Towards an Implementation of Differential Dynamic Logic in PVS

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

César Muñoz*
NASA Langley Research Center
Hampton, Virginia, USA

Swee Balachandran†
National Institute of Aerospace
Hampton, Virginia, USA

Mariano Moscato
National Institute of Aerospace
Hampton, Virginia, USA

Aaron Dutle
NASA Langley Research Center
Hampton, Virginia, USA

Paolo Masci
National Institute of Aerospace
Hampton, Virginia, USA

Lauren White
NASA Langley Research Center
Hampton, Virginia, USA

Abstract

This paper describes an ongoing effort to embed and verify differential dynamic logic (dL) in the Prototype Verification System (PVS). dL is a logic for specifying and formally reasoning about *hybrid systems*, i.e., systems that employ both continuous and discrete dynamics. There are several benefits of this effort. First, the embedding of dL in PVS offers an independent formal verification of the semantics and inference rules of dL. Second, the embedding is fully operational within PVS, giving PVS practitioners the ability to use dL in the formal specification and verification of hybrid systems. Third, the rich specification language, type system, and powerful interactive prover of PVS can be used on dL objects. In addition to the embedding and verification of dL, a custom extension for Visual Studio Code has been developed, so that a stylized dL syntax can be used to specify hybrid programs and their properties.

CCS Concepts: • **Theory of computation** → **Formal languages and automata theory**; *Logic*; • **Security and privacy** → *Logic and verification.*

Keywords: Differential Dynamic Logic, Prototype Verification System, Formal Verification, Hybrid Systems

*Currently at Amazon Web Services
†Currently at Xwing

SOAP '22, June 14, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9274-7/22/06…$15.00
https://doi.org/10.1145/3520313.3534661

ACM Reference Format:
J. Tanner Slagel, César Muñoz, Swee Balachandran, Mariano Moscato, Aaron Dutle, Paolo Masci, and Lauren White. 2022. Towards an Implementation of Differential Dynamic Logic in PVS. In *Proceedings of the 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '22), June 14, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3520313.3534661

1 Introduction

Hybrid Systems, which are characterized by the interplay of continuous and discrete dynamics, are ubiquitous in the safety-critical world. One example is automated aircraft navigating through crowded urban canyons and intersections which must avoid collisions with other aircraft and the surrounding landscape [2, 3]. Another similar example is the interaction of numerous groups of aircraft of many different types in wildfire response and mitigation [30]. These systems contain both continuous dynamics (e.g., the movement of the aircraft over time) and discrete dynamics (e.g., entrance into a defined operational airspace, an acceleration command given to an aircraft).

Differential dynamic logic (dL) [21] offers a natural way to model and reason about properties of such systems. dL allows a *Hybrid Program* to be written as a model of a discrete/continuous controller, and logical statements about the program can be stated and proven using deduction rules. dL was implemented first in the tool KeYmaera [25], and extended in KeYmaera X [7]. It has been used in verification of a wide variety of safety-critical applications [11, 12, 14, 26].

This paper describes an ongoing effort to embed and verify a fully-functional version of dL in the Prototype Verification System (PVS), a formal specification language with a tightly integrated interactive theorem prover. There are three main parts of this work:

1. A formal verification of the soundness of dL in PVS, which guarantees any statement proven using the deduction rules is correct;
2. A fully operational embedding of dL in PVS, including capability for specification of hybrid programs, writing logical statements about hybrid programs, and deduction rules proving these statements;
3. Automation of proof rules and pretty-printing of dL specifications to make the underlying formalization and embedding invisible to the casual user.

One benefit of such an embedding is that a user familiar with dL can specify and prove properties about hybrid systems in a very similar manner to existing dL implementations, with an additional layer of assurance provided by the PVS soundness proof. Another benefit is that the richness of the PVS system can be used alongside the dL prover. For example, properties about and relationships between entire *classes* of hybrid programs can be verified using PVS outside of dL.

There has been past work on verifying hybrid systems in PVS [1, 28], and using other formal methods tools like Event-B [6], Isabelle/HOL [8–10, 29], and Coq. Most similar to the current work is a formalization of dL performed in HOL and Coq [4], where a proof written in the dL language can be verified by a sound proof checker. The work here also verifies the axioms and rules of dL, and further implements them as strategies in PVS. Using these strategies, properties of hybrid programs can be verified *interactively* within the PVS proof assistant, similar to KeYmaera/KeYmaera X. Also, since this implementation of dL is in PVS, it has all the features and automation capabilities of PVS such as advanced real number reasoning [5, 15–19].

2 Prototype Verification System

The Prototype Verification System (PVS) is a formal specification and verification tool developed by SRI International [20]. PVS has a strongly typed functional specification language based on higher-order logic, which allows a user to write functions and logical statements. PVS also includes a tightly integrated interactive theorem prover, for the verification of the specified statements. To prove a statement in PVS, the user inputs proof commands to manipulate a logical sequent in a sound way to produce a list of sequents, referred to as branches, that conjunctively imply the original one. A sequent is read as the conjunction of the antecedent (above the *turnstile*) implying the disjunction of the consequent (below the turnstile, see Figure 1). A user enters proof commands until `TRUE` appears in the consequent, `FALSE` appears in the antecedent, or the same formula appears in both the antecedent and the consequent, for all branches. PVS allows users to invoke previously proven lemmas in the prover, and also includes a proof strategy language, where users can define proof strategies that combine commands and lemmas in sophisticated ways.

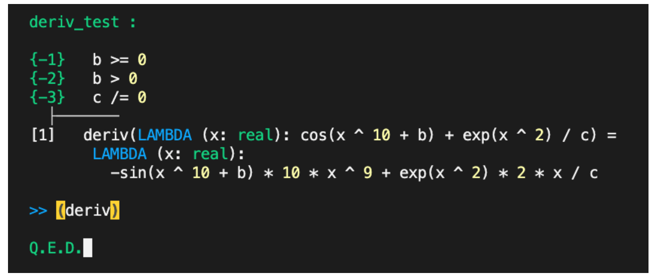

Figure 1. An example of the PVS theorem prover

For an example of the interactive theorem proving environment see Figure 1, which displays usage of the `(deriv)` strategy developed in the analysis library of NASAlib[1], that uses several lemmas to automate derivatives for a wide array of functions.

3 Differential Dynamic Logic

Differential dynamic logic (dL) was first conceived by Platzer in [21] and implemented in the original KeYmaera system [25]. There have been several iterations on, and extensions of, dL and KeYmaera since the original formulation [7, 22–24]. This section does not attempt to give a complete description of dL or implementations, as this would be far beyond the scope of this paper. Instead, a general description is given, and a few specific examples that will be followed through the paper are presented. For a complete formal description, see [21]. For a tutorial on differential dynamic logic in KeYmaera, see [27].

dL can be roughly divided into three main components: hybrid program specification, differential logic itself, and the deduction rules for the logic.

3.1 Hybrid Programs

Hybrid programs (HPs) are intended to describe systems that have discrete and continuous behavior combined. The two basic building blocks of HPs are discrete assignments $x_1 := t_1, \ldots, x_n := t_n$ where a variable x_i is assigned real value t_i, for each i such that $0 \le i \le n$, and continuous evolution of an ordinary differential equation (ODE) $\{x_1' = f_1, \ldots, x_i' = f_i \,\&\, D\}$, where the first entries define the differential system (where f_i is an expression that could contain the values of $x_1, \ldots, x_n$), and the last (optional) entry D gives the domain of the system. These two basic block types can be iteratively combined to produce more complex formulas, using sequential composition of programs (;), nondeterministic choice between two options ($\cup$), testing of a first order formula ($?P$), and nondeterministic repetition ($*$) of a block[2].

[1] NASALib is a collection of PVS formalizations maintained by the NASA Langley Formal Methods group: https://github.com/nasa/pvslib

[2] Implementations vary in their choice of combinators. For example, KeYmaera X adds an explicit "if-then-else" combinator.

An example hybrid program is

$$((a := a + 1); \{x' = v, v' = a\})^* \tag{1}$$

This program increments the value of the variable a by one, then evolves the differential equation where a is acceleration, v is velocity, and x is position. This evolution progresses for some amount of time, and then repeats this program a nondeterministic number of times. This example will be used throughout the paper to explain dL and its embedding in PVS.

3.2 Differential Logic

Differential logic formulas involve real numbers, hybrid programs, and several familiar logical connectors. The logic allows for comparison of real numbers (using $<, \leq, =, \geq, >, \neq$), quantification ($\forall$, $\exists$), negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), and (bi)implication ($\Rightarrow, \Leftrightarrow$) for formulas, and unique to hybrid programs, the operators *all runs* and *some runs*. The formula $[\alpha]\phi$ states that for all runs of the hybrid program α, the formula ϕ holds. Similarly $\langle\alpha\rangle\phi$ states that there is at least *one* run of the program α for which the formula ϕ holds.

Using differential logic allows properties of a hybrid program to be written. For example, using the program in (1),

$$\begin{aligned} &x \geq 1 \wedge v \geq 0 \wedge a \geq 0 \;\Rightarrow \\ &\qquad [((a := a + 1); \{x' = v, v' = a\})^*](x \geq 1). \end{aligned} \tag{2}$$

states that if x is at least one, and v and a are non-negative, then for all runs of the program in (1), x will remain at least one.

3.3 Deduction Rules

To prove a statement about a hybrid program, the logic statement can be written as a dL sequent. In the case of (2), this looks like:

$$\begin{aligned} &x \geq 1,\; v \geq 0,\;\; a \geq 0 \vdash \\ &\qquad [((a := a + 1); \{x' = v, v' = a\})^*](x \geq 1). \end{aligned} \tag{3}$$

Then, deduction rules of dL are applied to formally manipulate the dL sequent in a way that is sound (the sequents produced by the rule imply the truth of the original sequent). A list of the complete syntax, semantics and rules of dL can be found in [24], pages 637-639[3]. Here, two of the set of formalized and proven rules are described, with their embedding in PVS detailed in Section 4.

3.3.1 Discrete Loop.

Formally, the dL-loop rule is given by:

$$\frac{\Gamma \vdash J \quad J \vdash [\alpha]J \quad J \vdash P}{\Gamma \vdash [\alpha^*]P,}$$

where Γ represents the antecedent formulas, J represents the invariant condition, P represents the property trying to be shown, and α represents the hybrid program.

Intuitively, this rule is used to show that all runs of the looped program α^* satisfy a property P, by using a property J that is invariant through each pass through α. The dL-loop rule reduces the argument to three cases: the invariant condition is held as a precondition to the hybrid program ($\Gamma \vdash J$), J is indeed invariant to one application of the loop ($J \vdash [\alpha]J$), and the invariant conditions implies the desired postcondition ($J \vdash P$). Applying the dL-loop rule to the sequent in (3) with the instantiation $\Gamma = (x \geq 1,\;\; a \geq 0)$, $\alpha = ((a := a + 1); \{x' = v, v' = a\})$, $J = (x \geq 1 \wedge v \geq 0 \wedge a \geq 0)$, and $P = (x \geq 1)$, yields the three sequents:

$$x \geq 1,\;\; v \geq 0,\;\;\; a \geq 0 \vdash x \geq 1 \wedge v \geq 0 \wedge a \geq 0 \tag{4}$$

$$\begin{aligned} &x \geq 1,\;\; v \geq 0,\;\;\; a \geq 0 \vdash \\ &\qquad [((a := a + 1); \{x' = v, v' = a\})] \\ &\qquad\quad (x \geq 1 \wedge v \geq 0 \wedge a \geq 0) \end{aligned} \tag{5}$$

$$x \geq 1 \,\wedge\, v \geq 0 \wedge a \geq 0 \vdash x \geq 1 \tag{6}$$

The sequent (4) and (6) can be proven with rules of basic propositional logic. This leaves the sequent in (5).

3.3.2 Differential Invariant.

Formally, the differential invariant rule is given by:

$$\frac{\Gamma, q(x) \vdash p(x) \quad q(x) \vdash [x' := f(x)](p(x))'}{\Gamma \vdash [x' = f(x) \,\&\, q(x)]p(x),}$$

where Γ represents the antecedent formulas, $q(x)$ represents a set (or property) which the continuous dynamics $x' = f(x)$ are restricted to, $p(x)$ is the property trying to be shown, and $(p(x))'$ is the differential operator on p that guarantees that $p(x)$ remains true through the evolution $x' = f(x)$, i.e., showing that $p(x)$ is an invariant condition to the continuous system $x' = f(x) \,\&\, q(x)$.

Consider the following statement in dL[4]:

$$x \geq 1,\;\; a \geq 0 \vdash [\{x' = v, v' = a\} \,\&\, (v \geq 0)](x \geq 1) \tag{7}$$

To apply the differential invariant rule to (7) the user would instantiate: $\Gamma = (x \geq 1, a \geq 0)$, $(x' = f(x)) = \{x' = v, v' = a\}$, $q(x) = (v \geq 0)$, and $p(x) = (x \geq 1)$, where, $(p(x))' = (x' \geq 0)$. This yields the two sequents

$$x \geq 1,\;\; a \geq 0,\;\; v \geq 0 \vdash x \geq 1 \tag{8}$$

$$v \geq 0 \vdash [\{x' := v, v' := a\}](x' \geq 0). \tag{9}$$

Note that the formula in (8) has $x \geq 1$ in both the antecedent and the consequent and is therefore trivially true, and the formula in (9) can be proven using the property that $[x' := v, v' := a](x' \geq 0) = v \geq 0$, which is also a rule of dL. It should be noted that the computation of $(p(x))'$ for a general proposition $p(x)$ that could include proposition logical formulas and inequalities is nontrivial, and a specific calculus must be implemented to ensure soundness of such an approach [22].

[3] dL Rule "Cheat Sheet": https://symbolaris.com/logic/dL-sheet.pdf

[4] It can be shown that (7) implies (5) using other rules of dL. These details are omitted for space limitations.

4 Embedding dL in PVS

The goal of this work is to have a fully operational and sound implementation of dL in PVS. This section will provide an overview of this embedding.

4.1 Hybrid Programs

To define Hybrid programs in PVS, the structure of variables that will be input and output from the programs must be defined. The variables and their values are represented as a mapping `env: [nat -> real]`, where variables are represented by their index `i:nat` and their value is given by `env(i)`. This mapping is called an `Environment`:

```
Environment : TYPE = [ nat -> real ].
```

For instance, taking `x: nat = 0`, `y:nat = 1`,

```
env: Environment = (LAMBDA(i:nat): 0)
     WITH [(x) := 10, (y) := -sqrt(5)]
```

represents the environment where all variables are set to zero, except the first variable (represented by the index 0) which has value 10 and second variable (represented by the index 1) who has value $-\sqrt{5}$.

Given the environment of variables, types for predicates, quantified boolean expressions, and real valued functions are defined:

```
BoolExpr  : TYPE = [Environment -> bool]
QBoolExpr : TYPE = [real -> BoolExpr]
RealExpr  : TYPE = [Environment -> real],
```

These types use higher order abstract syntax to represent bounded variables. For instance,

```
val(i:nat): RealExpr
  = LAMBDA(env:Environment): env(i).
```

For the discrete assignment and continuous evolution, a list of ordered pairs is needed where each first entry is an index of a variable and each second entry is the desired assignment. To prevent double assignments or continuous evolutions for the same variable, the condition that a variable index may not be used twice is in the type of the list, i.e., that a variable can only be assigned once is enforced in the following way:

```
MapExpr : TYPE = [nat,RealExpr]
mapexpr_inj(l:list[MapExpr]) : bool =
  LET N = length(l) IN
  FORALL(i:below(N),j:subrange(i+1,N-1)):
  nth(l,i)`1 /= nth(l,j)`1
MapExprInj : TYPE = (mapexpr_inj)
Assigns    : TYPE = MapExprInj
ODEs       : TYPE = MapExprInj.
```

The syntax and semantics of hybrid programs can be defined with these in place. Syntactically, hybrid programs are a recursive data type defined in PVS as:

```
HP : DATATYPE BEGIN
ASSIGN(assigns:Assigns) : assign?
DIFF(odes:ODEs,be:BoolExpr) : diff?
TEST(be:BoolExpr) : test?
SEQ(stm1,stm2:HP) : seq?
UNION(stm1,stm2:HP) : union?
STAR(stm:HP) : star? END HP.
```

The semantics of a hybrid program defines how input environments and output environments are related to each other through different program constructs. This is done through the semantic relation:

```
semantic_rel(hp:HP)(envi:Environment)
(envo:Environment): INDUCTIVE bool = ...,
```

which is inductively defined on the structure of the programs. Because of its size the full definition is not given here, but a few cases are shown. For example, the semantics of `SEQ(stm1,stm2:hp)` is given by:

```
(EXISTS (env:Environment):
    semantic_rel(stm1(hp))(envi)(env) AND
    semantic_rel(stm2(hp))(env)(envo)).
```

This says that two environments `envi` and `envo` are semantically related through `SEQ(stm1,stm2)` when there is an environment `env` semantically related (as an output) to `envi` through `stm1`, and semantically related (as in input) to `envo` through `stm2`. The semantics of a hybrid program of the form `ASSIGN(l: MapExprInj)` is given by:

```
(FORALL(i:below(length(l))):
LET (k,re) = nth(l,i) IN
envo(k) = re(envi)) AND
FORALL(i:(not_in_map(l))): envo(i) = envi(i)).
```

This says that environments `envi` and `envo` are semantically related through `ASSIGN(l)`, meaning that `envo` is `envi` with the proper assignments made at the right indices.

4.2 Basics of dL in PVS

Sequents of dL are embedded in PVS as predicates on two lists of boolean expressions - one acting as the antecedent and one acting as the consequent:

```
|-(Gamma, Delta: list[BoolExpr] ): bool.
```

The semantics of the expression `Gamma |- Delta` state that for each environment that satisfies the conjunction of expressions in Gamma, one of the expressions in Delta is true, in PVS written as:

```
FORALL (env: Environment):
    (FORALL(i:below(length(Gamma))):
      nth(Gamma,i)(env)) IMPLIES
      (EXISTS (j:below(length(Delta))):
        nth(Delta,j)(env))
```

There are a number of common boolean expression defined in dL-PVS that allow first order logic statements in the dL-PVS sequent. This includes all of the operations mentioned in Section 3.2 including $<, \leq, =, \geq, >, \neq$, quantification, negation, conjunction, disjunction, and (bi)implication for formulas. In addition to these expressions, dL also uses the universal quantifier and existential quantifier for hybrid programs. These are denoted as ALLRUNS and SOMERUNS, respectively. The boolean

```
SOMERUNS(hp:HP,P:BoolExpr)(env:Environment)
```

is true when there exists an envo:Environment that is an output of the hybrid program hp with input env, such that P(envo) is true. The boolean

```
ALLRUNS(hp:HP,P:BoolExpr)(env:Environment)
```

is true when P(envo) is true for every envo:Environment that is an output of the hybrid program hp with input env. Using the operators of dL-PVS, the logical statement in (2) can be specified in PVS as:

```
hp_ex: LEMMA
LET alpha = STAR(SEQ(ASSIGN( (: a, val(a)+1:)),
DIFF((:(x, val(v)), (v, val(a)) :)))) IN
  (: :) |- (: DLIMPLIES((:val(x) >= cnst(1),
  val(v) >= cnst(0), val(a) >= 0 :),
    ALLRUNS(alpha, val(x) >= 1)) :)
```

4.3 Rules of dL in PVS

All the rules of dL have been specified and proven as lemmas in PVS. This includes a number of basic propositional-logic-based rules (such as dL-assert, shown in Figures 2 and 3), but also the more specialized rules that are specifically tailored for reasoning of hybrid systems. The rules discussed in Section 3.3 are shown below as PVS lemmas.

In PVS the dL-loop rule was written as the following lemma and proven

```
dl_loop : LEMMA
FORALL(Gamma: list[BoolExpr], J,P:BoolExpr, A:HP):
    (Gamma |- cons(J)) AND (J |- P) AND
    (J |- ALLRUNS(A,J)) IMPLIES
      (Gamma |- cons(ALLRUNS(STAR(A),P))).
```

The differential invariant rule was also specified and proven

```
dl_dI: LEMMA
FORALL (Gamma)(nnP,Q)(ode:ODEs):
  (cons(Q,Gamma) |- cons(nqb_to_be(nnP))) AND
  (Q |- ALLRUNS(ASSIGN_DIFT(ode)(nnP))) IMPLIES
    ( Gamma |- cons( ALLRUNS(DIFF(ode,Q),
    nqb_to_be(nnP)), Delta) ).
```

As a small example of the complexity of some of these specifications and proofs, note that for the differential invariant rule, a deep embedding of boolean expressions was required to define the appropriate derivative of such expressions in a recursive and executable way. The function nqb_to_be(nnP) represents the transformation from nnP, a representation of a boolean expression written in this deep embedding to a boolean expression. The function ASSIGN_DIFT(odes)(nnP) represents the derivative of nnP with the derivative assignments defined by the odes. The ASSIGN_DIFT(odes)(nnP) expression corresponds to the term $[x' := f(x)](p(x))'$ in the differential invariant rule in Section 3.3.2.

4.4 Extensions of dL in PVS

The operational implementation of dL in PVS is fully typed, and allows specification and reasoning about hybrid programs based on properties of the hybrid programs rather than particular instantiations. To illustrate these points consider the following predicate behind?:

```
behind?(odes:ODEs)(env:Environment): bool =
FORALL(i:below(max_var(odes))):env(i) <
env(i+1).
```

This predicate is true when values of the environment are increasing up to the index of the maximum variable used by odes.

The subtype of hybrid programs called behind is defined as all the hybrid programs that preserve behind?(odes):

```
behind: TYPE = hp: (diff?) |
    (: behind?(odes(hp)) :) |-
      (: ALLRUNS(hp,behind?(odes(hp))) :).
```

Similarly, the boolean expression slower?(odes:ODEs) ensures that variables of lower index are going slower than variables of higher index:

```
slower?(odes:ODEs)(env:Environment): bool =
FORALL(i:below(length(odes))): nth(odes,i)`1 = i
AND FORALL(i:below(length(odes)-1)):
(nth(odes,i)`2(env) < nth(odes,i+1)`2(env)).
```

The subtype of hybrid programs called slow are hybrid programs where variables of lower index go slower than variables of higher index, regardless of starting values:

```
slow: TYPE
= hp: (diff?) | (: :) |- (: slower?(odes(hp)) :).
```

Using a JUDGEMENT in PVS (which is similar to a lemma, but concerning relationships between types), the user can state and prove that hybrid program type slow is a subtype of behind:

```
slow_is_behind: JUDGEMENT slow SUBTYPE_OF behind.
```

This means that every hybrid program of type slow is also of type behind. This JUDGEMENT shows that hybrid programs of type (diff?) that have variables whose velocities and positions are monotonic according to index are guaranteed to preserve order of the variables. Notice that this property is for an entire *class* of hybrid programs, not a defined one. An *instance* of this class, which can be reasoned about in classical dL, can take on a wide variety of forms. This illustrates a benefit of dL embedded in PVS: the expressive language and proof system provides avenues for reasoning about hybrid programs or entire classes of them outside of dL.

5 Using dL-PVS

5.1 Implementation of dL rules as strategies

The formally verified dL deduction rules have been implemented as strategies for use in the interactive theorem prover

```
discrete_loop_ex :

    |-------
{1}   (: val(x) >= cnst(1), val(v) >= cnst(0), val(a) >= cnst(0) :) |-
       (: ALLRUNS(STAR(SEQ(ASSIGN((: (a, val(a) + cnst(1)) :)),
                           DIFF((: (x, val(v)), (v, val(a)) :),
                                DLBOOL(TRUE)))),
                  val(x) >= cnst(1)) :)

>> (dl-loop "val(x) >= cnst(1) AND val(v) >= cnst(0) AND val(a) >=cnst(0)")
discrete_loop_ex.1 :

    |-------
{1}   ((: val(x) >= cnst(1), val(v) >= cnst(0), val(a) >= cnst(0) :) |-
        (: DLAND(val(x) >= cnst(1),
                 DLAND(val(v) >= cnst(0), val(a) >= cnst(0))) :))

>> (dl-assert)
discrete_loop_ex.2 :

    |-------
{1}   ((: DLAND(val(x) >= cnst(1),
                DLAND(val(v) >= cnst(0), val(a) >= cnst(0))) :)
        |- (: val(x) >= cnst(1) :))

>> (dl-assert)
discrete_loop_ex.3 :

    |-------
{1}   ((: DLAND(val(x) >= cnst(1),
                DLAND(val(v) >= cnst(0), val(a) >= cnst(0))) :)
        |-
        (: ALLRUNS(SEQ(ASSIGN((: (a, val(a) + cnst(1)) :)),
                       DIFF((: (x, val(v)), (v, val(a)) :), DLBOOL(TRUE))),
                   DLAND(val(x) >= cnst(1),
                         DLAND(val(v) >= cnst(0), val(a) >= cnst(0)))) :))

>>
```

Figure 2. Example of using the dl-loop strategy in PVS

of PVS. As the strategy language works using basic deduction rules and proven lemmas, these strategies are sound with respect to PVS.

The lemma `dl_loop` was written as a proof strategy `(dl-loop <inv>)` where `<inv>` is the user defined invariant condition to be used in application of this rule. Figure 2 shows the use of the `dl-loop` strategy applied to prove the statement in (3). After applying the rule with the correct invariant condition three subgoals are created that correspond to Equation 4 (`discrete_loop_example.1`), Equation 6 (`discrete_loop_example.2`), and Equation 5 (`discrete_loop_example.3`).

Similarly, the strategy `(dl-diffinv)` captures the differential invariant rule specified and proven in the lemma `dl_dI`, in addition to automated procedures to calculate derivatives. Figure 3 shows this strategy being used to prove the statement in Equation 7. Note that `(dl-diffinv)` automatically calculates `ALLRUNS(ASSIGN_DIFT(ode)(nnP)) = (val(v) >= cnst(0))` and the proof is discharged without user instantiation or user input about differentiation. The cases generated in Figure 3 correspond to the expressions in Equations 8 and 9.

5.2 dL-PVS in Visual Studio Code

A Visual Studio Code extension is under development that will allow users of dL-PVS to specify hybrid programs in a way that is aesthetically similar to the traditional syntax of dL, see Figure 4. This capability will also be available during proof sessions, allowing the user to reason about hybrid programs in stylized syntax rather than through the

```
diff_inv_ex :

    |-------
{1}   ((: val(x) >= cnst(1), val(a) >= cnst(0) :) |-
        (: ALLRUNS(DIFF((: (x, val(v)), (v, val(a)) :), val(v) >= cnst(0)),
                   val(x) >= cnst(1)) :))

>> (dl-diffinv)
diff_inv_ex.1 :

    |-------
{1}   ((: val(v) >= cnst(0), (val(x) >= cnst(1)), val(a) >= cnst(0) :) |-
        (: (val(x) >= cnst(1)) :))

>> (dl-assert)
diff_inv_ex.2 :

    |-------
{1}   ((: val(v) >= cnst(0) :) |- (: val(v) >= cnst(0) :))

>> (dl-assert)
Q.E.D.
```

Figure 3. Example of using the differential invariant strategy in PVS

```
% Assignment with star
test1: PROBLEM
  (x>=0) |- [(x:=x + 1)*] (x>=0)
% Assignment with test
test2: PROBLEM
  (x=0) |- [?x>0;x:=x + 1] (x=0)
% Differential equations
test3: PROBLEM
  (x > 0) AND (y > 0) |-
     [x'= y, y'=x*y] (x > 0 AND y >0)
```

Figure 4. Example of the Visual Studio Code extension for specification of hybrid programs in dL-PVS

standard PVS interface. This work will be a continuation of the completed Visual Studio Code extension for PVS [13].

6 Conclusions and Future Work

This paper gives an overview of the effort to implement an operational embedding of dL in the theorem prover PVS. At this stage, the rules of dL have been proven in PVS, and many proof strategies are finished. The Visual Studio Code extension for interaction with the system is in active development. There are a number of directions for future work, such as additional specification capabilities and rules associated with liveness properties that could give the embedding of dL more capability. Specifically, reasoning about hybrid programs with an arbitrary number of variables will require additional work, as this ability is not part of the standard dL rule suite. Additionally, introducing more automation into the proving process, such as an automatic way to detect invariant conditions in hybrid programs, would benefit users of dL-PVS.

Acknowledgments

Research by the National Institute of Aerospace authors is supported by NASA under NASA/NIA Cooperative Agreement NNL09AA00A.

References

[1] Erika Ábrahám-Mumm, Ulrich Hannemann, and 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, 48–57. https://doi.org/10.1109/ICECCS.2001.930163

[2] Swee Balachandran, Christopher Manderino, César Muñoz, and María Consiglio. 2020. A decentralized framework to support uas merging and spacing operations in urban canyons. In *2020 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE, 204–210. https://doi.org/10.1109/ICUAS48674.2020.9213973

[3] Swee Balachandran, César Muñoz, and María Consiglio. 2018. Distributed consensus to enable merging and spacing of UAS in an urban environment. In *2018 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE, 670–675. https://doi.org/10.1109/ICUAS.2018.8453460

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

[5] Marc Daumas, David Lester, and César Munoz. 2008. Verified real number calculations: A library for interval arithmetic. *IEEE Trans. Comput.* 58, 2 (2008), 226–237. https://doi.org/10.1109/tc.2008.213

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

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

[8] Jonathan Julian Huerta y Munive. 2020. *Algebraic verification of hybrid systems in Isabelle/HOL*. Ph.D. Dissertation. University of Sheffield. https://etheses.whiterose.ac.uk/28886/

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

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

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

[12] Yanni Kouskoulas, David W. Renshaw, André Platzer, and Peter Kazanzides. 2013. Certifying the Safe Design of a Virtual Fixture Control Algorithm for a Surgical Robot. In *Hybrid Systems: Computation and Control (part of CPS Week 2013), HSCC'13, Philadelphia, PA, USA, April 8-13, 2013*, Calin Belta and Franjo Ivancic (Eds.). ACM, 263–272. https://doi.org/10.1145/2461328.2461369

[13] Paolo Masci and César A. Muñoz. 2019. An Integrated Development Environment for the Prototype Verification System. In *Proceedings Fifth Workshop on Formal Integrated Development Environment, F-IDE@FM 2019, Porto, Portugal, 7th October 2019 (EPTCS, Vol. 310)*, Rosemary Monahan, Virgile Prevosto, and José Proença (Eds.). 35–49. https://doi.org/10.4204/EPTCS.310.5

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

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

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

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

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

[19] Anthony Narkawicz, César Munoz, and 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 (2015), 285–326. https://doi.org/10.1007/s10817-015-9320-x

[20] Sam Owre, John M Rushby, and Natarajan Shankar. 1992. PVS: A prototype verification system. In *International Conference on Automated Deduction*. Springer, 748–752. https://doi.org/10.1007/3-540-55602-8_217

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

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

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

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

[25] André Platzer and Jan-David Quesel. 2008. KeYmaera: A hybrid theorem prover for hybrid systems (system description). In *International Joint Conference on Automated Reasoning*. Springer, 171–178. https://doi.org/10.1007/978-3-540-71070-7_15

[26] André Platzer and Jan-David Quesel. 2009. European Train Control System: A Case Study in Formal Verification. In *ICFEM (LNCS, Vol. 5885)*, Karin Breitman and Ana Cavalcanti (Eds.). Springer, 246–265. https://doi.org/10.1007/978-3-642-10373-5_13

[27] Jan-David Quesel, Stefan Mitsch, Sarah Loos, Nikos Aréchiga, and André Platzer. 2016. How to model and prove hybrid systems with KeYmaera: a tutorial on safety. *International Journal on Software Tools for Technology Transfer* 18, 1 (2016), 67–91. https://doi.org/10.1007/s10009-015-0367-0

[28] J Tanner Slagel, Lauren White, and 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*. 278–290. https://doi.org/10.1145/3437992.3439933

[29] 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*, Vol. 13047. Springer Nature, 367. https://doi.org/10.1007/978-3-030-90870-6_20

[30] Hannah S Walsh, Eleni Spirakis, Sequoia R Andrade, Daniel E Hulse, and Misty D Davies. 2020. *SMARt-STEReO: Preliminary concept of operations.* Technical Report. https://ntrs.nasa.gov/citations/20205007665