Embedding Differential Dynamic Logic in PVS

J Tanner Slagel
NASA Langley Research Center
Hampton, VA, 23666, USA
j.tanner.slagel@nasa.gov

Mariano Moscato
National Institute of Aerospace
Hampton, VA, 23666, USA

Lauren White
NASA Langley Research Center
Hampton, VA, 23666, USA

César Muñoz
NASA Langley Research Center*
Hampton, VA, 23666, USA

Swee Balachandran
National Institute of Aerospace*
Hampton, VA, 23666, USA

Aaron Dutle
NASA Langley Research Center
Hampton, VA, 23666, USA

This paper presents an operational embedding of Differential dynamic logic in the Prototype Verification System (PVS), for the formal verification of hybrid systems. Hybrid systems contain continuously evolving and discretely evolving components, and arise in many safety- and mission-critical applications. Differential dynamic logic (dL) is a framework for formally specifying and reasoning about hybrid programs, where the specification allows modeling of hybrid systems, and its proof calculus allows for reasoning about such potentially complex programs. The embedding of dL described here is an *operational* embedding, meaning that it leverages the internal logic of PVS, resulting in a version of dL whose proof calculus is not only formally verified, but is also executable within PVS itself. This **P**roper**l**y **A**ssured **I**mplementation of **D**ifferential Dynamic Logic for H**y**brid **P**rogram **V**erification and **S**pecification, **Plaidypvs**, supports standard dL style proofs, but further leverages the capabilities of PVS to allow reasoning beyond the scope of traditional dL.

1 Introduction

Formal reasoning about systems that contain both discrete and continuous dynamics, known as *hybrid systems*, has emerged in numerous mission- and safety-critical applications. It is often useful to model hybrid systems as *hybrid programs*, where the discrete variables are given assignments similar to traditional programs, and the continuous variables are defined by a system of differential equations. The boon of hybrid programs (HPs) is that they can model complex dynamics where the continuous and discrete dynamics are largely intertwined, but due to their complexity, efficient and effective formal reasoning about properties of such programs can be a challenge.

Differential dynamic logic (dL), allows specification and reasoning about HPs using a small set of proof rules [45, 47, 53, 55]. Conceptually dL can be split into two parts: a framework for the logical specifications of HPs and their properties, and a proof calculus that is a collection of axioms and rules for the formalized reasoning about these logical specifications. The KeYmaera X[1] theorem prover is a software implementation of dL built up from a small trusted core that assumes the axioms of dL, [18, 29, 25], with a web-based interface for specification and reasoning of HPs [28]. KeYmaera X has been used for formal verification of several cyber-physical systems [21, 27, 23, 7, 6, 19, 26, 33].

This paper focuses on embedding dL in the Prototype Verification System (PVS). PVS is a fully typed functional specification language with an integrated interactive theorem prover based on higher order logic. The prover interface allows users to reason and prove type conditions and user-specified lemmas with proof rules and strategies selected by the proof engineer. The main proof rules are built

*Institute at time of contribution.
[1]KeYmaera X webpage: https://keymaerax.org

Submitted to:
LSFA 2023

into the core of PVS, while strategies are written in a separate strategy language. Strategies only modify the PVS sequent by choosing and calling a sequence of proof rules–so that no additional soundness concerns are introduced by strategies. Users of PVS can use the definitions and lemmas from other PVS specifications and libraries (as long as a logically sound hierarchy of imports is used) to build on top of previously completed work. The largest such library, considered the standard library, is NASA's PVS Library of Formal Developments, NASAlib [2]. NASAlib contains over $35,000$ proven lemmas spanning across 50 folders related to a wide range of topics in mathematics, logic, and computer science.

The primary contribution of this work is a **P**roper**l**y **A**ssured **I**mplementation of **D**ifferential Dynamic Logic for H**y**brid **P**rogram **V**erification and **S**pecification, **Plaidypvs**[3]. Specifically, the contributions of Plaidypvs are:

1. **Specification** of dL-style HPs and their properties through an embedding in the PVS specification language.
2. **Verification** of the deduction rules of dL using the logical and mathematical scaffolding in PVS.
3. **Implementation** of these rules through the strategy language of PVS, resulting in a formally verified and interactive implementation of the proof calculus of dL within PVS.

While reasoning about HPs using a formally verified implementation of dL is already an achievement, the integration in PVS brings additional opportunities for extending the functionality of dL beyond what is available in a stand-alone dL system, like KeYmaera X. For example, new or existing functions and definitions in PVS can be used inside of the dL framework. This includes trigonometric and other transcendental functions already specified in NASAlib, as well as the corresponding properties concerning their derivatives and integrals. In addition, meta-reasoning about HPs and their properties can be performed in PVS using the dL embedding. Examples include specifying HPs with a parametric number of variables, which can be used to reason about situations with an unknown but finite number of actors; and reasoning about entire classes of HPs or relationships between HPs, specified using the type and subtype system in PVS.

The paper proceeds as follows. Section 2 details the formal development of HP specifications in Plaidypvs, while Section 3 gives an overview of the formal verification effort to prove dL in PVS, as well the implementation of the proof calculus of dL in the PVS prover interface. Section 4 shows an example of utilizing the features of Plaidypvs beyond the capabilities of dL alone, while related work is discussed in 5. Finally, conclusions and future work are discussed in 6.

Threaded through the paper is a simple example of a Dubins curve modeling an aircraft turning and then proceeding in a straight line, see Figure 1. This path is first defined implicitly as an HP in Example 2.2, with an invariant property specified in Example 2.3, proven in Examples 3.1, 3.2, and 3.3. Finally, the equivalence between this implicit formulation of the program to its explicit formulation is shown in Example 4.1, showing the reasoning abilities of Plaidypvs beyond a stand-alone implementation of dL.

2 Specification of hybrid programs

This section describes the syntax, semantics, and logical specifications of HPs developed in Plaidypvs. Before these are introduced, a few preliminary concepts are needed.

[2]NASAlib Github https://github.com/nasa/pvslib

[3]Pronounced Platypus. Link: https://github.com/nasa/pvslib/tree/master/dL

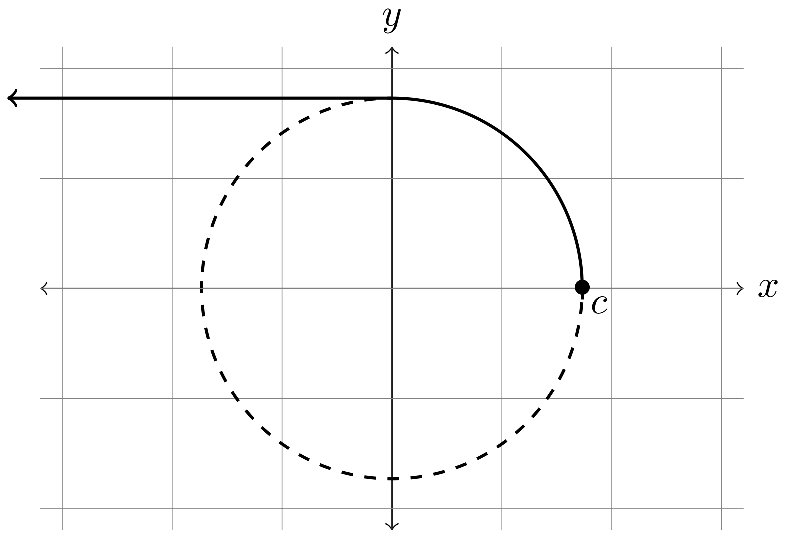


Figure 1: Dubins path modeling an aircraft turning

2.1 Environment, real expressions, Boolean expressions

For an HP, the state of the program at any moment is given by the values of the variables, captured in this development by an environment type $\mathscr{E} : [\mathbb{N} \to \mathbb{R}]$. The representation uses de Bruijn indices where the variables are given as the index to a function, and the value is the function value at the particular index [12].

With variable states $\mathscr{E}$ established, real expressions and Boolean expressions are defined by a shallow embedding meaning they are characterized by their evaluation functions:

$$\mathscr{R} := [\mathscr{E} \to \mathbb{R}], \;\; \mathscr{B} := [\mathscr{E} \to \mathbb{B}].$$

Real expressions are real-valued functions on the environment of variables, for example:

$$\textbf{val}(i) = \lambda(e : \mathscr{E}) : e(i), \;\; \textbf{cnst}(c) = \lambda(e : \mathscr{E}) : c,$$

represent the "value" function that return the ith variable's value, and the "constant" function that returns the value $c \in \mathbb{R}$ regardless of input, respectively. While real expressions can be arbitrary functions, Plaidypvs recognizes some basic combinations as real expressions. Given $r_1, r_2 \in \mathscr{R}$ and $r \in \mathbb{R}_{\geq 0}$ the following are known to be real expressions:

$$r_1 + r_2, \;\; r_1 - r_2, \;\; r_1/r_2, \;\; r_1 \cdot r_2, \;\; \sqrt{r_1}, \;\; r_1^r.$$

Boolean expressions represent predicates on an environment, and Booleans in Plaidypvs, for example:

$$\top = \lambda(e : \mathscr{E}) : \textbf{True}, \;\; \bot = \lambda(e : \mathscr{E}) : \textbf{False},$$

are the Boolean expressions representing "True" and "False", respectively, using the existing PVS Booleans **True** and **False**. Arbitrary Boolean functions can be specified, but similar to real expressions, the common propositional logical operators are defined to allow combining Boolean expressions. Given Boolean expressions $b_1, b_2 \in \mathscr{B}$, the following are all recognized as Boolean expressions:

$$b_1 \wedge b_2,\ b_1 \vee b_2,\ b_1 \rightarrow b_2,\ b_1 \leftrightarrow b_2,\ \neg b_1.$$

The use of basic real and Boolean expressions and their combinators assists in some automation aspects of the system, but the ability for users to define arbitrary custom expressions is one of the unique features of the Plaidypvs implementation of dL (see Section 4).

Example 2.1 (Environments, Real and Boolean Expressions) *For $x = 0,\ y = 1$ and $c \in \mathbb{R}_{\geq 0}$, define the environment $e = \lambda(i : \mathbb{N})0$ **with** $x \mapsto c/2, y \mapsto \sqrt{3}c/2$. The real expressions $\boldsymbol{val}(x)$, $\boldsymbol{val}(y)$, $\boldsymbol{cnst}(c)$, can be used to define the Boolean expression corresponding to a circle of radius c centered at $(0,0)$:*

$$\boldsymbol{val}(x)^2 + \boldsymbol{val}(y)^2 = \boldsymbol{cnst}(c)^2.$$

Furthermore

$$(\boldsymbol{val}(x)^2 + \boldsymbol{val}(y)^2 = \boldsymbol{cnst}(c)^2)(e) = \boldsymbol{True}.$$

The form of the expressions is similar to the actual implementation in PVS, and highlights the fact that these are indeed real-valued functions of an environment. However, for ease of presentation, the **val** and **cnst** notation is suppressed in much of the remainder of the paper. The Boolean expression above, for example, will be presented instead as $x^2 + y^2 = c^2$.

2.2 Hybrid programs

Hybrid programs are syntactically defined as a datatype $\mathscr{H}$ in PVS according to the grammar

$$\alpha ::= \mathbf{x} := \ell \mid \mathbf{x}' = \ell \& P \mid ?P \mid x := * \,\& qP \mid \alpha_1 ; \alpha_2 \mid \alpha_1 \cup \alpha_2 \mid \alpha_1^*.$$

Here, $\mathbf{x} := \ell$ is a list of elements of $\mathbb{N} \times \mathscr{R}$ where the first entries are unique, intended to represent a discrete assignment of the variables indexed by these first elements. The expression $\mathbf{x}' = \ell$ is another such list, and $P \in \mathscr{B}$ is a Boolean expression. The differential equation $\mathbf{x}' = \ell \& P$ is meant to symbolize the continuous evolution of the variables in $\mathbf{x}'$ according to the first order differential equation described by ℓ, while guaranteeing that the solution satisfies P along the evolution. To reference a variable used in a discrete assignment or differential equation the notation $i \in \mathbf{x}$ ($i \in \mathbf{x}'$) will be used, and the analogous real expression associated with i will be denoted $\ell(i)$. The expression $?P$ represents a check of the Boolean expression P. In $x := * \,\& qP$, $qP \in [\mathbb{R} \to \mathscr{B}]$ is a Boolean expression with one free real variable, and the expression is meant to arbitrarily (discretely) assign the variable x a real value r such that $qP(r)$ holds. Note that use of the symbol & is distinct from Boolean conjunction, and is used here purely syntactically. The expression $\alpha_1 ; \alpha_2$ represents sequential execution of the sub-programs, while $\alpha_1 \cup \alpha_2$ symbolizes an nondeterministic choice between two subprograms. Finally, α_1^* represents repetition of a HP a fixed but unknown (possibly zero) number of times.

Formally, the function **s_rel** defines the semantic relation for a hybrid program. For environments

$e_i, e_o \in \mathscr{E}$, $\mathbf{s\_rel}(\alpha)(e_i)(e_o)$ is true exactly when:

$$\begin{cases} \forall k : k \notin \mathbf{x} \to e_o(k) = e_i(k) & \text{if } \alpha = (\mathbf{x} := \ell), \\ \quad \wedge k \in \mathbf{x} \to e_o(k) = \ell(k)(e_i) & \\ \exists D : \mathbf{s\_rel\_diff}(D, \mathbf{x}', \ell, P, e_i, e_o) & \text{if } \alpha = (\mathbf{x}' = \ell \,\&\, P), \\ e_o = e_i \wedge P(e_i) & \text{if } \alpha = ?P, \\ \exists r : e_o(x) = r \wedge Q(r)(e_i) & \text{if } \alpha = (x := * \,\&\, Q), \\ \exists e : \mathbf{s\_rel}(\alpha_1)(e_i)(e) & \text{if } \alpha = \alpha_1 ; \alpha_2, \\ \quad \wedge \mathbf{s\_rel}(\alpha_2)(e)(e_o) & \\ \mathbf{s\_rel}(\alpha_1)(e_i)(e_o) & \text{if } \alpha = \alpha_1 \cup \alpha_2, \\ \vee \mathbf{s\_rel}(\alpha_2)(e_i)(e_o) & \\ e_o = e_i \vee & \text{if } \alpha = \alpha_1^*. \\ \exists e : \mathbf{s\_rel}(\alpha_1)(e_i)(e) & \\ \quad \wedge \mathbf{s\_rel}(\alpha)(e)(e_o) & \end{cases}$$

The correspondence between the informal description of semantics and the **s_rel** function is fairly standard in all cases except the differential equation branch. For differential equations, the domain D is $\mathbb{R}_{\geq 0}$, or some closed interval starting at 0, and the semantics is given by the function

$$\begin{aligned} \mathbf{s\_rel\_diff}(D, \mathbf{x}', \ell, P, e_i, e_o) = {} & \exists r, \exists! f : D(r) \wedge \mathbf{sol?}(D, \mathbf{x}', \ell, e_i)(f) \wedge \\ & e_o = \mathbf{e\_at\_t}(\mathbf{x}', \ell, f, e_i)(r) \wedge \\ & \forall t : (D(t) \wedge t \leq r) \\ & \quad \to P(\mathbf{e\_at\_t}(\mathbf{x}', \ell, f, e_i)(t)). \end{aligned}$$

Unpacking this further,

$$\mathbf{e\_at\_t}(\mathbf{x}', \ell, f, e_i) = \lambda(r : \mathbb{R}) \lambda(j : \mathbb{N}) \begin{cases} e_i(j) & \text{if } j \notin \mathbf{x}', \\ f(j)(r) & \text{if } j \in \mathbf{x}', \end{cases}$$

is a function that characterizes the environment e_i, with the continuously evolving variables $\mathbf{x}'$ replaced by values from a function $f : [\mathbb{R}^k \to [\mathbb{R} \to \mathbb{R}]]$. The definition

$$\begin{aligned} \mathbf{sol?}(D, \mathbf{x}', \ell, e_i)(f) = {} & \forall (i \in \mathbf{x}', t \in \mathbf{D}) : \\ & (f(i))'(t) = \ell(i)(\mathbf{e\_at\_t}(\mathbf{x}', \ell, f, e_i)(t)) \end{aligned}$$

ensures that f is the solution to the k-dimensional differential equation $\mathbf{x}' = \ell$ throughout the domain D. Note in the definition of **s_rel_diff** this solution f is further assumed to be unique on the domain D.

Example 2.2 (HP) *For $x = c$, $y = 0$, and $c \in \mathbb{R}_{\geq 0}$, the HP*

$$\begin{gathered} ((?(x > 0); (x' = -y, y' = x, \& x \geq 0)) \cup \\ (?(x \leq 0); (x' = -c, y' = 0)))^*, \end{gathered}$$

represents the dynamics where x and y progress according to the differential equation $x' = -y$, $y' = x$ when $x > 0$, but when $x \leq 0$ the variables progress according to the differential equation $x' = -c$, $y' = 0$. Note that the test $?$ statements determine which branch of $\cup$ in the HP is applicable, and the domain $x \geq 0$ in the first differential equation prevents the dynamics from continuing when $x = 0$, forcing the other branch of the HP to take place. The $*$ allows repetition to happen in order to carry out both branches of the dynamics.

2.3 Quantified statements about hybrid programs

Notice that an HP, unlike a traditional deterministic and discrete assignment program, can have potentially many different executions, or *runs*. This means that given an input environment e_i, there may be infinitely many output environments e_o semantically related to it (by repetition, random assignment, etc.). To reason about these, universal and existential quantifiers over the potentially infinite number of executions of an HP are denoted by allruns $[\cdot]$ and someruns $\langle\cdot\rangle$ respectively. For $\alpha \in \mathscr{H}$ and $P \in \mathscr{B}$ $[\alpha]P \in \mathscr{B}$ is defined as

$$\lambda(e_i : \mathscr{E})\forall e_o : \mathbf{s\_rel}(\alpha)(e_i)(e_o) \rightarrow P(e_o),$$

and $\langle\alpha\rangle P$ is defined as

$$\lambda(e_i : \mathscr{E})\exists e_o : \mathbf{s\_rel}(\alpha)(e_i)(e_o) \wedge P(e_o).$$

These say that every (some, respectively) run of the HP α starting at environment e_i satisfies P.

Example 2.3 (Allruns) *Let α be the HP in Example 2.2, $\boldsymbol{circ}(c) = x^2 + y^2 = c^2$ and*

$$\boldsymbol{path}(c) = (x > 0 \rightarrow \boldsymbol{circ}(c)) \wedge (x \leq 0 \rightarrow y = c).$$

Then

$$(x = c \wedge y = 0) \rightarrow [\alpha]\boldsymbol{path}(c),$$

is the Boolean expression stating that if the value of x is c and the value of y is 0*, then for all runs of the HP α, the values of x and y stay inside* $\boldsymbol{path}(c)$*. In other words, x and y stay on the circle of radius c until $x = 0$ and then stay on the line $y = c$.*

3 Embedding differential dynamic logic

With the formal specification of hybrid programs established, the embedding of the sequent calculus of dL in PVS can be discussed. First the dL-sequent will be defined, then a description of the formal verification process encoding the axioms and rules of dL as lemmas in PVS is provided.

3.1 The dL-sequent

The dL-sequent in PVS is defined by a function $\vdash$, which takes in two lists of Boolean expressions Γ and Δ, known as the dL-antecedent and dL-consequent respectively

$$\Gamma \vdash \Delta,$$

and returns the Boolean value

$$\forall e \in \mathscr{E}, \bigwedge \Gamma(e) \implies \bigvee \Delta(e),$$

where $\implies$ is the PVS implication. Intuitively, this means that the conjunction of the antecedent formulas implies the disjunction of the consequent formulas.

The dL method for proving a statement about a hybrid program is to use the defined rules of dL to manipulate (sometimes producing multiple branches) the sequent so that the conjunction of resulting sequent implies the original. To accomplish this in a formally verified way, each rule of dL is specified as a PVS lemma, which takes essentially the following form.

$$\textbf{notR}\quad \frac{\Gamma, P \vdash \Delta}{\Gamma \vdash \neg P, \Delta}$$

$$\textbf{notL}\quad \frac{\Gamma \vdash P, \Delta}{\Gamma, \neg P \vdash \Delta}$$

$$\textbf{andR}\quad \frac{\Gamma \vdash P, \Delta \quad \Gamma \vdash Q, \Delta}{\Gamma \vdash P \wedge Q, \Delta}$$

$$\textbf{andL}\quad \frac{\Gamma, P, Q \vdash \Delta}{\Gamma, P \wedge Q \vdash \Delta}$$

$$\textbf{orR}\quad \frac{\Gamma \vdash P, Q, \Delta}{\Gamma \vdash P \vee Q, \Delta}$$

$$\textbf{orL}\quad \frac{\Gamma, P \vdash \Delta \quad \Gamma, Q \vdash \Delta}{\Gamma, P \vee Q \vdash \Delta}$$

$$\textbf{cut}\quad \frac{\Gamma \vdash C, \Delta \quad \Gamma, C \vdash \Delta}{\Gamma \vdash \Delta}$$

$$\textbf{weakR}\quad \frac{\Gamma \vdash P, \Delta \quad P \vdash Q}{\Gamma \vdash Q, \Delta}$$

$$\textbf{impliesR}\quad \frac{\Gamma, P \vdash Q, \Delta}{\Gamma \vdash P \rightarrow Q, \Delta}$$

$$\textbf{impliesL}\quad \frac{\Gamma \vdash P, \Delta \quad \Gamma, Q \vdash \Delta}{\Gamma, P \rightarrow Q \vdash \Delta}$$

$$\textbf{iffR}\quad \frac{\Gamma, P \vdash Q, \Delta \quad \Gamma, Q \vdash P, \Delta}{\Gamma \vdash P \leftrightarrow Q, \Delta}$$

$$\textbf{iffL}\quad \frac{\Gamma, P \wedge Q \vdash \Delta \quad \Gamma, \neg P \wedge \neg Q \vdash \Delta}{\Gamma, P \leftrightarrow Q \vdash \Delta}$$

$$\textbf{falseL}\quad \frac{}{\Gamma, \bot \vdash \Delta}$$

$$\textbf{trueR}\quad \frac{}{\Gamma \vdash \top, \Delta}$$

$$\textbf{axiom}\quad \frac{}{\Gamma, P \vdash P, \Delta}$$

$$\textbf{weakL}\quad \frac{P, \Gamma \vdash \Delta \quad Q \vdash P}{\Gamma, Q \vdash \Delta}$$

Figure 2: Propositional dL rules

Lemma <dL-rule-name> *For all lists of Boolean expressions* Γ, Δ*, and predicate* A *on* Γ, Δ*,*

$$\left(A(\Gamma, \Delta) \wedge \bigwedge_{i=1}^{k} \Gamma_i \vdash \Delta_i \right) \implies \Gamma \vdash \Delta.$$

In the above specification, A represents a way to ensure that the rule applies to the original sequent. The sequents $\{\Gamma_i \vdash \Delta_i\}_{i=1}^{k}$ represent a transformation of the original antecedent and consequent into the result of applying the rule being specified. For some rules of dL, the specification and proof of these lemmas are simple (e.g., propositional logic rules). Some can be difficult to *specify* due to the need to parse the structure of a hybrid program to identify if a rule applies and to rewrite particular parts for the rule transformation. Finally, some of the rules are difficult to *prove* because they rely on applying mathematics that either is complex or had to be developed in PVS to support this functionality.

With such a lemma proven in PVS, a user can bring the lemma into a proof environment and instantiate each of the expressions needed to use the rule. In order to automate this, these lemmas are developed into *strategies* in PVS. These strategies have the ability to parse the current sequent, identify instantiations that apply, hide unneeded formulas, and prove type check conditions that may appear, among other capabilities. Further batch strategies can employ several of these atomic strategies at once to simplify the proof process. Some of these rules, including details about their specification, verification, and implementation as strategies in PVS, are discussed below. A Plaidypvs "cheat sheet" is available for users with the development.[4]

3.2 Basic logical and structural rules of dL

Many logical rules allow manipulations of the dL-sequent. For example, the rule **impliesR** allows an implication in the dL-consequent, $P \rightarrow Q$, to be simplified to P in the dL-antecedent and Q in the dL-consequent.

[4]Plaidypvs cheat sheet: https://github.com/nasa/pvslib/tree/master/dL/cheatsheet.pdf

$$\textbf{existsR}\quad \frac{\Gamma \vdash p(e), \Delta}{\Gamma \vdash \exists x : p(x), \Delta}\ (\text{any } e)$$

$$\textbf{forallL}\quad \frac{\Gamma, p(e) \vdash \Delta}{\Gamma, \forall x : p(x) \vdash \Delta}\ (\text{any } e)$$

$$\textbf{forallR}\quad \frac{\Gamma \vdash p(y), \Delta}{\Gamma \vdash \forall x : p(x), \Delta}\ (y \text{ Skolem symbol})$$

$$\textbf{existsL}\quad \frac{\Gamma, p(y) \vdash \Delta}{\Gamma, \exists x : p(x) \vdash \Delta}\ (y \text{ Skolem symbol})$$

Figure 3: Quantifier dL rules

$$\textbf{existsR}\quad \frac{\Gamma \vdash p(e), \Delta}{\Gamma \vdash \exists x : p(x), \Delta}\ (\text{any } e)$$

$$\textbf{forallL}\quad \frac{\Gamma, p(e) \vdash \Delta}{\Gamma, \forall x : p(x) \vdash \Delta}\ (\text{any } e)$$

$$\textbf{forallR}\quad \frac{\Gamma \vdash p(y), \Delta}{\Gamma \vdash \forall x : p(x), \Delta}\ (y \text{ Skolem symbol})$$

$$\textbf{existsL}\quad \frac{\Gamma, p(y) \vdash \Delta}{\Gamma, \exists x : p(x) \vdash \Delta}\ (y \text{ Skolem symbol})$$

Figure 4: Structural dL rules

$$\textbf{impliesR}\quad \frac{\Gamma, P \vdash Q, \Delta}{\Gamma \vdash P \rightarrow Q, \Delta.}$$

Here $\Gamma \vdash P \rightarrow Q, \Delta$ is the dL-sequent that **impliesR** can be applied to, and $\Gamma, P \vdash Q, \Delta$ is the result. Note that the standard logical notation being used for **impliesR** above is for ease of presentation, whereas the PVS specification of such a rule, generally hidden from a user by a strategy, is closer to that described in Section 3.1. There are a number of propositional rules in dL similar to **impliesR** which allow manipulation of the basic logical connectives ($\wedge$, $\vee$, $\neg$, $\rightarrow$, $\iff$) and operators ($\top$, $\bot$) in the dL-sequent, see Figure 2. Additionally, there are quantifier rules for Skolemization and instantiation in the dL sequent, see Figure 3, and there are structural rules that allow expressions to be moved or hidden, see Figure 4.

The proofs of the lemmas for logical and structural rules largely follow from the analogous logical properties in PVS. Each of the individual rules are implemented as strategies, but the batch strategies are generally simpler to apply. For example, **flatten** repeatedly applies propositional rules which disjunctively simplify a dL sequent, separating implications using the **impliesR** rule, flattening conjunctions in the antecedent, and flattening disjunctions in the consequent. A list of batch commands in Plaidypvs is given in Figure 7.

Example 3.1 (dL-sequent example) *Attempting to prove the validity of the expression from Example 2.3, the sequent begins as:*

$$\vdash (x = c \wedge y = 0) \rightarrow [\alpha]\boldsymbol{path}(c).$$

Invoking the rule ***flatten*** *to the sequent above applies* ***impliesR*** *and* ***andL****, which separates conjunctions in the antecedent, producing the following sequent:*

$$x = c, y = 0 \vdash [\alpha]\boldsymbol{path}(c). \qquad (1)$$

boxd	$\langle\alpha\rangle P \leftrightarrow \neg[\alpha]\neg P$
assignb	$[\mathbf{x} := \ell]\, P = \mathbf{SUB}(\mathbf{x} := \ell)(P)$
assignd	$\langle \mathbf{x} := \ell\rangle\, P = \mathbf{SUB}(\mathbf{x} := \ell)(P)$
testb	$[?Q]\, P = Q \rightarrow P$
testd	$\langle ?Q\rangle\, P = Q \wedge P$
choiceb	$[\alpha_1 \cup \alpha_2]\, P \leftrightarrow [\alpha_1]\, P \wedge [\alpha_2]\, P$
choiced	$\langle\alpha_1 \cup \alpha_2\rangle\, P \leftrightarrow \langle\alpha_1\rangle\, P \vee \langle\alpha_2\rangle\, P$
composeb	$[\alpha_1 ; \alpha_2]\, P \leftrightarrow [\alpha_1]\,[\alpha_2]\, P$
composed	$\langle\alpha_1 ; \alpha_2\rangle\, P \leftrightarrow \langle\alpha_1\rangle\,\langle\alpha_2\rangle\, P$
iterateb	$[\alpha^*]\, P = P \wedge [\alpha]\,[\alpha^*]\, P$
iterated	$\langle\alpha^*\rangle\, P = P \vee \langle\alpha\rangle\,\langle\alpha^*\rangle\, P$
anyb	$[x := * \,\&\, Q(x)]\, P(x) = \forall x : Q(x) \rightarrow P(x)$
anyd	$\langle x := * \,\&\, Q(x)\rangle\, P(x) = \exists x : Q(x) \wedge P(x)$

Figure 5: Hybrid program rewrites.

3.3 Hybrid program rewrites and rules

While the rules in Section 3.2 manipulate the logical structure of a dL-sequent, further rules act on the hybrid program components of such a sequent. Properties given in Figure 5 allow direct rewriting of hybrid programs. Other rules about hybrid programs in a sequent are given in Figure 6. The majority of these manipulate the allruns $[\cdot]$ or someruns $\langle\cdot\rangle$ operators, and the proofs were largely concerned with reasoning about the semantic relation function, **s_rel**.

In addition to each of these rules becoming strategies, the batch strategy **assert** uses all of the hybrid program rewrites in Table 5 to simplify an expression.

There are a few intricacies worth mentioning in the formal verification and implementation of these rewrites and rules in PVS. In the rewrites **assignb** and **assignd**, an allruns or someruns of an assignment HP is equated to a substitution. Substitution is defined at the environment level, where **assign_sub**$(\mathbf{x} := \ell)(e)$ is the environment such that

$$\mathbf{assign\_sub}(\mathbf{x} := \ell)(e)(i) = \begin{cases} \ell(i)(e) & \text{if } i \in \mathbf{x} \\ e(i) & \text{if } i \notin \mathbf{x}. \end{cases} \tag{2}$$

Substitution of a general Boolean expression is therefore defined

$$\mathbf{SUB}(\mathbf{x} := \ell)(P) = \lambda(e : \mathscr{E})\, P(\mathbf{assign\_sub}(\mathbf{x} := \ell)(e)).$$

While the definition of substitution above applies to any Boolean expression P and can be reasoned about by a user of Plaidypvs, the standard level of manipulation in dL is not often at the environment level. To increase the level of automation, a number of rewrites for reducing expressions containing **SUB** have been implemented. This led to formally verifying substitution properties for real expressions, inequalities of real expressions, and hybrid programs, so that a substitution at the top level of an expression could be pushed down to the level of **val** and **cnst**, where atomic substitutions are applied. The implementation of these rules required a calculus for reducing the substitution down to atomic expressions, written in the strategy language of PVS. This allows the **assignb** and **assignd** strategies to automatically compute a

$$\textbf{Mb}\ \frac{\vdash P \to Q}{\Gamma \vdash [\alpha]P \to [\alpha]Q, \Delta} \qquad \textbf{Gb}\ \frac{\vdash P}{\Gamma \vdash [\alpha]P, \Delta}$$

$$\textbf{Md}\ \frac{\vdash P \to Q}{\Gamma \vdash \langle\alpha\rangle P \to \langle\alpha\rangle Q, \Delta} \qquad \textbf{Gd}\ \frac{\vdash \langle\alpha\rangle\top \quad \vdash P}{\Gamma \vdash \langle\alpha\rangle P, \Delta}$$

$$\textbf{K}\ \frac{\Gamma \vdash [\alpha](P \to Q), \Delta}{\Gamma \vdash [\alpha]P \to [\alpha]Q, \Delta} \qquad \textbf{VRb}\ \frac{\Gamma \vdash P, \Delta}{\Gamma \vdash [\alpha]P, \Delta}\ \textbf{fresh?}(P)(\alpha)$$

$$\textbf{loop}\ \frac{\Gamma \vdash J, \Delta \quad J \vdash [\alpha]J \quad J \vdash P}{\Gamma \vdash [\alpha^*]P, \Delta} \qquad \textbf{VRd}\ \frac{\vdash \langle\alpha\rangle\top \ \Gamma \vdash P, \Delta}{\Gamma \vdash \langle\alpha\rangle P, \Delta}\ \textbf{fresh?}(P)(\alpha)$$

$$\textbf{mbR}\ \frac{\Gamma \vdash [\alpha]Q, \Delta \quad Q \vdash P}{\Gamma \vdash [\alpha]P, \Delta} \qquad \textbf{mdR}\ \frac{\Gamma \vdash \langle\alpha\rangle Q, \Delta \quad Q \vdash P}{\Gamma \vdash \langle\alpha\rangle P, \Delta}$$

$$\textbf{mbL}\ \frac{\Gamma, [\alpha]Q \vdash \Delta \quad P \vdash Q}{\Gamma, [\alpha]P \vdash \Delta} \qquad \textbf{mdL}\ \frac{\Gamma, \langle\alpha\rangle Q \vdash \Delta \quad P \vdash Q}{\Gamma, \langle\alpha\rangle P \vdash \Delta}$$

$$\textbf{ghost}\ \frac{\Gamma \vdash [y := e]P, \Delta}{\Gamma \vdash P, \Delta}\ \textbf{fresh?}(P)(y)$$

Figure 6: Hybrid program rules.

substitution for any propositional expression composed of equalities and inequalities of polynomial real expressions. For example the substitution

$$\textbf{SUB}(x := y, y := 10)(x^2 + y^2 = 11),$$

is transformed automatically into

$$y^2 + 10^2 = 11.$$

The individual steps of this are a number of lemma applications in PVS:

$$\begin{aligned}
\textbf{SUB}(x := y, y := 10)(x^2 + y^2 = 11) &= \big(\textbf{SUB\_re}(x := y, y := 10)(x^2 + y^2) = \textbf{SUB\_re}(x := y, y := 10)(11)\big)\\
&= \big(\textbf{SUB\_re}(x := y, y := 10)(x^2) + \textbf{SUB\_re}(x := y, y := 10)(y^2) = 11\big)\\
&= \big(\textbf{SUB\_re}(x := y, y := 10)(x)^2 + \textbf{SUB\_re}(x := y, y := 10)(y)^2 = 11\big)\\
&= y^2 + 10^2 = 11,
\end{aligned}$$

where **SUB_re** is substitution defined on real expressions $r \in \mathscr{R}$ as:

$$\textbf{SUB\_re}(\mathbf{x} := \ell)(r) = \lambda(e : \mathscr{E})\, r(\textbf{assign\_sub}(\mathbf{x} := \ell)(e)).$$

The automated substitution of more general Boolean expressions (for example, a statement of the form $[\alpha]P$) is still incomplete in Plaidypvs, and an area of future work.

Another challenge in formal verification occurs in some hybrid program rules. The **ghost**, **VRb**, and **Vrd** rules require the concept of *freshness*. A fresh variable y is defined as

$$\textbf{fresh?}(P)(y) = \forall e \in \mathscr{E},\, r \in \mathbb{R},\, P(e) = P(e \text{ with } y \mapsto r)].$$

In other words, the value of the Boolean expression P does not depend on the value of the variable y. Analogous definitions exist to express that a variable is fresh relative to a real expression or a hybrid program. Furthermore, an entire hybrid program can be checked for freshness relative to a Boolean expression

$$\mathbf{fresh?}(P)(\alpha)=\begin{cases}\forall k\in\mathbf{x}\ \ \mathbf{fresh?}(P)(k) & \text{if } \alpha=(\mathbf{x}:=\ell),\\ \forall k\in\mathbf{x}'\ \ \mathbf{fresh?}(P)(k) & \text{if } \alpha=(\mathbf{x}'=\ell\,\&\,Q),\\ \mathbf{True} & \text{if } \alpha=?Q,\\ \mathbf{fresh?}(P)(x) & \text{if } \alpha=(x:=*\,\&\,Q),\\ \mathbf{fresh?}(P)(\alpha_1)\wedge\mathbf{fresh?}(P)(\alpha_2) & \text{if } \alpha=\alpha_1;\alpha_2,\\ \mathbf{fresh?}(P)(\alpha_1)\wedge\mathbf{fresh?}(P)(\alpha_2) & \text{if } \alpha=\alpha_1\cup\alpha_2,\\ \mathbf{fresh?}(P)(\alpha) & \text{if } \alpha=\alpha_1^*.\end{cases} \tag{3}$$

Note that the recursive definition of freshness above ensures the value of P does not change for any run of the hybrid program α by checking if all the variables potentially changing in α are fresh relative to P.

The need for generation of a fresh variable, as in **ghost**, requires a mechanism for producing a fresh variable (i.e., the smallest natural number in a dL-sequent not being used as a variable index), and it also requires a method for automatically proving freshness of the variable in PVS. This was completed in the strategy language of PVS that utilizes a number of rewrites related to freshness proven in PVS as lemmas.

Example 3.2 (dL-sequent example continued) *Expanding α in the sequent from Example 3.1, eq. 1 and using* ***loop*** *with $J=(\boldsymbol{path}(c)\wedge y\geq 0)$ produces three subgoals,*[5] *one of which is*

$$\begin{aligned}\boldsymbol{path}(c),y\geq 0\vdash\big[(?(x>0);(x'=-y,y'=x,\&x\geq 0))\cup\\ (?(x\leq 0);(x'=-c,y'=0))\big]\boldsymbol{path}(c)\wedge y\geq 0.\end{aligned}$$

Using ***assert*** *to simplify with hybrid program rewrites and applying propositional simplifications with the batch command* ***ground****, the result is the following two cases:*

$$\begin{aligned}(x>0,\boldsymbol{circ}(c),y\geq 0)\vdash\big[x'=-y,y'=x,\&x\geq 0))\big]\boldsymbol{path}(c)\wedge y\geq 0,\\ (x\leq 0,y=c)\vdash\big[(x'=-c,y'=0)\big]\boldsymbol{path}(c)\wedge y\geq 0.\end{aligned} \tag{4}$$

3.4 Rules for differential equations

The rules for differential equations are given in Figure 8. The differential equation rules required significant mathematical underpinnings to be added to PVS for their formal verification. For the implementation of the **dI** rule, a calculus to automatically compute the derivative of a Boolean expression P was necessary. To do this, an embedding of Boolean expressions was developed as a data type, with the grammar:

$$b ::= b_1\wedge_{\mathbf{nqB}}b_2\mid b_1\vee_{\mathbf{nqB}}b_2\mid\neg_{\mathbf{nqB}}b_1\mid\mathbf{rel}_{\mathbf{nqB}}(r_1,r_2),$$

where $\mathbf{rel}_{\mathbf{nqB}}$ is of type **NQB_rel** which is itself an embedding of the inequality operators:

$$\mathbf{rel}_{\mathbf{nqB}} ::= \ \leq_{\mathbf{nqB}}\ \mid\ \geq_{\mathbf{nqB}}\ \mid\ <_{\mathbf{nqB}}\ \mid\ >_{\mathbf{nqB}}\ \mid\ =_{\mathbf{nqB}}\ \mid\ \neq_{\mathbf{nqB}}\ .$$

[5]The other two dL-sequents generated can be proven easily. For full details of the examples in this paper, see the PVS implementation at https://github.com/nasa/pvslib/tree/master/dL/examples

flatten	Disjunctively simplifies the dL sequent by applying **trueR**, **falseR**, **orR**, **impliesR**, **notR**, **axiom**, **falseL**.
ground	Disjunctively and conjunctively simplifies the dL sequent by applying **flatten** and additional splitting lemmas **andR**, **orL**, and **impliesL**.
inst	Instantiates a universal quantifier in the dL-antecedent by applying **forallL** or an existential quantifier in the dL-consequent by applying **existsL**.
skolem	Skolemizes an existential quantifier in dL-antecedent by applying **existsR** or an universal quantifier in the dL-consequent by applying **forallR**.
grind	Repeatedly uses **ground** and **skolem**, and a number of rewrites related to real expressions. This strategy has the option to use the MetitTarski automatic theorem prover as an outside oracle to discharge the proof if possible.
assert	Repeatedly applies hybrid program rewrites in Figure 5.

Figure 7: Batch dL commands

With this structure, the derivative b' of a Boolean expression b is defined as:

$$\begin{cases} b_1' \wedge b_2' & \text{if } b = b_1 \wedge_{\mathbf{nqB}} b_2 \text{ or } b = b_1 \vee_{\mathbf{nqB}} b_2 \\ r_1' \leq r_2' & \text{if } b = r_1 \leq_{\mathbf{nqB}} r_2 \text{ or } b = r_1 <_{\mathbf{nqB}} r_2 \\ r_1' \geq r_2' & \text{if } b = r_1 \geq_{\mathbf{nqB}} r_2 \text{ or } b = r_1 >_{\mathbf{nqB}} r_2 \\ r_1' = r_2' & \text{if } b = \left(r_1 =_{\mathbf{nqB}} r_2\right) \text{ or } b = \left(r_1 \neq_{\mathbf{nqB}} r_2\right). \end{cases}$$

In the PVS implementation of this, $[x' := f(x)]\,(P)'$ is computed in a single step, where P is replaced by a equivalent non-quantified Boolean, and the derivative of any real expression r occurring in P is the real expression given by:

$$r' = \sum_{i \in \mathbf{x}} \partial r_i \cdot \ell(i).$$

This is the derivative of the real expression r in terms of the explicit variable that all the variables in $\mathbf{x}$ are a function of. To arrive at this formulation, differentiability and partial differentiability had to be defined for real expressions as well as the multivariate chain rule.

For the Differential Ghost rule **dG**, adding an equation to the differential equation $x' = \ell$ required that the new differential equation $x' = \ell, y' = a(x) \cdot y + b(x)$ had a unique solution. The Picard-Lindelöff theorem can be used to show that if a and b are continuous on Q, then there is a unique solution to $y' = a(x) \cdot y + b(x)$. Given a solution to $x' = \ell$ that is contained in Q, it follows that $x' = \ell, y' = a(x) \cdot y + b(x)$ has a unique solution. These properties of differential equations, including the Picard-Lindelöff theorem, were developed in PVS specifically to prove these rules.

Example 3.3 (dL-sequent example continued) *Applying **dC** with $C = \textbf{circ}(c)$ to the first branch of the proof in Example 3.2, eq. 4 produces two subgoals, the first of which (with expanded **circ**) is*

$$(x > 0, x^2 + y^2 = c^2, y \geq 0) \vdash \left[x' = -y, y' = x, \& x \geq 0))\right] (x^2 + y^2 = c^2).$$

Using **dI** reduces to two cases:

$$\begin{gathered} x \geq 0 \vdash (2 \cdot x \cdot -y + 2 \cdot y \cdot x = 0) \\ (x \geq 0, x^2 + y^2 = c^2, y \geq 0) \vdash x^2 + y^2 = c^2, \end{gathered} \tag{5}$$

$$\textbf{dinit}\quad \frac{\Gamma, Q \vdash [x' = f(x)\,\&\, Q]\,P, \Delta}{\Gamma \vdash [x' = f(x)\,\&\, Q]\,P, \Delta}$$

$$\textbf{dW}\quad \frac{Q \vdash P}{\Gamma \vdash [x' = f(x)\,\&\, Q]\,P, \Delta}$$

$$\textbf{dI}\quad \frac{\Gamma, Q \vdash P, \Delta \;\; Q \vdash [x' := f(x)]\,(P)'}{\Gamma \vdash [x' = f(x)\,\&\, Q]\,P, \Delta}$$

$$\textbf{dC}\quad \frac{\Gamma \vdash [x' = f(x)\,\&\, Q]\,C, \Delta \quad \Gamma \vdash [x' = f(x)\,\&\,(Q \wedge C)]\,P, \Delta}{\Gamma \vdash [x' = f(x)\,\&\, Q]\,P, \Delta}$$

$$\textbf{dG}\quad \frac{\Gamma \vdash G,\, G \vdash P,\, \Gamma \vdash \exists y\,[x' = f(x),\, y' = a(x)\cdot y + b(x)\,\&\, Q]\,G, \Delta}{\Gamma \vdash [x' = f(x)\,\&\, Q]\,P, \Delta}\;\textbf{fresh?}(y)$$

$$\textbf{dS}\quad \frac{\Gamma \vdash \forall t \geq 0\,(\forall 0 \leq s \leq t\, Q(y(s))) \to [x := y(t)]\,P}{\Gamma \vdash [x' = f(x)\,\&\, Q]\,P}$$

Figure 8: Differential Equation Rules. For **dG**, a and b are continuous on Q, and y is fresh relative to $x' = f(x)$, Q, a, b, P, Γ and Δ.

both of which can be proven with basic algebraic and logical simplifications included in batch command **grind**.

4 Using Plaidypvs

Plaidypvs has the functionality of dL within the PVS environment. Numerous examples of this can be found in the examples directory of the Plaidypvs library. Figures 9 and 10 illustrate using dL for specification and verification of hybrid systems in Plaidypvs. However, Plaidypvs is not limited to just these applications, the embedding allows additional features to be used for formal reasoning of hybrid programs. For example, the definition of other functions from PVS libraries can be imported into an Plaidypvs file, and meta-properties about hybrid programs can be specified and proven. The example below illustrates these points and has been implemented in Plaidypvs.

Example 4.1 (Verified connection to Dubins paths) *The following example shows the capability of Plaidypvs to allow meta reasoning about hybrid programs that other implementations of* dL*, like KeYmaera X, cannot perform. An aircraft moving at a constant speed $c > 0$, with a turn rate of 1 can be modeled by a Dubins path:*

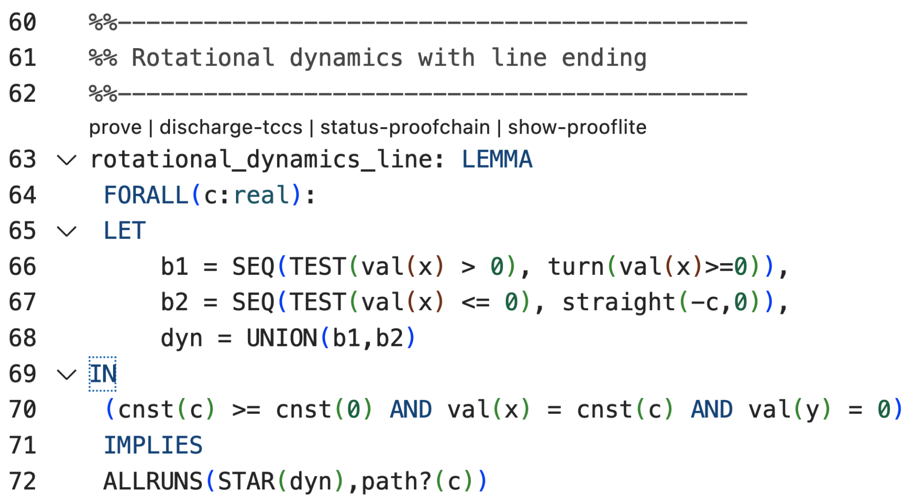
$$\theta' = 1, x' = -c\sin(\theta), y' = c\cos(\theta).$$

Furthermore, it can be shown that the hybrid program β defined as

$$((?(x \geq 0); (\theta' = 1, x' = -c\sin(\theta), y' = c\cos(\theta)\& x \geq 0)) \cup$$
$$(?(x < 0); (x' = -c, y' = 0)))^*,$$

*is equivalent to the hybrid program α defined in Example 2.2, for appropriate initial values. Formally, this is a property relating the **s_rel** function associated with each of these programs, namely for environments e_i, e_o such that $e_i(x) = c$ and $e_i(y) = 0$:*

$$(\exists t\ \textbf{\textit{s\_rel}}(\beta)(e_i\ \textit{with}\ [\theta := 0])(e_o\ \textit{with}\ \ [\theta := t])) \iff$$
$$\textbf{\textit{s\_rel}}(\alpha)(e_i)(e_o\ \textit{with}\ [\theta := e_i(\theta)]).$$



```
%%-------------------------------------------
%% Rotational dynamics with line ending
%%-------------------------------------------
rotational_dynamics_line: LEMMA
 FORALL(c:real):
 LET
     b1 = SEQ(TEST(val(x) > 0), turn(val(x)>=0)),
     b2 = SEQ(TEST(val(x) <= 0), straight(-c,0)),
     dyn = UNION(b1,b2)
IN
 (cnst(c) >= cnst(0) AND val(x) = cnst(c) AND val(y) = 0)
 IMPLIES
 ALLRUNS(STAR(dyn),path?(c))
```

Figure 9: The specification of Example 2.3 in Plaidypvs.

Note the property above is specified in PVS logical specification since it is a meta property of hybrid programs, and involves generic hybrid programs rather than particular instances, both features are unique to Plaidypvs. Thus, for a Boolean expression Q that does not change according to θ:

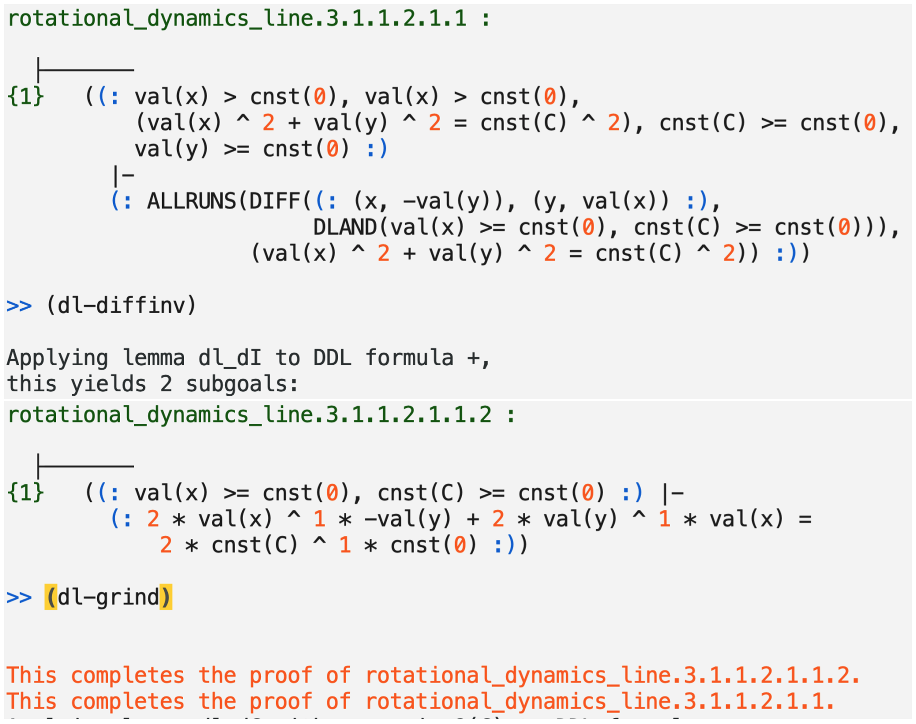
$$(x = c, y = 0 \to [\alpha]\,Q) \iff (x = c, y = 0, \theta = 0 \to [\beta]\,Q).$$

5 Related work

There is a long line of research on the formal verification of hybrid systems. The development of dL itself ([45, 47, 53, 55]) and its use in formal verification of hybrid systems ([6, 7, 19, 21, 23, 26, 27, 33]) is well-known. Additionally, there has been significant work done in the PVS theorem prover [1, 57], Event-B [14], and Isabelle/HOL [17, 34, 35, 36, 56, 59, 61, 62] verifying hybrid systems outside of the dL framework.

The most similar verification effort to the current development is [5], where the authors formally verified the soundness of dL in Coq and Isabelle. The work in [5] focuses on a full formal verification of soundness of dL, with the goal of a formally verified prover kernel for KeYmaera X. The result are proof checkers in Coq and Isabelle for dL proofs. The goal of Plaidypvs is a verified operational embedding of dL in the theorem prover PVS, allowing specification and reasoning about HPs *interactively* within PVS.

While the work in [5] proves soundness of most of the proof calculus of dL, the present work focuses on verifying the proof *rules* of dL. Particularly, the substitution axiom in dL that allows rules and axioms to be applied to specifications of HPs in dL is proven in [5], but is not directly proven for the PVS embedding. Instead, substitution is handled by the instantiation functionalities of the PVS itself, specifically when dL rules and axioms are applied as strategies to a particular dL-sequent in the interactive prover. Additionally, there are several places where the embedding of dL in this work is more general than the work in [5]. Differential Ghost and Differential Effect in [5] are shown for a single ordinary differential equation rather than the more general *system* of ordinary differential equations. Differential Solve is only shown for differential equations with linear solutions, whereas the corresponding rule in



```
rotational_dynamics_line.3.1.1.2.1.1 :

  ├───────
{1}   ((: val(x) > cnst(0), val(x) > cnst(0),
          (val(x) ^ 2 + val(y) ^ 2 = cnst(C) ^ 2), cnst(C) >= cnst(0),
          val(y) >= cnst(0) :)
        |-
        (: ALLRUNS(DIFF((: (x, -val(y)), (y, val(x)) :),
                       DLAND(val(x) >= cnst(0), cnst(C) >= cnst(0))),
                  (val(x) ^ 2 + val(y) ^ 2 = cnst(C) ^ 2)) :))

>> (dl-diffinv)

Applying lemma dl_dI to DDL formula +,
this yields 2 subgoals:
rotational_dynamics_line.3.1.1.2.1.1.2 :

  ├───────
{1}   ((: val(x) >= cnst(0), cnst(C) >= cnst(0) :) |-
        (: 2 * val(x) ^ 1 * -val(y) + 2 * val(y) ^ 1 * val(x) =
            2 * cnst(C) ^ 1 * cnst(0) :))

>> (dl-grind)


This completes the proof of rotational_dynamics_line.3.1.1.2.1.1.2.
This completes the proof of rotational_dynamics_line.3.1.1.2.1.1.
```

Figure 10: The proof steps that complete the proof discussed in Example 3.3.

Plaidypvs automatically solves differential equations with linear and quadratic solutions and is proven for any ODE where the solution is known. Differential Invariant in [5] is restricted to propositions of the form $P = (f(x) \geq g(x))$ and $P = (f(x) > g(x))$ and it is remarked that other cases can be derived in dL from these two cases, but in Plaidypvs Differential Invariant is fully implemented for any proposition that is the conjunction or disjunction of inequalities.

The current work formalizes a version of dL based on Parts I and II in [55], though there are many extensions as well. For adversarial cyber-physical systems there is differential game logic in [52, 54], and Part 5 of [55]. There are also extensions for distributed hybrid systems (quantified differential dynamic logic, [50]), stochastic hybrid systems (stochastic differential dynamic logic, [51]), differential algebraic programs (differential-algebraic dynamic logic, [48]), and a temporal extension of dL called differential temporal dynamic logic [46], [49, Chapter 4].

In addition to verification of hybrid systems, the present work falls more generally into the category of formal verification or simulation of logical systems inside theorem provers. PVS0 is an embedding of a fragment of the specification language of PVS *within* PVS, used in termination analysis of recursive functions [15]. Other efforts to model or verify theorem provers include work on the prover kernel of Hol Light [20], the type checker of Coq [58], the soundness of ACL2 [11]. The goal of Plaidypvs is to add to hybrid systems reasoning to the toolbox of PVS, to increase its proving capabilities. PVS has been applied to a number of applications including formal verification of aircraft avoidance systems including detect-and-avoid logic and algorithms [38], path planning algorithms [3, 8], unmanned aircraft systems [39], position reporting algorithms of aircraft [16, 31] sensor uncertainty mitigation [43] floating point error analysis [30, 60], genetic algorithms [44], nonlinear control systems [4], and requirements written in linear temporal logic and FRETish [9]. PVS has decades of developments that give it a number of unique features and automation capabilities. In addition to advanced real number reasoning [10, 37, 41, 32, 40, 42], previous work has connected PVS to the automated theorem prover MetiTarski [2], for automated reasoning of universally quantified statements about real numbers, including a number of transcendental functions [13]. This capability to use MetiTarski in PVS is leveraged in the **assert** command in Plaidypvs.

6 Conclusion and future work

This work describes Plaidypvs, the operational embedding of dL in PVS. Plaidypvs allows usage of a version of dL within the interactive theorem prover PVS. This embedding extends the formal verification abilities of PVS by giving a framework for specifying and reasoning about HPs and allowing features of PVS to be used naturally within the dL embedding. These include support for importing user-defined functions and theories including the extensive math and computer science developments in NASAlib. Additionally, this embedding allows for meta reasoning about HPs and dL at the PVS level. An example was given that shows the functionality of Plaidypvs that could not be completed in a stand-alone implementation of dL alone, like KeYmaera X.

With the first version of Plaidypvs established, the door is open to much future work. One natural step is to apply Plaidypvs to safety-critical applications of interest to NASA. This will include formal verification of hybrid systems related to urban air mobility and wildland fire fighting among others. Another direction is to increase the usability of Plaidypvs. To do so, a Visual Studio Code extension is under development to display specifications and the proof calculus in a natural and user-friendly way. To increase the automation of dL within Plaidypvs, a more complete substitution calculus to include boolean expressions containing statements about hybrid programs will be implemented. Additionally, formal verification of liveness properties are intended, with implementations of strategies to match. Furthermore, a more robust ordinary differential equation solver to enhance the capabilities of the *differential solve* command would increase the usability of Plaidypvs greatly. Finally, a detailed description of the multivariate analysis and ordinary differential equation library developed to support this embedding will be written, similar to the semi-algebraic set library ([57]) which was done to support verification of liveness properties in upcoming work.

The semantic structure of dL in Plaidypvs is based on the input/output semantics. Future work on defining the trace semantics of hybrid programs will extend the analysis capabilities of the embedding, such as being able to define properties in linear temporal logic, similar to the work in [22]. It has been noted that Quantifier elimination, is often the bottleneck for formal verification of hybrid programs, due to the computational complexity of the general problem. Implementation of techniques to make this process faster would help the usability of Plaidypvs. There are many directions to go for this effort, but one direction will be implementation of the active corners method for a specific class of quantifier elimination [24] geared towards formalized reasoning of aircraft operations.

References

[1] Erika Ábrahám-Mumm, Ulrich Hannemann & Martin Steffen (2001): *Verification of hybrid systems: Formalization and proof rules in PVS*. In: *Proceedings Seventh IEEE International Conference on Engineering of Complex Computer Systems*, IEEE, pp. 48–57, doi:10.1109/ICECCS.2001.930163.

[2] Behzad Akbarpour & Lawrence Charles Paulson (2010): *MetiTarski: An automatic theorem prover for real-valued special functions*. *Journal of Automated Reasoning* 44(3), pp. 175–205.

[3] Swee Balachandran, Anthony Narkawicz, César Muñoz & María Consiglio (2017): *A path planning algorithm to enable well-clear low altitude UAS operation beyond visual line of sight*. In: *Twelfth USA/Europe Air Traffic Management Research and Development Seminar (ATM2017)*, sn, p. 26.

[4] Cinzia Bernardeschi & Andrea Domenici (2016): *Verifying safety properties of a nonlinear control by interactive theorem proving with the Prototype Verification System*. *Information Processing Letters* 116(6), pp. 409–415.

[5] Brandon Bohrer, Vincent Rahli, Ivana Vukotic, Marcus Völp & André Platzer (2017): *Formally verified differential dynamic logic*. In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pp. 208–221, doi:10.1145/3018610.3018616.

[6] Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Andrew Sogokon & André Platzer (2019): *A Formal Safety Net for Waypoint Following in Ground Robots*. *IEEE Robotics and Automation Letters* 4(3), pp. 2910–2917.

[7] Rachel Cleaveland, Stefan Mitsch & André Platzer (2023): *Formally Verified Next-Generation Airborne Collision Avoidance Games in ACAS X*. *ACM Trans. Embed. Comput. Syst.* 22(1), pp. 1–30, doi:10.1145/3544970.

[8] Brendon K Colbert, J Tanner Slagel, Luis G Crespo, Swee Balachandran & César Muñoz (2020): *PolySafe: A Formally Verified Algorithm for Conflict Detection on a Polynomial Airspace*. *IFAC-PapersOnLine* 53(2), pp. 15615–15620.

[9] Esther Conrad, Laura Titolo, Dimitra Giannakopoulou, Thomas Pressburger & Aaron Dutle (2022): *A compositional proof framework for FRETish requirements*. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 68–81.

[10] Marc Daumas, David Lester & César Munoz (2008): *Verified real number calculations: A library for interval arithmetic*. *IEEE Transactions on Computers* 58(2), pp. 226–237, doi:10.1109/tc.2008.213.

[11] Jared Davis & Magnus O Myreen (2015): *The reflective Milawa theorem prover is sound (down to the machine code that runs it)*. *Journal of Automated Reasoning* 55(2), pp. 117–183.

[12] Nicolaas Govert De Bruijn (1972): *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*. In: *Indagationes Mathematicae (Proceedings)*, 75, Elsevier, pp. 381–392.

[13] William Denman & César Munoz (2014): *Automated real proving in PVS via MetiTarski*. In: *International Symposium on Formal Methods*, Springer, pp. 194–199.

[14] Guillaume Dupont (2021): *Correct-by-construction design of hybrid systems based on refinement and proof*. Ph.D. thesis. Available at https://oatao.univ-toulouse.fr/28190/1/Dupont_Guillaume.pdf.

[15] Aaron Dutle (2015): *Proving Program Termination with Matrix Weighted Digraphs*. In: *Cumberland Conference on Combinatorics, Graph Theory and Computing*, NF1676L-21163.

[16] Aaron Dutle, Mariano Moscato, César Muñoz, Gregory Anderson, François Bobot et al. (2021): *Formal analysis of the compact position reporting algorithm*. *Formal Aspects of Computing* 33(1), pp. 65–86.

[17] Simon Foster, Jonathan Julián Huerta y Munive, Mario Gleirscher & Georg Struth (2021): *Hybrid systems verification with Isabelle/HOL: Simpler syntax, better models, faster proofs*. In: *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings 24*, Springer, pp. 367–386.

[18] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp & André Platzer (2015): *KeYmaera X: An axiomatic tactical theorem prover for hybrid systems*. In: *International Conference on Automated Deduction*, Springer, pp. 527–538, doi:10.1007/978-3-319-21401-6_36.

[19] Nathan Fulton & André Platzer (2018): *Safe Reinforcement Learning via Formal Methods: Toward Safe Control Through Proof and Learning*. In Sheila McIlraith & Kilian Weinberger, editors: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, February 2-7, 2018, New Orleans, Louisiana, USA.*, AAAI Press, pp. 6485–6492.

[20] John Harrison (2006): *Towards self-verification of HOL Light*. In: *International Joint Conference on Automated Reasoning*, Springer, pp. 177–191.

[21] Jean-Baptiste Jeannin, Khalil Ghorbal, Yanni Kouskoulas, Aurora Schmidt, Ryan Gardner, Stefan Mitsch & André Platzer (2017): *A Formally Verified Hybrid System for Safe Advisories in the Next-generation Airborne Collision Avoidance System*. *STTT* 19(6), pp. 717–741, doi:10.1007/s10009-016-0434-1.

[22] Jean-Baptiste Jeannin & André Platzer (2014): *dTL 2: differential temporal dynamic logic with nested temporalities for hybrid systems*. In: *International Joint Conference on Automated Reasoning*, Springer, pp. 292–306.

[23] Aditi Kabra, Stefan Mitsch & André Platzer (2022): *Verified Train Controllers for the Federal Railroad Administration Train Kinematics Model: Balancing Competing Brake and Track Forces*. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 41(11), pp. 4409–4420, doi:10.1109/TCAD.2022.3197690.

[24] Nishant Kheterpal, Elanor Tang & Jean-Baptiste Jeannin (2022): *Automating Geometric Proofs of Collision Avoidance with Active Corners*. In: *CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN–FMCAD 2022*, p. 359.

[25] Stefan Mitsch (2021): *Implicit and explicit proof management in keymaera x*. *arXiv preprint arXiv:2108.02965*.

[26] Stefan Mitsch, Marco Gario, Christof J. Budnik, Michael Golm & André Platzer (2017): *Formal Verification of Train Control with Air Pressure Brakes*. In Alessandro Fantechi, Thierry Lecomte & Alexander Romanovsky, editors: *RSSRail*, *LNCS* 10598, Springer, pp. 173–191, doi:10.1007/978-3-319-68499-4_12.

[27] Stefan Mitsch, Khalil Ghorbal, David Vogelbacher & André Platzer (2017): *Formal Verification of Obstacle Avoidance and Navigation of Ground Robots*. *I. J. Robotics Res.* 36(12), pp. 1312–1340, doi:10.1177/0278364917733549.

[28] Stefan Mitsch & André Platzer (2017): *The keymaera X proof IDE-concepts on usability in hybrid systems theorem proving*. *arXiv preprint arXiv:1701.08469*.

[29] Stefan Mitsch & André Platzer (2020): *A Retrospective on Developing Hybrid System Provers in the KeYmaera Family: A Tale of Three Provers*. In: *Deductive Software Verification: Future Perspectives: Reflections on the Occasion of 20 Years of KeY*, Springer, pp. 21–64.

[30] Mariano Moscato, Aaron Dutle, César A Muñoz et al. (2017): *Automatic estimation of verified floating-point round-off errors via static analysis*. In: *International Conference on Computer Safety, Reliability, and Security*, Springer, pp. 213–229.

[31] Mariano M Moscato, César A Muñoz, Aaron Dutle, François Bobot et al. (2018): *A formally verified floating-point implementation of the compact position reporting algorithm*. In: *International Symposium on Formal Methods*, Springer, pp. 364–381.

[32] Mariano M Moscato, César A Muñoz & Andrew P Smith (2015): *Affine arithmetic and applications to real-number proving*. In: *International Conference on Interactive Theorem Proving*, Springer, pp. 294–309, doi:10.1007/978-3-319-22102-1_20.

[33] Andreas Müller, Stefan Mitsch, Werner Retschitzegger, Wieland Schwinger & André Platzer (2017): *Change and Delay Contracts for Hybrid System Component Verification*. In Marieke Huisman & Julia Rubin, editors: *FASE*, *LNCS* 10202, Springer, pp. 134–151, doi:10.1007/978-3-662-54494-5_8.

[34] Jonathan Julian Huerta y Munive (2020): *Algebraic verification of hybrid systems in Isabelle/HOL*. Ph.D. thesis, University of Sheffield. Available at https://etheses.whiterose.ac.uk/28886/.

[35] Jonathan Julián Huerta y Munive & Georg Struth (2018): *Verifying hybrid systems with modal Kleene algebra*. In: *International Conference on Relational and Algebraic Methods in Computer Science*, Springer, pp. 225–243, doi:10.1007/978-3-030-02149-8_14.

[36] Jonathan Julián Huerta y Munive & Georg Struth (2022): *Predicate Transformer Semantics for Hybrid Systems*. *Journal of Automated Reasoning* 66(1), pp. 93–139, doi:10.1007/s10817-021-09607-x.

[37] César Munoz & Anthony Narkawicz (2013): *Formalization of Bernstein polynomials and applications to global optimization*. Journal of Automated Reasoning 51(2), pp. 151–196, doi:10.1007/s10817-012-9256-3.

[38] César Muñoz, Anthony Narkawicz, George Hagen, Jason Upchurch, Aaron Dutle, María Consiglio & James Chamberlain (2015): *DAIDALUS: detect and avoid alerting logic for unmanned systems*. In: 2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC), IEEE, pp. 5A1–1.

[39] César A Muñoz, Aaron Dutle, Anthony Narkawicz & Jason Upchurch (2016): *Unmanned aircraft systems in the national airspace system: a formal methods perspective*. ACM SIGLOG News 3(3), pp. 67–76.

[40] Anthony Narkawicz & César Munoz (2013): *A formally verified generic branching algorithm for global optimization*. In: Working Conference on Verified Software: Theories, Tools, and Experiments, Springer, pp. 326–343, doi:10.1007/978-3-642-54108-7_17.

[41] Anthony Narkawicz & César Muñoz (2014): *A Formally Verified Generic Branching Algorithm for Global Optimization*. In Ernie Cohen & Andrey Rybalchenko, editors: Proceedings of the 5th International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2013), Lecture Notes in Computer Science 8164, Springer, Menlo Park, CA, US, pp. 326–343, doi:10.1007/978-3-642-54108-7_17.

[42] Anthony Narkawicz, César Munoz & Aaron Dutle (2015): *Formally-verified decision procedures for univariate polynomial computation based on Sturm's and Tarski's theorems*. Journal of Automated Reasoning 54(4), pp. 285–326, doi:10.1007/s10817-015-9320-x.

[43] Anthony Narkawicz, César Muñoz & Aaron Dutle (2018): *Sensor uncertainty mitigation and dynamic well clear volumes in DAIDALUS*. In: 2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC), IEEE, pp. 1–8.

[44] M Saqib Nawaz, M IkramUllah Lali & MA Pasha (2013): *Formal verification of crossover operator in genetic algorithms using prototype verification system (PVS)*. In: 2013 IEEE 9th International Conference on Emerging Technologies (ICET), IEEE, pp. 1–6.

[45] André Platzer (2007): *Differential Dynamic Logic for Verifying Parametric Hybrid Systems*. In Nicola Olivetti, editor: TABLEAUX, LNCS 4548, Springer, pp. 216–232, doi:10.1007/978-3-540-73099-6_17.

[46] André Platzer (2007): *A Temporal Dynamic Logic for Verifying Hybrid System Invariants*. In Sergei N. Artëmov & Anil Nerode, editors: LFCS, LNCS 4514, Springer, pp. 457–471, doi:10.1007/978-3-540-72734-7_32.

[47] André Platzer (2008): *Differential dynamic logic for hybrid systems*. Journal of Automated Reasoning 41(2), pp. 143–189, doi:10.1007/s10817-008-9103-8.

[48] André Platzer (2010): *Differential-algebraic Dynamic Logic for Differential-algebraic Programs*. J. Log. Comput. 20(1), pp. 309–352, doi:10.1093/logcom/exn070. Advance Access published on November 18, 2008.

[49] André Platzer (2010): *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, Heidelberg, doi:10.1007/978-3-642-14509-4.

[50] André Platzer (2010): *Quantified Differential Dynamic Logic for Distributed Hybrid Systems*. In Anuj Dawar & Helmut Veith, editors: CSL, LNCS 6247, Springer, pp. 469–483, doi:10.1007/978-3-642-15205-4_36.

[51] André Platzer (2011): *Stochastic Differential Dynamic Logic for Stochastic Hybrid Programs*. In Nikolaj Bjørner & Viorica Sofronie-Stokkermans, editors: CADE, LNCS 6803, Springer, pp. 446–460, doi:10.1007/978-3-642-22438-6_34.

[52] André Platzer (2015): *Differential Game Logic*. ACM Trans. Comput. Log. 17(1), pp. 1:1–1:51, doi:10.1145/2817824.

[53] André Platzer (2017): *A complete uniform substitution calculus for differential dynamic logic*. Journal of Automated Reasoning 59(2), pp. 219–265, doi:10.1007/s10817-016-9385-1.

[54] André Platzer (2017): *Differential Hybrid Games*. ACM Trans. Comput. Log. 18(3), pp. 19:1–19:44, doi:10.1145/3091123.

[55] André Platzer (2018): *Logical Foundations of Cyber-Physical Systems*. Springer, Cham, doi:10.1007/978-3-319-63588-0.

[56] Huanhuan Sheng, Alexander Bentkamp & Bohua Zhan (2023): *HHLPy: Practical Verification of Hybrid Systems Using Hoare Logic*. In: *Formal Methods: 25th International Symposium, FM 2023, Lübeck, Germany, March 6–10, 2023, Proceedings*, Springer, pp. 160–178.

[57] J Tanner Slagel, Lauren White & Aaron Dutle (2021): *Formal verification of semi-algebraic sets and real analytic functions*. In: *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 278–290, doi:10.1145/3437992.3439933.

[58] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau & Théo Winterhalter (2019): *Coq coq correct! verification of type checking and erasure for coq, in coq*. *Proceedings of the ACM on Programming Languages* 4(POPL), pp. 1–28.

[59] Georg Struth (2021): *Hybrid Systems Verification with Isabelle/HOL: Simpler Syntax, Better Models, Faster Proofs*. In: *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*, 13047, Springer Nature, p. 367, doi:10.1007/978-3-030-90870-6_20.

[60] Laura Titolo, Marco A. Feliú, Mariano M. Moscato & César A. Muñoz (2018): *An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs*. In Isil Dillig & Jens Palsberg, editors: *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings*, *Lecture Notes in Computer Science* 10747, Springer, pp. 516–537, doi:10.1007/978-3-319-73721-8_24. Available at https://doi.org/10.1007/978-3-319-73721-8_24.

[61] Shuling Wang, Naijun Zhan & Liang Zou (2015): *An improved HHL prover: an interactive theorem prover for hybrid systems*. In: *Formal Methods and Software Engineering: 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings 17*, Springer, pp. 382–399.

[62] Liang Zou, Naijun Zhan, Shuling Wang & Martin Fränzle (2015): *Formal verification of Simulink/Stateflow diagrams*. In: *Automated Technology for Verification and Analysis: 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings 13*, Springer, pp. 464–481.