Formally verified differential dynamic logic

Bohrer, Brandon; Rahli, Vincent; Vukotic, Ivana; Völp, Marcus; Platzer, André

DOI:
10.1145/3018610.3018616

License:
Other (please specify with Rights Statement)

Document Version
Peer reviewed version

Citation for published version (Harvard):
Bohrer, B, Rahli, V, Vukotic, I, Völp, M & Platzer, A 2017, Formally verified differential dynamic logic. in Y Bertot & V Vafeiadis (eds), *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017.* Association for Computing Machinery (ACM), pp. 208-221, 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, 16/01/17. https://doi.org/10.1145/3018610.3018616

Formally Verified Differential Dynamic Logic *

Brandon Bohrer[1] Vincent Rahli[2] Ivana Vukotic[2] Marcus Völp[2] André Platzer[1]

[1]Computer Science Department
Carnegie Mellon University
Pittsburgh, USA
{bbohrer,aplatzer}@cs.cmu.edu

[2]SnT, University of Luxembourg
Luxembourg
firstname.lastname@uni.lu

Abstract

We formalize the soundness theorem for differential dynamic logic, a logic for verifying hybrid systems. To increase confidence in the formalization, we present two versions: one in Isabelle/HOL and one in Coq. We extend the metatheory to include features used in practice, such as systems of differential equations and functions of multiple arguments. We demonstrate the viability of constructing a verified kernel for the hybrid systems theorem prover KeYmaera X by embedding proof checkers for differential dynamic logic in Coq and Isabelle. We discuss how different provers and libraries influence the design of the formalization.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification

Keywords differential dynamic logic, hybrid systems verification, KeYmaera X, formalization

1. Introduction

Cyber-physical systems such as autonomous cars or water supply systems operate in close proximity of humans or control life-critical resources, and therefore require strong safety guarantees. The highest level of assurance currently known to mankind is obtained through formal verification of such safety properties using proof assistants. Because many cyber-physical systems can be modeled as hybrid systems, formally verifying hybrid systems is an important task. Differential dynamic logic (dL) [36, 37, 39] offers an effective approach for formal verification of hybrid systems: Compared with model-checking-based approaches [15] it offers a high level of expressiveness and precision, and in contrast to other logics [43] it provides general, compositional rules for hybrid systems reasoning. The theorem prover KeYmaera X [17] implements the proof rules of dL and extensive tactic-based automation. Because verifying hybrid systems is important, it is equally important to ensure the verification tool is correct. KeYmaera X simplifies correctness significantly by maintaining a small soundness-critical core of approximately 1700 lines of code, using a proof calculus based on uniform substitution and supported by mathematical proof [39].

But, even if small and straightforward, uniform substitutions still need to be implemented correctly. Moreover, the soundness proofs themselves are involved and rely on nontrivial theorems about differential equations. The purpose of this paper, thus, is

1. to provide independent justification of the correctness of differential dynamic logic by formalizing[1] its syntax, semantics, axiomatization, and soundness proofs in Isabelle [33, 34] and Coq [1, 9], and
2. to obtain verified prover kernels for dL from these mechanizations of the uniform substitution calculus, first embedded in Isabelle and Coq, but in the future also extracted as stand-alone programs.

Ironically, this increases the trusted computing base of the individual kernels. They now depend on the correctness of their proof assistant as well as the definitions in our formalizations. However, the guarantees we gain are of a fundamentally different nature: formalization gives us confidence in the soundness proof for dL, which cannot be addressed by reducing the size of the core. Multiple cores with independent justification are always

* This material is based upon work supported by the National Science Foundation under NSF CAREER Award CNS-1054246, by DARPA under agreement number FA8750-12-2-0291, by the SnT and the National Research Fund Luxembourg (FNR), through PEARL grant FNR/P14/8149128.

[1] Our Isabelle and Coq implementations are available at `https://github.com/LS-Lab/Isabelle-dL` and `https://github.com/LS-Lab/Coq-dL`, respectively.

more trustworthy than an individual core: mistakes now would have to go unnoticed in all of them. Moreover, the chosen provers are under substantial scrutiny.

A verified prover kernel is only ultimately useful if the verification addresses the reasoning principles used in practice. This is important since prover implementations often contain features that are used in practice but not addressed by the theory. In the case of KeYmaera X this includes generalizing single ODEs [39] to systems of ODEs vectorially [36], as well as bound and uniform variable renaming. Indeed, our verification effort exposed a subtle soundness bug in the implementation of bound variable renaming in KeYmaera X, which has since been fixed.

2. Background

2.1 Differential Dynamic Logic

Differential dynamic logic (dL) [36, 37, 39] is a logic for proving properties of hybrid systems expressed as *hybrid programs*, a programming language with constructs for continuous dynamics. Theorem proving in dL consists of determining whether a formula is *valid*, i.e., true in all states ν and interpretations I. States assign meaning to *flexible/assignable* symbols: program variables x and differential symbols x'. Interpretations assign meaning to *rigid* symbols: function symbols f, predicate symbols p, q, program symbols a, b, predicational (a.k.a. quantifier) symbols C and differential program symbols c.

Expressions (e) of dL are terms (θ), differential programs (ODE), hybrid programs (α) and formulas (ϕ). In the following, $\mathcal{V}$ denotes the set of all *variables*. For any $U \subseteq \mathcal{V}$ we write $U' \overset{\text{def}}{=} \{x' : x \in U\}$ for the set of *differential symbols* x' of variables x in U. Our Isabelle formalization considers variables $\mathcal{V}$ and differential symbols $\mathcal{V}'$ disjoint as in [38]. Our Coq formalization considers differential symbols as variables, i.e. $\mathcal{V}' \subseteq \mathcal{V}$, as in [39], allowing higher differential symbols, such as x''.

Definition 1. *Terms* are defined by the grammar:

$$\theta, \eta \ ::= \ x \mid x' \mid r \mid f(\theta_1, \ldots, \theta_k) \mid \theta + \eta \mid \theta \cdot \eta \mid (\theta)'$$

where θ, η, θ_i are terms and $r \in \mathbb{R}$ real-valued literals. The semantics $I\nu[\![\theta]\!]$ assigns a value $r \in \mathbb{R}$ to every term θ in interpretation I and state ν. Like variables, differential symbols x' receive their values from the state ν and are assignable. *Differentials* $(\theta)'$ express how the values of arbitrary terms change, but are not assignable.

All terms are locally Lipschitz continuous, which implies that all ordinary differential equations (ODEs) have solutions. This requires interpretations of function symbols f to be locally Lipschitz continuous as well.

Definition 2. Systems of ODEs, expressed as *Differential Programs* (ODEs), are defined by the following grammar, where ODE_i are differential programs and c is a differential program constant symbol:

$$\text{ODE} \ ::= \ c \mid x' = \theta \mid \text{ODE}_1, \text{ODE}_2$$

The semantics of a differential program in a given interpretation is a state-dependent vector field of type $\mathbb{R}^{\mathcal{V}} \to \mathbb{R}^{\mathcal{V}'}$ that defines the derivative of every variable in every state. Differential program symbols c stand for arbitrary differential programs. Singleton systems $x' = \theta$ define the continuous evolution of a single variable x. Systems of equations are constructed as products $\text{ODE}_1, \text{ODE}_2$, which perform the parallel composition of two differential programs.

Definition 3. *Hybrid programs* (HPs) are defined as:

$$\alpha, \beta \ ::= \ a \mid x := \theta \mid x' := \theta \mid ?\psi \mid \text{ODE} \,\&\, \psi \mid \alpha \cup \beta \mid \alpha; \beta \mid \alpha^*$$

where α, β are HPs, and ψ a dL formula. The semantics of a program in a given interpretation is a reachability relation $R \subseteq (\mathbb{R}^{\mathcal{V} \cup \mathcal{V}'} \times \mathbb{R}^{\mathcal{V} \cup \mathcal{V}'})$ on states. *Assignments* $x := \theta$ and *differential assignment* $x' := \theta$ update the value of variable x or differential symbol x', respectively, to θ. *Tests* $?\psi$ succeed iff the formula ψ is true and do not affect the state. *Differential programs* $\text{ODE} \,\&\, \psi$ follow the evolution of ODE for any duration within the domain constraint ψ. *Nondeterministic choices* $\alpha \cup \beta$ behave as either α or β. *Sequential compositions* $\alpha; \beta$ run β on the state produced by α. *Nondeterministic repetition* α^* runs α an arbitrary number of times. *Program constants* a stand for arbitrary programs.

Definition 4. *Formulas* of dL are defined by the grammar (with dL formulas ϕ, ψ, predicate symbol p, and predicational symbol C):

$$\phi, \psi \ ::= \ \theta \geq \eta \mid p(\theta_1, ..., \theta_k) \mid C(\phi) \mid \neg\phi \mid \phi \wedge \psi \mid \exists x\, \phi \mid \langle\alpha\rangle\phi$$

The semantics of a formula in a given interpretation is the set of states $S \subseteq \mathbb{R}^{\mathcal{V} \cup \mathcal{V}'}$ in which it is true. Operators $>, \leq, <, \vee, \to, \leftrightarrow, [\alpha]\phi, \mathit{true}, \mathit{false}$, and $\forall$ are definable, e.g., $[\alpha]\phi$ as $\neg\langle\alpha\rangle\neg\phi$. The modal formulas $[\alpha]\phi$ and $\langle\alpha\rangle\phi$ express that ϕ holds after all or some runs of α, respectively. *Unary predicational symbols* C (with formula ϕ as argument) are higher-order predicates binding any variables and correspond to functions from formulas to formulas. *Nullary predicational symbols* P, Q are derivable by specifying a ϕ of constant truth-value, i.e., $C(\text{true})$. Unary predicational symbols are used for contextual congruence reasoning, nullary ones used in axioms. The semantics of dL is defined in [39, §2.2].

Example 1 (Uncontrolled Continuous Car Model). As a simple example, consider an uncontrolled continuous car [41, §5.1]. The following dL formula says that if the acceleration and velocity are initially nonnegative, the velocity will always be nonnegative when following a differential equation where the derivative of position is velocity and the derivative of velocity is acceleration:

$$A() \geq 0 \wedge v \geq 0 \rightarrow [v' = A(), x' = v \,\&\, true]\, v \geq 0$$

2.2 Uniform Substitution Calculus

Several proof calculi are available for dL [36, 37, 39]. The one implemented in KeYmaera X and discussed here is a minimalistic Hilbert calculus based on uniform substitution [39], where most reasoning principles are expressed as concrete axioms instantiated by substitution and combined using a small number of standard Hilbert rules. It is from this small number of rules that we gain simplicity of implementation and verification.[2]

A main source of complexity in practical proof calculi is that most axioms and rule schemata are only sound given certain side conditions. The uniform substitution rule confronts these side conditions once and for all through its (computables) notion of *admissibility.*

Theorem 1 ([39, Thm.26]). *The following proof rule with uniform substitution σ is sound:*

$$\text{(US)}\quad \frac{\phi}{\sigma(\phi)} \quad \textit{if}\ \texttt{admissible}(\phi, \sigma)$$

Thus showing soundness of dL is reduced to: (1) showing soundness of rule US, and (2) showing the validity of each axiom or axiomatic rule. Because axioms in a uniform substitution calculus are but individual concrete formulas, there is no need for side conditions: any subtleties have been made explicit in the statement of the axiom. Similarly, we have verified soundness of uniform substitutions on proofs [39, Thm.27] to instantiate concrete representations of proof rules.

2.3 Axiomatization

The axioms and axiomatic proof rules of dL [39, §4–5] implement reasoning about programs, ODEs, differentiation of terms, contextual equivalence, and modal and propositional operators. Here, we focus on the differential axioms: DW, DE, DC, DS, DI, and DG in Fig. 1, which implement the ODE reasoning at the heart of dL.

Differential Weakening DW states that the evolution constraint of an ODE holds after the ODE. *Differential Effect* DE states that differential symbols agree with the vector field of an ODE at the end of the ODE. *Differential Cut* DC states that a formula can be added to an evolution constraint if it always holds after the ODE. *(Constant) Differential Solve* DS states that constant ODEs are uniquely solved by linear functions. *Differential Induction* DI enables inductive reasoning over the flow of an ODE. *Differential Ghost* DG states that one can add equations to ODEs as long as they are linear, implying they have solutions of the same duration.

$$\begin{array}{ll}
\text{DW} & [x' = f(x) \,\&\, q(x)]q(x) \\
\text{DE} & [x' = f(x) \,\&\, q(x)]p(x, x') \leftrightarrow \\
& [x' = f(x) \,\&\, q(x)][x' := f(x)]p(x, x') \\
\text{DC} & \big([x' = f(x) \,\&\, q(x)]p(x) \leftrightarrow \\
& \quad [x' = f(x) \& q(x) \wedge r(x)]p(x)\big) \\
& \leftarrow [x' = f(x) \,\&\, q(x)]r(x) \\
\text{DS} & [x' = f \,\&\, q(x)]p(x) \leftrightarrow \\
& \forall t{\geq}0\, \big((\forall 0{\leq}s{\leq}t\, q(x + fs)) \rightarrow [x := x + ft]p(x)\big) \\
\text{DI} & [x' = f(x) \,\&\, q(x)]p(x) \leftarrow \\
& \big(q(x) \rightarrow p(x) \wedge [x' = f(x) \,\&\, q(x)](p(x))'\big) \\
\text{DG} & [x' = f(x) \,\&\, q(x)]p(x) \leftrightarrow \\
& \exists y\, [x' = f(x), y' = a(x)y + b(x) \,\&\, q(x)]p(x) \\
[:=] & [x := f]p(x) \leftrightarrow p(f) \\
\text{K} & [a](P \rightarrow Q) \rightarrow ([a]P \rightarrow [a]Q)
\end{array}$$

Figure 1. Selected dL axioms

2.4 Uniform Substitution

Church introduced a Uniform Substitution operation [12, §35] (denoted $\check{S}$) and proof rule [12, §40] in order to replace axiom schemata by a finite number of axioms. Differential dynamic logic adopted uniform substitution [39, §3], because of its minimality.

A substitution σ maps rigid symbols to concrete replacements. The difficulty lies in identifying when it is sound to perform a substitution. Consider the following application of axiom [:=] from Fig. 1:

$$\text{clash}\lightning\ \frac{[x := f]p(x) \leftrightarrow p(f)}{[x := 1]x = 1 \leftrightarrow x = 1}$$

where the premiss is valid, but the conclusion false in every state except $x = 1$. This invalid conclusion would be provable from [:=] using the substitution $\sigma = \{f \mapsto 1, p(\cdot) \mapsto x = 1\}$ if we ignored admissibility. *Admissibility* defines which uses of substitutions are sound, as verified by the soundness theorems for US [39, Thm. 26–27]. Admissibility is defined in terms of dL's *static semantics*: signatures, free, bound and must-bound variables [39, §2.4]. The signature $\Sigma(e)$ of an expression is the set of rigid symbols which can influence its dynamics. The free variables $\text{FV}(e)$ of an expression are all assignables (i.e., x and x') which can influence its dynamic semantics. Bound variables $\text{BV}(\alpha)$/must-bound variables $\text{MBV}(\alpha)$ of a program are the assignables that are modified by α on some/all paths.

For example, the signature of Example 1 is the singleton set containing A, its free variables are x and v, and its bound variables are x, x', v, and v'.

We illustrate a high-level proof of Example 1 using generalizations of the axioms presented in Fig. 1 to systems of ODEs. Using the US rule with $\sigma = \{f \mapsto$

[2] For efficiency reasons, KeYmaera X also implements a propositional sequent calculus with Skolemization, which we have only partially formalized.

$A(), q(\cdot) \mapsto true, p(\cdot) \mapsto \cdot \geq 0\}$ on the DI axiom reduces the proof of Example 1 to a proof of

$$[v' = A(), x' = v \,\&\, true]\,(v)' \geq 0$$

We then use US on DE such that it remains to prove

$$[v' = A(), x' = v \,\&\, true][v' := A()]\,(v)' \geq 0$$

Using US on Gödel's G [39, Fig.2] it remains to prove

$$[v' := A()]\,(v)' \geq 0$$

Using US and x' axiom [39, Fig.3] it remains to prove

$$[v' := A()]\,v' \geq 0$$

Finally, we conclude with US by $\sigma = \{f \mapsto A(), p(\cdot) \mapsto \cdot \geq 0\}$ on $[:=]$, after renaming x to v' and using $A() \geq 0$.

3. Formalization: Fundamentals

The following sections describe our formalizations in detail. An outline of the formalizations is given in Fig. 2 showing the dependencies within the formalization. The final result of the formalization is a soundness theorem for an embedded dL proof checker. This depends on the soundness of axioms and rules, which in turn depends on the semantics, which depend on the syntax.

This section first explains the formalization of the syntax and (denotational) dynamic semantics of dL [39, §2.1–§2.2]. These definitions constitute the trusted specification for our formalization: our results depend on the fact that we have defined the semantics of dL correctly. We then define the static semantics of dL [39, §2.4], which is proven correct with respect to the dynamic semantics through *coincidence* [39, Lem.10-12] and *bound effect* [39, Lem.9] lemmas, which are the key building blocks for Theorem 1.

3.1 Syntax

Because the substitution theorem (Theorem 1) is fundamentally syntactic and because our ultimate goal is a verified prover core, both implementations use a deep embedding of dL, i.e., syntactic expressions are represented explicitly as a datatype in the formalization.

Isabelle To simplify the proofs, we define many connectives as derived forms, then validate their definitions by showing they have the expected dynamic semantics. For convenience we define functions to take a fixed, arbitrary number of arguments. Lower-arity functions are derived by supplying constants for all other arguments. Expressions are paramaterized by an arbitrary finite identifier space, so that all vectors are finite-dimensional as needed by the analysis libraries. As usual, we express this dependency on the number of identifiers by encoding the natural number n as a type *'a* of typeclass finite containing n values. Substitution will demand that the number of function symbols (*'nFun*) and predicational symbols (*'nPnl*) change independently from the numbers of all other identifiers (*'n*). Thus we generalize, representing these numbers by the type variables *'nFun*::finite, *'nPnl*::finite, and *'n*::finite respectively. Because terms and ODEs contain functions and variables, they have two type arguments: (*'nFun*, *'n*) trm and (*'nFun*, *'n*) ODE. Because formulas and hybrid programs also contain predicational symbols and variables, they have three type arguments: (*'nFun*, *'nPnl*, *'n*) hp and (*'nFun*, *'nPnl*, *'n*) formula.

Coq Our Coq formalization has an infinite number of identifiers. We also make selected use of dependent types. For example, we encode function arguments using vectors (in terms of the form $f(\theta_1, \ldots, \theta_k)$). Coq's vector type is a dependent type for a list of a given length. The advantage of using vectors over plain lists is that when applying function symbols to arguments there is no need to syntactically check that the list's length is equal to the function symbol's arity. One drawback is that some operations become cumbersome as explained in [20].

3.2 Real Analysis

Because dL relies on significant results from real analysis, our choice of analysis libraries has a significant impact everywhere throughout the formalization.

Our Isabelle formalization uses the standard libraries for analysis, wherein the classical reals are implemented as Cauchy sequences of the rationals. We use the multivariate analysis library of Hölzl et al. [19] which uses Isabelle's typeclass mechanism to generalize prior results and uses a notion of *filters* to elegantly describe the domain on which a function is differentiable. Our treament of differential equations uses a library by Immler [22] which provides important results about differential equations, most notably the Picard-Lindelöf theorem for existence and uniqueness of solutions.

Our Coq implementation relies on Coquelicot [11], a user-friendly library, which extends Coq's standard real analysis library with several widely used results, such as the mean value theorem. For usability, Coquelicot separates the act of defining a function from the act of showing that desired properties such as differentiability are satisfied. This is in contrast to the way the standard library is written where such properties are encoded by the types of the functions. It is similar to the treatment in Isabelle, where expressing complex properties as types is difficult. Coq's standard library provides an axiomatization of classical real numbers described as a complete Archimedean field, but no concrete implementation. This is unlike in other libraries such as the CoRN library [26], which provides an implementation of constructive reals as Cauchy sequences. The two libraries are compatible with each other to some extent [24].

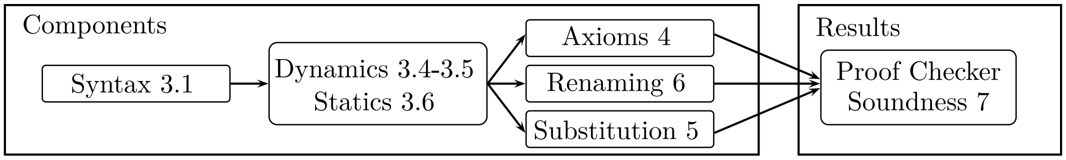

Figure 2. Outline of Formalization

3.3 States and Differential Symbols

To define dL's dynamic semantics, we first choose a representation of states. Because all assignables are real-valued, the only choice is in the number of assignables.

Isabelle Because all expressions have a finite number of identifiers *'n*, states are finite-dimensional as well. For each identifier x, the state assigns a value to the variable x and its associated differential symbol x'. The Isabelle formalization prohibits higher differentials and symbols x'' for the sake of simplicity, because they can be implemented from first-order differentials by introducing additional variables. We encode states *'n* state as pairs of vectors of reals. When we need just the variables or just the differential symbols, it is called *'n* half_state.

Coq Unlike the Isabelle formalization, the Coq formalization supports higher-order differentials x'', x''', etc., since differential symbols are variables as in [39]. We define the Assign type as the type of both non-primed variables and (higher-)differential symbols. The semantics of terms maps states to reals where states map assignables to reals, i.e., we define State as the function type Assign → R, where R is the type of reals.

3.4 Dynamic Semantics of Differential Terms

Differential term $(\theta)'$ could be thought of as derivatives of the term θ with respect to time. However, time is only meaningful during a differential equation, not during a discrete computation. Thus, the semantics of $(\theta)'$ are defined instead by its *spatial* derivatives with respect to all variables. The differential substitution lemma [39, Lem. 35] then shows that the spatial and temporal derivatives agree during the evolution of a differential equation, because the differential symbols x' then agree with the time derivatives of each variable x.

That is, a differential term $(\theta)'$ is interpreted [39, §2.2] as the sum of derivatives of the value of θ w.r.t. all variables $x \in \mathcal{V}$ times the respective $\nu(x')$ value:

$$I\nu[\![(\theta)']\!] = \sum_{x \in \mathcal{V}} \nu(x') \frac{\partial I\nu_x^r[\![\theta]\!]}{\partial r}(\nu(x))$$

where ν_x^r denotes the state that agrees with state ν except for the value of variable x, which is changed to $r \in \mathbb{R}$. Unlike the Isabelle formalization, the Coq formalization supports higher differentials, so that $\mathcal{V}$ includes differential symbols. The Coq definition sums over $\mathrm{FV}(\theta)$, but this is equivalent to summing over $\mathcal{V}$ because all other terms are 0.

Coq In order for $I\nu[\![(\theta)']\!]$ to be well-defined, we proved by induction on terms that the partial derivative $\frac{\partial I\nu_x^r[\![\theta]\!]}{\partial r}$ exists. To handle the differential case $(\theta)'$, we proved that terms are C^∞ smooth, i.e., that the n^{th}-partial derivative of $I\nu[\![\theta]\!]$

$$\frac{\partial \ldots \frac{\partial F(\nu_{x_1 \cdots x_n}^{r_1 \cdots r_n})}{\partial r_n}(\nu(x_n))}{\partial r_1}(\nu(x_1)) \tag{1}$$

exists for all n, where F is the following real-valued function on states: $\lambda\nu.I\nu[\![(\theta)']\!]$. In Coq, this n^{th}-partial derivative translates as the following recursive function, which differentiates F over the list of assignables l from left to right (i.e., the list $x_1, \ldots, x_n$ in Equation 1):

```
Fixpoint partial_derive
        (F : state → R) (l : list Assign) : state → R :=
  match l with
  | [] ⇒ F
  | x :: l ⇒
    fun (st : state) ⇒
      Derive
        (fun r : R ⇒ partial_derive F l (upd_state st x r))
        (st x)
  end.
```

where [] is the empty list; :: is the cons operator; the state (upd_state st x r) is the modified state st_x^r; and Derive is the Coquelicot abstraction that computes the derivative of a real-valued function at a point. Coquelicot guarantees that (Derive f x) computes the derivative of f at point x if the derivative exists, i.e. if ex_derive f x is true. If l is n copies of an assignable y, then partial_derive is simply an n^{th}-derivative:

```
partial_derive f l s
= Derive_n (fun r ⇒ f (upd_state s y r)) (length l) (s y)
```

where (Derive_n f n pt) is a Coquelicot abstraction that computes the n^{th} derivative of f at point pt.

Of course, in order to prove that the derivative in Equation 1 exists, we assumed that function symbols are C^∞ smooth, i.e., their interpretations satisfy[3]:

```
Definition smooth_fun {m : nat} (f : Vector R m → R) :=
```

[3] Coq automatically generates the implicit argument {m : nat} to the definition of smooth_fun from the type of f.

```
∀ (Is : Vector (state → R) m) (a : Assign) (l : list Assign),
  (∀ I w l',
      Vector.In I Is
      → sublist (w :: l') (a :: l)
      → ex_partial_derive I w l')
  → ex_partial_derive (fun s ⇒ f(map(revApp s)Is)) a l.
```

where (revApp a f) is defined as $f(a)$, and where

```
Definition ex_partial_derive (f : state → R) a l : Prop :=
∀ pt s,
  ex_derive (fun r ⇒ partial_derive f l (upd_state s a r)) pt.
```

states that f is differentiable over the list of assignables $a :: l$. The smooth_fun abstraction states that f is partially differentiable over the list of assignables $a :: l$ assuming that its arguments, provided by the Is vector of real-valued functions over states, are partially differentiable over sublists of $a :: l$.

Isabelle The Isabelle formalization builds on the preexisiting formalization for a closely related notion of derivative: the Fréchet derivative. Given a function $f : \mathbb{R}^n \to \mathbb{R}^m$, the Fréchet derivative, $f'(x) : \mathbb{R}^n \to \mathbb{R}^m$, of f at a point $x : \mathbb{R}^n$ is the linear function defined by the dot product $f'(x)(x') = x' \cdot \nabla f(x)$ where $\nabla f(x)$ is the gradient. We give a syntactic characterization:

```
frechet::"('nFun, 'nPnl, 'n) interp
          ⟹ ('nFun, 'n) trm
          ⟹ 'n half_state
          ⟹ 'n half_state
          ⟹ real"
```

of the Fréchet derivative and prove that it matches the differential of a term (frechet_correctness). Validity of the differential axioms [39, §5] follows easily. [4]

The predicates dfree and dsafe identify differential-free terms (which we call *simple* terms or sterms) and terms allowing non-nested differentials (called *differential terms* or dterms), respectively. Likewise osafe, fsafe, and hpsafe characterize ODE systems, formulas and programs where all subterms are dsafe. The semantics of a dterm depends on the whole state, while sterms only depend on the variables. Hence, we define two semantic functions sterm_sem (which takes a 'n half_state) and dterm_sem (which takes a 'n state). The lemma dsem_to_ssem confirms that both agree on sterms:

```
lemma dsem_to_ssem:
 "dfree ϑ ⟹ dterm_sem I ϑ ν = sterm_sem I ϑ (fst ν)"
```

A side benefit of the distinction between sterm and dterm is that we can safely add non-differentiable or even discontinuous terms to the language. For example, if we so wished, we could add support for conditionals and interpreted functions such as min, max, and abs which are desired in practice for complex proofs such as the airborne collision avoidance system ACAS X [23].

Formalizing the derivative as the Fréchet derivative leads to an equivalent semantics for differential terms:

```
definition directional_derivative ::
  "('sf, 'sc, 'cz) interp ⇒ ('sf, 'sz) trm ⇒ 'sz state ⇒ real"
where "directional_derivative I t =
     (λv. frechet I t (fst v) (snd v))"
```

That is, the differential symbol state (snd v) provides coefficients to the gradient. Inside the evolution of an ODE, these coefficients agree with the time-derivative of each variable, thus the semantics of all terms agree with their time-derivatives. During the discrete stages of a program, time is meaningless, but all axioms over differentials hold because the equalities captured in frechet hold in any (differential) state.

Comparison Allowing higher differentials in the Coq formalization results in more complex arguments about derivatives and the requirement that all functions are C^∞ smooth. Prohibiting higher differentials in the Isabelle implementation simplifies these arguments and relaxes the smoothness requirement to C^1, also opening the possibility of non-smooth terms. However, this comes at the cost of more complicated data structure invariants, which will complicate the substitution proofs in Sec. 5. The use of Fréchet derivatives in the Isabelle implementation aids in reuse of existing results, but will complicate the lemmas required in Sec. 4.3 for DG, and makes extensions to infinite-dimensional states more difficult. By contrast, because the Coq implementation explicitly defines that only free variables contribute to the differential of a term, it is easier to show that differentials exist in the presence of infinite states.

3.5 Dynamic Semantics of ODEs

We use the semantics[5] of KeYmaera X for differential programs ODE where differential program symbols c receive meaning from the interpretation I in analogy to program constant symbols. We use an unconventional twist for the semantics of differential products, though, that agrees with the semantics of systems of ODEs [36].

Variables and differential symbols only change when mentioned explicitly. For example, the dynamic semantics of $(x' = \theta, y' = 0)$ and $(x' = \theta)$ differ subtly: the first changes y' to 0, but the latter leaves it intact. This complicates our formalizations, because the ODE libraries require our formalization to distinguish variables that come from the solution of an ODE from those kept from the initial state. Differential program constants must reveal their bound variables as part of the interpretation.

Isabelle The semantics of a differential program is a vector field assigning values to the differential symbols

[4] The chain rule is currently omitted due to time constraints and because it is not needed in practice.

[5] Note: While KeYmaera X has recently improved its semantics to support systems DG and DE, we leave this as future work.

given the state for the variables. In Isabelle's vector notation, $(\chi i.\ e)$ introduces a vector indexed by i where the k'th element is computed by substitution as $[k/i]e$. The constructors OVar, OSing, and OProd represent differential program symbols that vary over systems, singleton systems, and products of systems, respectively.

```
fun ODE_sem:: "('nFun, 'nPnl, 'n) interp
                  ⇒ ('nFun, 'n) ODE
                  ⇒ 'n half_state
                  ⇒ 'n half_state"
where
  "ODE_sem I (OVar x) = ODEs I x"
| "ODE_sem I (OSing x ϑ) =
      (λν. (χ i. if i = x then sterm_sem I ϑ ν else 0))"
| "ODE_sem I (OProd ODE1 ODE2) =
      (λν. ODE_sem I ODE1 ν + ODE_sem I ODE2 ν)"
```

This additive semantics for products is highly amenable to verification, but defies intuition. It is brought into harmony with intuition by the predicate osafe: in well-formed ODEs, each variable is bound at most once, so adding the vector fields of two systems componentwise coincides with parallel composition.

Coq As in Def. 2 and our Isabelle formalization, in Coq we defined a function dynamic_semantics_ode_fun that interprets ODEs as functions of type state → state. This function is the Coq counterpart of the ODE_sem Isabelle function. Given this function we define dynamic_semantics_ode, a predicate that states that the function *phi* from $[0, r]$ to state is a solution of the ODE *ode*, w.r.t. an interpretation I, as follows:

```
∀ (z : preal_upto r), (x : Assign)
  In x (ode_assigns I ode)
  → (ex_derive (fun t ⇒ phi t x) z
        ∧ phi z x' = Derive (fun t ⇒ phi t x) z
        ∧ dynamic_semantics_term I (phi z) x'
            = dynamic_semantics_ode_fun I ode (phi z) x')
```

where preal_upto r is the type of positive real numbers between 0 and r; and where (ode_assigns I *ode*) is the set of assignables bound in *ode*, where I provides the bound assignables of ODE constants. The denotational semantics of a program of the form *ode* & ψ is:

```
fun v w ⇒
∃ (r : preal) (phi : R → state),
  equal_states_except v (phi 0) (ode_footprint_diff I ode)
  ∧ w = phi r
  ∧ dynamic_semantics_ode I ode r phi
  ∧ ∀ (z : preal_upto r),
      dynamic_semantics_formula I ψ (phi z)
     ∧ equal_states_except (phi 0) (phi z) (ode_footprint I ode)
```

where v is the initial state of the program *ode* & ψ and w is its final state. This formula states that (1) the initial state v has to be equal to *phi* 0 on all assignables except on the differential symbols bound in *ode* (extracted by ode_footprint_diff); (2) the final state w has to be equal to *phi* r; (3) *phi* has to be a solution of *ode* as stated by dynamic_semantics_ode; (4) ψ has to satisfy the ODE at all points in time between 0 and r; (5) and finally, assignables not bound in *ode* (i.e. not in (ode_footprint I *ode*)) do not evolve over time. Given an ode *ode*, ode_footprint extracts all the assignables of the form x such that $x' = \theta$ occurs in *ode*, while ode_footprint_diff extracts all the assignables of the form x' such that $x' = \theta$ occurs in *ode*.

3.6 Static Semantics

In this section, we verify the static semantics of dL [39, §2.4]. We verify *coincidence* lemmas [39, Lem. 10-12] showing that the dynamic semantics of expressions depend only on their signatures and free variables, as well as the bound effect lemma [39, Lem. 9] showing that *only* bound variables can be affected by a program.

Because we wish to generate verified prover cores from our formalizations, we must ensure that the free, bound and must-bound variables of expressions are computable. The nontrivial part is that differential terms $(\theta)'$ and predicational symbols $C(\phi)$ have all variables as free variables. In the Isabelle formalization, this is easy because it supports finite numbers of assignables. However, in Coq we support infinitely many identifiers, so we must choose a finite representation. As in KeYmaera X, the only infinite sets that arise in the static semantics are cofinite, meaning their complement is finite. Thus we have a finite representation of sets, i.e., all sets in the static semantics are either finite or cofinite:

```
Inductive FCset {T : Type} : Type :=
| FCS_finite   (l : list T) : FCset
| FCS_infinite (l : list T) : FCset.
```

where a set of the form FCS_finite l is a finite set that only contains the elements in l; and FCS_infinite l is an infinite set that does not contain the elements in l, i.e., it is the complement of the set FCS_finite l. If the type T has decidable equality, then predicates such as membership, subset or disjoint become decidable. Interestingly, in order to prove one property of the subset predicate (called ifset_subset_iff in our formalization), in addition to having decidable equality, we required the type T to come with a "fresh" operation that, given a list l of elements of T, generates a fresh element that is not in l. These are the only two operations we ever used to develop our FCset library.

With this set representation, the proofs of coincidence and bound effect mirror the structure of prior proofs [39], in both the Coq and Isabelle formalizations.

Comparison The finite-dimensional state space of the Isabelle implementation provided the useful simplifying assumption that all sets are finite, making all necessary operations easily computable. However, because formulas may have a large number of variables, using standard finite set data structures may not be desirable in practice. The infinite state space of the Coq formalization forces us to confront this issue head-on, developing and verifying a data structure analogous to the one used by KeYmaera X in practice, further increasing confidence in the implementation of the prover core.

4. Formal Verification of Differential Dynamic Logic Axioms

Since uniform substitutions reduce soundness to validity of the axioms, we formalize the proofs that dL's axioms are valid [39]. Except for DI, DS and DG, most axioms have simple local proofs. DI uses the mean-value theorem. DS and DG use the Picard-Lindelöf theorem.

4.1 DI: Differential Invariants

We have implemented differential invariants for atomic formulas $\theta_1 \geq \theta_2$ and $\theta_1 > \theta_2$ as the following DI axioms:

$$\begin{aligned}&(Q \to [x' = h(x) \,\&\, Q](f(x))' \geq (g(x))')\\ \to\ &(Q \to f(x) \geq g(x)) \to [x' = h(x) \,\&\, Q] f(x) \geq g(x)\\ &(Q \to [x' = h(x) \,\&\, Q](f(x))' \geq (g(x))')\\ \to\ &(Q \to f(x) > g(x)) \to [x' = h(x) \,\&\, Q] f(x) > g(x)\end{aligned}$$

The other cases can be derived in dL [39]. For readability, we only showed axioms for single ODEs. Generalizations to ODE systems are straightforward. For example the DI axiom that we have proved for atomic formulas of the form $\theta_1 \geq \theta_2$ uses a differential program constant c that can be instantiated to any system of ODEs:

$$\begin{aligned}&(Q \to [c \,\&\, Q](f(x))' \geq (g(x))')\\ \to\ &(Q \to f(x) \geq g(x)) \to [c \,\&\, Q] f(x) \geq g(x)\end{aligned}$$

4.2 DS: Differential Solution

DS states that constant ODEs $x' = c$ are solved by the function $f(t) = x_0 + t \cdot c$ on the domain $\mathbb{R}$ and that this solution is unique. Proving DS proceeds in these steps.

Isabelle For the uniqueness direction of DS, we assume the existence of a solution `sol` on an closed interval $[0, t]$ and show that solution is equal to $f(t) = x_0 + t \cdot c$. In Isabelle we show uniqueness of solutions with Picard-Lindelöf, which says any locally Lipschitz ODE has a unique solution on an open interval. We do so with an instance of the locale `ll_on_open` named `ll`:

```
interpret ll: ll_on_open UNIV
        "(λ_. ODE_sem I (OSing vid1 (f0 fid1)))" UNIV 0
```

This gives us the existence of a unique solution called `flow`. We prove that the maximal existence interval for a constant ODE is $\mathbb{R}$ with `ll.existence_ivl_eq_domain` and prove that $f(t) = x_0 + t \cdot c$ equals the unique solution of the ODE with `ll.equals_flowI`.

Coq Because Coquelicot does not provide the Picard-Lindelöf theorem, the Coq implementation proves uniqueness directly. Since DS is for constant ODEs (other ODEs combine axioms [39]), it suffices to show for any function g that has a constant derivative c that:

```
∀ (c r : R) (g : R → R),
  (0 ≤ r)
  → (∀ x : preal_upto r, ex_derive g x ∧ Derive g x = c)
  → g r = (g 0 + c * r).
```

This result can easily be proved by first turning the goal into `g r - g 0 = c * r`, and then by integrating both sides over the interval $[0, r]$ and using the following simplified form of the fundamental theorem of calculus (for constant derivatives and $a \leq b$):

```
(∀ x, a ≤ x ≤ b → ex_derive g x ∧ Derive g x = c)
→ RInt (Derive g) a b = g b - g a.
```

4.3 DG: Differential Ghost

The axiom DG allows adding or removing an ODE $y' = \theta$ to another ODE if θ is linear in y (but not necessarily linear in time). For example, we extend the ODE $x' = x^2$ to $(x' = x^2, y' = x^3 \cdot y + 2 \cdot y)$. We use the solution ϕ_x to construct a time-dependent ODE for y: $y(0) = y_0, y'(t) = \phi_x(t)^3 \cdot y + 2 \cdot y$. Because this is still linear in y it has a solution, even though it is not linear in t. Linearity ensures the solution exists at least as long as the solution for the rest of the system, which is needed for soundness. The proof is by Picard-Lindelöf.

Isabelle The existence of unique solutions provides a unique solution on the open maximal existence interval of an ODE (see `ll_on_open_it.flow_usolves_ode`). We need solutions on the compact interval $[0, t]$. Thus, our proof first observes that when a solution exists on $[0, t]$, then $[0, t]$ is a subset of the maximal existence interval. We then observe that the ODE for y' has the same existence interval as the ODE for x' and thus $[0, t]$ is a subset of that existence interval. Thus we can restrict the solution for y' to $[0, t]$.

Applying `ll_on_open_it.flow_usolves_ode` requires showing that the ODE for y' is locally Lipschitz, which we show by first showing it has a continuous derivative. Because we use Fréchet derivatives, the derivative $f'(x)$ at a point x is a (bounded linear) function. Continuity for derivatives is defined using a metric space on bounded linear functions with operator norm:

$$\|f\| = \sup_{x \neq 0} \frac{\|f(x)\|}{\|x\|}$$

This lets us prove a lemma `continuous_blinfun_vec'` which allows us to show derivatives are continuous by showing each component individually.

Coq For lack of a Picard-Lindelöf theorem in Coquelicot, only the easy → direction of DG is proved in our Coq formalization, i.e. that $[x' = f(x) \,\&\, q(x)]p(x)$ implies $\exists y\, [x' = f(x), y' = a(x)y + b(x) \,\&\, q(x)]p(x)$.

5. Formally Verified Uniform Substitution

First- vs. Second-Order As presented in [39, Fig.1] (for unary functions), to apply a uniform substitution σ to a term of a form $f(\theta_1, \ldots, \theta_n)$, one substitutes $\cdot_i$ by $\sigma(\theta_i)$ in u, if σ maps f to u, i.e., $\sigma(f(\theta_1, \ldots, \theta_n)) = \{\cdot_1 \mapsto \sigma(\theta_1), \ldots, \cdot_n \mapsto \sigma(\theta_n)\}(u)$. The reserved constant symbols of the form $\cdot_i$ mark the positions where the arguments will end up in u. For example, σ could map the ternary symbol f to $\cdot_1 + \cdot_2 + \cdot_3$. The dot terms are replaced with the concrete arguments by a recursive call. While dot terms exist only in the substitution data structure, they still appear in the arguments to substitution. In order to simplify the termination argument for substitution, we split the substitution algorithm into two separate phases that we call *first-order uniform substitution* and *second-order uniform substitution*, borrowing terminology from Nuprl [8, 13]. Second-order uniform substitution is equivalent to the substitution operation of [39]. However, when we substitute the arguments of a function into the body, we call a separate first-order substitution instead of making a recursive call to second-order substitution. First-order substitution is only responsible for substituting nullary symbols representing arguments. This greatly simplifies the formal termination argument for both operations.

Error Monad vs. Separate Admissiblity Check Our Isabelle implementation separates the admissibility check from the application of a uniform substitution to an expression. Uniform substitution is therefore implemented as a recursive total function. The soundness theorem assumes that admissibility holds, which is a decidable, inductively defined predicate.

In Coq we combine both passes and use an option monad to deal with the fact that uniform substitution fails if one admissibility check fails.

The advantage of our Isabelle implementation is that uniform substitution is a total function, while the advantage of our Coq implementation is that we only do one pass over expressions. Even though using a monad slightly complicates our Coq definition of uniform substitution, because we are only ever interested in the case when uniform substitution succeeds, using the right tactics, we never have to explicitly deal with the monad.

Adjoint Interpretation The substitution proof proceedes by defining for each substitution σ, interpretation I and state ν an adjoint interpretation [39, §3.1] that equivalently captures the effect of σ semantically. Since validity means truth in every interpretation and state, validity of the substituted formula follows from the validity of the original formula in the adjoint interpretation [39, §3.2].

Coq Because the adjoint interpretation of a uniform substitution s needs to capture the effect of s on expressions, it associates the following function with function symbol g of arity n:

```
fun d : Vector R n ⇒
  dynamic_semantics_term
    (upd_interpretation_dots I d) v (lookup_func s g n)
```

where (dynamic_semantics_term I v t) implements $Iv[\![t]\!]$; (upd_interpretation_dots I d) agrees with I except for the interpretation of the symbol $\cdot_i$, which is changed to the i^{th} element of d; (lookup_func s g n) returns u if s maps g to the term u, and $g(\cdot_0, \ldots, \cdot_{n-1})$ otherwise. In Coq, for the adjoint interpretation to be well-defined, we have to prove that the above function satisfies the smooth_fun predicate. We prove this by induction on the term (lookup_func s g n). To get a strong enough induction hypothesis to prove the differential term case, we prove instead the following by induction on the term t:

```
smooth_fun (fun d : Vector R m ⇒
    partial_derive (fun v ⇒ dynamic_semantics_term
            (upd_interpretation_dots I m d) v t) l s).
```

for any natural number m, state s, assignable list l, and interpretation I. Note that smooth_fun differentiates over the interpretation used to compute the semantics of t, while the inner partial_derive differentiates over the state. In the case where t is a function symbol g, we only know that g is smooth, i.e., it satisfies smooth_fun. In order to apply our smooth_fun hypothesis we combine the two partial derivatives mentioned above into one. We then uncombine the two partial derivatives in order to apply the induction hypothesis.

In order to combine nested partial derivatives into a single partial derivative, one has to carefully avoid variable name clashes through renaming:

```
partial_derive
  (fun s1 ⇒ partial_derive (fun s2 ⇒ F s1 s2) l2 s2)
  l1
  s1
= partial_derive
    (fun s ⇒ F (update_state_st s s1 l2')
               (update_state_st_rebase s2 s l2' l2))
    (l1 ++ l2')
    (update_state_st_rebase s1 s2 l2 l2').
```

where *l2'* is a fresh renaming of the list *l2* that is disjoint from *l1*; and uncombine them as follows:

```
partial_derive
  (fun s ⇒ F (update_state_st s s1 l2')
             (update_state_st_rebase s2 s l2' l2))
  (sl1 ++ sl2')
  s
= partial_derive
    (fun s1 ⇒ partial_derive
                (fun s2 ⇒ F s1 s2)
                sl2
                (update_state_st_rebase s2 s l2' l2))
    sl1
    (update_state_st s s1 l2')
```

where the state (update_state_st *s1 s2 l*) is defined as *s2* on assignables in *l*, and as *s1* otherwise; and where the state (update_state_st_rebase *s1 s2 l1 l2*) returns (*s2* (rename *l2 l1 a*)) on assignables *a* in *l2*, and is defined as *s1* otherwise, where (rename *l2 l1 a*) returns *a* if it does not occur in *l2*, and otherwise returns the n^{th} element of *l1* if *a* is the n^{th} element of *l2*.

Isabelle Because we explicitly represent the number of (possible) function symbols in the type of a term, we can express the fact that argument symbols are absent in the result of first-order substitution through the type of the first-order substitution functions. Consider the type for first-order substitution on terms:

```
primrec TsubstFO::"('nFun + 'b::finite, 'n) trm
        ⇒ ('b ⇒ ('nFun, 'n) trm) ⇒ ('nFun, 'n) trm"
```

That is, in (TsubstFO θ σ) the term θ distinguishes the *'nFun* permanent symbols from the *'b* temporary symbols that stand for arguments. The substitution σ specifies a replacement for all *'b* of the temporary symbols, and those replacements do not refer to any of the *'b* symbols, after which it returns a term that also does not refer to *'b* symbols. When second-order substitution calls first-order substitution on a term θ, then θ always comes from the substitution σ and not from the original input expression, thus we have the freedom to distinguish argument symbols from other function symbols in types.

Substitution requires showing that data structure invariants are maintained. The result of substitution is always safe according to the predicates dsafe, etc. These properties follow by induction.

Substitution is perhaps where the less restrictive invariants of the Coq formalization pay off most heavily: because substitution makes extensive changes to a formula, any structural invariants also require extensive arguments that they are maintained.

6. Renaming

Uniform renaming and *bound renaming* are the primary operations that KeYmaera X implements that were not addressed in the theory [39]. Whereas uniform substitution replaces a *symbol* with an arbitrary term, renaming renames two *variables* to each other. This renaming operation is a primitive operation of Nominal Logic [35] where it is referred to as *swapping*. We have proved that renaming preserves the validity of formulas. Renaming is necessary when applying an axiom that refers to a concrete variable name, such as the assignment axiom [:=]. Bound renaming is a small extension to uniform renaming that only renames bound variables.

Uniform Renaming Uniform renaming swaps two variables x and y uniformly everywhere in an expression. The correctness argument is analogous to that for substitution: to show that substitution preserves validity, we constructed an adjoint interpretation whose effect is equivalent to the substitution. Because variables receive meaning from the state, we instead construct an *adjoint state* whose effect is equivalent to swapping x and y (specifically, the state where x is swapped with y and x' is swapped with y'). We show by induction that the result of renaming is true in a given state iff the initial formula is true in the adjoint state, so by validity of the initial formula, the renamed formula is valid.

Bound Renaming Given an assignment inside a modality $[x := \theta]\phi$, bound renaming renames the destination variable x, but differs from uniform renaming because it does not affect the right-hand side θ, only the destination x and the formula ϕ. The correctness proof of bound renaming consists of applying uniform renaming to the formula ϕ and using the coincidence theorem on formulas to show that the result is true in the state reached after the assignment $y := \theta$.

Verification Reveals a Bug The proof for bound renaming exposed an exploitable soundness bug in the KeYmaera X implementation. The correct admissibility criterion for bound renaming of x and y in $[x := \theta]\phi$ is

$$\{y, y', x'\} \cap \mathrm{FV}(\phi) = \emptyset$$

The need to include x' in this condition is counterintuitive, which led to a prover bug where this variable was not checked. However, once the bug is discovered, it is straightforward to construct an example where this bug leads to a soundness violation, for example:

$$\text{BRclash}\lightning\ \frac{[x := x']x = x'}{[y := x']y = y'}$$

The premiss is valid, but the conclusion is not. Thankfully, no existing code in KeYmaera X depended on the presence of this bug, so changing the precondition as indicated above was sufficient to fix the bug.

7. Applications

The immediate application of this work is the creation of verified prover kernels for the theorem prover KeYmaera X. While we leave a verified standalone kernel as future work, our formalizations include verified proof checkers for dL embedded in Coq and Isabelle which interpret dL proof terms inspired by LPd$\mathcal{L}$ [16].

We evaluate the completeness of our proof checkers with a few example proofs. For our first example, we implement the $\wedge$ case of DI in dL. The idea is to reduce $[c \,\&\, Q](P_1 \wedge P_2)$ equivalently to $[c \,\&\, Q]P_1 \wedge [c \,\&\, Q]P_2$

using $[\alpha](P \wedge Q) \leftrightarrow [\alpha]P \wedge [\alpha]Q$, which is a consequence of the K axiom, and then using DI separately on both conjuncts to obtain $[c \,\&\, Q](P_1)' \wedge [c \,\&\, Q](P_2)'$ as the remaining subgoal provided that $P_1 \wedge P_2$ holds initially.

Our second example is a proof of Example 1 from p. 2. Despite its simplicity, it demonstrates many reasoning techniques for concrete systems. For example, the differential equation is easily solvable in closed form, but the same axioms enable checking properties of nonlinear ODEs that have no closed form solution.

Our embedded proof checkers make initial steps at bridging the theory with KeYmaera X. KeYmaera X implements an extension of the Uniform Substitution calculus [39] with a propositional sequent calculus [36]. Its proofs do not operate on formulas but on locally-sound derived rules (i.e. the conclusion is valid in any interpretation in which the premisses are). Thanks to uniform substitutions [39, Thm.27], derived rules, as with axioms, can be represented concretely as a conclusion C and list of premisses $(SG_1, \ldots, SG_n)$.

We implement and verify enough of the theory of sequent calculus and derived rules to implement our examples, but gaps remain. A systematic gap is that our formalizations do not provide explicit support for arithmetic reasoning. Currently, support is limited to leaving arithmetic facts as open goals or adding them as axioms and proving them in Coq or Isabelle. In future work, we plan on using a witness-producing real arithmetic solver and verified witness checker based on semidefinite programming, which is competitive with second-tier decision procedures for real arithmetic [40]. Furthermore, our DG axiom does not support systems and the DE axiom is less flexible than in KeYmaera X. Implementing them in full generality requires a modest extension of differential program symbols.

8. Lessons Learned

The process of formalization has largely been one of discovering nuances in what came before, rather than discovering outright flaws.

It is perhaps remarkable that we found only one explicit bug, and at the same time the location of that bug was telling. Renaming was until now the only part of the KeYmaera X core that was not justified by a proof, and it was the only part in which we found a bug. While having an informal proof as the foundation of the prover core did not give us the level of assurance we desired, it clearly went a long way to improve the robustness of the system in practice.

Our two formalizations confirm that both approaches to higher differentials [38, 39] are viable options. The formalizations also confirm design decisions for simpler soundness arguments. For example, differential invariants used to employ a meta-operator on formulas [37] that the uniform substitution approach abandoned to obtain a minimalistic basis [39]. Our formalizations confirm that it is far easier for correctness arguments to decompose DI into separate axioms for each case.

Both formalizations independently chose to separate first-order and second-order substitution to remove the need for custom well-founded orders, which are easier for humans than for proof assistants. There is sometimes a tradeoff between the complexity of an algorithm and the complexity of its correctness arguments. In our case both formalizations chose a longer algorithm, thus requiring more proofs, each of which was simpler.

9. Related Work

Verification of Theorem Provers Barras and Werner [7] verified a typechecker for a fragment of Coq in Coq. Harrison [18] verified (1) a weaker version of HOL Light's kernel in HOL Light and (2) HOL Light in a stronger variant of HOL Light. Myreen et al. have extended this work, verifying HOL Light in HOL4 [28, 32] and using their verified compiler CakeML [27] to ensure these guarantees apply at the machine-code level. Myreen and Davis proved the soundness of the ACL2-like theorem prover Mitawa in HOL4 [31]. Anand, Bickford, and Rahli [5, 42] proved the relative consistency of Nuprl's type theory [3, 13] in Coq with the goal of generating a verified prover core. We share a common goal: formally verify that theorem provers are correct. However, the underlying theory of our prover greatly differs (dL intimately deals with programs with differential equations), leading to substantially different proofs (in our case, intricate proofs about real analysis).

Verification of Hybrid Systems Hybrid systems verification is an actively studied field. For example, SpaceEx [15] is a model checker that provides an automated reachability analysis for linear hybrid automata with a soundness-critical core of about 100 000 LOC.

We focus on approaches that strive for a justification of the verification technique itself. Immler [21] verifies a set-based reachability analysis for ODEs in Isabelle using his differential equations library [22]. His use of Isabelle makes his analysis more trustworthy than SpaceEx, but it is currently less automated and, more fundamentally, lacks the expressiveness and scalability of dL. Völker [44] defines the semantics of hybrid automata in Isabelle, but no verification techniques. Ábrahám-Mumm et al. formalized in PVS [2] an automaton-based approach similar to timed automata. They implement Floyd's inductive assertion reasoning method. However, they only check invariants when transitioning between discrete states, as opposed to our differential invariants which hold continuously. Furthermore, they only support continuous dynamics given as explicit solutions. Thus they cannot reason by differential invariant, nor can they express systems whose solutions only exist for finite time. StarL [30] is a framework for programming and simulating Android applications that control robots. It uses a formalization of timed automata and has a discrete model of space in order to verify distributed applications. Anand and Knepper [4] develop the framework ROSCoq based on the Logic of Events for reasoning about distributed CPS in Coq with constructive reals and generating verified controllers. However, they provide limited support for reasoning about derivatives, requiring extensive manual proofs by the user. VeriDrone by Ricketts et al. [43] is a framework for verifying hybrid systems in Coq that relies on a discrete-time temporal logic called RTLA, inspired by TLA [29]. They prove an analog of DI, but not the other ODE axioms. They use a combination of a deep and shallow embedding, allowing arbitrary Coq propositions to be embedded directly into RTLA. Their embedding reuses Coq's notion of substitution instead of defining its own. An advantage of uniform substitution is its generality: we can introduce new variable dependencies if they do not clash, which is necessary for most proof steps. For example, the V axiom $(p() \rightarrow [a]p())$ can immediately be instantiated to $x = 0 \rightarrow [y' = 1]x = 0$ because $\mathrm{BV}(y' = 1) \cap \mathrm{FV}(x = 0) = \emptyset$, introducing dependencies on x and y and y'. The permitted syntactic occurrence patterns for uniform substitutions are easily decidable, thereby enabling automatic clash detection and sound generalizations of axiom shapes. That needs the syntactic exposition of a deep embedding, though.

10. Conclusions

We have for the first time mechanized the metatheory for the soundness of dL (Fig. 3 summarizes what has been done so far). This mechanization is validated by some example proofs, demonstrating how to recover convenient reasoning over differential invariants, and how to use common techniques to reason about concrete systems. This validation shows that our formalizations put us well on our way to verified prover cores for dL that can be used in production to increase the trustworthiness of proofs about hybrid systems. Our choice to perform the formalization twice, largely independently, also offered a rare opportunity to discuss subtleties that arose during formalization and how to address them.

We cannot answer whether it is easier to get a prover core correct by minimizing its size without the support of formal proof (1 700 LOC of KeYmaera X) or by implementing it inside another proof assistant with a larger core (8 913 LOC of Isabelle's core Isabelle/Pure, 16 538 LOC for Coq's kernel, and 6 720 for its standalone checker of compiled files), because both rest on nontrivial stacks and our formalizations (14 614 total lines in Isabelle, 9 364/16 406 lines of Coq specifications/proofs) and the libraries they depend on need to be free of specification errors. Yet, without doubt, the combination of a well-engineered prover core with multiple formal soundness proofs in well-established, well-tested proof assistants dominates either approach alone. This is especially true because these formalizations increase confidence in the correctness of the supporting theory, which cannot be addressed by better engineering. Furthermore, these formalizations open up the possibility of synergistic collaboration between KeYmaera X, Coq and Isabelle.

	CADE'15 [38]	JAR'16 [39]	KeYmaera X	**Isabelle**	**Coq**
systems of ODEs	[36]		✓	✓	✓
multiple argument functions	✗	✗	✓	✓	✓
infinite number of identifiers	✓	✓	✓	✗	✓
explicit set representation	✗	✗	✓	✗	✓
higher-order differentials	✗	✓	✗	✗	✓
uniform substitution definitions	1	1	1	2	2
uniform variable renaming	✗	✗	✓	✓	✓
bound variable renaming	✗	✗	✓	✓	✓
sequent calculus	[36]		✓	(✓)	(✓)
DG	✓	✓	✓	✓	✗
DG+DE for systems of ODEs	[37]		✓	✗	✗

Figure 3. Comparison between various presentations/implementations of dL

Acknowledgements Sincere thanks to the Isabelle group at TU München for their kind advice, especially to Johannes Hölzl for advice on general Isabelle usage and Fabian Immler for advice on his ODE library.

References

[1] Coq Proof Assistant. URL `http://coq.inria.fr/`. Accessed: 2016-11-28.

[2] E. Ábrahám-Mumm, M. Steffen, and U. Hannemann. Verification of hybrid systems: Formalization and proof rules in PVS. In *ICECCS 2001*, pages 48–57. IEEE Computer Society, 2001. ISBN 0-7695-1159-7. doi: 10.1109/ICECCS.2001.930163. URL `http://doi.ieeecomputersociety.org/10.1109/ICECCS.2001.930163`.

[3] S. F. Allen, M. Bickford, R. L. Constable, R. Eaton, C. Kreitz, L. Lorigo, and E. Moran. Innovations in computational type theory using Nuprl. *J. Applied Logic*, 4(4):428–469, 2006. `http://www.nuprl.org/`.

[4] A. Anand and R. A. Knepper. ROSCoq: Robots powered by constructive reals. In C. Urban and X. Zhang, editors, *ITP 2015*, volume 9236 of *LNCS*, pages 34–50. Springer, 2015. ISBN 978-3-319-22101-4. doi: 10.1007/978-3-319-22102-1_3. URL `http://dx.doi.org/10.1007/978-3-319-22102-1_3`.

[5] A. Anand and V. Rahli. Towards a formally verified proof assistant. In Klein and Gamboa [25], pages 27–44. ISBN 978-3-319-08969-0. doi: 10.1007/978-3-319-08970-6_3. URL http://dx.doi.org/10.1007/978-3-319-08970-6_3.

[6] J. Avigad and A. Chlipala, editors. *CPP 2016*, 2016. ACM. ISBN 978-1-4503-4127-1. URL http://dl.acm.org/citation.cfm?id=2854065.

[7] B. Barras and B. Werner. Coq in Coq. Technical report, INRIA Rocquencourt, 1997.

[8] D. A. Basin and D. J. Howe. Some normalization properties of Martin-Löf's type theory, and applications. In T. Ito and A. R. Meyer, editors, *TACS '91*, volume 526 of *LNCS*, pages 475–494. Springer, 1991. ISBN 3-540-54415-1. doi: 10.1007/3-540-54415-1_60. URL http://dx.doi.org/10.1007/3-540-54415-1_60.

[9] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004. ISBN 3540208542. http://www.labri.fr/perso/casteran/CoqArt.

[10] S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors. *ITP'13*, volume 7998 of *LNCS*, 2013. Springer. ISBN 978-3-642-39633-5.

[11] S. Boldo, C. Lelay, and G. Melquiond. Coquelicot: A user-friendly library of real analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62, 2015. doi: 10.1007/s11786-014-0181-1. URL http://dx.doi.org/10.1007/s11786-014-0181-1.

[12] A. Church. *Introduction to Mathematical Logic*. Princeton, Princeton University Press, 1956.

[13] R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P.Panangaden, J. Sasaki, and S. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986. ISBN 0-13-451832-2.

[14] A. P. Felty and A. Middeldorp, editors. *CADE-25*, volume 9195 of *LNCS*, 2015. Springer. ISBN 978-3-319-21400-9. doi: 10.1007/978-3-319-21401-6. URL http://dx.doi.org/10.1007/978-3-319-21401-6.

[15] G. Frehse, C. L. Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable verification of hybrid systems. In G. Gopalakrishnan and S. Qadeer, editors, *CAV 2011*, volume 6806 of *LNCS*, pages 379–395. Springer, 2011. ISBN 978-3-642-22109-5. doi: 10.1007/978-3-642-22110-1_30. URL http://dx.doi.org/10.1007/978-3-642-22110-1_30.

[16] N. Fulton and A. Platzer. A logic of proofs for differential dynamic logic: toward independently checkable proof certificates for dynamic logics. In Avigad and Chlipala [6], pages 110–121. ISBN 978-1-4503-4127-1. doi: 10.1145/2854065.2854078. URL http://doi.acm.org/10.1145/2854065.2854078.

[17] N. Fulton, S. Mitsch, J.-D. Quesel, M. Völp, and A. Platzer. KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In Felty and Middeldorp [14], pages 527–538. ISBN 978-3-319-21400-9. doi: 10.1007/978-3-319-21401-6_36. URL http://dx.doi.org/10.1007/978-3-319-21401-6.

[18] J. Harrison. Towards self-verification of HOL Light. In U. Furbach and N. Shankar, editors, *IJCAR 2006*, volume 4130 of *LNCS*, pages 177–191. Springer, 2006. ISBN 3-540-37187-7.

[19] J. Hölzl, F. Immler, and B. Huffman. Type classes and filters for mathematical analysis in Isabelle/HOL. In Blazy et al. [10], pages 279–294. ISBN 978-3-642-39633-5. doi: 10.1007/978-3-642-39634-2_21. URL http://dx.doi.org/10.1007/978-3-642-39634-2_21.

[20] C.-K. Hur. Heq: a Coq library for heterogeneous equality. Presented at the 2nd Coq Workshop, 2010. URL http://www.mpi-sws.org/~gil/Heq/.

[21] F. Immler. Verified reachability analysis of continuous systems. In C. Baier and C. Tinelli, editors, *TACAS-21*, volume 9035 of *LNCS*, pages 37–51. Springer, 2015. ISBN 978-3-662-46680-3. doi: 10.1007/978-3-662-46681-0_3. URL http://dx.doi.org/10.1007/978-3-662-46681-0_3.

[22] F. Immler and C. Traut. The flow of ODEs. In J. C. Blanchette and S. Merz, editors, *ITP 2016*, volume 9807 of *LNCS*, pages 184–199. Springer, 2016. ISBN 978-3-319-43143-7. doi: 10.1007/978-3-319-43144-4_12. URL http://dx.doi.org/10.1007/978-3-319-43144-4_12.

[23] J. Jeannin, K. Ghorbal, Y. Kouskoulas, A. Schmidt, R. Gardner, S. Mitsch, and A. Platzer. A formally verified hybrid system for safe advisories in the next-generation airborne collision avoidance system. *STTT*, 2016. doi: 10.1007/s10009-016-0434-1.

[24] C. Kaliszyk and R. O'Connor. Computing with classical real numbers. *J. Formalized Reasoning*, 2(1):27–39, 2009. doi: 10.6092/issn.1972-5787/1411. URL http://dx.doi.org/10.6092/issn.1972-5787/1411.

[25] G. Klein and R. Gamboa, editors. *ITP 2014*, volume 8558 of *LNCS*, 2014. Springer. ISBN 978-3-319-08969-0. doi: 10.1007/978-3-319-08970-6. URL http://dx.doi.org/10.1007/978-3-319-08970-6.

[26] R. Krebbers and B. Spitters. Type classes for efficient exact real arithmetic in Coq. *Logical Methods in Computer Science*, 9(1), 2011. doi: 10.2168/LMCS-9(1:01)2013. URL http://dx.doi.org/10.2168/LMCS-9(1:01)2013.

[27] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In S. Jagannathan and P. Sewell, editors, *POPL'14*, pages 179–192. ACM, 2014. ISBN 978-1-4503-2544-8.

[28] R. Kumar, R. Arthan, M. O. Myreen, and S. Owens. Self-formalisation of higher-order logic - semantics, soundness, and a verified implementation. *J. Autom. Reasoning*, 56(3):221–259, 2016. doi: 10.1007/s10817-015-9357-x. URL http://dx.doi.org/10.1007/s10817-015-9357-x.

[29] L. Lamport. Hybrid systems in TLA$^+$. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *LNCS*, pages 77–102. Springer, 1992. ISBN 3-540-57318-6. doi: 10.1007/3-540-57318-6_25. URL http://dx.doi.org/10.1007/3-540-57318-6_25.

[30] Y. Lin and S. Mitra. Starl: Towards a unified framework for programming, simulating and verifying distributed robotic systems. In S. H. Noh, S. Fischmeister, and J. Xue, editors, *LCTES 2015*, pages 9:1–9:10. ACM, 2015. ISBN 978-1-4503-3257-6. doi: 10.1145/2670529.2754966. URL http://doi.acm.org/10.1145/2670529.2754966.

[31] M. O. Myreen and J. Davis. The reflective Milawa theorem prover is sound - (down to the machine code that runs it). In Klein and Gamboa [25], pages 421–436. ISBN 978-3-319-08969-0. doi: 10.1007/978-3-319-08970-6_27. URL http://dx.doi.org/10.1007/978-3-319-08970-6_27.

[32] M. O. Myreen, S. Owens, and R. Kumar. Steps towards verified implementations of HOL Light. In Blazy et al. [10], pages 490–495. ISBN 978-3-642-39633-5.

[33] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[34] L. C. Paulson. *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, volume 828 of *LNCS*. Springer, 1994. ISBN 3-540-58244-4.

[35] A. M. Pitts. Nominal logic: A first order theory of names and binding. In N. Kobayashi and B. C. Pierce, editors, *TACS 2001*, volume 2215 of *LNCS*, pages 219–242. Springer, 2001. ISBN 3-540-42736-8. doi: 10.1007/3-540-45500-0_11. URL http://dx.doi.org/10.1007/3-540-45500-0_11.

[36] A. Platzer. Differential dynamic logic for hybrid systems. *J. Autom. Reas.*, 41(2):143–189, 2008. ISSN 0168-7433. doi: 10.1007/s10817-008-9103-8.

[37] A. Platzer. Logics of dynamical systems. In *LICS*, pages 13–24. IEEE Computer Society, 2012. ISBN 978-1-4673-2263-8. doi: 10.1109/LICS.2012.13. URL http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6275587.

[38] A. Platzer. A uniform substitution calculus for differential dynamic logic. In Felty and Middeldorp [14], pages 467–481. ISBN 978-3-319-21400-9. doi: 10.1007/978-3-319-21401-6_32. URL http://dx.doi.org/10.1007/978-3-319-21401-6.

[39] A. Platzer. A complete uniform substitution calculus for differential dynamic logic. *J. Autom. Reas.*, 2016. doi: 10.1007/s10817-016-9385-1.

[40] A. Platzer, J.-D. Quesel, and P. Rümmer. Real world verification. In R. A. Schmidt, editor, *CADE*, volume 5663 of *LNCS*, pages 485–501. Springer, 2009. ISBN 978-3-642-02958-5. doi: 10.1007/978-3-642-02959-2_35.

[41] J.-D. Quesel, S. Mitsch, S. Loos, N. Aréchiga, and A. Platzer. How to model and prove hybrid systems with KeYmaera: A tutorial on safety. *STTT*, 18(1):67–91, 2016. doi: 10.1007/s10009-015-0367-0.

[42] V. Rahli and M. Bickford. A nominal exploration of intuitionism. In Avigad and Chlipala [6], pages 130–141. ISBN 978-1-4503-4127-1. doi: 10.1145/2854065.2854077. URL http://doi.acm.org/10.1145/2854065.2854077.

[43] D. Ricketts, G. Malecha, M. M. Alvarez, V. Gowda, and S. Lerner. Towards verification of hybrid systems in a foundational proof assistant. In *MEMOCODE 2015*, pages 248–257. IEEE, 2015. ISBN 978-1-5090-0237-5. doi: 10.1109/MEMCOD.2015.7340492. URL http://dx.doi.org/10.1109/MEMCOD.2015.7340492.

[44] N. Völker. Towards a HOL framework for the deductive analysis of hybrid control systems. In *ADPM2000*, 2000.