbance on top of Example 7. The specific differential inequality $v' \leq ad$ used in this example models that the effective braking force and the effective acceleration force are subject to disturbance d; the disturbance is negligible when the acceleration or braking force is small, but it grows with increasing force. This model avoids disturbance when the car does not accelerate ($a = 0$), which means that disturbance alone will not cause the car to move when it is stopped.

Example 8 Stop sign controller with braking disturbance

$$\text{init} \rightarrow [(\mathit{ctrl}; \mathit{plant})^*](\text{req}) \tag{58}$$

$$\text{init} \equiv v \geq 0 \wedge A > 0 \wedge B \geq b > 0 \wedge p + \frac{v^2}{\boldsymbol{2bd}} \leq S \wedge \varepsilon > 0 \wedge \boldsymbol{d > 0} \tag{59}$$

$$\begin{aligned} \mathit{ctrl} \equiv\ & (?\text{safe}_\varepsilon; a := *; ? -B \leq a \leq A) \\ & \cup (?v = 0; a := 0) \\ & \cup (a := *; ? -B \leq a \leq -b) \end{aligned} \tag{60}$$

$$\text{safe}_\varepsilon \equiv p + \frac{v^2}{\boldsymbol{2bd}} + \left(\frac{A}{b} + 1\right)\left(\frac{A\boldsymbol{d}}{2}\varepsilon^2 + \varepsilon v\right) \leq S \tag{61}$$

$$\mathit{plant} \equiv c := 0;\ p' = v, \boldsymbol{v' \leq ad}, c' = 1 \ \&\ v \geq 0 \wedge c \leq \varepsilon \tag{62}$$

$$\text{req} \equiv p \leq S \tag{63}$$

Example 8 uses the same loop of sequential execution of controller and plant as Example 7, cf. (58). We adapt the initial condition in formula (59) to reflect that disturbance may reduce the braking force of the car to bd, but does not eliminate the braking force ($d > 0$). The controller itself remains the same as in Example 7, cf. (60). The main difference is in the controller's safety condition given in formula (61), which considers the fact that disturbance may reduce the effective braking force of the car ($\frac{v^2}{2bd}$) as well as its acceleration ($\frac{Ad}{2}$) until the brakes apply. Here, the disturbance affects braking and acceleration alike. For models with asymmetric disturbance (e. g., stronger acceleration and weaker brakes, see [36]), the braking term $\frac{v^2}{2b}$ and the acceleration term $\frac{A}{2}\varepsilon^2$ would be affected in different ways. Finally, the plant (62) replaces the differential equation of Example 7 with the differential inequality model.

5.9 Example 9: Lyapunov Functions and Invariants

We have seen in the previous sections that knowing system invariants ahead of time can greatly simplify the proof process. In this section, we discuss how Lyapunov functions, which are an important tool in control design, can be used to provide invariants that are useful for verification.

We will first define Lyapunov functions, and then use them to verify an invariant for a proportional-derivative (PD) controller for our car model. We will use a Lyapunov function to verify that a specific class of sets constitutes invariants, and then we will use these invariants to design and verify a discrete-time trajectory genera-

tor for our car so that it can approach the intersection without violating it.

A *Lyapunov function* is a generalization of the idea of energy for mechanical systems. A thorough exposition of Lyapunov functions and its role in control theory can be found in [31]. For our purposes, we will only consider global Lyapunov functions for continuous systems. Consider a continuous system $x' = f(x)$, such that $f(x_{eq}) = 0$ for some state value x_{eq}. We say that the point x_{eq} is an *equilibrium* of the system, because the system will not evolve when at x_{eq}. Note that in general x can be a vector. We say that a function V is a global Lyapunov function of the system if

1. $V(x_{eq}) = 0$, i. e., it is zero at the equilibrium,
2. $V(x) > 0$ for all $x \neq x_{eq}$, and
3. $\frac{dV}{dx} f < 0$ for all $x \neq x_{eq}$.

These conditions ensure that the system will gravitate towards the equilibrium. The first and second conditions together guarantee that the equilibrium is a global minimum of the Lyapunov function. The third condition says that the dynamics of the system descend the gradient of the function V. As a result, the system will evolve in a way that minimizes V, and will stop at the equilibrium.

A Lyapunov function is often called a *generalized energy function*, because energy of a system is always positive (2) and must always be dissipated (3), according to the laws of physics. A Lyapunov function, however, is a more general idea which can be used in cases when a straightforward notion of energy is not available or does not make physical sense.

The key property of Lyapunov functions which we will use in this section is that for any positive constant c, the sublevel set $V(x) \leq c$ is an invariant of the system. Intuitively, this follows from the fact that along the system dynamics, the Lyapunov function is decreasing. Then along all future states, $V(x)$ must be less than the initial state, so that $V(x) \leq c$ for all future states. Lyapunov functions are often available as a side effect of controller design, which can be leveraged for verification.

As an example, consider the model in Example 9a. This example shows our familiar car model, but with the addition of a continuous proportional-derivative (PD) control law in (67) that controls the acceleration of the car with the goal of stabilizing the system around a reference position p_r. The constant K_p is called the proportional gain, which allows the controller to act on the difference between the current car position p and the desired position p_r. The constant K_d is called the derivative gain, which allows the controller to respond to the velocity. With this form, the controller prescribes zero acceleration when $p = p_r$ and $v = 0$. The controller gains in Example 9a have been chosen according to a standard control design procedure known as pole placement [12], and as a side effect of the control design procedure it is known that the controlled system has the Lyapunov function given below.

$$V(p, p_r, v) = \frac{5}{4}(p - p_r)^2 + \frac{(p - p_r)v}{2} + \frac{v^2}{4}$$

In Example 9a, the proof task is to show that the set of positions and velocities such that $V(p, p_r, v) < c$ is an invariant set, for any positive constant c. This is a simple verification task that only requires telling KeYmaera to treat the safety condition as a differential invariant (see Section 5.10), and the proof follows from the properties of the Lyapunov function.

Example 9a A PD control law for the car model

$$\text{init} \rightarrow [\mathit{plant}]\,(\text{req}) \tag{64}$$

$$\text{init} \equiv v \geq 0 \wedge \boldsymbol{c > 0 \wedge K_p = 2 \wedge K_d = 3} \tag{65}$$

$$\wedge\, \boldsymbol{V(p, p_r, v) < c} \tag{66}$$

$$\mathit{plant} \equiv p' = v, v' = \boldsymbol{-K_p(p - p_r) - K_d v} \tag{67}$$

$$\text{req} \equiv V(p, p_r, v) < c \tag{68}$$

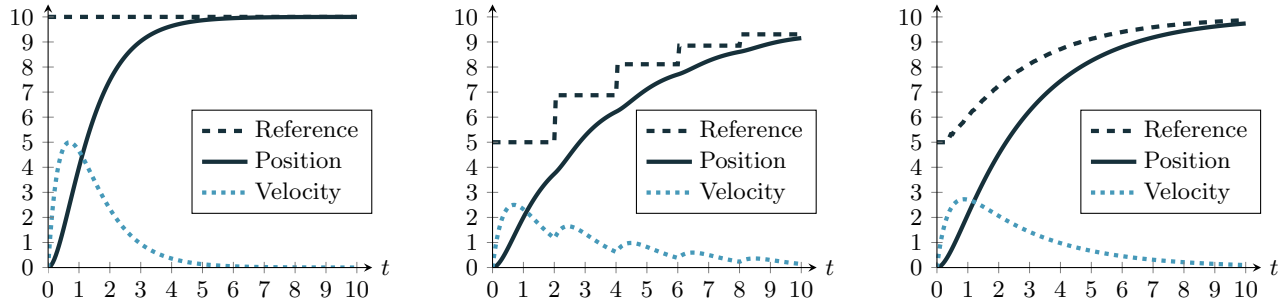
In general, the goal of a controller design task is to feed inputs to the physical system such that the state of the system moves from some initial value x_i to a desired final value x_f. Traditionally, this task is decoupled into the design of two subsystems;

1. a *regulator*, which stabilizes or *regulates* the state of the system around a given setpoint, and
2. a *trajectory generator*, which produces a sequence of setpoints that the system should use as references on its way to the target state.

Note that the feedback controller shown in Fig. 3 is a regulator, since it seeks to stabilize the system state around a desired reference r (in Example 9a, the controller tries to stabilize the car at a reference position p_r). This reference would in practice be generated by a trajectory generator, which is not shown in Fig. 3.

A naïve design for a trajectory generator is to simply feed the desired final state as the reference state value, and to let the regulator do all of the work of bringing the system to the desired state. In practice, this is not a good solution, because attempting to move the state between two distant values will result in a large control effort. In the example of a car, this means that the controller will accelerate maximally. This will strain the engine, forcing it to operate outside of the conditions for which it was calibrated. As a result, performance across numerous benchmarks such as fuel efficiency and emissions reduction will degrade. Additionally, a passenger in such a car will be made uncomfortable by the sudden jolt of acceleration.

For general physical systems, sudden control shocks may damage the equipment, and in general will cause excessive wear, shortening its lifespan. Working together,



(a) Regulator with reference value set to the position of the stop sign

(b) Trajectory generator: reference update second by second

(c) Trajectory generator: fast reference update with Lyapunov-based condition (72)

Fig. 9: Difference between approaching the stop sign with a regulator or a trajectory generator (Example 9b)

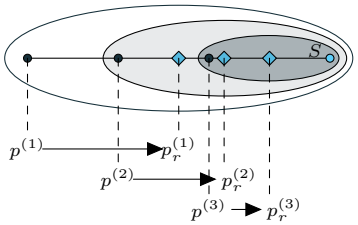


Fig. 10: Illustration of reference points set by the trajectory generator: point $p^{(1)}$ moves towards reference point $p_r^{(1)}$ until close enough, i.e. condition (72) holds at a point $p^{(2)}$, which is when the trajectory generator switches the reference point to $p_r^{(2)}$. Finally, at $p^{(3)}$ the generator switches to $p_r^{(3)}$.

Example 9b A trajectory generator for the stop sign controller

$$\text{init} \rightarrow [(\mathit{ctrl}; \mathit{plant})^*](\text{req}) \tag{69}$$

$$\text{init} \equiv v \geq 0 \wedge p_m \leq p \leq S \wedge \boldsymbol{p_r = \frac{p_m + S}{2}} \wedge K_p = 2 \wedge K_d = 3 \wedge \boldsymbol{V(p, p_r, v) \leq \left(\frac{S - p_m}{2}\right)^2} \tag{70}$$

$$\mathit{ctrl} \equiv \Big(\boldsymbol{p_m := p;\ p_r := \frac{p_m + S}{2};} \tag{71}$$

$$\boldsymbol{?V(p, p_r, v) \leq \left(\frac{S - p_m}{2}\right)^2}\Big) \tag{72}$$

$$\cup\ \boldsymbol{?\text{true}} \tag{73}$$

$$\mathit{plant} \equiv p' = v, v' = -K_p(p - p_r) - K_d v\ \boldsymbol{\&\ v \geq 0} \tag{74}$$

$$\text{req} \equiv p \leq S \tag{75}$$

a combination of a regulator and a trajectory generator will allow the system to glide gracefully along a trajectory to the desired final state. Fig. 9 illustrates the difference between using only a regulator and using a trajectory generator: note that the maximum velocity in Fig. 9a considerably exceeds the velocity in Fig. 9b, where a trajectory generator updates the reference point every other second. The velocity curve becomes smooth when the trajectory generator runs fast, see Fig. 9c.

In Example 9b, we design a discrete-time trajectory generator that chooses the reference point p_r such that the reference point is always in front of the current position, and the car never violates the stop sign. One strategy is to have a sampling controller that chooses as reference the midpoint between the current position and the stop sign. The controller may choose a new reference point whenever the car is "near" the old reference point, in a way that we will make more precise below. The controller moves along a chain of reference points being equilibria of the Lyapunov function, see Fig. 10.

The controller measures the position of the car and saves its value into a variable called p_m. Then, the controller chooses as a reference the mid point between p_m and the stop sign at S, which is at a distance of $(p_m + S)/2$ (71). We consider the reference point as the equilibrium of the Lyapunov function, so that the value of the Lyapunov function at the stop sign is $\left(\frac{S-p_m}{2}\right)^2$. If the car is inside the sublevel set $V(p, p_r, v) \leq \left(\frac{S-p_m}{2}\right)^2$, the Lyapunov function would have to increase for the car to be able to exceed the stop sign, but it cannot. It follows that the system can never exceed the stop sign. Hence, the Lyapunov function sublevel set $V(p, p_r, v) \leq \left(\frac{S-p_m}{2}\right)^2$ is an invariant. However, it may be the case that the car has not yet reached a position at which it is inside this invariant. To remedy this, the trajectory generator checks in (72) to see whether it has entered the sublevel set for the next trajectory point, and otherwise simply keeps the old one by testing whether the formula true holds, which is a no-op (73). The corresponding model in Example 9b can be easily verified by using the loop invariant (76).

$$V(p,p_r,v) \leq \left(\frac{S-p_m}{2}\right)^2 \land p_m \leq p \land p_r = \frac{p_m+S}{2} \land v \geq 0 \tag{76}$$

The first conjunct represents the sublevel set of the Lyapunov function that is active when the trajectory generator calculates the reference point. The second conjunct says that the measured position is always less than or equal to the current position—i. e., the measured position can be outdated, but it can never exceed the position. The third conjunct says that the reference position is the midpoint between the last measured position and the stop sign. These two conjuncts contain information that is visible in the discrete portion (71) of the program, but not the continuous part. By lifting this into the loop invariant, we are able to use this information when proving goals about the overall program. Similarly, the fourth conjunct says that the velocity is always positive. This is a property of the continuous portion (74) of the program, which we lift into the loop invariant so that it can be used in the proof of safety of the overall system. Note that we also require that this invariant holds at the start of the system evolution in (70), so that the system is not unsafe before the reference trajectory generator can even act on it. The controller tests that it holds for the new trajectory point before switching to it (72). Otherwise, the new reference point p_r is discarded and the old reference point is kept in (73) instead. At runtime, this means that the controller never switches to a distant target point before it made enough progress towards the current target point, so that the next target point can be used safely.

5.10 Example 10: Car Controller for Nonlinear Dynamics

In the previous examples we have modeled motion of the car on a one-dimensional straight lane. In the next step, we model the behavior of a car with steering, so that it drives on a two-dimensional lane. Steering lets the car choose a steering angle, which influences the turn radius of the car: keeping the front wheels straight means the car drives straight, turning them a little results in a slight turn with a large radius, turning them hard results in a sudden turn with a small radius. This means the car drives a sequence of circular arcs with varying radii as a trajectory. An animation of this type of controller, which discretely changes curve radius to control steering, can be viewed online: *Modeling Discrete Steering* [1]. A sample trajectory is depicted in Fig. 11.

Such motion leads to nonlinear dynamics models. As a safety property, we want to prove that the car controller manages to stay within the bounds of the lane, i. e., it does not deviate from the center of the lane too much.

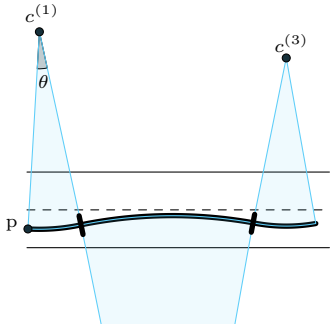


Fig. 11: Motion on a road as a sequence of circular arcs: Curved trajectory of a car that stays on its lane following a sequence of circular arcs around the respective centers $c^{(i)}$ of varying radius and varying arc lengths.

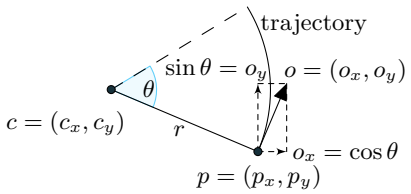


Fig. 12: State illustration of a car on a two-dimensional plane. The car has position $p = (p_x, p_y)$, orientation $o = (o_x, o_y)$, and drives on circular arcs of radius r and angle θ around curve center points $c = (c_x, c_y)$.

But first let us consider how we must extend the one-dimensional car model to become two-dimensional, as illustrated in Fig. 12. The car now needs two coordinates to describe its current position $p = (p_x, p_y)$ and we need to describe where it is heading. Note that in the previous models, orientation was implicitly encoded in the velocity of the car (the car was heading towards increasing position values). Here, we describe the orientation of the car with a two-dimensional vector that points in the direction where the car is currently driving: $o = (o_x, o_y)$. We make o a unit vector, because otherwise it would implicitly also describe the velocity v of the car.

But how is this orientation vector linked to the curve of the car and how does it change over time? Let us model a curve to find out. An important property of a curve is its radius r. The car makes a sharp turn if the radius r is small. If the radius becomes larger then the curve becomes increasingly straightened; the car drives a straight line if the radius is infinite ($r = \infty$). We do not need to model the center of the curve, because we can infer it from the orientation of the car and the radius using the following formulas: $o_x = -\frac{p_y - c_y}{r}$ and $o_y = \frac{p_x - c_x}{r}$. The center of the curve must be on an axis perpendicular to the orientation of the car, and it is either r to the left or to the right of the car on this axis, depending on

whether the car drives a left turn or a right turn. Does this mean we have to introduce a new variable telling us whether the car drives clockwise or counter-clockwise? Luckily no: in this case, we can spare an additional variable by encoding whether the curve bends left or right in the radius r. A positive radius $r > 0$ means driving counter-clockwise (i. e., left curve as in Fig. 12), while a negative radius $r < 0$ means driving clockwise (i. e., right curve).

Now that we know how the curve is linked to the current orientation of the car, let us consider how we need to change the orientation along the curve. We know that the car drives with linear velocity v, which changes according to the acceleration of the car $v' = a$. On a straight line as in the previous examples this lets us easily compute new positions of the car through Newton's laws of motion. On a circular curve, we can consult rigid body planar motion to compute the angular velocity ω on the circle from the linear velocity v of the car and the radius r of the curve: $r\omega = v$. As a result, angular velocity also changes according to the acceleration of the car $\omega' = \frac{a}{r}$. But more importantly, the angular velocity ω lets us derive position changes of the car on the curve in terms of an *angle* θ between the car's current position and its new positions. The differential equation for changing the angle is simply $\theta' = \omega$. Knowing the angle on the curve now makes it easy to compute the orientation of the car using trigonometry: $o = (o_x, o_y) = (\cos\theta, \sin\theta)$. To sum up, the orientation changes of our car could be modeled using the differential equations

$$\begin{aligned} o_x' &= (\cos\theta)' = (-\sin\theta)\theta', \\ o_y' &= (\sin\theta)' = (\cos\theta)\theta', \\ \theta' &= \omega, \text{ and } \omega' = \frac{a}{r} \ . \end{aligned}$$

Unfortunately, we cannot easily use trigonometric functions, nor any other transcendental functions in a proof, because they result in undecidable arithmetic. So we need to find a different representation. On close examination, the differential equations do not actually need to make the angle θ explicit, because we encode the angle implicitly in the orientation of the car $o_x = \cos\theta$ and $o_y = \sin\theta$. So we can rephrase the differential equations as $o_x' = (-\sin\theta)\theta' = -o_y\omega$ and $o_y' = (\cos\theta)\theta' = o_x\omega$. This technique of differential axiomatization [46] was also applied to handle curved flight maneuvers and the motion of autonomous robots [36]. Even though these differential equations do not have a closed-form polynomial solution, it turns out that we can handle such differential equations using differential cuts and differential invariants. But before we dive into the details of these advanced verification techniques for differential equation systems, let us briefly summarize the model.

In Example 10 the car is initially stopped at the center of the lane ($p_y = l_y$), its initial steering points to a curve of positive radius ($r > 0$), and the orientation of the car is a unit vector ($\|o\| = 1$). As in the previous models, the car has three control choices: in (79) it may choose a new trajectory when it is safe to do so by choosing a new radius ($r \neq 0$) and adjusting its angular velocity to fit the linear velocity ($\omega r = v$); in (80) it may stay stopped by not accelerating $a = 0$ when already being stopped; and in (81) it may brake nondeterministically $-B \leq a \leq -b$ on its current trajectory to stay on the lane. The plant is modeled in (83)–(86) using the differential equation systems introduced above. The overall pattern of the plant is still similar to $p' = v, v' = a$ from the previous examples, except that the direction into which the position p moves is in two-dimensional space and changing. The safety property req in (87) formalizes what it means for a car to stay within lane bounds: the position of the car p_y deviates from the center of the lane l_y by at most half the lane width l_w. Here, we assume that the lane is oriented along the x-axis of our space and its width along the y-axis.

Example 10 Car controller with nonlinear dynamics

$$\text{init} \rightarrow [(\mathit{ctrl}; \mathit{plant})^*]\,(\text{req}) \tag{77}$$

$$\begin{aligned} \text{init} \equiv\ & l_w > 0 \land \boldsymbol{p_y = l_y} \land v \geq 0 \land \boldsymbol{r > 0} \land \boldsymbol{\|o\| = 1} \\ & \land A > 0 \land B \geq b > 0 \land \varepsilon > 0 \end{aligned} \tag{78}$$

$$\begin{aligned} \mathit{ctrl} \equiv\ & (?\text{safe}_\varepsilon; a := *; ? - B \leq a \leq A); \\ & \boldsymbol{\omega := *}; \\ & \boldsymbol{r := *; ?r \neq 0 \land \omega r = v} \end{aligned} \tag{79}$$

$$\cup\ (?v = 0; a := 0; \boldsymbol{\omega := 0}) \tag{80}$$

$$\cup\ (a := *; ? - B \leq a \leq -b) \tag{81}$$

$$\text{safe}_\varepsilon \equiv |\boldsymbol{p_y - l_y}| + \frac{v^2}{2b} + \left(\frac{A}{b} + 1\right)\left(\frac{A}{2}\varepsilon^2 + \varepsilon v\right) < \boldsymbol{l_w} \tag{82}$$

$$\mathit{plant} \equiv c := 0; \tag{83}$$

$$\boldsymbol{p_x' = vo_x, p_y' = vo_y}, v' = a, \tag{84}$$

$$\boldsymbol{o_x' = -o_y\omega, o_y' = o_x\omega, \omega' = \frac{a}{r}}, c' = 1 \tag{85}$$

$$\&\ v \geq 0 \land c \leq \varepsilon \tag{86}$$

$$\text{req} \equiv \boldsymbol{|p_y - l_y| < l_w} \tag{87}$$

The proof of Example 10 uses differential cuts, differential weakening and differential induction, since the differential equations in Example 10 do not have a polynomial solution. We describe these proof techniques in detail below.

Differential Invariants, Differential Cuts, and Differential Induction When a differential equation system does not have a closed-form solution, we cannot follow our practice from the examples so far and simply replace the differential equation system with its solution after introducing a time variable. To handle such systems, KeYmaera provides proof rules for differential induc-

tion, differential invariants, differential cut and differential weakening [46, 47, 50, 52]. In the following paragraphs we illustrate how to prove properties about systems with nonlinear dynamics using these proof rules.

A simple proof rule for handling differential equation systems is *differential weakening*. Differential weakening replaces a differential equation system with its evolution domain constraints. For example, let us assume we want to use differential weakening to prove $[plant](v \geq 0)$, where *plant* denotes the differential equation system of lines (83)–(86) of Example 10. When we apply differential weakening, we have to show

$$\underbrace{v \geq 0 \land c \leq \varepsilon}_{\text{evolution domain}} \rightarrow \underbrace{v \geq 0}_{\text{safety condition}} ,$$

because $v \geq 0 \land c \leq \varepsilon$ is the evolution domain (86) of the differential equation system. Here, we can show our requirement, because the evolution domain constraints are sufficiently informative to let us prove the requirement. More often than not, however, the evolution domain constraints do not carry enough information. For example, we cannot prove $[plant](c \geq 0)$ using differential weakening alone, since $v \geq 0 \land c \leq \varepsilon \rightarrow c \geq 0$ is not true.

We can use *differential cuts* to make the evolution domain sufficiently informative. A differential cut adds information that we have proved about an ODE to the evolution domain constraints of a differential equation system. This way, we can increasingly strengthen the evolution domain by successively applying differential cuts, until eventually the differential equation can be resolved by differential weakening.

Let us use a differential cut to enrich the evolution domain constraint of Example 10. Here, we use our domain knowledge about the system to find appropriate differential cuts. By inspecting the clock variable, we see that it is reset to 0 at the beginning and then evolves with constant slope 1. We can therefore use the differential cut rule to provide $c \geq 0$ as an additional evolution domain constraint, so that the evolution domain constraint (86) becomes $v \geq 0 \land c \leq \varepsilon \land c \geq 0$. Using this new evolution domain constraint, we can now use differential weakening to prove $[plant](c \geq 0)$ from above.

However, before we get to use such an additional constraint, we first have to prove that it holds along the differential equation system, since otherwise additional constraints in the evolution domain would change the system dynamics. We call such constraints *differential invariants*.

Differential invariants are somewhat similar to inductive invariants for discrete loops: a differential invariant has to be true at the beginning of the continuous evolution, it has to stay true throughout the evolution, and it has to be strong enough so that we can prove our safety condition from it. Thus, differential invariants define an induction principle for differential equations [50].

The corresponding proof rule in KeYmaera is called *differential invariant*. Intuitively, differential invariants show that a formula is getting "more true" when following the differential equation system. To prove that $c \geq 0$ is a differential invariant of the differential equation system in line (85), we have to show that $c \geq 0$ is true at the beginning and remains true throughout. The constraint is true at the beginning because the clock is reset in line (83) and $0 \geq 0$ holds trivially. Showing that the constraint remains true throughout needs a little explanation. The differential invariant rule lets us assume the current evolution domain constraint, because the continuous evolution is not allowed to leave it. It requires us to show that the syntactic derivative of the new constraint is true [52]:

$$\underbrace{v \geq 0 \land c \leq \varepsilon}_{\text{evolution domain}} \rightarrow \underbrace{c' \geq 0'}_{\text{syntactic derivative}} .$$

Primed variables and real numbers in this syntactic derivative are further substituted with their actual term as defined in the differential equation system. So we have to show $v \geq 0 \land c \leq \varepsilon \rightarrow 1 \geq 0$, because $0' = 0$ by definition and $c' = 1$ as specified in line (85) of Example 10.

In the complete example we use further differential cuts similar to our robot case study [36] to strengthen the evolution domain constraint with differential invariants on orientation, velocity, and traveled distance. More in-depth information about differential weakening, cuts, and induction can be found in [52].

For step-by-step instructions on how to use differential invariants, differential cuts, and differential weakening in KeYmaera, watch the *Differential Cuts, Invariants, and Weakening* video [1].

6 Advanced Modeling Concepts and Pitfalls

In this section, we introduce advanced modeling concepts and we discuss modeling pitfalls resulting in faulty models that vacuously satisfy any property.

6.1 Hybrid Time

A common assumption in hybrid systems, as already mentioned, is that discrete actions do not consume time. Because discrete actions are assumed not to consume time, multiple discrete actions can occur at the same real point in time. To capture this mathematically, the time axis used in hybrid systems models contains a natural number component which counts the number of discrete actions. That is, a hybrid point in time is a pair $(r, n) \in \mathbb{R}_{\geq 0} \times \mathbb{N}$. For each real-valued point in time r there is a discrete time axis that reflects the order of discrete actions. Hence the time model for hybrid systems, called *hybrid time*, is given by $\mathbb{R}_{\geq 0} \times \mathbb{N}$, see [53] for details.

Table 3: Summary of tutorial examples: modeling aspects and proof strategies

	Description	**ODE**	**Strategy**		**Statistics**	
	Incremental Change		Design	Proof	Steps[i]	Time
Ex. 1	Uncontrolled plant	Linear	Design ODE	Automated	0 / 7	1.1 s
Ex. 2	Discrete controller, nondeterministic repetition	Linear	Find invariant (automated in some cases)	Automated	0 / 73	1.3 s
Ex. 3a	Event-triggered controller	Linear	Find event trigger	Automated	0 / 85	0.7 s
Ex. 4	Pitfalls in event-triggered control	Linear	Avoid pitfalls	No proof since not valid	–	–
Ex. 5	Time-triggered controller	Linear	Find safe control conditions	Automated	0 / 107	0.7 s
Ex. 6	Guarded nondeterministic assignment	Linear	Model assignment guards	Automated	0 / 114	1.1 s
Ex. 7	Nondeterministic overapproximation of formulas	Linear	Model formula bounds	Automated	0 / 131	3.1 s
Ex. 8	Actuation disturbance	Diff. Ineq.[ii]	Model disturbance bounds	Diff. inequality elimination	61 / 416	1.8 s
Ex. 9	Lyapunov functions, PD control	Linear	Find Lyapunov functions, controller gains	Differential invariant	0 / 29 52 / 239	2.9 s 14.8 s
Ex. 10	Curved motion in two-dimensional space	Nonlin.	Avoid transcendental functions	Differential cut, invariant	170 / 737	2.1 s

[i] Proof steps: interactive / total steps [ii] Differential inequality

6.2 Discrete Actions with Nonzero Duration

We have assumed that the computations by the controller take zero time because this is a common assumption to simplify controller design and analysis. However, in many cases it is desirable to consider delays due to sensors, computation time, and actuators. The general idea is to model the start of the desired action and store the result in ghost variables, then allow the continuous dynamics to evolve for the desired duration, and finally store the values of the ghost variables into the program variables when they take effect. Here we will illustrate the case of computation delay. Suppose, for example, that we are controlling the acceleration of a car as before, and that the computation time takes up to half a second. Then the controller can be modeled by the following hybrid program.

Example 11 Controller with nonzero computation time

$$ctrl \equiv z := -b; \tag{88}$$
$$t := 0;\ (p' = v,\ v' = a,\ t' = 1\ \&\ t \leq 0.5); \tag{89}$$
$$a := z; \tag{90}$$

First, in line (88) the desired acceleration of $-b$ is stored into the ghost variable z. Then, in line (89) the plant is allowed to evolve as long as a time variable t is less than or equal to 0.5, meaning it will delay subsequent actions for up to half a second. Finally, in line (90) we update the acceleration commanded by the controller to the value of the ghost variable, which was computed before the delay. This style can also be used to address sensor and actuator delay.

6.3 Disjoint Tests and Evolution Domains

When combining choices and tests it is important to make sure that the model does not get blocked in an unnatural way. For example, the program

$$(?v < 3; v' = A) \cup (?v > 5; v' = -B) \tag{91}$$

cannot evolve if v is between 3 and 5. Therefore, it is good modeling practice to have at least one branch in the program for each case, so (91) should be augmented with a case $\cup ?3 \leq v \leq 5;\ v' = \ldots$. Evolution domain constraints also need to be designed with care. For example, the HP

$$((v' = -B\ \&\ v \geq 0) \cup (v' = -b\ \&\ v < 0))^* \tag{92}$$

has disjoint evolution domain constraints. When $v = 0$, the system cannot switch to the second choice, because its evolution constraint $v < 0$ is not satisfied for the initial state. There is an infinitesimal gap in the model, see [54]. So, the second branch should use the evolution domain constraint $v \leq 0$ instead of $v < 0$.

6.4 Non-Existence of Systems

Tests outside nondeterministic choices must be used carefully, since they potentially entail non-existence of the

modeled system. For example, the dL statement

$$A > 0 \land v > 0 \rightarrow [(v' = A)\,;\ ?v = 0]\,(v = 0)$$

results in an empty set of executions, since none of the values of v will satisfy the test $?v = 0$. Thus, the property $v = 0$ is vacuously true simply because the system never runs successfully. Such issues can be detected by liveness proofs using the diamond modality $\langle\alpha\rangle$: the dL statement $v > 0 \rightarrow \langle(v' = A)\,;\ ?v = 0\rangle\,(v = 0)$ is only true, if at least one run satisfies the requirement, which is not the case if $A > 0$.

6.5 Safety Throughout vs. Safety Finally

The evolution of a differential equation system is allowed to nondeterministically stop at any time (even zero) before the evolution domain becomes false. Thus, dL properties of the form $[(x' = \theta\ \&\ H)]\,\phi$ usually verify safety *throughout* system execution. A subsequent test, as in $[(x' = \theta\ \&\ c \leq \varepsilon)\,;\ ?c = \varepsilon]\,\phi$, however, means that all traces that end before the clock reached its maximum time will not be considered during the proof. Thus, we only verify that the requirement ϕ will be true at the end of the evolution after exactly time ε, which is weaker than safety throughout. The lesson here is to always take care when using tests, as discussed in Section 5.6, or to add temporal logic dTL, see [30, 47].

7 Summary and Outlook

In this tutorial, we presented the basic modeling and proof techniques provided by KeYmaera to verify parametric hybrid systems, focusing on safety properties.

The modeling and proof process followed in this tutorial gradually develops the relevant features of hybrid system models and its associated proof techniques. Such an incremental development is not only effective for learning KeYmaera but continues to be helpful in most hybrid system applications for managing proof complexity. Since the dynamics of applications is often complex and their behavior rather subtle, it is almost impossible to get them correct without first considering simpler models and simpler controllers and basing the subsequent design of extensions on provably correct designs of the simpler systems. Incremental proofs for incremental system designs make it easier to localize which part of a controller design is responsible for violating safety. Proof strategies are also often easier to find in simpler settings and then transferred to subsequent more complex system designs. In this tutorial, we incrementally developed models and their associated proof strategies, as summarized in Table 3.

KeYmaera's proofs enable strong guarantees about the correctness of system design models. If the real system fits to the model, its behavior is guaranteed to satisfy the correctness properties verified w.r.t. the model. KeYmaera supports ModelPlex, a method to automatically turn verified models into provably correct runtime monitors, so that proofs about models transfer to the running system in verifiably correct ways [38].

In the following paragraphs, we briefly list further features of KeYmaera. In addition to safety properties, KeYmaera is able to show liveness properties. These can be expressed using the diamond modality $\langle\cdot\rangle$. Where in the proofs of safety properties invariants were necessary to prove properties of loops now *variants*, provided using @variant(...) annotations, are required [45]. A variant can be seen as a formula encoding progress (cf. a termination function in discrete program verification). KeYmaera can also be used to prove general formulas of dL with arbitrary nesting of quantifiers and modalities. We refer to previous work [45, 47, 58] for such examples, which is helpful, for instance, for controllability and reactivity properties. KeYmaera can be used to reason about hybrid games by means of *differential dynamic game logic* [60]. Here, the number of interactions between box and diamond modalities is not fixed a priori but instead statements about arbitrary alternations of these can be made. Therefore, this extension can be used, for instance, to reason about the existence of a controller rather than the correct functioning of a specific one.

In addition KeYmaera can be used to reason about distributed hybrid systems with an *a priori* unknown number of interacting agents. For this, hybrid programs can be extended to quantified hybrid programs and the logic allows quantifiers over additional domains such as the domain of all cars. The resulting logic is called *quantified differential dynamic logic* [48, 49]. This allows, among other things, to reason explicitly about the number of cars involved in a lane change maneuver during the proof instead of having to apply some argument why it is sufficient to consider only a certain limited number of cars [34].

While invariants are an integral part of a system design, KeYmaera also supports techniques for automatically generating invariants and differential invariants [55], for which significant progress has been made recently [24], leading to a decision procedure for algebraic invariants.

Finally, we have been investigating proof-aware refactorings to support incremental model and control designs with incremental proofs [40]. The idea is to perform transformations on the hybrid programs in a structured way in order to minimize the effort required for reproving properties about these programs. This specifically supports an iterative development of hybrid systems.

Acknowledgments. The authors would like to thank the anonymous reviewers for their very constructive and detailed feedback.

References

1. Online KeYmaera tutorial videos. `http://video.symbolaris.com`
2. Alur, R.: Formal verification of hybrid systems. In: Chakraborty, S., Jerraya, A., Baruah, S.K., Fischmeister, S. (eds.) EMSOFT. pp. 273–278. ACM (2011)
3. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theor. Comput. Sci. 138(1), 3–34 (1995)
4. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.H.: Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In: Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H. (eds.) Hybrid Systems. LNCS, vol. 736, pp. 209–229. Springer (1992)
5. Alur, R., La Torre, S., Pappas, G.J.: Optimal paths in weighted timed automata. Theor. Comput. Sci. 318(3), 297–322 (2004)
6. Aréchiga, N., Loos, S.M., Platzer, A., Krogh, B.H.: Using theorem provers to guarantee closed-loop system properties. In: Tilbury, D. (ed.) ACC. pp. 3573–3580 (2012)
7. Asarin, E., Dang, T., Maler, O.: The d/dt tool for verification of hybrid systems. In: Brinksma, E., Larsen, K.G. (eds.) CAV. LNCS, vol. 2404, pp. 365–370. Springer (2002)
8. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach, LNCS, vol. 4334. Springer (2007)
9. Behrmann, G., Fehnker, A.: Efficient guiding towards cost-optimality in UPPAAL. In: Margaria, T., Yi, W. (eds.) TACAS. LNCS, vol. 2031, pp. 174–188. Springer (2001)
10. Behrmann, G., Fehnker, A., Hune, T., Larsen, K.G., Pettersson, P., Romijn, J., Vaandrager, F.W.: Minimum-cost reachability for priced timed automata. In: Benedetto, M.D.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC. LNCS, vol. 2034, pp. 147–161. Springer (2001)
11. Brown, C.W.: QEPCAD B: A program for computing with semi-algebraic sets using CADs. SIGSAM Bull. 37(4), 97–108 (2003)
12. Chen, C.T.: Linear System Theory and Design. Oxford University Press, 3rd edn. (1999)
13. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV, LNCS, vol. 8044, pp. 258–263. Springer (2013)
14. Clarke, Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, MA, USA (1999)
15. Damm, W., Dierks, H., Disch, S., Hagemann, W., Pigorsch, F., Scholl, C., Waldmann, U., Wirtz, B.: Exact and fully symbolic verification of linear hybrid automata with large discrete state spaces. Sci. Comput. Program. 77(10-11), 1122–1150 (2012)
16. Davenport, J.H., Heintz, J.: Real quantifier elimination is doubly exponential. Journal of Symbolic Computation 5(1/2), 29–35 (1988)
17. Davoren, J.M., Nerode, A.: Logics for hybrid systems. IEEE 88(7), 985–1010 (2000)
18. Deshpande, A., Göllü, A., Varaiya, P.: SHIFT: A formalism and a programming language for dynamic networks of hybrid automata. In: Antsaklis, P.J., Kohn, W., Nerode, A., Sastry, S. (eds.) Hybrid Systems. LNCS, vol. 1273, pp. 113–133. Springer (1996)
19. Dolzmann, A., Sturm, T.: Redlog: Computer algebra meets computer logic. ACM SIGSAM Bull. 31, 2–9 (1997)
20. Eggers, A., Ramdani, N., Nedialkov, N., Fränzle, M.: Improving SAT modulo ODE for hybrid systems analysis by combining different enclosure methods. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM. LNCS, vol. 7041, pp. 172–187. Springer (2011)
21. Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. JSAT 1(3-4), 209–236 (2007)
22. Frehse, G.: PHAVer: algorithmic verification of hybrid systems past HyTech. STTT 10(3), 263–279 (2008)
23. Frehse, G., Guernic, C.L., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV. LNCS, vol. 6806, pp. 379–395. Springer (2011)
24. Ghorbal, K., Platzer, A.: Characterizing algebraic invariants by differential radical invariants. In: Ábrahám, E., Havelund, K. (eds.) TACAS. LNCS, vol. 8413, pp. 279–294. Springer (2014)
25. Harel, D., Kozen, D., Tiuryn, J.: Dynamic logic. MIT Press (2000)
26. Henzinger, T.A.: The theory of hybrid automata. In: LICS. pp. 278–292. IEEE Computer Society (1996)
27. Henzinger, T.A., Ho, P.H., Wong-Toi, H.: HyTech: A model checker for hybrid systems. STTT 1(1-2), 110–122 (1997)
28. Ioannu, P., Xu, Z., Eckert, S., Clemons, D., Sieja, T.: Intelligent cruise control: Theory and experiment. In: CDC. pp. 1885–1890 (1993)
29. Jeannin, J.B., Ghorbal, K., Kouskoulas, Y., Gardner, R., Schmidt, A., Platzer, E.Z.A.: A formally verified hybrid system for the next-generation airborne collision avoidance system. In: Baier, C., Tinelli, C. (eds.) TACAS. LNCS, Springer (2015)
30. Jeannin, J.B., Platzer, A.: dTL^2: Differential temporal dynamic logic with nested temporalities for hybrid systems. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR. LNCS, vol. 8562, pp. 292–

306. Springer (2014)
31. Khalil, H.K.: Nonlinear Systems. Prentice Hall, 3rd edn. (2001)
32. Kouskoulas, Y., Renshaw, D.W., Platzer, A., Kazanzides, P.: Certifying the safe design of a virtual fixture control algorithm for a surgical robot. In: Belta, C., Ivancic, F. (eds.) HSCC. pp. 263–272. ACM (2013)
33. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. STTT 1(1+2), 134–152 (1997)
34. Loos, S.M., Platzer, A., Nistor, L.: Adaptive cruise control: Hybrid, distributed, and now formally verified. In: Butler, M., Schulte, W. (eds.) FM. LNCS, vol. 6664, pp. 42–56. Springer (2011)
35. Loup, U., Scheibler, K., Corzilius, F., Ábrahám, E., Becker, B.: A symbiosis of interval constraint propagation and cylindrical algebraic decomposition. In: Bonacina, M.P. (ed.) CADE. LNCS, vol. 7898, pp. 193–207. Springer (2013)
36. Mitsch, S., Ghorbal, K., Platzer, A.: On provably safe obstacle avoidance for autonomous robotic ground vehicles. In: Newman, P., Fox, D., Hsu, D. (eds.) Robotics: Science and Systems (2013)
37. Mitsch, S., Loos, S.M., Platzer, A.: Towards formal verification of freeway traffic control. In: Lu, C. (ed.) ICCPS. pp. 171–180. IEEE (2012)
38. Mitsch, S., Platzer, A.: Modelplex: Verified runtime validation of verified cyber-physical system models. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV. LNCS, vol. 8734, pp. 199–214. Springer (2014)
39. Mitsch, S., Quesel, J.D., Platzer, A.: From safety to guilty and from liveness to niceness. In: 5th Workshop on Formal Methods for Robotics and Automation (2014)
40. Mitsch, S., Quesel, J.D., Platzer, A.: Refactoring, refinement, and reasoning – a logical characterization for hybrid systems. In: Jones, C.B., Pihlajasaari, P., Sun, J. (eds.) FM. LNCS, vol. 8442, pp. 481–496. Springer (2014)
41. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
42. Olderog, E.R., Dierks, H.: Real-time systems - formal specification and automatic verification. Cambridge University Press (2008)
43. Plaku, E., Kavraki, L.E., Vardi, M.Y.: Hybrid systems: from verification to falsification by combining motion planning and discrete search. Form. Methods Syst. Des. 34(2), 157–182 (2009)
44. Platzer, A.: Differential dynamic logic for verifying parametric hybrid systems. In: Olivetti, N. (ed.) TABLEAUX. LNCS, vol. 4548, pp. 216–232. Springer (2007)
45. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reas. 41(2), 143–189 (2008)
46. Platzer, A.: Differential-algebraic dynamic logic for differential-algebraic programs. J. Log. Comput. 20(1), 309–352 (2010)
47. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer, Heidelberg (2010)
48. Platzer, A.: Quantified differential dynamic logic for distributed hybrid systems. In: Dawar, A., Veith, H. (eds.) CSL. LNCS, vol. 6247, pp. 469–483. Springer (2010)
49. Platzer, A.: A complete axiomatization of quantified differential dynamic logic for distributed hybrid systems. Logical Methods in Computer Science 8(4), 1–44 (2012), special issue for selected papers from CSL'10
50. Platzer, A.: The complete proof theory of hybrid systems. In: LICS. pp. 541–550. IEEE (2012)
51. Platzer, A.: Logics of dynamical systems. In: LICS. pp. 13–24. IEEE (2012)
52. Platzer, A.: The structure of differential invariants and differential cut elimination. Logical Methods in Computer Science 8(4), 1–38 (2012)
53. Platzer, A.: Analog and hybrid computation: Dynamical systems and programming languages. Bulletin of the EATCS 114, 151–200 (2014)
54. Platzer, A.: Foundations of cyber-physical systems. Lecture Notes 15-424/624, Carnegie Mellon University (2014), `http://symbolaris.com/course/fcps14/fcps14.pdf`
55. Platzer, A., Clarke, E.M.: Computing differential invariants of hybrid systems as fixedpoints. Form. Methods Syst. Des. 35(1), 98–120 (2009)
56. Platzer, A., Clarke, E.M.: Formal verification of curved flight collision avoidance maneuvers: A case study. In: Cavalcanti, A., Dams, D. (eds.) FM. LNCS, vol. 5850, pp. 547–562. Springer (2009)
57. Platzer, A., Quesel, J.D.: KeYmaera: A hybrid theorem prover for hybrid systems. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR. LNCS, vol. 5195, pp. 171–178. Springer (2008)
58. Platzer, A., Quesel, J.D.: European Train Control System: A case study in formal verification. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM. LNCS, vol. 5885, pp. 246–265. Springer (2009)
59. Pratt, V.R.: Semantical considerations on Floyd-Hoare logic. In: FOCS. pp. 109–121. IEEE Computer Society (1976)
60. Quesel, J.D., Platzer, A.: Playing hybrid games with KeYmaera. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR. LNCS, vol. 7364, pp. 439–453. Springer (2012)
61. Ratschan, S., She, Z.: Safety verification of hybrid systems by constraint propagation-based abstraction refinement. ACM Trans. Embed. Comput. Syst. 6(1) (2007)
62. Tarski, A.: A Decision Method for Elementary Algebra and Geometry. University of California Press,

Berkeley, 2nd edn. (1951)
63. Tomlin, C., Pappas, G.J., Sastry, S.: Conflict resolution for air traffic management: a study in multi-agent hybrid systems. IEEE T. Automat. Contr. 43(4), 509–521 (1998)
64. Umeno, S., Lynch, N.A.: Safety verification of an aircraft landing protocol: A refinement approach. In: Bemporad, A., Bicchi, A., Buttazzo, G.C. (eds.) HSCC. LNCS, vol. 4416, pp. 557–572. Springer (2007)
65. Wolfram, S.: The Mathematica book (5th edn.). Wolfram-Media (2003)