# SAT Modulo Differential Equation Simulations

Tomáš Kolárik* and Stefan Ratschan†

May 12, 2020

**Abstract**

Differential equations are of immense importance for modeling physical phenomena, often in combination with discrete modeling formalisms. In current industrial practice, properties of the resulting models are checked by testing, using simulation tools. Research on SAT solvers that are able to handle differential equations has aimed at replacing tests by correctness proofs. However, there are fundamental limitations to such approaches in the form of undecidability, and moreover, the resulting solvers do not scale to problems of the size commonly handled by simulation tools. Also, in many applications, classical mathematical semantics of differential equations often does not correspond well to the actual intended semantics, and hence a correctness proof wrt. mathematical semantics does not ensure correctness of the intended system.

In this paper, we head at overcoming those limitations by an alternative approach to handling differential equations within SAT solvers. This approach is usually based on the semantics used by tests in simulation tools, but still may result in mathematically precise correctness proofs wrt. that semantics. Experiments with a prototype implementation confirm the promise of such an approach.

## 1 Introduction

The design of cyber-physical systems is more and more being based on models that can be simulated before the actual system even exists. Here, the most natural way of modeling the physical part is based on differential equations. The resulting models can then be simulated using numerical solvers for ordinary differential equations (ODEs), or tools such as Xcos or Simulink. However, the computational support for automatically analyzing (e.g., testing, verifying) such models is still far from satisfactory.

This has been addressed by SAT solvers [10, 11] that do not only offer efficient discrete (i.e., Boolean) reasoning, but that, in addition, are also able to handle differential equations by integrating interval ODE solvers [17, 19]. Handling ODEs in

*Czech Technical University in Prague, Faculty of Information Technology

†Institute of Computer Science of the Czech Academy of Sciences, ORCID: 0000-0003-1710-1513

such a way is extremely difficult, and most related verification problems are undecidable [5]. The resulting SAT modulo ODE solvers can handle benchmark examples that are impressive, but still quite far away from the size of the problems that may occur in industrial practice.

A further reason why such tools are a poor fit to the needs coming from industrial applications is the fact that classical mathematical solutions usually do *not* correctly represent the intended behavior of industrial models [16], since the design process is based on the results of numerical simulations, and *not* on a mathematical analysis of the underlying differential equations. The numerical simulations differ from mathematical solutions due to discretization and floating-point computation. Hence, the output of the used simulation tool is the authoritative description of the behavior of the model, *not* traditional mathematical semantics. This holds even in cases when the model was designed based on ODEs corresponding to physical laws ("from first principles"), because even in such cases, the parameters of the model are estimated based on simulations. This is becoming all the more important due to the increasing popularity of data driven modeling approaches, for example, based on machine learning.

Therefore, the existing SAT modulo ODE approaches prove correctness wrt. semantics that differs from the notion of correctness used during simulation and testing. We overcome this mismatch by formalizing the semantics of SAT modulo ODE based on numerical simulations. We prove decidability of the resulting problem, and design a simple solver. We provide a syntactical characterization of the kind of inputs for which one can expect an efficient solution from such a solver, and support this by experiments using a prototype implementation.

We also address another restriction of existing SAT modulo ODE approaches. Their support for differential equations has the form of monolithic building blocks that contain a full system of ordinary differential equations within which no Boolean reasoning is allowed. In contrast to that, in this paper we provide a direct integration of ODEs into a standard SAT modulo theory (SMT) framework [3], which results in a tight integration of the syntax of the theory into Boolean formulas, as usual for theories in SMT-LIB [1].

The problem of verifying differential equations wrt. simulation semantics has been addressed before [16, 4], but not in a SAT modulo theory context. Also floating point arithmetic has been realized to be an important domain for verification tools [6, 14], resulting in a floating point theory in SMT-LIB. However, this concentrates on the intricacies of floating point arithmetic, which we largely ignore here, and concentrate instead on the handling of ODEs.

In the next section, we will introduce an illustrative toy example. Then we will present our integration of ODEs into SMT, first using classical mathematical semantics (Section 3), then using simulation semantics (Section 4). In the next three sections, we prove decidability of the resulting theory, design a simple solver, and study its theoretical properties. In Section 8, we present some computational experiments with a prototype implementation, and in Section 9, we conclude the paper.

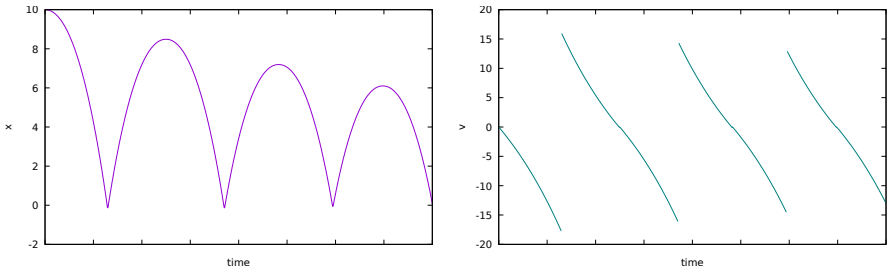


Figure 1: Example Trajectories

This work was funded by institutional support of the Institute of Computer Science (RVO:67985807) and by CTU project SGS20/211/OHK3/3T/18.

## 2 Example

For explaining the intuition behind the syntax of our language and the structure of formulas that we expect to handle, we describe an illustrative toy example. The example corresponds to a bounded model checking problem for a bouncing ball with linear drag—Figure 1 shows the height of the ball on the left-hand side and its speed on the right-hand side.

$$
\begin{aligned}
&g = 9.81 \wedge K = 0.9 \wedge \neg up_1 \wedge \mathit{init}(x_1) = 10 \wedge \mathit{init}(v_1) = 0 \wedge \\
&\dot{x}_1 = v_1 \wedge \left(up_1 \Rightarrow \dot{v}_1 = -g - \tfrac{v_1}{100}\right) \wedge \left(\neg up_1 \Rightarrow \dot{v}_1 = -g + \tfrac{v_1}{100}\right) \wedge \\
&x_1 \geq 0 \wedge (up_1 \Rightarrow v_1 \geq 0) \wedge (\neg up_1 \Rightarrow v_1 \leq 0) \wedge \\
&up_1 \Rightarrow (\mathit{final}(v_1) \leq 0 \wedge \neg up_2 \wedge \mathit{init}(x_2) = \mathit{final}(x_1) \wedge \mathit{init}(v_2) = 0) \wedge \\
&\neg up_1 \Rightarrow (\mathit{final}(x_1) \leq 0 \wedge up_2 \wedge \mathit{init}(x_2) = 0 \wedge \mathit{init}(v_2) = -K\mathit{final}(v_1)) \wedge \\
&\dot{x}_2 = v_2 \wedge \left(up_2 \Rightarrow \dot{v}_2 = -g - \tfrac{v_2}{100}\right) \wedge \left(\neg up_2 \Rightarrow \dot{v}_2 = -g + \tfrac{v_2}{100}\right) \wedge \\
&x_2 \geq 0 \wedge (up_2 \Rightarrow v_2 \geq 0) \wedge (\neg up_2 \Rightarrow v_2 \leq 0) \wedge \\
&up_2 \Rightarrow (\mathit{final}(v_2) \leq 0 \wedge \neg up_3 \wedge \mathit{init}(x_3) = \mathit{final}(x_2) \wedge \mathit{init}(v_3) = 0) \wedge \\
&\neg up_2 \Rightarrow (\mathit{final}(x_2) \leq 0 \wedge up_3 \wedge \mathit{init}(x_3) = 0 \wedge \mathit{init}(v_3) = -K\mathit{final}(v_2)) \wedge \\
&\ldots \\
&\mathit{final}(x_{23}) \geq 8
\end{aligned}
$$

In the example, the variables $up_1, \ldots$ range over the Booleans, the variables $K$ and $g$ range over real numbers, and the variables $x_i, v_i$, $i \in \{1, \ldots, 23\}$ range over functions from corresponding intervals $[0, \tau_i]$ to the real numbers, where the lengths $\tau_i$ are *not* fixed a priori. The example does not provide this information explicitly—we will introduce notation to do so, later. Also, all constraints on those variables (i.e., all invariants) are intended to hold *for all* elements of those intervals. Again we will introduce formal details later.

The dot operator denotes differentiation, *init* denotes the value of the argument function at 0 and *final* the value at $\tau_i$. Note that the example uses the Boolean vari-

ables $up_1, \ldots$ to activate different differential equations and bounds on the variables $x_1, v_1, x_2, v_2, \ldots$.

The ball starts at height 10 with speed zero, and for each $x_i, v_i$, $i \in \{1, \ldots, 23\}$ the pair $x_i, v_i$ models one falling or rising phase of the ball (the figure shows 7 of those). The update $init(v_{i+1}) = -Kfinal(v_i)$ results in a non-continuous change between the last point of $v_i$ and the initial point of $v_{i+1}$. The example checks whether a state with height greater or equal 8 is reached after falling and rising a certain number of times. For illustrative purposes, the modeled behavior is completely deterministic, although our method can also handle non-determinism.

We want to check whether there are values for the variables that satisfy such formulas when interpreting the differential equations using simulation tools. Before going into details we will analyze the structure of the above formula.

First of all, the variables have indices 1, 2, and 3 corresponding to stages of a bounded model checking problem. The indices are just part of the names of the corresponding variables, but still, they clarify the fact that the variables in the formulas also occur in stages. Especially, the variables with the same index belong to the same stage, and within stage $i$, each functional variable, that is, $x_i$ and $v_i$, is determined by a differential equation.

Further, understanding that $x_i$ models the height of a bouncing ball, we see that if the ball is moving down, the constraints $x_i \geq 0$ eventually must be violated, bounding the length of the functional variables. If the ball is moving up, this is ensured by the constraints $v_i \geq 0$.

And finally, the stages also define a specific order on how one can assign values to variables: The first line of the formula assigns values to the variables $g$ and $K$, and initial values of $x_1$ and $v_1$. Then it states differential equations describing the evolution of $x_1$ and $v_1$. Moreover, it states invariants that should hold on those solutions. Next, it describes how the initial value of $x_2$ and $v_2$ depends on the final value of $x_1$ and $v_1$. Then it analogously repeats the above statements for $x_2$ and $v_2$, and so on. In other words, solving the real part of the above formula may proceed in stages, avoiding any circular reasoning.

## 3 Formalization: SAT modulo ODE

In this section, we will tightly integrate ODEs into SAT, roughly following the SMT framework of Barrett and Tinelli [3]. Note that SMT uses *first*-order predicate logic as its basis, while here we want to reason about functions (the solutions of ODEs). We overcome this seeming mismatch by simply handling those functions as first-order objects[1].

The signature of our theory contains the sort symbols $\mathcal{R}$ and $(\mathcal{F}_k)_{k\in\kappa}$ for a finite index set $\kappa$. Intuitively, the sort $\mathcal{R}$ corresponds to real numbers, and each sort symbol

[1]While this is new in the context of SAT modulo ODE, this is quite common in mathematics. For example, Zermelo-Fraenkel set theory uses such an approach to define sets, relations, etc. within *first-order* predicate logic.

$\mathcal{F}_k$, $k \in \kappa$ to real functions, with the argument modeling time over a certain time interval. In the illustrative example, the variables $x_1$, $v_1$ belong to the same sort (e.g., $\mathcal{F}_1$), the variables $x_2$, $v_2$ to another one (e.g., $\mathcal{F}_2$), and so on.

The allowed predicate symbols are $\{=, \geq\}$ and the function symbols include $\{0, 1, +, -, \cdot, \exp, \log, \sin, \cos, \tan\}$, all of the usual arity. All of those predicate and function symbols are defined on all sorts (i.e., not only on $\mathcal{R}$, but also on $(\mathcal{F}_k)_{k\in\kappa}$). Still, we always require all arguments and results to be from the same sort.

We will have additional function symbols that we will also call *functional operators*: The function symbols $\mathit{init}_k : \mathcal{F}_k \to \mathcal{R}$ and $\mathit{final}_k : \mathcal{F}_k \to \mathcal{R}$, $k \in \kappa$ model the initial and final value of the argument function. The function symbol $\mathit{diff}_k : \mathcal{F}_k \to \mathcal{F}_k$, $k \in \kappa$, models differentiation, and hence we will usually write $\mathit{diff}(z)$ as $\dot{z}$. We also assume the function symbols $\mathit{embed}_k : \mathcal{R} \to \mathcal{F}_k$, $k \in \kappa$, that convert real numbers to functions. However, we will not write the function symbols $\mathit{embed}_k, k \in \kappa$ explicitly, but implicitly assume them whenever an argument from $\mathcal{F}_k$ is expected and an argument from $\mathcal{R}$ present. In the example, this is the case in the differential equation $\dot{v}_2 = -g$ which would actually read $\dot{v}_2 = -\mathit{embed}_2(g)$, or $\dot{v}_2 = \mathit{embed}_2(-g)$. For all functional operators we will not write the index, if clear from the context.

Since we do not allow quantifiers, we will not work with a separate set of variables, but simply call 0-ary predicate symbols *Boolean variables*, 0-ary function symbols from $\mathcal{R}$ *numerical variables*, and 0-ary function symbols from $\mathcal{F}_k, k \in \kappa$, *k-function variables* and often just *function variables*. We denote the set of Boolean variables by $\mathcal{V}_\mathcal{B}$, and for every sort $\mathcal{S}$, we denote the corresponding set of variables by $\mathcal{V}_\mathcal{S}$. We also define the set of all such variables $\mathcal{V} := \mathcal{V}_\mathcal{B} \cup \mathcal{V}_\mathcal{R} \cup \bigcup_{k\in\kappa} \mathcal{V}_{\mathcal{F}_k}$.

**Definition 1** *An* atomic formula *is either a Boolean variable or an atomic theory formula. An* atomic theory formula *is of one of the three following kinds:*

- *An atomic* real-valued formula *is a formula of the form $p(\eta_1, \ldots, \eta_n)$ where $p$ is an $n$-ary predicate symbol from $\mathcal{R}$ and $\eta_1, \ldots, \eta_n$ are terms built in the usual way using function symbols from $\mathcal{R}$ and the functional operators init and final, whose argument is allowed to be a function variable.*

- *An atomic $k$-*differential formula *is a differential equation of the form $\dot{z} = \eta$ where $z$ is a function variable from $\mathcal{F}_k$, and $\eta$ is a term of type $\mathcal{F}_k$ not containing any functional operator except for $\mathit{embed}_k$.*

- *An atomic $k$-*function formula *is a formula of the form $p(\eta_1, \ldots, \eta_n)$, where $p$ is an $n$-ary predicate symbol from $\mathcal{F}_k$ and $\eta_1, \ldots, \eta_n$ are terms built in the usual way using function symbols from $\mathcal{F}_k$, and not containing any functional operators except for $\mathit{embed}_k$.*

*A* literal *is either an atomic formula or the negation of an atomic formula. A* formula *is an arbitrary Boolean combination of literals. A* theory formula *is a formula without Boolean variables.*

For example, $g = 9.81$ and $\mathit{init}(v_2) = -K\mathit{final}(v_1)$ are examples of atomic real-valued formulas, $\dot{x}_1 = v_1$ is an example of an atomic differential formula, and $x_1 \geq 0$ is an example of an atomic function formula.

The resulting formulas have the usual mathematical semantics where we interpret the sort $\mathcal{R}$ over the real numbers $\mathbb{R}$ and $\mathcal{F}_k, k \in \kappa$ over smooth functions in $[0, \tau_k] \to \mathbb{R}$, where $\tau_k \in \mathbb{R}^{\geq 0}$. Hence the length $\tau_k$ will be the same for all elements belonging to the same sort $\mathcal{F}_k$. These functions will usually arise as solutions of differential equations, hence the domain $[0, \tau_k]$ usually models time.

We interpret all symbols in $\mathcal{R}$ according to their usual meaning over the real numbers. To extend this to arithmetical predicate and function symbols with function arguments, that is, with arguments from $\mathcal{F}_k$, we simply lift their meaning over the reals to the whole domain $[0, \tau_k]$ of our functions in $[0, \tau_k] \to \mathbb{R}$. For example, the constant 1 in $\mathcal{F}_k$ is the function that assigns to each element of $[0, \tau_k]$ the constant 1. The atomic function formula $z \geq 1$ expresses the fact that the $k$-function variable $z$ is greater or equal than the constant function 1 at every element of $[0, \tau_k]$. In general, for a function symbol $f$ of type $\mathcal{F}_k \times \cdots \times \mathcal{F}_k \to \mathcal{F}_k$, its interpretation $f_{\mathcal{F}_k}$ is such that for $z_1, \ldots, z_n : [0, \tau_k]$, for all $t \in [0, \tau_k]$, $f_{\mathcal{F}_k}(z_1, \ldots, z_n)(t) = f_{\mathcal{R}}(z_1(t), \ldots, z_n(t))$, where $f_{\mathcal{R}}$ is the interpretation of the corresponding function symbol $f$ of type $\mathcal{R} \times \cdots \times \mathcal{R} \to \mathcal{R}$. For a predicate symbol $p$ of type $\mathcal{F}_k \times \cdots \times \mathcal{F}_k$, its interpretation $p_{\mathcal{F}_k}$ is such that $p_{\mathcal{F}_k}(z_1, \ldots, z_n)$ iff for all $t \in [0, \tau_k]$, $p_{\mathcal{R}}(z_1(t), \ldots, z_n(t))$, where again $p_{\mathcal{R}}$ is the interpretation of the corresponding predicate symbol $p$ of type $\mathcal{R} \times \cdots \times \mathcal{R}$.

Note that, as a result, $\neg z \geq 1$ is *not* equivalent to $z < 1$: The former means that not all the time $z$ is greater or equal one, whereas the latter means that all the time $z$ is less than one. Due to this, we will also call such atomic function formulas *invariants*.

Finally, we interpret the function operators as follows: The interpretation of $\mathit{init}_k$ takes a function $z : [0, \tau_k] \to \mathbb{R}$ and returns $z(0)$, whereas the interpretation of $\mathit{final}_k$ returns $z(\tau_k)$. The interpretation of $\mathit{embed}_k$ takes a real number $x$, and returns the constant function that takes the value $x$ on its whole domain $[0, \tau_k]$. Finally, we interpret $\mathit{diff}_k$ as the usual differential operator from mathematical analysis.

We call a function that assigns values of corresponding type to all elements of $\mathcal{V}$, and the above meaning to all other function and predicate symbols, an $\mathrm{ODE}_{\mathbb{R}}$-*interpretation*. Based on this, we get the usual semantical notions from predicate logic. The main problem is to check, for a given formula, whether it is satisfiable by an $\mathrm{ODE}_{\mathbb{R}}$-interpretation.

# 4 Formalization: SAT Modulo ODE Simulations

In this section, we will introduce alternative semantics to formulas based on floating point arithmetic. Since there are various variants of floating point arithmetic (e.g., 32 and 64 bit IEEE 754 arithmetic), including different formalizations [6, 14], and moreover, a plethora of methods for solving differential equations [12], the resulting semantics will be parametric in the used variant of floating point arithmetic and

ODE solver.

Now we interpret the $\mathcal{R}$-variables over the floating point numbers $\mathbb{F}$, and the $\mathcal{F}_k$-variables over functions from $\{t\Delta \mid t \in \{0, \ldots, \frac{\tau_k}{\Delta}\}\} \to \mathbb{F}$ (*trajectories*), for a given $k \in \kappa$. Here, we require $\tau_k$ to be a multiple of the step size $\Delta$. We interpret all function and predicate symbols—including the functional operators—in the obvious floating point analogue to the formalization from the previous section, with the usual rounding to the nearest floating point number. Especially, we interpret function symbols on $\mathcal{F}_k$, $k \in \kappa$ point-wise on the elements of $\{t\Delta \mid t \in \{0, \ldots, \frac{\tau_k}{\Delta}\}\}$. However, when lifting predicates to type $\mathcal{F}_k$, we only require the lifted predicate to hold for $t \in \{0, \ldots, \frac{\tau_k}{\Delta} - 1\}$, that is, without the final point. For explaining why we refer to the illustrative example. If $\neg up$, it uses an invariant $x \geq 0$. At the same time it allows switching to $up$ if and only if $x \leq 0$. When interpreting $x$ as a continuous function, this makes perfect sense: the switch occurs exactly when both $x \geq 0$ and $x \leq 0$, that is, when $x = 0$. This does not work in our approximate interpretation because it is highly unlikely that, after discretization, a point is reached for which precisely $x = 0$. To circumvent this problem, we allow the invariant to be violated at the very final point of $x$ which at the same time is the first point that allows switching.

To concentrate on our main point, we will ignore special floating point values modeling overflow and similar intricacies of floating point arithmetic. Still, our approach is compatible with such values, since we do not require that every floating point number have a corresponding real number.

Before turning to differential equations, we first describe how they are usually solved in practice [12]. The input to such a solver is a *system* of differential equations which, in our terminology, is a conjunction of $n$ atomic $k$-differential formulas in $n$ variables. Solvers then compute a solution for the *whole system*, using discrete steps in time. For example, writing the system of differential equations as $\dot{\boldsymbol{z}} = \boldsymbol{F}(\boldsymbol{z})$, where boldface indicates vectors, Euler's method—the most widely known explicit solution method for ODEs—uses the rule

$$\boldsymbol{z}(t + \Delta) = \boldsymbol{z}(t) + \boldsymbol{F}(\boldsymbol{z}(t))\Delta.$$

As a result, the solution satisfies this equality at each point in time.

Since in our case, differential equations do not directly occur in systems, but in individual atomic formulas, we separate this rule into conditions for the individual formulas, instead of conditions for the individual time steps. In the case of Euler's method, denoting the individual $\mathcal{F}_k$-variables by $z_1, \ldots, z_n$, for a differential formula $\dot{z}_i = f(z_1, \ldots, z_n)$, the resulting condition is

$$\forall t \in \{0, \ldots, \frac{\tau_k}{\Delta} - 1\} \,.\, z_i((t+1)\Delta) = z_i(t\Delta) + f(z_1(t\Delta), \ldots, z_n(t\Delta))\Delta.$$

In general, the rules used by explicit solvers are based on an equality with left-hand side $\boldsymbol{z}(t + \Delta)$ which allows an analogical natural separation into conditions on the individual components of the solution.

We call a function that assigns Boolean values to the elements of $\mathcal{V}_\mathcal{B}$, floating point numbers to the elements of $\mathcal{V}_\mathcal{R}$, trajectories to the elements of $\mathcal{V}_{\mathcal{F}_k}, k \in \kappa$ and the

above meaning to all other function and predicate symbols, an $\text{ODE}_{\mathbb{F}}$-*interpretation*. This again defines all the usual semantical notions from predicate logic using the same notation as in the previous section. For any interpretation $\mathcal{I}$ we denote by $\mathcal{I}(\eta)$ the value of the term $\eta$ in $\mathcal{I}$, by $\mathcal{I} \models \phi$ the satisfiability of $\phi$ in $\mathcal{I}$, and so on. For the rest of the paper we assume a floating point interpretation $\mathcal{I}_{\mathbb{F}}$ for $\mathcal{V} = \emptyset$ that we will extend with values for a non-empty set $\mathcal{V}$.

In the rest of the paper, we design and analyze tools for checking whether a given formula is satisfiable by an $\text{ODE}_{\mathbb{F}}$-interpretation, in which case we will also simply say that it is satisfiable.

# 5 Theory Solver

The common SMT approaches use separate solvers for handling the Boolean part and the specific logical theory, respectively. In this section we concentrate on the latter. So, for a given theory formula $\phi$ (i.e., formula without Boolean variables), we want to check whether $\phi$ is satisfiable by an $\text{ODE}_{\mathbb{F}}$-interpretation.

As an example consider the formula

$$g = 9.81 \wedge \mathit{init}(v) = 10 \wedge v \geq 0 \wedge \mathit{final}(v) \leq 0 \wedge \dot{v} = -g - \frac{v}{100}$$

which is satisfiable by an interpretation that assigns to $g$ the value 9.81 and to $v$ a trajectory that starts with the value 10, then decreases according to the given differential equation, and stays non-negative, except for the very last step which is non-positive.

We first prove that unlike $\text{ODE}_{\mathbb{R}}$, in our case there is no fundamental theoretical hurdle caused by undecidability.

**Theorem 1** $\text{ODE}_{\mathbb{F}}$-*satisfiability is algorithmically decidable.*

**Proof.** Assume a theory formula $\phi$. W.l.o.g. we assume $\phi$ to be a conjunction. Let $|\mathbb{F}|$ be the cardinality of the set of floating point numbers. The key observation is that $|\mathbb{F}|$ is finite. The problem is that $\text{ODE}_{\mathbb{F}}$-interpretations satisfying $\phi$ may assign trajectories of arbitrary length to function variables. However, if there is an interpretation that satisfies $\phi$ then there is also an interpretation satisfying $\phi$ that has length smaller than $|\mathbb{F}|^{\max\{\frac{\tau_k}{\Delta} \mid k \in \kappa\}}$: Assume an interpretation $\mathcal{I}$ satisfying $\phi$ that for some $k \in \kappa$ assigns trajectories longer than this bound to the variables in $\mathcal{F}_k$. Due to the finite cardinality of $|\mathbb{F}|$, there must be $t$, $t'$ s.t. $t \neq t'$ and for every $z \in \mathcal{V}_k$, $\mathcal{I}(z)(t) = \mathcal{I}(z)(t')$. This means that the interpretation that coincides with $\mathcal{I}$, but for every $z \in \mathcal{V}_k$, the section between $t+1$ and $t'$ is removed, also satisfies $\phi$. We can repeat this process until the interpretation satisfying $\phi$ is short enough.

Due to this we can check the satisfiability of $\phi$ using brute force search, checking whether the finite set of interpretations assigning trajectories of length smaller than $|\mathbb{F}|^{\max\{\frac{\tau_k}{\Delta} \mid k \in \kappa\}}$ contains an element that satisfies $\phi$. ■

This proof is based on the fact that the set of floating point numbers has finitely many elements. However, due to the sheer number of those elements, the algorithm used in the proof is far from practically useful.Possibly with the exception of minifloats—floating point numbers using a small number of bits for their representation

In the rest of the section we will design a solver that is able to solve theory formulas that arise from the specific structure identified in Section 2 much more efficiently. This structure will allow each step of the solver to assign a value to a variable. Since we will also want to compute initial values of function variables, we introduce a set $\mathcal{V}_{init} := \{init(z) \mid z \in \mathcal{V}_{\mathcal{F}_k}, k \in \kappa\}$ each ranging over the floating point numbers $\mathbb{F}$. We can represent the computed values as follows:

**Definition 2** *A* state $\sigma$ *is a function that assigns to each element of* $\mathcal{V}_{\mathcal{R}} \cup \mathcal{V}_{\mathcal{F}} \cup \mathcal{V}_{init}$ *either an object of the corresponding type or the special value undef.*

For a state $\sigma$, we denote the extension of $\mathcal{I}_{\mathbb{F}}$ with the values defined by $\sigma$ on $\mathcal{V}_{\mathcal{R}} \cup \mathcal{V}_{\mathcal{F}}$ (ignoring the values for $\mathcal{V}_{init}$) by $\mathcal{I}_\sigma$. Our theory solver will use inference rules to fill the state $\sigma$ with values until those values allow us to evaluate the given formula. For this we define for a term $\eta$, $ev_\sigma(\eta)$ to be *undef*, if the term $\eta$ contains a variable $v$ for which $\sigma(v) = undef$, and the result of term evaluation $\mathcal{I}_\sigma(\eta)$, otherwise. For example, for a state $\sigma = \{K \mapsto 0.5, v_1 \mapsto undef\}$, $ev_\sigma(-Kfinal(v_1)) = undef$, but for $\sigma = \{K \mapsto 0.5, v_1 \mapsto \rho\}$, where $\rho$ is a trajectory whose final value is 2.0, $ev_\sigma(-Kfinal(v_1)) = -1.0$.

In a similar way, for an atomic formula $A$ we define $ev_\sigma(A) = undef$, if $\sigma$ assigns *undef* to a variable in $A$, and otherwise $ev_\sigma(A) = \top$, if $\mathcal{I}_\sigma \models A$, and $ev_\sigma(A) = \bot$, if $\mathcal{I}_\sigma \not\models A$. For example, for the state $\sigma$ just mentioned, $ev_\sigma(-Kfinal(v_1) \geq 0) = \bot$.

Based on the usual extension of the Boolean operators $\neg, \wedge, \vee$ to three values, in our case $\{\bot, undef, \top\}$, this straightforwardly extends to formulas, in general. For example, $ev_{\{x \mapsto 0, y \mapsto undef\}}(x = 1 \wedge y = 1) = ev_{\{x \mapsto 0, y \mapsto undef\}}(x = 1) \wedge ev_{\{x \mapsto 0, y \mapsto undef\}}(y = 1) = \bot \wedge undef = \bot$. This implies that whatever value $y$ has, the formula will not be satisfiable.

**Theorem 2** $\phi$ *is* $\mathrm{ODE}_{\mathbb{F}}$*-satisfiable iff there is a state* $\sigma$ *with* $ev_\sigma(\phi) = \top$.

**Proof.** If $ev_\sigma(\phi) = \top$ then $\mathcal{I}_\sigma \models \phi$ and hence $\phi$ is satisfiable. In the other direction, if $\phi$ is satisfiable by an $\mathrm{ODE}_{\mathbb{F}}$-interpretation $\mathcal{I}$, then for the state $\sigma$ with $\sigma(x) = \mathcal{I}(x)$, for $x \in \mathcal{V}_{\mathcal{R}} \cup \mathcal{V}_{\mathcal{F}}$, and $\sigma(init(z)) = \mathcal{I}(z)(0)$, for every $z \in \mathcal{V}_{\mathcal{F}_k}, k \in \kappa$, $ev_\sigma(\phi) = \top$. ■

The question is, how to find such a state efficiently, if $\phi$ is satisfiable, and how to decide that it does not exist in the case where it is unsatisfiable. We will assume the input formula to be a conjunction of literals, since disjunctions will be handled by the Boolean solver (see Section 7). By misuse of notation we will also view $\phi$ as the set of its literals.

As already discussed, we will use inference rules on states. The first rule uses the fact that both sides of an equality have to evaluate to the same value in $\sigma$:

**Definition 3** *For two states $\sigma$ and $\sigma'$, $\sigma \to_{\mathcal{R}} \sigma'$ iff*

- *there is a literal of the form $x = \eta$ or $\eta = x$ in $\phi$ with $x \in \mathcal{V}_{\mathcal{R}} \cup \mathcal{V}_{init}$,*
- *$\sigma(x) = undef$,*
- *$ev_\sigma(\eta) \neq undef$, and*
- *$\sigma'$ is s.t. for all $v \in \mathcal{V}_{\mathcal{R}} \cup \mathcal{V}_{\mathcal{F}} \cup \mathcal{V}_{init}$, $\sigma'(v) = \begin{cases} ev_\sigma(\eta), & \text{if } v = x, \text{ and} \\ \sigma(v), & \text{otherwise.} \end{cases}$*

For example, if $\sigma = \{K \mapsto 0.5, v_1 \mapsto \rho, init(v_2) \mapsto undef\}$, with $\rho$ again a trajectory with final value 2.0, and the input formula contains the literal $init(v_2) = -K final(v_1)$, then $\sigma \to_{\mathcal{R}} \sigma'$, where $\sigma' = \{K \mapsto 0.5, v_1 \mapsto \rho, init(v_2) \mapsto -1.0\}$.

The second inference solves differential equations. For a state $\sigma$ and $k \in \kappa$ we define $IVP_\phi(\sigma, k)$ (for "initial value problem") as the formula

$$\bigwedge_{\dot{z}=\eta \,\in\, \phi, z \in \mathcal{V}_{\mathcal{F}_k}} \dot{z} = \eta \wedge \bigwedge_{z \in \mathcal{V}_{\mathcal{F}_k}} init(z) = \sigma(init(z)) \wedge \bigwedge_{x \in \mathcal{V}_{\mathcal{R}}, \sigma(x) \neq undef} x = \sigma(x).$$

Here we assume for all $z \in \mathcal{F}_k$, $\sigma(init(z)) \neq undef$, and for all $x \in \mathcal{V}_{\mathcal{R}}$ occurring in a $k$-differential equation in $\phi$, $\sigma(x) \neq undef$. Then one can find an assignment to the variables in $\mathcal{V}_{\mathcal{F}_k}$ satisfying the formula $IVP_\phi(\sigma, k)$ using a numerical ODE solver whose method corresponds to the one used for defining formula semantics. This assignment is unique up to the length of the assigned trajectories. In practice, the solver might fail, e.g. due to floating point overflows, but we ignore this complication for simplicity of exposition.

**Definition 4** *For two states $\sigma$ and $\sigma'$, $k \in \kappa$ and $t \in \mathbb{N}_{\geq 0}$, $\sigma \to_{\mathcal{F}_k, t} \sigma'$ iff*

- *for every variable $z \in \mathcal{V}_{\mathcal{F}_k}$, $\sigma(z) = undef$,*
- *for every variable $z \in \mathcal{V}_{\mathcal{F}_k}$, $\sigma(init(z)) \neq undef$,*
- *for every variable $x \in \mathcal{V}_{\mathcal{R}}$ occurring in a $k$-differential equation in $\phi$, $\sigma(x) \neq undef$*
- *for every variable $z \in \mathcal{V}_{\mathcal{F}_k}$, $\phi$ contains exactly one literal of the form $\dot{z} = \eta$*
- *$\sigma'$ is identical to $\sigma$ except that it assigns to the variables $z \in \mathcal{V}_{\mathcal{F}_k}$ the corresponding trajectories of length $t$ satisfying $IVP_\phi(\sigma, k)$.*

It would not be difficult to also handle the case when $\phi$ contains more than one differential literal with the same left-hand side. But usually this is not practically useful, and hence the rules does not consider this case.

Now we can apply several inference steps in a row, starting from the everywhere undefined state $\sigma_{undef}$. This always terminates, since every inference step creates a state with less undefined elements and $\mathcal{V}_{\mathcal{R}} \cup \mathcal{V}_F \cup \mathcal{V}_{init}$ is finite.

Remember that our goal is to use the inference rules to arrive at a state $\sigma$, for which $ev_\sigma(\phi) = \top$ or to decide that no such state exists. Since the inferences do not introduce new undefined values, it does not make sense to do further inferences on a state $\sigma$, for which $ev_\sigma(\phi) \neq \mathit{undef}$.

If we arrive at a state $\sigma$ for which $ev_\sigma(\phi) = \bot$, can we conclude that $\phi$ is unsatisfiable? Certainly not: A different sequence of inferences might have found a state that evaluates to $\top$, showing satisfiability. For example, for the formula $\mathit{init}(z) = 0 \wedge \dot{z} = 1 \wedge \mathit{final}(z) \geq 10$, one inference step from $\sigma_{\mathit{undef}}$ wrt. $\rightarrow_{\mathcal{R}}$ results in the state $\{\mathit{init}(z) \mapsto 0, z \mapsto \mathit{undef}\}$, but from this state, an inference wrt. $\rightarrow_{\mathcal{F},t}$ only results in a state that evaluates to $\top$, if $t$ is big enough for the final state of the assigned trajectory to be larger of equal 10.

Still, for given fixed lengths $t$ of inferences $\rightarrow_{\mathcal{F},t}$, the order of inferences does not matter.

**Theorem 3** *Let $\lambda : \kappa \to \mathbb{N}_{\geq 0}$. Let $\sigma_1$ and $\sigma_2$ be the final states of two sequences of inferences using $\rightarrow_{\mathcal{R}}$ and $\rightarrow_{\mathcal{F}_k,\lambda(k)}, k \in \kappa$ s.t. neither from $\sigma_1$ nor from $\sigma_2$, further inferences are possible. Then $ev_{\sigma_1}(\phi) = ev_{\sigma_2}(\phi)$.*

So it may be necessary to try different trajectory lengths for arriving at a state that evaluates to $\top$, showing satisfiability. We represent this search using a tree:

**Definition 5** *An* inference tree *is a tree whose vertices are formed by states, where the root is $\sigma_{\mathit{undef}}$, and every vertex state $\sigma$ that is no leaf either has*

- *precisely one successor vertex $\sigma'$ with $\sigma \rightarrow_{\mathcal{R}} \sigma'$ or*
- *successor vertices $\sigma'_0, \sigma'_1, \ldots, \sigma'_n$ s.t. for an arbitrary, but fixed $k \in \kappa$*
  - *for every $i \in \{0, \ldots, n\}$, $\sigma \rightarrow_{\mathcal{F}_k,i} \sigma'_i$, , and*
  - *there is an $m < n$ s.t. for all $z \in \mathcal{V}_{\mathcal{F}_k}$, $\sigma'_n(z)(m) = \sigma'_n(z)(n)$.*

For example, for the formula $\mathit{init}(v) = 10 \wedge v \geq 0 \wedge \mathit{final}(v) \leq 0 \wedge \dot{v} = -9.81 - \frac{v}{100}$, and the tree root $\sigma_{\mathit{undef}}$, the inference rule $\rightarrow_{\mathcal{R}}$, results in the successor state $\sigma = \{\mathit{init}(v) \mapsto 10, v \mapsto \mathit{undef}\}$. Now we use the second inference rule which branches the tree. For the successor state wrt. $\rightarrow_{\mathcal{F}_k,0}$, the final state of the trajectory computed for $v$ is equal to its initial state 10, which violates the condition $\mathit{final}(v) \leq 0$, and hence this successor state evaluates to $\bot$. The successor states wrt. $\rightarrow_{\mathcal{F}_k,1}$, $\rightarrow_{\mathcal{F}_k,2}$, ... will have longer trajectories with the respective final states getting smaller and smaller, until—at the point when the trajectory has become long enough—the final state will finally satisfy $\mathit{final}(v) \leq 0$. The resulting successor state then evaluates to $\top$ which shows satisfiability of the formula.

Even if we would not find a state that evaluates to $\top$, we would not have to search infinitely many successors, since an inference tree includes only a finite subset of the infinite set of possible inferences wrt. $\rightarrow_{\mathcal{F}_k,t}$, $t \in \mathbb{N}_0$. The termination condition $\sigma'_n(z)(m) = \sigma'_n(z)(n)$ must be satisfied for some $m$, again due to the finite cardinality of the set of floating point numbers. Still, inference trees cover the search space completely, allowing us to conclude the input to be unsatisfiable in some cases:

**Theorem 4** *Assume an inference tree such that for all leaves $\sigma$, $ev_\sigma(\phi) = \bot$. Then $\phi$ is unsatisfiable.*

The theorem follows from two facts: First, for states $\sigma$ and $\sigma'_1, \sigma'_1, \ldots$ s.t. $\sigma \to_{\mathcal{F}_k,1} \sigma'_1, \sigma \to_{\mathcal{F}_k,2} \sigma'_2, \ldots$, and $\phi$ is satisfiable by an $ODE_{\mathbb{F}}$-interpretation that coincides with a state $\sigma$ on its defined elements, there is an $i \in \mathbb{N}_0$ s.t. $\phi$ is satisfiable by $\sigma'_i$. Second, due to the same reasoning as used in the proof of Theorem 1, if this is the case for an arbitrary $i \in \mathbb{N}_0$ then this is also the case for an $i < n$, since $\sigma'_n(z)(m) = \sigma'(z)(n)$.

Now an algorithm can simply check all leaves of such an inference tree to check satisfiability. There is various possibilities to do so, for example using simple recursive depth-first search:

$ODESAT(\sigma)$
let $\Sigma$ be the set of successor vertices of $\sigma$ in an inference tree.
**if** $\Sigma = \emptyset$ **then return** $ev_\sigma(\phi)$
**else if** there is a $\sigma' \in \Sigma$ s.t. $ODESAT(\sigma') = \top$ then **return** $\top$
**else if** for all $\sigma' \in \Sigma$ s.t. $ODESAT(\sigma') = \bot$ then **return** $\bot$
**else return** *undef*

The initial call should be $ODESAT(\sigma_{undef})$, and the result $\top$ can be interpreted as `SAT`, the result *undef* as `UNKNOWN`, and the result $\bot$ as `UNSAT`.

We call any such algorithm that returns its result based on application of Theorems 4 and 2 to the leaves of an inference tree an *evaluation based $ODE_{\mathbb{F}}$-solver*. Here we may encounter two problems:

- The tree might have some leaves $\sigma$ with $ev_\sigma(\phi) = undef$, resulting in the answer `UNKNOWN`.
- The tree may be huge, resulting in a long run-time of the algorithm.

The first problem may happen, for example, if a numerical variable is constrained by an equation such as $x^2 = 1$ that cannot be solved by our rules (of course, such an equation can be easily solved, for example by methods for solving polynomial equations, but here we are interested in getting as far as possible without such techniques). In such a case, one can fall back to brute-force search of Theorem 1. Of course, this is inefficient, and we want to avoid it, which leads us back to the second problem, the problem of a large search tree.

One possibility is to simply give up on completeness by not exploring the full search space. For example, as usual for SAT modulo ODE solvers [10, 11], we might only search for trajectories up to a certain length, that is, use the strategy of bounded model checking. In this case, the solver will not decide satisfiability, but satisfiability by trajectories up to a certain length.

Also, in some cases, we might have good heuristics available. Especially, for a given inference tree, finding a vertex $\sigma$ with $ev_\sigma(\phi) = \top$ is a tree search problem

which enables the usage of well-known tree search algorithms [9], allowing us to efficiently find a leaf that shows satisfiability of the input.

In any case, we will see in the next section, that for formulas having a structure similar to the toy example, these problems can be avoided.

# 6 Formula Structure

In the previous section we identified a search tree that allows us to replace brute-force search by inferences. In this section we show how to use syntactical restrictions on the input formula to

- ensure that the inference tree does not end in undefined leaves, and to
- restrict the size of the inference tree.

First we will show how to avoid undefined leaves by ensuring that the value of every non-Boolean variable be deducible from the value of another variable without the need for circular reasoning.

**Definition 6** *A formula $\phi$ is* orientable *iff there is a total order $r_1, \ldots, r_{|\mathcal{V}_\mathcal{R} \cup \mathcal{V}_\mathcal{F}|}$ on the variables in $\mathcal{V}_\mathcal{R} \cup \mathcal{V}_\mathcal{F}$ s.t. for every $i \in \{1, \ldots, |\mathcal{V}_\mathcal{R} \cup \mathcal{V}_\mathcal{F}|\}$,*

- *if $r_i \in \mathcal{V}_\mathcal{R}$, then there is a literal in $\phi$ that is of the form $r_i = \eta$ or $\eta = r_i$, where $\eta$ does not contain any variable from $r_i, \ldots, r_{|\mathcal{V}_\mathcal{R} \cup \mathcal{V}_\mathcal{F}|}$, and*
- *if $r_i \in \mathcal{V}_\mathcal{F}$, then*
  - *there is a literal in $\phi$ that is of the form $init(r_i) = \eta$ or $\eta = init(r_i)$, where $\eta$ does not contain any variable from $r_i, \ldots, r_{|\mathcal{V}_\mathcal{R} \cup \mathcal{V}_\mathcal{F}|}$ and*
  - *there is exactly one literal in $\phi$ that has the form $\dot{r}_i = \eta$, and the term $\eta$ does not contain any variable that is both in $\mathcal{V}_\mathcal{R}$ and in $r_i, \ldots, r_{|\mathcal{V}_\mathcal{R} \cup \mathcal{V}_\mathcal{F}|}$.*

Note that the non-circularity condition for atomic differential formulas only includes variables in $\mathcal{V}_\mathcal{R}$ but not variables in $\mathcal{V}_\mathcal{F}$, allowing the formulation of systems of ordinary differential equations. For illustration of Definition 6, consider the formula $x = \sin y \wedge y = 2x \wedge \dot{z}_1 = z_1 + z_2 \wedge \dot{z}_2 = z_1 - z_2 \wedge init(z_1) = 10 \wedge init(z_2) = 10$ that is not orientable, but $\Psi \wedge x = 0$, where $\Psi$ is the previous formula, is orientable using the order $x, y, z_1, z_2$.

**Theorem 5** *Assume a formula $\phi$ that is orientable, and a corresponding inference tree $\Omega$ whose leaves do not allow further inferences. Then for every state $\sigma$ at a leaf of $\Omega$, for every variable $v \in \mathcal{V}_\mathcal{R} \cup \mathcal{V}_\mathcal{F}$, $\sigma(v) \neq undef$.*

**Proof.** Consider the order Definition 6 ensures on the variables in $\mathcal{V}_\mathcal{R} \cup \mathcal{V}_\mathcal{F}$, assume a leaf $\sigma$ of $\Omega$, and assume that there is a variable $v \in \mathcal{V}_\mathcal{R} \cup \mathcal{V}_\mathcal{F}$ s.t. $\sigma(v) = undef$.

W.l.o.g. let $v$ be the smallest such variable wrt. to the order given by Definition 6. Then there is a state $\sigma'$ with $\sigma \to \sigma'$ which is a contradiction to $\sigma$ being a leaf. ■

Hence, in such a case, we can decide satisfiability of the input formula using inferences alone, never returning `UNKNOWN`.

**Theorem 6** *Every* evaluation based $ODE_{\mathbb{F}}$-solver *that is based on an inference tree whose leaves do not allow further inferences, is a decision procedure for all orientable $ODE_{\mathbb{F}}$-formulas.*

Still, the inference tree may be huge, since we might have to search for extremely long trajectories. To avoid this, we analyze the example from Section 2 once more. Here, the invariant $x_1 \geq 0$ ensures, that the ball will eventually stop falling. In a similar way, the invariant $v_1 \geq 0$ detects that the ball stops rising. As soon as those two invariants stop to hold, we do not have to search for longer trajectories. The following theorem generalizes this.

**Theorem 7** *If $\phi$ contains an atomic $k$-functional formula $A$ with $ev_\sigma(A) = \bot$ then for every state $\sigma'$ s.t. for every function variable $z$, $\sigma'(z)$ is at least as long as $\sigma(z)$, and $\sigma'$ is equal to $\sigma$ for all elements defined in $\sigma$, including function variables up to their length, $ev_{\sigma'}(A) = \bot$.*

Hence, as soon as the application of the inference rule $\sigma \to_{\mathcal{F}_k,t} \sigma'$ results in a state $\sigma'$ such that $ev_{\sigma'}(A) = \bot$, we do not have to expand further successors of $\sigma$ using the same rule with bigger $t$.

# 7 Solver Integration

Now we also allow disjunctions and Boolean variables in the input formula $\phi$. We follow the common architecture [18] of SMT solvers where a SAT solver handles the Boolean structure, a theory solver (in our case the one from Section 5) handles conjunctions of non-Boolean literals, and the integrating SMT solver handles communication between the two.

We include the case where either the Boolean solver or the theory solver is incomplete, in which the combination may return `UNKNOWN`. Even though many SMT schemes expect the underlying theory solver to be complete, one can usually use an incomplete solver, as well, by letting the theory solver return `UNSAT` in the place of `UNKNOWN`. If this happens during execution of the SMT solver, it should return `UNKNOWN` when it would otherwise have returned `UNSAT`. In any case, if the input is identified to be satisfiable, the solver still can reliably return `SAT` as the final result.

The usual SMT solvers proceed by replacing all atomic formulas from the theory by elements of a set of fresh Boolean variables $\mathcal{V}_A$, and finding a Boolean assignment $\alpha : \mathcal{V}_\mathcal{B} \cup \mathcal{V}_A \to \{\bot, \top\}$ satisfying the resulting purely Boolean formula. Denoting by

$\gamma(A)$ the atomic formula in $\phi$ corresponding to the variable $A \in \mathcal{V}_A$ one can then use the theory solver to check whether the formula

$$\Sigma(\alpha) := \bigwedge_{A \in \mathcal{V}_A, \alpha(A) = \top} \gamma(A) \wedge \bigwedge_{A \in \mathcal{V}_A, \alpha(A) = \bot} \neg\gamma(A)$$

is satisfiable. If this is the case, the original input formula $\phi$ is satisfiable, as well.

If we want to ensure that $\Sigma(\alpha)$ fulfills the syntactical restrictions of Section 6, we have to ensure that for every such Boolean assignment the formula $\Sigma(\alpha)$ fulfills those restrictions. Returning to the illustrative example from Section 2, the order $g, K, x_1, v_1, x_2, v_2, x_3, v_3$ ensures that $\Sigma(\alpha)$ is always orientable. Here, the variables $up_1$, $up_2$ always activate the necessary atomic formulas.

In the case of our theory, in practically reasonable formulas, in a similar way as in our illustrative example, atomic differential and function formulas occur positively, without a negation. If a Boolean assignment satisfies the abstraction of such a formula, then also any Boolean assignment that assigns $\top$ instead of $\bot$ to a Boolean variable in $\mathcal{V}_A$ that corresponds to a differential or function literals. Hence, whenever the SMT solver asks the theory solver to check satisfiability of a formula where such a literal occurs negatively, then the theory solver may ignore those, and check the rest of the formula for satisfiability.

When integrating the theory solver into SMT [18], several levels of integration are possible. Following the classification of Nieuwenhuis et. al. [18], the lowest level is the naive lazy approach. Here a SAT solver finds a satisfiable Boolean assignment $\alpha$ of the Boolean abstraction of the input formula, and the theory solver checks the formula $\Sigma(\alpha)$. If the answer is `SAT`, the input formula is satisfiable. If the answer is `UNSAT`, the solver should identify a sub-formula of $\Sigma(\alpha)$ that is still unsatisfiable. The negation of the abstraction of this formula is formed (a so-called *conflict clause*) and added to the original input formula. Then the whole process is repeated with restarting the SAT solver from scratch. All of this can be easily supported by our theory solver described in Section 5 by recording inference in the usual way into a so-called implication graph.

A further level of integration is to use an incremental theory solver. This means that in the case where the theory solver answers either `SAT` or `UNKNOWN` for some input formula $\phi$, the SMT solver may later ask us to check satisfiability of an extended formula $\phi \wedge \psi$. Later, the SMT solver might ask the theory solver to backtrack to an earlier state. Again, it is no problem for our theory solver to support all of this. The new part $\psi$ of the extended formula may allow additional inferences and the algorithm can simply continue from the state where it finished the analysis of the original formula $\phi$.

Note that here the SMT solver might associate a certain strength with the query [18, Section 4.1], asking for a definitive answer only in the situation when the formula will not further be extended. Here it makes sense to wait with using the $\rightarrow_{\mathcal{F}_k}$-inferences until all $k$-function literals appear in the formula to check since those literals may play an essential role in keeping the inference tree small by applying Theorem 7.

Ideally, the SMT solver supports this by adding such literals always together with the corresponding differential equations.

So-called online SMT solvers do not restart the SAT solver from scratch in further iterations, but only backtrack to an earlier point that did not yet result into an unsatisfiable theory formula.

A further feature of advanced SMT solvers is theory propagation. In this case, the SMT solver not only asks the theory solver to check satisfiability of some formula $\phi$ but, in addition, to also identify elements from a set of literals that are entailed by $\phi$. This is easy to do in the case of inferences that have only one successor state, especially, inferences wrt. $\rightarrow_{\mathcal{R}}$, but will probably not pay off for inferences that require search.

# 8 Computational Experiments

In this section we will study the behavior of a prototype implementation of the techniques introduced in this paper. Especially, we are interested in how far our theoretical finding that the SAT modulo ODE problem is easier for simulation semantics than for classical mathematical semantics (Theorem 1) also holds in practice. Our solver (UN/SOT) is based on the naive lazy approach to SMT, the simplest possible one described in Section 7, using the SAT solver Minisat[2] and our theory solver implementation which currently uses the ODE solver ODEINT[3]. It avoids the full evaluation of its input formula after each inference step: Instead it cycles through all literals in the input formula, handling all $k$-differential literals and $k$-function literals for each $k \in \kappa$ as one block. Whenever the current literal or $k$-block allows an inference, the corresponding inference rule is applied, and whenever the current literal can be evaluated based on the current state, it is evaluated. As soon as a literal evaluates to $\bot$ or all literals evaluate to $\top$, the corresponding result is returned to the SMT solver. If no more inferences are possible, and no result has been found up to this point, the solver returns `UNKNOWN`. This happens only in cases not following the structure identified in Section 6. In the case where the answer is `UNSAT`, the theory solver forms conflict clauses from the sub-formula involved in the inferences necessary to arrive at the answer. We only do simple backtracking, no backjumping.

As a solver with classical mathematical semantics we used dReal[4], that is based on the ODE solver CAPD[5] that builds on decades of research on validated ODE integration [15, 17, 19]. But again, the goal of this section is *not* to measure the efficiency of the used algorithms, but rather, the inherent practical difficulty of the respective problems.

We present experiments based on a hybrid system model of inpatient glycemic control of a patient with type 1 diabetes [7]. The patient is represented by 18

[2] http://minisat.se/
[3] http://www.odeint.com
[4] http://dreal.github.io
[5] http://capd.ii.uj.edu.pl

specific parameters and by initial values of the insulin system (5 function variables) and the glucose system (2 function variables). The whole process is divided into two phases. In the first phase, the patient is being monitored to ensure his or her stability for the surgery. If this fails, the surgery is canceled and the process ends. Otherwise, the second phase (and the surgery) follows, where the controller starts operating—it drives insulin and glucose inputs—wrt. observed condition of the patient. The patient's condition is sampled approximately every 30 minutes (using 1 minute timing jitter). We have two verification tasks, safety—surgery starts, and the glucose level stays in a certain set of safe states, and unsafety—surgery starts, and the set of safe states is left.

For translating the hybrid system to an SMT problem, we unrolled it wrt. its discrete transitions. We did two types of experiments for both solvers, first with fixing a certain initial state, and second with intervals of initial states. For experiments of the second type we equidistantly cover the initial states with a number of sample points and specify the initial state using a disjunction over these sample points. For dReal, we use the original interval. The same applies also for modeling the timing jitter (for both types of experiments), where, in the case of our solver, the equidistance is an input parameter.

The results of the first case, with a fixed initial state, can be seen in the following tables. We examined two different scenarios, where satisfiability amounts to the terminal state being safe and unsafe, respectively. The tables on the top show a variant with an initial state for which the system stays within the safe states, the tables at the bottom a variant where it reaches an unsafe state after the fifth unrollment, and stays there. The column $N$ lists the number of unrollments, $s$ the equidistance of the timing jitter, and the column headed by the tool names the run-time in seconds. The time-ratio should not serve for any efficiency comparison between the two tools but across different test cases.

"Safe" init. state

| | | Verifying safety | | | | | Verifying unsafety | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $N$ | $s$ | Result | UN/SOT | dReal | Ratio | $N$ | $s$ | Result | UN/SOT | dReal | Ratio |
| 3 | 1 | `sat` | 0.15 | 26 | 172 | 3 | 1 | `unsat` | 0.14 | 6 | 44.1 |
| 3 | $\frac{1}{4}$ | `sat` | 0.13 | 26 | 197 | 3 | $\frac{1}{4}$ | `unsat` | 0.33 | 6 | 18 |
| 6 | 1 | `sat` | 0.88 | 50000 | 56804 | 6 | 1 | `unsat` | 4.04 | 52119 | 12911 |
| 6 | $\frac{1}{4}$ | `sat` | 1.51 | 50000 | 33101 | 6 | $\frac{1}{4}$ | `unsat` | 363 | 52119 | 143 |
| 12 | 1 | `sat` | 4.71 | × | × | 8 | 1 | `unsat` | 28 | × | × |
| 12 | $\frac{1}{4}$ | `sat` | 7.12 | × | × | 8 | $\frac{1}{4}$ | `unsat` | 36761 | × | × |

"Unsafe" init. state

| $N$ | $s$ | Result | UN/SOT | dReal | Ratio | $N$ | $s$ | Result | UN/SOT | dReal | Ratio |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 1 | `sat` | 0.18 | 26.2 | 145 | 3 | 1 | `unsat` | 0.14 | 5.8 | 40.8 |
| 3 | $\frac{1}{4}$ | `sat` | 0.11 | 26.2 | 230 | 3 | $\frac{1}{4}$ | `unsat` | 0.31 | 5.8 | 18.6 |
| 6 | 1 | `unsat` | 0.78 | 107980 | 138809 | 6 | 1 | `sat` | 0.65 | 5428 | 8296 |
| 6 | $\frac{1}{4}$ | `unsat` | 8.95 | 107980 | 12064 | 6 | $\frac{1}{4}$ | `sat` | 1.19 | 5428 | 4545 |
| 12 | 1 | `unsat` | 0.87 | × | × | 8 | 1 | `sat` | 1.00 | × | × |
| 12 | $\frac{1}{4}$ | `unsat` | 10.2 | × | × | 8 | $\frac{1}{4}$ | `sat` | 1.07 | × | × |

Here, not unexpectedly, our approach scales quite well against the parameter $N$. This contrasts the behavior of methods based on interval computation, that have to fight with the so-called dependency problem that tends to blow up intervals over long time horizons.

The results of the second case, with intervals of initial points, are shown in the following tables. This time, the tables on the top show a variant with smaller ranges of possible initial states, and the tables at the bottom a variant with larger ones.

Smaller intervals

| | Verifying safety | | | | | | Verifying unsafety | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $N$ | $s$ | Result | UN/SOT | dReal | Ratio | $N$ | $s$ | Result | UN/SOT | dReal | Ratio |
| 3 | 1 | `sat` | 0.2 | 24045 | 122261 | 3 | 1 | `unsat` | 4455 | 6.3 | 0.001 |
| 3 | $\frac{1}{2}$ | `sat` | 5.04 | 24045 | 4770 | 3 | $\frac{1}{2}$ | `unsat` | 7776 | 6.3 | 0.001 |
| 6 | 1 | `sat` | 2.38 | × | × | 4 | 1 | `unsat` | 25042 | 124 | 0.005 |
| 6 | $\frac{1}{2}$ | `sat` | 2.17 | × | × | 4 | $\frac{1}{2}$ | `unsat` | $> 36000$ | 124 | × |
| 12 | 1 | `sat` | 5.83 | × | × | 5 | 1 | `unsat` | $> 36000$ | 2478 | × |
| 12 | $\frac{1}{2}$ | `sat` | 8.28 | × | × | 5 | $\frac{1}{2}$ | `unsat` | × | 2478 | × |

Larger intervals

| $N$ | $s$ | Result | UN/SOT | dReal | Ratio | $N$ | $s$ | Result | UN/SOT | dReal | Ratio |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 1 | `sat` | 29.9 | $> 82800$ | × | 3 | 1 | `unsat` | overflow | 9.3 | × |
| 3 | $\frac{1}{2}$ | `sat` | 28.2 | $> 82800$ | × | 3 | $\frac{1}{2}$ | `unsat` | × | 9.3 | × |
| 6 | 1 | `sat` | 1.04 | × | × | 4 | 1 | ? | $> 36000$ | $> 54000$ | × |
| 6 | $\frac{1}{2}$ | `sat` | 70.6 | × | × | 4 | $\frac{1}{2}$ | ? | × | $> 54000$ | × |
| 12 | 1 | `sat` | 58.9 | × | × | 5 | 1 | `sat` | 887 | $> 86400$ | × |
| 12 | $\frac{1}{2}$ | `sat` | 6.96 | × | × | 5 | $\frac{1}{2}$ | `sat` | 8598 | $> 86400$ | × |

Here, in the case with intervals, the unsafe state can be reached only with the larger ranges. Also, of course, with the intervals and with the unsatisfiable result, the performance of our tool degrades heavily, when choosing more sample points in the interval. In the worst case, it has to check the finite set of all floating point numbers in the interval, while dReal uses more sophisticated techniques. The result "overflow" means that the program crashed due to restrictions of our implementation.

All experiments were performed on a personal laptop machine with CPU Intel® i7-4702MQ, 8GB memory, running on OS Arch Linux with 4.19.60 Linux kernel.

Note that the original model [7] contains a few mistakes, which we had to correct. The main problem is that the dynamics can block switching between adjacent modes, leading to unintended `UNSAT` results.

We showed that the (corrected) model is *not* safe. This contradicts the original results [7] that proved the model to be safe, apparently only due the mentioned modeling mistakes. To support our statement, we attach a trajectory of a concrete counterexample in Figure 2. Here, a dangerous glucose level is reached (variable $G_p$, the dotted curve) for the initial values $I_p(0) = 29, X(0) = 290, I_1(0) = 120, I_d(0) = 144, I_l(0) = 10, G_p(0) = 238, G_t(0) = 50$.

We refer curious readers to our experimental report [13] for more details. The input data of these and many further experiments, along with the source code of

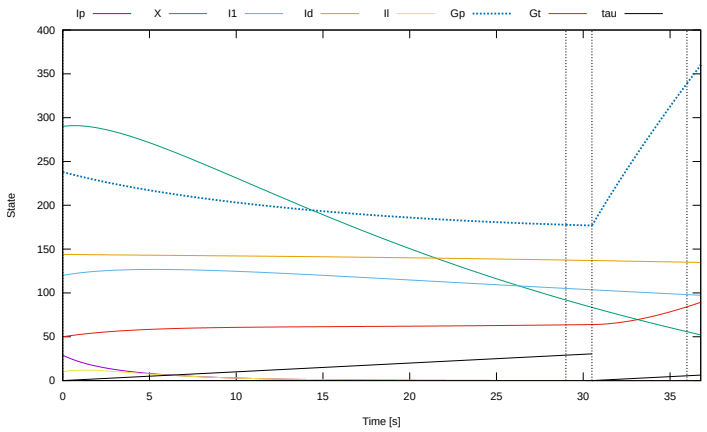


Figure 2: Unsafety witness for the glucose model

the tool, are available on the website of the tool[6].

Finally, we would like to mention that we found the dReal immensely useful for developing, tuning, and debugging our own tool.

# 9 Conclusion

In this paper we introduced an alternative approach to handling differential equations in a SAT context. Motivated by industrial practice, the approach uses the semantics of simulation tools instead of classical mathematical semantics as its basis. Also, the approach allows inputs that integrate ODEs more tightly into SAT problems than was the case for existing methods. Computational experiments with a simple prototype implementation indicate that this problem formulation allows the efficient solution of problems that are highly difficult for start-of-the-art tools based on classical mathematical semantics, especially in satisfiable cases.

In the future, we intend to work on a tighter algorithmic integration between the Boolean and the theory solver [18, 2, 8], on search techniques for efficient handling of satisfiable inputs, and on deduction techniques to prune the inference tree for unsatisfiable inputs. Finally, it will be important to support advanced ODE solving techniques such as root finding (for locating events happening between two simulation steps) and adaptive step sizes.

# References

[1] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[6] https://gitlab.com/Tomaqa/unsot, subdirectory doc/experiments/v0.7

[2] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in SAT modulo theories. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 512–526. Springer, 2006.

[3] C. Barrett and C. Tinelli. Satisfiability modulo theories. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*. Springer International Publishing, 2018.

[4] O. Bouissou, S. Mimram, and A. Chapoutot. Hyson: Set-based simulation of hybrid systems. In *23rd IEEE International Symposium on Rapid System Prototyping (RSP)*, pages 79–85. IEEE, 2012.

[5] O. Bournez and M. L. Campagnolo. A survey on continuous time computations. In S. Cooper, B. Löwe, and A. Sorbi, editors, *New Computational Paradigms*, pages 383–423. Springer New York, 2008.

[6] M. Brain, C. Tinelli, P. Rümmer, and T. Wahl. An automatable formal semantics for IEEE-754 floating-point arithmetic. In *22nd IEEE Symposium on Computer Arithmetic*, pages 160–167. IEEE, 2015.

[7] S. Chen, M. O'Kelly, J. Weimer, O. Sokolsky, and I. Lee. An intraoperative glucose control benchmark for formal verification. In *Analysis and Design of Hybrid Systems ADHS*, volume 48 of *IFAC-PapersOnLine*, pages 211–217. Elsevier, 2015.

[8] L. de Moura and D. Jovanović. A model-constructing satisfiability calculus. In *14th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI, Rome, Italy*, 2013.

[9] S. Edelkamp and S. Schroedl. *Heuristic search: theory and applications*. Morgan Kaufmann, 2012.

[10] A. Eggers, M. Fränzle, and C. Herde. SAT modulo ODE: A direct SAT approach to hybrid systems. In *Automated Technology for Verification and Analysis*, volume 5311 of *LNCS*, 2008.

[11] S. Gao, S. Kong, and E. M. Clarke. Satisfiability modulo ODEs. In *2013 Formal Methods in Computer-Aided Design*, pages 105–112. IEEE, 2013.

[12] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I.* Springer-Verlag, 1987.

[13] T. Kolárik. UN/SOT v0.7 experiments report. `https://gitlab.com/Tomaqa/unsot/blob/master/doc/experiments/v0.7/report.pdf`, 2020.

[14] G. Melquiond. Floating-point arithmetic in the Coq system. *Information and Computation*, 216:14–23, 2012.

[15] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction to Interval Analysis*. SIAM, 2009.

[16] P. J. Mosterman, J. Zander, G. Hamon, and B. Denckla. A computational model of time for stiff hybrid systems applied to control synthesis. *Control Engineering Practice*, 20(1):2–13, 2012.

[17] N. S. Nedialkov. Implementing a rigorous ODE solver through literate programming. In *Modeling, Design, and Simulation of Systems with Uncertainties*, pages 3–19. Springer, 2011.

[18] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: from an abstract Davis–Putnam–Logemann-Loveland procedure to DPLL(T). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.

[19] D. Wilczak and P. Zgliczyński. $\mathcal{C}^r$-Lohner algorithm. *Schedae Informaticae*, 20:9–46, 2011.