

HHLPy: Practical Verification of Hybrid Systems using Hoare Logic

Huanhuan Sheng^{1,2}, Alexander Bentkamp¹, and Bohua Zhan^{1,2}

¹ Institute of Software, Chinese Academy of Sciences, Beijing, China
{shenghh, bentkamp, bzhan}@ios.ac.cn

² University of Chinese Academy of Sciences, Beijing, China

Abstract. We present a tool for verification of hybrid systems expressed in the sequential fragment of HCSP (Hybrid Communicating Sequential Processes). The tool permits annotating HCSP programs with pre- and postconditions, invariants, and proof rules for reasoning about ordinary differential equations. Verification conditions are generated from the annotations following the rules of hybrid Hoare logic. We designed labeling and highlighting mechanisms to distinguish and visualize different verification conditions. The tool is implemented in Python and has a web-based user interface. We evaluated the effectiveness of the tool on translations of Simulink/Stateflow models and on KeYmaera X benchmarks.

Keywords: Hybrid systems · Hoare logic · Formal verification.

1 Introduction

Hybrid systems refer to systems that have both continuous and discrete behaviors. They occur in diverse areas of science and engineering, ranging from transportation and spaceflight, to robots and medical devices. Hence, verifying that hybrid systems meet certain specifications is an important problem. Apart from methods such as monitoring and model checking, theorem proving is one of the major approaches to verifying hybrid systems.

There is a substantial amount of previous work on verification of hybrid systems based on theorem proving. One major framework is Platzer’s differential dynamic logic ($d\mathcal{L}$) [21, 23], and the associated KeYmaera/KeYmaera X prover [9, 24]. Recently, a Hoare logic has been introduced for $d\mathcal{L}$ and implemented within the Isabelle proof assistant [19]. We review these works in detail in Section 8 of this paper.

Another approach is to model hybrid systems using HCSP (Hybrid CSP) [12, 33], an extension of CSP (Communicating Sequential Processes) to include continuous evolution. Its semantics of continuous evolution is more deterministic than $d\mathcal{L}$ ’s, so it can be used naturally for capturing Simulink/Stateflow models. A hybrid Hoare logic has been developed for HCSP, and is implemented in Isabelle [28]. However, the practical application of the tool is complicated by its steep learning curve. To use this tool, the user need to be familiar with the

Isabelle proof assistant and with the particular library of theorems. This is in stark contrast to KeYmaera X, which allows users to verify hybrid programs by choosing menu actions and offers highly specialized automation.

This paper introduces HHLPy³, a tool for verification of the sequential part of HCSP with a friendly graphical user interface. The tool is based on a Hoare logic for reasoning about sequential HCSP programs and supports various Hoare logic rules to verify the behavior of differential equations. These rules are closely related to rules in $d\mathcal{L}$, but due to the semantic differences of HCSP, we adapted some of the rules and informally proved our rules to be sound (Section 3).

Based on these Hoare logic rules for differential equations, we devised an automatic verification condition generation procedure that computes verification conditions (VCs) from a given annotated HCSP program (Section 4).

One slight difference between our work and traditional VC generation is that we express VCs as a set of conditions, rather than a single condition composing individual VCs with ‘ \wedge ’. With each VC, we associate a label to distinguish between different VCs, so that users can choose solvers (currently either Z3 [17] or Wolfram Engine [29]) for each VC individually (Section 5).

To visualize to the user where each VC originates from, a highlighting mechanism highlights the set of code fragments in the annotated program that contributed to generating the VC (Section 6).

We implemented the tool using Python and JavaScript and evaluated it on Simulink/Stateflow models and on KeYmaera X benchmarks (Section 7). We translated two Simulink/Stateflow models using the toolchain developed by Zou et al. [34, 35] and verified them in our tool. Due to differences in the semantics of $d\mathcal{L}$ and HCSP, we translated each KeYmaera X benchmark by hand, trying to maintain semantic equivalence as much as possible. In this way, we succeeded to use our tool to solve most of the verification problems in the basic and nonlinear KeYmaera X benchmarks.

2 Preliminaries

In this section, we present the sequential fragment of HCSP, with an informal explanation of its semantics. We further give an overview of the existing toolchain on translation of Simulink/Stateflow models into HCSP.

2.1 Sequential fragment of HCSP

Hybrid CSP (HCSP), introduced in [12, 33], is an extension of Hoare’s Communicating Sequential Processes to include continuous evolution. It can model communicating processes running in parallel, where each process may have both continuous and discrete behavior. In this paper, we focus on the sequential fragment of HCSP, consisting of the following commands:

$$S, T ::= \text{skip} \mid x := e \mid x := * (B) \mid S; T \mid \text{if } B \text{ then } S \text{ else } T \mid S ++ T \mid S* \\ \mid \langle \dot{x} = e \ \& \ D \rangle$$

³ The tool is available at <https://github.com/bzhan/mars/tree/master/hhlpy>

Here `skip` does nothing. $x := e$ assigns the value of expression e to variable x . $x := *(B)$ is nondeterministic assignment of some value satisfying condition B to x . $S;T$ and `if B then S else T` are regular sequential composition and conditional. $S++T$ is a nondeterministic choice between S and T . $S*$ runs S a nondeterministic number of times (including zero).

The ordinary differential equation (ODE) command $\langle \dot{\mathbf{x}} = \mathbf{e} \ \& \ D \rangle$ specifies continuous evolution in HCSP. It makes the vector of variables \mathbf{x} evolve according to ODE $\dot{\mathbf{x}} = \mathbf{e}$ until the domain D becomes false. If D is false from the start, the ODE is skipped. In contrast to $\text{d}\mathcal{L}$, where continuous evolution may stop at any point within the specified domain, in HCSP it always continues up to the boundary. In this paper, we assume D is given by a polynomial inequality of the form $p(\mathbf{x}) < 0$, so it represents an open set in \mathbb{R}^n .

We assume in this paper that all expressions appearing in an HCSP program (as well as in annotations to be discussed later) are polynomials, hence continuity conditions are trivially satisfied.

For a formal treatment of semantics of HCSP (including communication and parallel composition), we refer to Zhan et al. [31, Chapter 6].

2.2 Translation from Simulink/Stateflow

The HCSP language is located at the center of a toolchain that also includes translation from Simulink/Stateflow models, simulation and code generation [2]. The original translation algorithms from Simulink [35] and Stateflow [34] produce HCSP programs that involve communication between parallel processes. However, more recent methods by Xu et al. [30] and Guo et al. [10] produce sequential HCSP programs. We use these translation methods for verification of Simulink/Stateflow models in Section 7.

3 Proof Rules of Hybrid Hoare Logic

In this section, we present the Hoare logic that forms the basis of our verification tool. The Hoare triple for partial correctness, written as $\{P\}c\{Q\}$, means starting from a state satisfying assertion P , any terminating execution of c reaches a state satisfying assertion Q . The Hoare rules for ordinary commands (except ODEs) are standard, and are presented in Appendix A.

Hence, we focus on the Hoare rules for ODEs. These rules are mostly adapted from rules for $\text{d}\mathcal{L}$, as given in [23, 26]. Due to the difference in semantics between HCSP and $\text{d}\mathcal{L}$, several of the rules take on different forms. We do not aim to present a minimal set of rules, instead providing users a wide range of choices.

3.1 Proof rules based on invariants

In order to state proof rules based on invariants of ODEs, we require an additional kind of judgments, called invariant triples.

Definition 1 (Invariant Triple). We say that Q is an invariant of ODE $\dot{\mathbf{x}} = \mathbf{e}$ under domain P , written as

$$\llbracket P \rrbracket \langle \dot{\mathbf{x}} = \mathbf{e} \rangle \llbracket Q \rrbracket$$

if for any solution $\gamma : [0, T] \rightarrow \mathbb{R}^n$ such that $\gamma(t)$ satisfies P for all $t \in [0, T]$ and such that $\gamma(0)$ satisfies Q , $\gamma(t)$ satisfies Q for all $t \in [0, T]$.

Differential Weakening The differential weakening rule (dW) reduces a Hoare triple goal to an invariant triple, incorporating the domain condition.

$$\frac{\llbracket \bar{D} \rrbracket \langle \dot{\mathbf{x}} = \mathbf{e} \rangle \llbracket I \rrbracket \quad \partial D \wedge I \rightarrow Q}{\{(D \rightarrow I) \wedge (\neg D \rightarrow Q)\} \langle \dot{\mathbf{x}} = \mathbf{e} \rangle \{Q\}} \text{dW}$$

Note the rule is in the form that allows us to derive a precondition from any postcondition. The precondition $(D \rightarrow I) \wedge (\neg D \rightarrow Q)$ corresponds to the two cases for the state before ODE: if the state satisfies domain D , then it should satisfy the invariant. Otherwise it should satisfy the postcondition Q directly. An informal proof for the (dW) rule (as well as for all of the ODE rules below) is given in Appendix A. Two special cases of the rule, for I set to **true** and **false**, provide further intuition. They correspond to cases where no invariant is needed, and where the starting state is known to satisfy $\neg D$.

$$\frac{\partial D \rightarrow Q}{\{\neg D \rightarrow Q\} \langle \dot{\mathbf{x}} = \mathbf{e} \rangle \{Q\}} \text{dWT} \quad \frac{}{\{\neg D \wedge Q\} \langle \dot{\mathbf{x}} = \mathbf{e} \rangle \{Q\}} \text{dWF}$$

Differential Invariant The differential invariant rule (dI) is essentially the same as that in $\text{d}\mathcal{L}$. It concludes invariants from computation of Lie derivatives.

$$\frac{P \rightarrow \dot{f} = 0}{\llbracket P \rrbracket \langle \dot{\mathbf{x}} = \mathbf{e} \rangle \llbracket f = 0 \rrbracket} \text{dI}_=$$

Here \dot{f} denotes the Lie derivative of f under the differential equation $\dot{\mathbf{x}} = \mathbf{e}$. The corresponding rules for inequality and disequality are as follows, where \succcurlyeq denotes either $>$ or \geq .

$$\frac{P \rightarrow \dot{f} \geq 0}{\llbracket P \rrbracket \langle \dot{\mathbf{x}} = \mathbf{e} \rangle \llbracket f \succcurlyeq 0 \rrbracket} \text{dI}_{\succcurlyeq} \quad \frac{P \rightarrow \dot{f} = 0}{\llbracket P \rrbracket \langle \dot{\mathbf{x}} = \mathbf{e} \rangle \llbracket f \neq 0 \rrbracket} \text{dI}_{\neq}$$

Differential Cut The differential cut rule (dC) inserts an intermediate invariant to be proved, and afterwards permits the use of this invariant to show further invariants. In contrast to $\text{d}\mathcal{L}$, it is not possible to record previously proved invariants as conjuncts in the domain. Instead we place them in the premise of the invariant triple. Indeed this is the primary motivation for introducing the concept of invariant triples.

$$\frac{\llbracket P \rrbracket \langle \dot{\mathbf{x}} = \mathbf{e} \rangle \llbracket Q_1 \rrbracket \quad \llbracket P \wedge Q_1 \rrbracket \langle \dot{\mathbf{x}} = \mathbf{e} \rangle \llbracket Q_2 \rrbracket}{\llbracket P \rrbracket \langle \dot{\mathbf{x}} = \mathbf{e} \rangle \llbracket Q_1 \wedge Q_2 \rrbracket} \text{dC}$$

The (dC) rule can be used multiple times to show conjunction of more than two invariants. For example, if we wish to show three invariants Q_1, Q_2, Q_3 in that order, first apply the (dC) rule with Q_1 and Q_2 to obtain $\llbracket P \rrbracket \langle \dot{\mathbf{x}} = \mathbf{e} \rangle \llbracket Q_1 \wedge Q_2 \rrbracket$, then apply the (dC) rule again as follows:

$$\frac{\llbracket P \rrbracket \langle \dot{\mathbf{x}} = \mathbf{e} \rangle \llbracket Q_1 \wedge Q_2 \rrbracket \quad \llbracket P \wedge Q_1 \wedge Q_2 \rrbracket \langle \dot{\mathbf{x}} = \mathbf{e} \rangle \llbracket Q_3 \rrbracket}{\llbracket P \rrbracket \langle \dot{\mathbf{x}} = \mathbf{e} \rangle \llbracket Q_1 \wedge Q_2 \wedge Q_3 \rrbracket} \text{dC}$$

This permits Q_1 to be used when showing invariance of Q_2 , and both Q_1 and Q_2 to be used when showing invariance of Q_3 .

Differential Ghost The differential ghost rule (dG) adds new variables satisfying some differential equations to help prove the Hoare triple of the original differential equations.

$$\frac{\llbracket \overline{D} \rrbracket \langle \dot{\mathbf{x}} = \mathbf{e}, \dot{\mathbf{y}} = \mathbf{f}(\mathbf{x}, \mathbf{y}) \rangle \llbracket I \rrbracket \quad \partial D \wedge I \rightarrow Q}{\{(D \rightarrow \exists \mathbf{y}. I) \wedge (\neg D \rightarrow Q)\} \langle \dot{\mathbf{x}} = \mathbf{e} \& D \rangle \{Q\}} \text{dG}$$

Here, \mathbf{y} are fresh variables that do not occur in $\langle \dot{\mathbf{x}} = \mathbf{e} \& D \rangle$ or Q , and $\mathbf{f}(\mathbf{x}, \mathbf{y})$ satisfies the Lipschitz condition.

Barrier Certificate The barrier certificate rule (bc) concludes invariants from the definition of barrier certificate.

$$\frac{P \wedge f = 0 \rightarrow \dot{f} > 0}{\llbracket P \rrbracket \langle \dot{\mathbf{x}} = \mathbf{e} \rangle \llbracket f \succ 0 \rrbracket} \text{bc}$$

Here, e is a continuous function. f is a 1-time continuously differentiable function.

Darboux The Darboux rule (dbx) exploits properties of Darboux invariants, which are inspired by Darboux polynomials. Darboux equality and inequality rules are as follows.

$$\frac{P \rightarrow \dot{f} = gf}{\llbracket P \rrbracket \langle \dot{\mathbf{x}} = \mathbf{e} \rangle \llbracket f = 0 \rrbracket} \text{dbx}_= \quad \frac{P \rightarrow \dot{f} \geq gf}{\llbracket P \rrbracket \langle \dot{\mathbf{x}} = \mathbf{e} \rangle \llbracket f \succ 0 \rrbracket} \text{dbx}_{\succ}$$

Here, f and g are polynomials of \mathbf{x} .

3.2 Solution Rule

The solution rule offers another way to conclude Hoare triples directly, independent of using the (dW) or (dG) rule followed by proving invariants. In the rule below, e is linear in \mathbf{x} , and $\mathbf{u}(t, \mathbf{x})$ is the unique solution to the differential

equation $\dot{\mathbf{x}} = \mathbf{e}$ with symbolic initial value \mathbf{x} (that is, $\frac{d\mathbf{u}(t, \mathbf{x})}{dt} = \mathbf{e}(\mathbf{u}(t, \mathbf{x}))$ and $\mathbf{u}(0, \mathbf{x}) = \mathbf{x}$). Let $P'(\mathbf{x})$ denote the following predicate on the starting state \mathbf{x} :

$$\forall t > 0. (\forall 0 \leq \tau < t. D(\mathbf{u}(\tau, \mathbf{x})) \wedge \neg D(\mathbf{u}(t, \mathbf{x})) \rightarrow Q(\mathbf{u}(t, \mathbf{x}))).$$

The solution rule for Hoare triples (sln) is:

$$\frac{}{\{(D \rightarrow P') \wedge (\neg D \rightarrow Q)\} \langle \dot{\mathbf{x}} = \mathbf{e} \ \& \ D \rangle \{Q\}} \text{sln}$$

4 Verification Condition Generation

The VC generation procedure operates on annotated sequential HCSP programs. For ODEs, there are two kinds of annotations: ghost variable (`gvar`) and invariant annotations (`ode_inv`):

$$\begin{aligned} \text{gvar} &::= \text{ghost } z \ (\dot{z} = f(\mathbf{x}, z)) \\ \text{ode_inv} &::= [I] \mid [I] \ \{\text{dbx } c\} \mid [I] \ \{\text{bc}\} \end{aligned}$$

Here, ‘ghost z ($\dot{z} = f(\mathbf{x}, z)$)’ denotes a ghost variable z following the ODE $\dot{z} = f(\mathbf{x}, z)$, where f must be linear in z to ensure global Lipschitz condition. The annotation $[I]$ denotes showing invariant I using the (dI) rule. The annotation $[I] \ \{\text{dbx } c\}$ denotes showing an invariant using the (dbx) rule, with c being the optional cofactor. The annotation $[I] \ \{\text{bc}\}$ denotes using the (bc) rule.

The syntax for annotated sequential HCSP programs is:

$$\begin{aligned} \mathcal{S}, \mathcal{T} &::= \text{skip} \mid x := e \mid x := *(B) \mid \mathcal{S}; \mathcal{T} \mid \text{if } B \text{ then } \mathcal{S} \text{ else } \mathcal{T} \mid \\ &\mathcal{S} \text{ ++ } \mathcal{T} \mid \mathcal{S} * \text{ invariant } [I_1] \dots [I_n] \mid \\ &\langle \dot{\mathbf{x}} = \mathbf{e} \ \& \ D \rangle \text{ invariant } \text{gvar}_1 \dots \text{gvar}_k, \text{ode_inv}_1 \dots \text{ode_inv}_n \mid \\ &\langle \dot{\mathbf{x}} = \mathbf{e} \ \& \ D \rangle \text{ solution} \end{aligned}$$

The only addition to the syntax of HCSP is that each loop is followed by a list of invariants I_1, \dots, I_n , and each ODE is either followed by a list of ghost variable declarations and a list of invariant annotations, each of which specify an invariant to be proved using one of (dI), (dbx), or (bc) rules, or followed by the annotation “solution” to indicate that the (sln) rule is to be used.

To generate the necessary VCs for a given Hoare triple, we devised a procedure using weakest preconditions [3, 4]. To be able to refer to preconditions and VCs individually, we consider sets of conditions instead of composing predicates by \wedge .

Given a Hoare triple $\{P_1 \wedge \dots \wedge P_m\} \mathcal{S} \{Q_1 \wedge \dots \wedge Q_n\}$ to verify, we define the set of all VCs to be

$$\begin{aligned} \text{VC}(\{P_1 \wedge \dots \wedge P_m\} \mathcal{S} \{Q_1 \wedge \dots \wedge Q_n\}) = \\ \{P_1 \wedge \dots \wedge P_m \rightarrow R \mid R \in \text{pre}(\mathcal{S}, \{Q_1, \dots, Q_n\})\} \cup \quad (\text{pre}) \\ \{\tilde{P}_1 \wedge \dots \wedge \tilde{P}_{\tilde{m}} \rightarrow R \mid R \in \text{vc}(\mathcal{S}, \{Q_1, \dots, Q_n\})\} \quad (\text{vc}) \end{aligned}$$

where $\tilde{P}_1, \dots, \tilde{P}_{\tilde{m}}$ is the subset of the preconditions P_1, \dots, P_m whose variables are never reassigned in \mathcal{S} , and the functions `pre` and `vc` are defined below.

Given an annotated program \mathcal{S} and a set $\{Q_1, \dots, Q_n\}$ of postconditions, we denote the set of derived preconditions as $\text{pre}(\mathcal{S}, \{Q_1, \dots, Q_n\})$, defined as follows.

$$\begin{aligned}
 \text{pre}(\mathcal{S}, \{Q_1, \dots, Q_n\}) &= \text{pre}(\mathcal{S}, Q_1) \cup \dots \cup \text{pre}(\mathcal{S}, Q_n) && \text{(pre-multi)} \\
 \text{pre}(\text{skip}, Q) &= Q && \text{(pre-skip)} \\
 \text{pre}(x := e, Q) &= Q[e/x] && \text{(pre-assn)} \\
 \text{pre}(\mathcal{S}; \mathcal{T}, Q) &= \text{pre}(\mathcal{S}, \text{pre}(\mathcal{T}, Q)) && \text{(pre-seq)} \\
 \text{pre}(\text{if } B_1 \text{ then } \mathcal{S}_1 \text{ else if } \dots \text{ else if } B_{n-1} \text{ then } \mathcal{S}_{n-1} \text{ else } \mathcal{S}_n, Q) &= \\
 &\quad \{\neg(B_1 \vee \dots \vee B_{i-1}) \wedge B_i \rightarrow P \mid P \in \text{pre}(\mathcal{S}_i, Q), 1 \leq i \leq n-1\} \cup && \text{(pre-if)} \\
 &\quad \{\neg(B_1 \vee \dots \vee B_{n-1}) \rightarrow P \mid P \in \text{pre}(\mathcal{S}_n, Q)\} && \text{(pre-else)} \\
 \text{pre}(\mathcal{S}_1 ++ \dots ++ \mathcal{S}_n, Q) &= \text{pre}(\mathcal{S}_1, Q) \cup \dots \cup \text{pre}(\mathcal{S}_n, Q) && \text{(pre-choice)} \\
 \text{pre}(x := * (B), Q) &= B[y/x] \rightarrow Q[y/x] \text{ for a fresh variable } y && \text{(pre-nassn)} \\
 \text{pre}(\mathcal{S} * \text{invariant } [I_1] \dots [I_n], Q) &= \{I_j \mid 1 \leq j \leq n\} && \text{(pre-loop)} \\
 \text{pre}(\langle \dot{\mathbf{x}} = e \ \& \ D \rangle \text{invariant } \text{gvar}_1 \dots \text{gvar}_k, \text{ode.invs}_1 \dots \text{ode.invs}_n, Q) &= \\
 &\quad P_{\text{skip}} \cup P_{\text{init}} \\
 \text{pre}(\langle \dot{\mathbf{x}} = e \ \& \ D \rangle \text{solution}) &= P_{\text{skip}} \cup P_{\text{sln}}
 \end{aligned}$$

where

$$\begin{aligned}
 P_{\text{skip}} &= \{\neg D \rightarrow Q\} && \text{(pre-dWG-skip)} \\
 P_{\text{init}} &= \{D \rightarrow \exists z_1, \dots, z_k. I_1 \wedge \dots \wedge I_n\} \text{ if } k > 0 && \text{(pre-dG-init)} \\
 P_{\text{init}} &= \{D \rightarrow I_j, \mid 1 \leq j \leq n\} \text{ otherwise} && \text{(pre-dW-init)} \\
 P_{\text{sln}} &= \{D \rightarrow (\forall t > 0. (\forall 0 \leq \tau < t. D(\mathbf{u}(\tau, \mathbf{x}))) \wedge \\
 &\quad \neg D(\mathbf{u}(t, \mathbf{x})) \rightarrow Q(\mathbf{u}(t, \mathbf{x})))\} && \text{(pre-sln)}
 \end{aligned}$$

where z_1, \dots, z_k are the ghost variables provided in $\text{gvar}_1 \dots \text{gvar}_k$, and I_1, \dots, I_n are the invariants provided in $\text{ode.invs}_1, \dots, \text{ode.invs}_n$. If user chooses the (sln) rule, we verify that e is linear in \mathbf{x} and compute the unique solution $\mathbf{u}(\tau, \mathbf{x})$ to the ODE with symbolic initial value \mathbf{x} .

Given an annotated program \mathcal{S} and a set $\{Q_1, \dots, Q_n\}$ of postconditions, we denote the set of internal VCs as $\text{vc}(\mathcal{S}, \{Q_1, \dots, Q_n\})$, defined as follows.

$$\begin{aligned}
 \text{vc}(\mathcal{S}, \{Q_1, \dots, Q_n\}) &= \text{vc}(\mathcal{S}, Q_1) \cup \dots \cup \text{vc}(\mathcal{S}, Q_n) && \text{(vc-multi)} \\
 \text{vc}(\text{skip}, Q) &= \emptyset && \text{(vc-skip)} \\
 \text{vc}(x := e, Q) &= \emptyset && \text{(vc-assn)} \\
 \text{vc}(\mathcal{S}; \mathcal{T}, Q) &= \text{vc}(\mathcal{S}, \text{pre}(\mathcal{T}, Q)) \cup \text{vc}(\mathcal{T}, Q) && \text{(vc-seq)} \\
 \text{vc}(\text{if } B_1 \text{ then } \mathcal{S}_1 \text{ else if } \dots \text{ else if } B_{n-1} \text{ then } \mathcal{S}_{n-1} \text{ else } \mathcal{S}_n, Q) &= \\
 &\quad \text{vc}(\mathcal{S}_1, Q) \cup \dots \cup \text{vc}(\mathcal{S}_n, Q) && \text{(vc-ite)} \\
 \text{vc}(\mathcal{S}_1 ++ \dots ++ \mathcal{S}_n, Q) &= \text{vc}(\mathcal{S}_1, Q) \cup \dots \cup \text{vc}(\mathcal{S}_n, Q) && \text{(vc-choice)} \\
 \text{vc}(x := * (B), Q) &= \emptyset && \text{(vc-nassn)} \\
 \text{vc}(\mathcal{S} * \text{invariant } [I_1] \dots [I_n], Q) &= \\
 &\quad \text{vc}(\mathcal{S}, \{I_1, \dots, I_n\}) \cup && \text{(vc-loop-body)} \\
 &\quad \{(I_1 \wedge \dots \wedge I_n) \rightarrow Q\} \cup && \text{(vc-loop-exit)} \\
 &\quad \{(I_1 \wedge \dots \wedge I_n) \rightarrow P \mid P \in \text{pre}(\mathcal{S}, \{I_1, \dots, I_n\})\} && \text{(vc-loop-maintain)} \\
 \text{vc}(\langle \dot{\mathbf{x}} = e \ \& \ D \rangle \text{invariant } \text{gvar}_1 \dots \text{gvar}_m, \text{ode.invs}_1 \dots \text{ode.invs}_n, Q) &= C_{\text{exec}} \cup C_{\text{dC}} \\
 \text{vc}(\langle \dot{\mathbf{x}} = e \ \& \ D \rangle \text{solution}, Q) &= \emptyset
 \end{aligned}$$

where we set $C_{\text{exec}} = \emptyset$ if the only invariant is `false`, or else

$$C_{\text{exec}} = \{I_1 \wedge \dots \wedge I_n \wedge \partial D \rightarrow Q\} \quad (\text{vc-dWG-exec})$$

$$C_{\text{dC}} = \{I_1 \wedge \dots \wedge I_{j-1} \rightarrow R \mid R \in \text{vc}(\langle \dot{\mathbf{x}} = \mathbf{e} \ \& \ D \rangle, \text{ode_inv}_j, Q), \\ 1 \leq j \leq n\} \quad (\text{vc-dC})$$

Here, I_1, \dots, I_n are the invariants provided in `ode_inv_1, \dots, ode_inv_n`. If no invariants are specified, we set a single invariant $I_1 = \text{true}$ by default. We write $\text{vc}(\langle \dot{\mathbf{x}} = \mathbf{e} \ \& \ D \rangle, \text{ode_inv}_j, Q)$ for the VC generated from annotation `ode_inv_j`, defined as follows.

$$\begin{aligned} \text{vc}(\langle \dot{\mathbf{x}} = \mathbf{e} \ \& \ D \rangle, [\text{true}], Q) &= \emptyset && (\text{vc-true}) \\ \text{vc}(\langle \dot{\mathbf{x}} = \mathbf{e} \ \& \ D \rangle, [\text{false}], Q) &= \emptyset && (\text{vc-false}) \\ \text{vc}(\langle \dot{\mathbf{x}} = \mathbf{e} \ \& \ D \rangle, [f = 0], Q) &= \{\bar{D} \rightarrow \dot{f} = 0\} && (\text{vc-dI1}) \\ \text{vc}(\langle \dot{\mathbf{x}} = \mathbf{e} \ \& \ D \rangle, [f \succcurlyeq 0], Q) &= \{\bar{D} \rightarrow \dot{f} \geq 0\} && (\text{vc-dI2}) \\ \text{vc}(\langle \dot{\mathbf{x}} = \mathbf{e} \ \& \ D \rangle, [f \neq 0], Q) &= \{\bar{D} \rightarrow \dot{f} = 0\} && (\text{vc-dI3}) \\ \text{vc}(\langle \dot{\mathbf{x}} = \mathbf{e} \ \& \ D \rangle, [f = 0] \{ \text{dbx } g \}, Q) &= \{\bar{D} \rightarrow \dot{f} = gf\} && (\text{vc-dbx1}) \\ \text{vc}(\langle \dot{\mathbf{x}} = \mathbf{e} \ \& \ D \rangle, [f \succcurlyeq 0] \{ \text{dbx } g \}, Q) &= \{\bar{D} \rightarrow \dot{f} \geq gf\} && (\text{vc-dbx2}) \\ \text{vc}(\langle \dot{\mathbf{x}} = \mathbf{e} \ \& \ D \rangle, [f \succcurlyeq 0] \{ \text{bc} \}, Q) &= \{\bar{D} \wedge f = 0 \rightarrow \dot{f} > 0\} && (\text{vc-bc}) \end{aligned}$$

All Lie derivatives are computed with respect to $\dot{\mathbf{x}} = \mathbf{e}$ and the equations given in `gvar_1 \dots gvar_m`. For the (dbx) rule, if no cofactor g is provided, we attempt to compute the cofactor automatically. In the case of an equality invariant, this reduces to simplifying \dot{f}/f into polynomial form. In the case of an inequality invariant, we attempt to find a polynomial quotient of \dot{f} and f with a non-negative remainder.

Theorem 1. *A Hoare triple $\{P_1 \wedge \dots \wedge P_m\} \mathcal{T} \{Q_1 \wedge \dots \wedge Q_n\}$ holds if all conditions in $\text{VC}(\{P_1 \wedge \dots \wedge P_m\} \mathcal{T} \{Q_1 \wedge \dots \wedge Q_n\})$ hold.*

Proof. We give the full proof in Appendix B. In short, we proceed by structural induction on \mathcal{T} . The difficult case is when \mathcal{T} is an ODE. We apply the (sln), (dW) or (dG) rule, depending on the annotations. In the cases of (dW) and (dG), we proceed to isolate each invariant by applying the (dC) rule. Then we use (dI), (dbx) or (bc) rules, depending on the annotation, to conclude the proof for each invariant.

5 Labels

VCS generated by the procedure in Section 4 will be proved using Z3 or Wolfram Engine. In this section, we introduce a labeling mechanism to store which solver is used for each VC, in a way that is robust to minor modifications of the program or its annotations.

As indicated in Section 4, the generation of a VC starts from a postcondition or invariant and proceeds bottom up through the program. We call the postcondition or invariant at the beginning of this process the conclusion assertion

of the VC. We associate each VC to its conclusion assertion. Labels are used to distinguish between multiple VCs from the same conclusion assertion. They can arise for the following reasons:

- Loop and ODE invariants produce VCs for showing that they initially hold, and for showing that they are maintained by the loop or ODE.
- If-then-else and nondeterministic choice produce multiple preconditions, at least one for each branch.
- Each ODE produces preconditions for both the case when the domain D holds initially, and for when D does not hold.

A label consists of two parts: a category label and a branch label. The category label is either empty or one of “init”, “maintain”, “init_all”. The branch label is a list, separated by “.”, of either “skip”, “exec”, or $n(b)$ where n is a positive integer and b is a branch label itself. We write n instead of $n()$ when the inner branch label is empty.

Category Labels Category labels use “init” (“init_all”) and “maintain” to distinguish between VCs with loop or ODE invariants as conclusion assertions. For loops, the VCs for showing the invariant holds initially are labeled “init”, and the VCs that result from showing the invariant is maintained by the loop are labeled “maintain” (when there are nested loops or ODEs in the loop body, multiple VCs are computed in the loop body, this applies only to those with the invariant as conclusion assertion).

For an ODE, the VC coming from (pre-dW-init) (resp. (pre-dG-init)) are labeled “init” (resp. “init_all”). The VCs coming from (vc-dC), for showing each invariant is maintained during evolution, are labeled “maintain”.

The category label is empty in all other cases.

Branch Labels Branch labels help to distinguish VCs generated by executing different branches of programs.

The positive integer n handles branches created by ‘if B then \mathcal{S}_1 else \mathcal{S}_2 ’ or ‘ $\mathcal{S}_1 ++ \mathcal{S}_2$ ’. Each value of n (starting from 1) corresponds to one branch. Sequence labels $b.b$ are used for sequences of such commands. For example, the branches for ‘ $\mathcal{S}_1 ++ \mathcal{S}_2; \mathcal{S}_3 ++ \mathcal{S}_4$ ’ have labels 1.1, 1.2, 2.1 and 2.2. Nested labels $n(b)$ are used for nested commands. For example, the branches for if B then $\mathcal{S}_1 ++ \mathcal{S}_2$ else \mathcal{S}_3 have labels 1(1), 1(2) and 2, corresponding to \mathcal{S}_1 , \mathcal{S}_2 and \mathcal{S}_3 , respectively.

The labels “skip” and “exec” are used for branches of the ODE. The branch where the initial state does not satisfy domain D is labeled “skip”. The other branch, where the ODE is executed, is labeled “exec”. They come from applying the rules (vc-dWG-skip) and (vc-dWG-exec), respectively.

Example 1. This example illustrates assignments, nondeterministic choice, and loops.

$$\begin{aligned}
& \{x \leq 0\} \\
& x := -x; \\
& (x := x + 1 \text{ ++ } x := x + 2)^* \\
& \text{invariant } [x \geq 0]; \\
& x := x + 1 \\
& \{x \geq 1\}
\end{aligned}$$

The computation starts at postcondition $x \geq 1$. Applying (pre-assn) and (vc-loop-exit), we get the VC $x \geq 0 \rightarrow x + 1 \geq 1$. Applying (pre-loop), the loop's precondition is $x \geq 0$. The whole program's precondition is $-x \geq 0$ by applying (pre-assn) again. The loop body yields the preconditions $x + 1 \geq 0$ and $x + 2 \geq 0$ by (pre-choice) and (pre-assn). Then we get $x \geq 0 \rightarrow x + 1 \geq 0$ and $x \geq 0 \rightarrow x + 2 \geq 0$ by applying (vc-loop-maintain). The overall VCs and their labels are:

VC	Conclusion Assertion	Label
$x \leq 0 \rightarrow -x \geq 0$	$x \geq 0$ (inv)	init
$x \geq 0 \rightarrow x + 1 \geq 1$	$x \geq 1$ (post)	ϵ
$x \geq 0 \rightarrow x + 1 \geq 0$	$x \geq 0$ (inv)	maintain 1
$x \geq 0 \rightarrow x + 2 \geq 0$	$x \geq 0$ (inv)	maintain 2

Example 2. This example illustrates non-deterministic, assignments and ODEs (#4 of KeYmaera X's basic benchmarks):

$$\begin{aligned}
& \{x \geq 0\} \\
& x := x + 1; t := * (t \geq 0); \\
& \langle \dot{t} = -1, \dot{x} = 2 \ \& \ t > 0 \rangle \text{ invariant } [x \geq 1] \\
& \{x \geq 1\}
\end{aligned}$$

The computation of pre starts at postcondition $x \geq 1$. By (pre-dWG-skip) and (pre-dW-init), the ODE's preconditions are $\neg t > 0 \rightarrow x \geq 1$ and $t > 0 \rightarrow x \geq 1$. By (pre-nassn) and (pre-assn), the whole program's preconditions are $t_1 \geq 0 \rightarrow t_1 > 0 \rightarrow x \geq 1$ and $t_1 \geq 0 \rightarrow \neg t_1 > 0 \rightarrow x + 1 \geq 1$. The VCs $x \geq 1 \wedge t = 0 \rightarrow x + 1 \geq 1$ and $t \geq 0 \rightarrow 2 \geq 0$ come from (vc-dWG-exec) and (vc-dI2), respectively. The overall list of VCs is:

VC	Conclusion Assertion	Label
$x \geq 0 \rightarrow t_1 \geq 0 \rightarrow t_1 > 0 \rightarrow x + 1 \geq 1$	$x \geq 1$ (inv)	init
$x \geq 0 \rightarrow t_1 \geq 0 \rightarrow \neg t_1 > 0 \rightarrow x + 1 \geq 1$	$x \geq 1$ (post)	skip
$x \geq 1 \wedge t = 0 \rightarrow x \geq 1$	$x \geq 1$ (post)	exec
$t \geq 0 \rightarrow 2 \geq 0$	$x \geq 1$ (inv)	maintain

Example 3. Finally, we consider an example with multiple ghost variables (#18 of KeYmaera X's basic benchmarks):

$$\begin{aligned}
& \{x \geq 0\} \\
& t := * (t \geq 0); \langle \dot{x} = x, \dot{t} = -1 \ \& \ t > 0 \rangle; \\
& \text{invariant ghost } y (\dot{y} = -y) \text{ ghost } z (\dot{z} = z/2) \\
& [xy \geq 0] [yz^2 = 1]; \\
& \{x \geq 0\}
\end{aligned}$$

VC	Conclusion Assertion	Label
$x \geq 0 \rightarrow t_1 \geq 0 \rightarrow t_1 > 0 \rightarrow \exists y z. xy \geq 0 \wedge yz^2 = 1$	invariants	init_all
$x \geq 0 \rightarrow t_1 \geq 0 \rightarrow \neg t_1 > 0 \rightarrow x \geq 0$	$x \geq 0$ (post)	skip
$xy \geq 0 \wedge yz^2 = 1 \wedge t = 0 \rightarrow x \geq 0$	$x \geq 0$ (post)	exec
$t \geq 0 \rightarrow x \cdot (-y) + xy \geq 0$	$xy \geq 0$ (inv)	maintain
$xy \geq 0 \rightarrow t \geq 0 \rightarrow yz(z/2) + (y(z/2) + (-y)z)z = 0$	$yz^2 = 1$ (inv)	maintain

The first VC comes from (pre-dG-init). The remaining VCs are similar to Example 2, except that there is one VC for maintaining each invariant. When verifying the second invariant, the (dC) rule allows us to assume the first invariant.

6 Highlighting

In this section, we explain the highlighting mechanism we devised to help the user understand how each VC is derived from the program. Essentially, when the user hovers over a VC, we highlight all parts of the program that contributes to the computation of the VC, including commands, assertions and domain constraints. Why3 [5] has a similar mechanism, but the highlighting is less detailed.

We highlight any assertion that contributes to the VC. In particular, invariants of an ODE that are already proved will be highlighted when proving the next invariant because they are added as assumptions in (vc-dC). Preconditions whose variables are never reassigned will be highlighted because they are added as assumptions in (vc).

Domain constraints of ODEs will be highlighted if they are used in the VC (e.g. the domain constraint D in the VC generated by (vc-dWG-exec)).

Atomic commands are highlighted if they are traversed during VC generation. ODE commands are highlighted for VCs computed by (vc-dC) or (pre-sln). For if-then-else and nondeterministic choice, only the branch that is actually traversed during VC generation will be considered for highlighting.

Fig. 1 and 2 show the highlighting for some the VCs from Examples 1 and 2.

7 Implementation and Evaluation

In this section, we present the implementation of HHLPy and evaluate it on Simulink/Stateflow models and on KeYmaera X benchmarks.

7.1 Implementation

The backend of HHLPy is implemented in Python, and the graphical user interface is implemented using JavaScript. Fig. 3 shows the architecture of the tool. The user inputs HCSP programs and annotations in the editor (the HCSP programs can also come from translation of Simulink/Stateflow models). The core HHLPy engine then parses the input and generates VCs. The user interface displays the VCs and allows users to choose a solver for each VC. The solver will be invoked, with the results displayed to the user interface. A screenshot of the user interface is shown in Fig. 4.

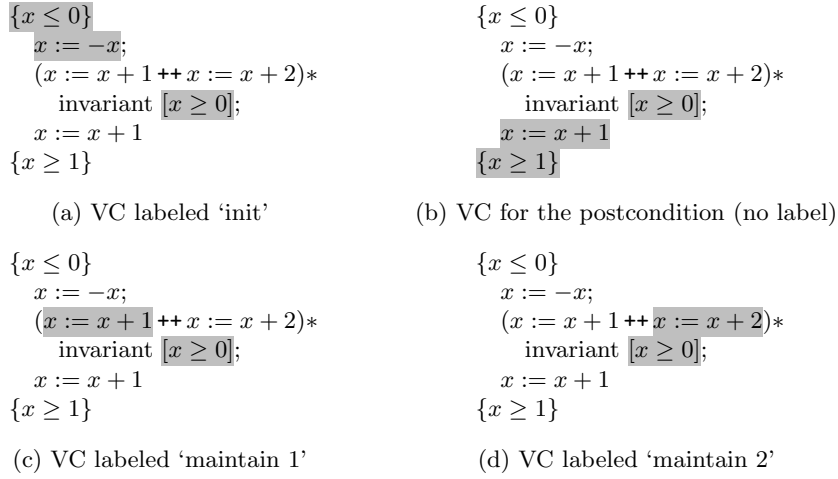


Fig. 1: Highlighting for the four VCs in Example 1

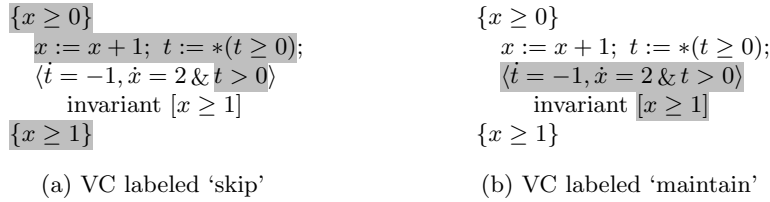


Fig. 2: Highlighting for the two of the VCs in Example 2

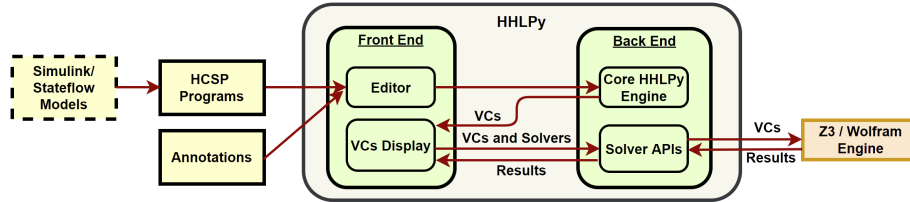


Fig. 3: Architecture of HHLPy

7.2 Evaluation on Simulink/Stateflow Models

To illustrate the use of our tool as part of an existing toolchain to verify correctness of Simulink/Stateflow models, we show two example models, one from Simulink and one from Stateflow.

Cruise Control System The first example is a cruise control system of an automotive vehicle [13]. The system stabilizes the speed of a vehicle around some desired speed (15 m/s in our case) using a PI controller. The PI controller

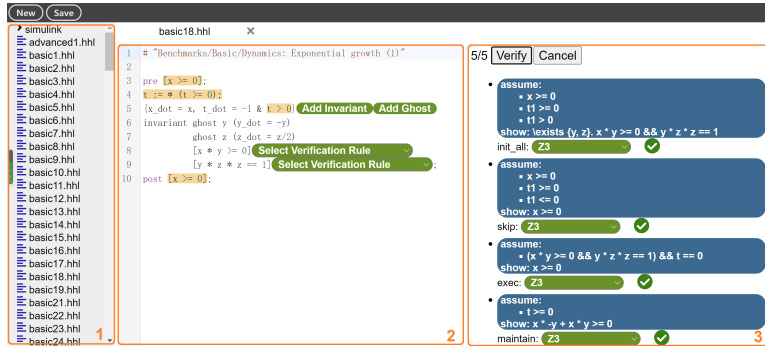


Fig. 4: Screenshot of user interface. The left panel (1) shows a list of example files. The middle panel (2) is the editor area, where user can edit the program and add annotations either directly as text or by clicking on buttons. The right panel (3) shows the VCs. When hovering over each VC, the relevant part of the code is highlighted.

adjusts the control force according to the difference between actual speed and desired speed as well as its integral. The vehicle follows its physical dynamics. The Simulink models are presented in Appendix C.

We first applied the approach by Xu et al. [30] to translate the Simulink models into an HCSP program. In the program, the controller and vehicle dynamics are combined into a single ODE. Given the initial speed $v = 14$ and initial integral value $I = 700$ of the controller, which are close to the stable point ($v = 15, I = 750$), we want to verify that the speed remains in the interval $[13.5, 16.5]$.

To verify the Hoare triple, we annotated the ODE and loop with the invariant $1.3 * (I - 750)^2 - 198 * (I - 750) * (v - 15) + 12192 * (v - 15)^2 \leq 5542$, and used (dI) rule to prove the ODE invariant. The tool generated seven VCs, and Z3 can prove all of them.

```

1  pre [v == 14][I == 700];
2
3  t := 0;
4  _tick := 0;
5  tt := 0;
6
7  {
8    {tt_dot = 1, l_dot = (15 - v) * 40, v_dot = ((15 - v) * 600 + I - v * 50) * 0.001 & tt < 1}
9    invariant [1.3*(I-750)^2 - 198 * (I-750)*(v-15)+12192*(v-15)^2<=5542];
10   t := t + tt;
11   _tick := _tick + 1;
12   tt := 0;
13 }*
14 invariant [1.3*(I-750)^2 - 198 * (I-750)*(v-15)+12192*(v-15)^2<=5542];
15 post [v >= 13.5][v <= 16.5];

```

Sawtooth Wave The sawtooth wave is a Stateflow model generating a signal that alternates between increasing from 0 to 1 and decreasing from 1 to 0. It

illustrates functionality in Stateflow such as hierarchical states and specifying ODEs in a state. The Stateflow model is presented in Appendix C. The signal x follows the ODE $\dot{x} = y$, with y switching between 1 and -1 per unit time. We want to verify that every time y switches, x is still between 0 and 1.

We translated the Stateflow model into an HCSP program with the approach by Guo et al. [10] (code shown in Appendix C). To verify the program, the loop is annotated with four invariants (mostly having to do with relationship between Stateflow locations and value of variable x), and the ODE is annotated with “solution”. A total of 62 VCs are generated and proved to be true by Z3.

7.3 Evaluation on Benchmarks from KeYmaera X

We also evaluated our tool on the basic and nonlinear benchmarks⁴ from KeYmaera X. We first translated the examples from $d\mathcal{L}$ to HCSP manually, trying to maintain semantic equivalence as much as possible. Due to the differences between $d\mathcal{L}$ and HCSP, some examples can not be translated into HCSP programs. We annotated the programs with invariants and proof rules, mostly following the existing proofs in KeYmaera X.

Given the annotations, HHLPy can verify 50 out of 60 examples in the basic benchmarks. In comparison, KeYmaera X solves 58 examples in the scripted mode (with detailed proof scripts), and 55 examples in the hints mode (with invariants annotated in the model) [16]. Of the ten unsolved examples, we are unable to translate eight of them to HCSP due to use of $d\mathcal{L}$ -specific constructs; one is non-polynomial; and the last one makes use of invariants containing old versions of variables. For the nonlinear benchmarks, HHLPy can verify 100 out of 141 examples (compared to 108 in the scripted mode and 95 in the hints mode for KeYmaera X). Most of the unsolved ones are because we are unable to find the invariants or their VCs cannot be proved in reasonable time by Z3 or Wolfram Engine.

In the 150 examples solvable by HHLPy, the user only needs to add annotations including loop/ODE invariants and ODE rules; just a couple of annotations are needed per problem. For some problems, it is necessary to switch the backend solver from the default Z3 to Wolfram Engine. After this, HHLPy can finish the proof automatically. These experiments show that our tool can be used to solve a wide range of examples from existing benchmarks with little manual effort.

8 Related Work

Differential dynamic logic ($d\mathcal{L}$) [21, 23] models hybrid systems by extending dynamic logic with continuous evolution. Reasoning rules about continuous evolution include differential invariants, differential weakening, differential cut, and differential ghosts. The rules are stated in the form of a uniform substitution

⁴ The benchmarks are available at <https://github.com/LS-Lab/KeYmaeraX-projects/tree/master/benchmarks>

calculus [22], and they are complete [26]. Differential dynamic logic has been implemented in KeYmaera [24] and KeYmaera X [9], whose user interfaces display current subgoals in sequent calculus form and allow users to point and click to construct proofs. The Bellerophon language allows users to perform proofs using a tactic language [8]. The KeYmaera X tool produces proofs that can be independently checked in Isabelle and Coq [1]. Liebreuz et al. developed a method to translate Simulink models to $d\mathcal{L}$ and to verify them in KeYmaera X [15].

Huerta y Munive and Struth represented $d\mathcal{L}$ programs using Kleene algebras, and built verification components for hybrid systems in Isabelle/HOL [19, 20]. Foster et al. proposed Hoare logic rules and refinement calculi for hybrid programs [7] and extended the verification components in Isabelle/HOL [6], e.g., with syntax translation to obtain more user-friendly modeling and specification languages and with proof automation using Eisbach. Huerta y Munive and Struth also described formalization of solutions to affine and linear systems of ODEs, with applications to verifying correctness of such systems [18].

Both of the above series of works focused on hybrid programs modeled using $d\mathcal{L}$. As discussed in Section 2 and 3, the semantics of continuous evolution is different from that in $d\mathcal{L}$, hence the proof rules need to be adapted, resulting in particular to significant changes to differential weakening and differential cut rules. Compared to the previous version of hybrid Hoare logic [28], we focus only on the sequential fragment of HCSP, resulting in much simpler rules that permits automatic VC generation. On the other hand, we consider a full set of reasoning rules for ODEs, rather than only the invariant rule in [28].

The design of our tool is similar to many other (semi-)automatic program verification tools, such as Dafny [14], VeriFast [11], and Why3 [5], in that annotations are inserted into the program code. Our work differs from these tools firstly in being able to handle hybrid programs. Moreover, we designed detailed labeling and highlighting mechanisms to improve robustness of the annotations and help visualization. These improvements are not limited to hybrid programs, and can potentially be incorporated into other program verification tools as well.

9 Conclusion

We presented HHLPy, a tool for verification of hybrid programs written in the sequential fragment of HCSP. The backend of the tool implements a Hoare logic that includes rules for reasoning about continuous evolution adapted from $d\mathcal{L}$. We also designed labeling and highlighting mechanisms to improve user interaction. We demonstrated the capabilities of the tool on HCSP programs translated from Simulink/Stateflow models and on KeYmaera X benchmarks.

In this paper, the deduction system and its proof of soundness is presented only informally. We leave extension of the deduction system to handle communication, interrupts, and parallel composition, together with formal treatments of logical issues such as completeness, to future work. On the side of implementation and applications, we intend to further extend the tool to be able to handle non-polynomial ODEs and invariants and permit interactive proofs of VCs.

References

1. Bohrer, R., Rahli, V., Vukotic, I., Völpl, M., Platzer, A.: Formally verified differential dynamic logic. In: Bertot, Y., Vafeiadis, V. (eds.) *Conference on Certified Programs and Proofs (CPP 2017)*. pp. 208–221. ACM (2017)
2. Chen, M., Han, X., Tang, T., Wang, S., Yang, M., Zhan, N., Zhao, H., Zou, L.: MARS: A toolchain for modelling, analysis and verification of hybrid systems. In: Hinchey, M.G., Bowen, J.P., Olderog, E. (eds.) *Provably Correct Systems*, pp. 39–58. *NASA Monographs in Systems and Software Engineering*, Springer (2017)
3. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* **18**(8), 453–457 (1975)
4. Dijkstra, E.W.: *A Discipline of Programming*. Prentice-Hall (1976)
5. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) *European Symposium on Programming, (ESOP 2013)*. LNCS, vol. 7792, pp. 125–128. Springer (2013)
6. Foster, S., Huerta y Munive, J.J., Gleirscher, M., Struth, G.: Hybrid systems verification with Isabelle/HOL: Simpler syntax, better models, faster proofs. In: Huisman, M., Pasareanu, C.S., Zhan, N. (eds.) *Formal Methods (FM 2021)*. LNCS, vol. 13047, pp. 367–386. Springer (2021)
7. Foster, S., Huerta y Munive, J.J., Struth, G.: Differential Hoare logics and refinement calculi for hybrid systems with Isabelle/HOL. In: Fahrenberg, U., Jipsen, P., Winter, M. (eds.) *Relational and Algebraic Methods in Computer Science (RAMiCS 2020)*. LNCS, vol. 12062, pp. 169–186. Springer (2020)
8. Fulton, N., Mitsch, S., Bohrer, R., Platzer, A.: Bellerophon: Tactical theorem proving for hybrid systems. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) *Interactive Theorem Proving (ITP 2017)*. LNCS, vol. 10499, pp. 207–224. Springer (2017)
9. Fulton, N., Mitsch, S., Quesel, J., Völpl, M., Platzer, A.: KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. In: Felty, A.P., Middeldorp, A. (eds.) *Conference on Automated Deduction (CADE-25)*. LNCS, vol. 9195, pp. 527–538. Springer (2015)
10. Guo, P., Zhan, B., Xu, X., Wang, S., Sun, W.: Translating a large subset of Stateflow to hybrid CSP with code optimization. *J. Syst. Archit.* **130**, 102665 (2022)
11. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) *NASA Formal Methods (NFM 2011)*. LNCS, vol. 6617, pp. 41–55. Springer (2011)
12. Jifeng, H.: *From CSP to Hybrid Systems*, p. 171–189. Prentice Hall International (UK) Ltd., GBR (1994)
13. Kekatos, N.: Verifying a cruise control system using Simulink and SpaceEx. *CoRR abs/2101.00102* (2021)
14. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16)*. LNCS, vol. 6355, pp. 348–370. Springer (2010)
15. Liebrez, T., Herber, P., Glesner, S.: Deductive verification of hybrid control systems modeled in Simulink with KeYmaera X. In: Sun, J., Sun, M. (eds.) *International Conference on Formal Engineering Methods (ICFEM 2018)*. LNCS, vol. 11232, pp. 89–105. Springer (2018)
16. Mitsch, S., Jin, X., Zhan, B., Wang, S., Zhan, N.: ARCH-COMP21 category report: Hybrid systems theorem proving. In: Frehse, G., Althoff, M. (eds.) *International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH21)*. EPiC Series in Computing, vol. 80, pp. 120–132. EasyChair (2021)

17. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*. LNCS, vol. 4963, pp. 337–340. Springer (2008)
18. Huerta y Munive, J.J.: Affine systems of ODEs in Isabelle/HOL for hybrid-program verification. In: de Boer, F.S., Cerone, A. (eds.) *Software Engineering and Formal Methods (SEFM 2020)*. LNCS, vol. 12310, pp. 77–92. Springer (2020)
19. Huerta y Munive, J.J., Struth, G.: Verifying hybrid systems with modal Kleene algebra. In: Desharnais, J., Guttman, W., Joosten, S. (eds.) *Relational and Algebraic Methods in Computer Science (RAMiCS 2018)*. LNCS, vol. 11194, pp. 225–243. Springer (2018)
20. Huerta y Munive, J.J., Struth, G.: Predicate transformer semantics for hybrid systems. *J. Autom. Reason.* **66**(1), 93–139 (2022)
21. Platzer, A.: Differential dynamic logic for hybrid systems. *J. Autom. Reason.* **41**(2), 143–189 (2008)
22. Platzer, A.: A complete uniform substitution calculus for differential dynamic logic. *J. Autom. Reason.* **59**(2), 219–265 (2017)
23. Platzer, A.: *Logical Foundations of Cyber-Physical Systems*. Springer (2018)
24. Platzer, A., Quesel, J.: KeYmaera: A hybrid theorem prover for hybrid systems (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) *International Joint Conference on Automated Reasoning (IJCAR 2008)*. LNCS, vol. 5195, pp. 171–178. Springer (2008)
25. Platzer, A., Tan, Y.K.: Differential equation axiomatization: The impressive power of differential ghosts. In: Dawar, A., Grädel, E. (eds.) *Logic in Computer Science (LICS 2018)*. pp. 819–828. ACM (2018)
26. Platzer, A., Tan, Y.K.: Differential equation invariance axiomatization. *J. ACM* **67**(1), 6:1–6:66 (2020)
27. Prajna, S., Jadbabaie, A., Pappas, G.J.: A framework for worst-case and stochastic safety verification using barrier certificates. *IEEE Trans. Autom. Control.* **52**(8), 1415–1428 (2007)
28. Wang, S., Zhan, N., Zou, L.: An improved HHL prover: An interactive theorem prover for hybrid systems. In: Butler, M.J., Conchon, S., Zaïdi, F. (eds.) *Formal Methods and Software Engineering (ICFEM 2015)*. LNCS, vol. 9407, pp. 382–399. Springer (2015)
29. Wolfram Research, Inc.: Wolfram Engine, Version 13.1, <https://www.wolfram.com/engine>, Champaign, IL, 2022
30. Xu, X., Zhan, B., Wang, S., Talpin, J.P., Zhan, N.: A denotational semantics of Simulink with higher-order UTP. *J. Log. Algebraic Methods Program.* p. 100809
31. Zhan, N., Wang, S., Zhao, H.: *Formal Verification of Simulink/Stateflow Diagrams*. Springer (2017)
32. Zhao, H., Zeng, X., Chen, T., Liu, Z., Woodcock, J.: Learning safe neural network controllers with barrier certificates. *Formal Aspects Comput.* **33**(3), 437–455 (2021)
33. Zhou, C., Wang, J., Ravn, A.P.: A formal description of hybrid systems. In: *Hybrid Systems III: Verification and Control*, Proceedings of the DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems. pp. 511–530 (1995)
34. Zou, L., Zhan, N., Wang, S., Fränzle, M.: Formal verification of Simulink/Stateflow diagrams. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) *Automated Technology for Verification and Analysis (ATVA 2015)*. LNCS, vol. 9364, pp. 464–481. Springer (2015)
35. Zou, L., Zhan, N., Wang, S., Fränzle, M., Qin, S.: Verifying Simulink diagrams via a Hybrid Hoare Logic prover. In: Ernst, R., Sokolsky, O. (eds.) *International Conference on Embedded Software (EMSOFT 2013)*. pp. 9:1–9:10. IEEE (2013)

A More on the Proof Rules

In this section, we provide more details on the proof rules.

A.1 Proof Rules for Other Commands

The proof rules for commands other than the ODE are given in Fig. 5.

$$\begin{array}{c}
\frac{}{\{P\}\text{skip}\{P\}} \text{skip} \quad \frac{}{\{P[e/x]\}x := e\{P\}} \text{assign} \quad \frac{\{P_1\}c_1\{Q\} \quad \{P_2\}c_2\{Q\}}{\{P_1 \wedge P_2\}c_1 ++ c_2\{Q\}} \text{ichoice} \\
\frac{}{\{B[y/x] \rightarrow Q[y/x]\}x := *(B)\{Q\}} \text{nondet-assign} \quad \frac{\{P\}c_1\{Q\} \quad \{Q\}c_2\{R\}}{\{P\}c_1; c_2\{R\}} \text{seq} \\
\frac{\{P_1\}c_1\{Q\} \quad \{P_2\}c_2\{Q\}}{\{(B \rightarrow P_1) \wedge (\neg B \rightarrow P_2)\}\text{if } B \text{ then } c_1 \text{ else } c_2\{Q\}} \text{ite} \\
\frac{\{I\}c\{I\}}{\{I\}c * \{I\}} \text{loop} \quad \frac{P' \rightarrow P \quad \{P\}c\{Q\} \quad Q \rightarrow Q'}{\{P'\}c\{Q'\}} \text{conseq}
\end{array}$$

Fig. 5: Hoare rules for the non-ODE commands

A.2 Example and Proof for the (dW) Rule

We give some examples of applying the (dW) rule with different invariants, such as the trivial invariants `true` and `false` or the non-trivial invariant $x - y = 1$.

Example 4. In this example, the precondition guarantees that the ODE starts inside the domain D . The Hoare triple is:

$$\{x < 5\}\langle \dot{x} = 1 \ \& \ x < 5 \rangle\{x = 5\}$$

No invariant is needed (set $I = \text{true}$ by default). From the postcondition $x = 5$, we obtain precondition $\neg x < 5 \rightarrow x = 5$ and VC $x = 5 \rightarrow x = 5$. So the VCs are:

$$\begin{array}{l}
x < 5 \rightarrow \neg x < 5 \rightarrow x = 5 \\
x = 5 \rightarrow x = 5
\end{array}$$

Example 5. In this example, the precondition guarantees that the ODE starts outside the domain D . The Hoare triple is:

$$\{x > 10\}\langle \dot{x} = 1 \ \& \ x < 5 \rangle\{x > 8\}$$

Here we set invariant to be `false`. From the postcondition $x > 8$, we obtain precondition $\neg x < 5 \wedge x > 8$ and no VCs from the ODE. The overall VCs are:

$$x > 10 \rightarrow \neg x < 5 \wedge x > 8$$

Example 6. Now consider a case where the starting position may or may not lie within the domain D . The Hoare triple is:

$$\{x < 6\} \langle \dot{x} = 1 \ \& \ x < 5 \rangle \{x \geq 5 \wedge x < 6\}$$

No invariant is needed ($I = \text{true}$ is set by default). From the postcondition, we obtain the precondition $\neg x < 5 \rightarrow x \geq 5 \wedge x < 6$, and the VC $x = 5 \rightarrow x \geq 5 \wedge x < 6$. So the overall VCs are:

$$\begin{aligned} x < 6 \rightarrow \neg x < 5 \rightarrow x \geq 5 \wedge x < 6 \\ x = 5 \rightarrow x \geq 5 \wedge x < 6 \end{aligned}$$

Example 7. Now we consider an example where a non-trivial invariant must be introduced. The Hoare triple is:

$$\{x = 0 \wedge y = 0\} \langle \dot{x} = 1, \dot{y} = 1 \ \& \ x < 5 \rangle \{y = 5\}$$

We set the invariant to be $x - y = 0$, requiring to show

$$\llbracket x \leq 5 \rrbracket \langle \dot{x} = 1, \dot{y} = 1 \rangle \llbracket x - y = 0 \rrbracket$$

We get the VC $x = 5 \wedge x - y = 0 \rightarrow y = 5$ and the precondition $(x < 5 \rightarrow x - y = 0) \wedge (\neg x < 5 \rightarrow y = 5)$.

Proof (of the (dW) rule). Given starting state \mathbf{x} , we divide into two cases based on whether \mathbf{x} satisfies domain D .

1. If \mathbf{x} satisfies D , then there exists a solution $\gamma : [0, T] \rightarrow \mathbb{R}^n$, such that $\gamma(t)$ satisfies D for $t \in [0, T)$ and $\gamma(T)$ satisfies $\neg D$, and we wish to show that $\gamma(T)$ satisfies Q . By the continuity of γ , we get that $\gamma(t)$ satisfies \bar{D} for $t \in [0, T]$. Moreover, since $D \rightarrow I$ holds in the precondition, we get that $\gamma(0)$ satisfies I as well. Then from $\llbracket \bar{D} \rrbracket \langle \dot{x} = e \rangle \llbracket I \rrbracket$, we get that $\gamma(t)$ satisfies I for $t \in [0, T]$. From $\partial D \wedge I \rightarrow Q$ and the fact that $\gamma(T)$ satisfies I and ∂D , we get that $\gamma(T)$ satisfies Q , as desired.
2. If \mathbf{x} does not satisfy D , then the ODE is not executed, and we wish to show that \mathbf{x} satisfies Q . Since $\neg D \rightarrow Q$ holds in the precondition, we get that \mathbf{x} satisfies Q , as desired.

A.3 Example and Proof for the (dI) Rule

We give an example showing the use of the (dI) rule.

Example 8 (Example 7 continued). In this example, we continue with the invariant triple we get:

$$\llbracket x \leq 5 \rrbracket \langle \dot{x} = 1, \dot{y} = 1 \rangle \llbracket x - y = 0 \rrbracket$$

The Lie derivative of $x - y$ is $\dot{x} - \dot{y} = 1 - 1 = 0$. So the premise of the (dI) rule is $x \leq 5 \rightarrow 0 = 0$.

Proof (of the (dI) rule). We consider each of the rules in turn.

Rule dI₌ Consider a solution $\gamma : [0, T] \rightarrow \mathbb{R}^n$ of the ODE, satisfying the condition that $\gamma(t)$ satisfies P for all $t \in [0, T]$. Then, the derivative of $f(\gamma(t))$ is 0 for all $t \in [0, T]$. This shows that $f(\gamma(t))$ is constant within this interval. If furthermore $f(\gamma(0)) = 0$, then we get $f(\gamma(T)) = 0$, as desired.

Rule dI_⩾ Similar to before, except now the derivative of $f(\gamma(t))$ is greater than or equal to 0 for all $t \in [0, T]$, so $f(\gamma(t))$ is a non-decreasing function of t within this interval. If furthermore $f(\gamma(0)) \geq 0$, then we get $f(\gamma(T)) \geq 0$, as desired.

Rule dI_≠ Similar to before, we get that the derivative of $f(\gamma(t))$ is 0 for all $t \in [0, T]$. This shows that $f(\gamma(t))$ is constant within this interval. If furthermore $f(\gamma(0)) \neq 0$, then we get $f(\gamma(T)) \neq 0$, as desired.

A.4 Example and Proof for the (dC) Rule

Example 9. In this example, we insert an intermediate invariant to help prove the final invariant. The invariant triple is:

$$\llbracket y \leq 1 \rrbracket \langle \dot{x} = y, \dot{y} = 1 \rangle \llbracket y > 0 \wedge x > 0 \rrbracket$$

Using the (dC) rule, the two premises are:

$$\llbracket y \leq 1 \rrbracket \langle \dot{x} = y, \dot{y} = 1 \rangle \llbracket y > 0 \rrbracket$$

$$\llbracket y \leq 1 \wedge y > 0 \rrbracket \langle \dot{x} = y, \dot{y} = 1 \rangle \llbracket x > 0 \rrbracket$$

Both the two premises can be proved by using the (dI) rule.

Proof (of the (dC) rule). Consider a solution $\gamma : [0, T] \rightarrow \mathbb{R}^n$ of the ODE, satisfying the condition that $\gamma(t)$ satisfies P for all $t \in [0, T]$, and $\gamma(0)$ satisfies $Q_1 \wedge Q_2$. We wish to show $\gamma(t)$ satisfies $Q_1 \wedge Q_2$ for all $t \in [0, T]$. By the first premise, we get $\gamma(t)$ satisfies Q_1 for all $t \in [0, T]$. Then by the second premise, $\gamma(t)$ satisfies $Q_1 \wedge Q_2$ for all $t \in [0, T]$.

A.5 Example and Proof for the (dG) Rule

Example 10. In this example, the Hoare triple is:

$$\{x > 0 \wedge t = 0\} \langle \dot{x} = x, \dot{t} = 1 \ \& \ t < 10 \rangle \{x > 0\}$$

Using the (dG) rule, we introduce a ghost variable y with the differential equation $\dot{y} = -\frac{y}{2}$ and set the invariant for the new differential equations to be $xy^2 = 1$. Then the new goal is:

$$\llbracket t < 10 \rrbracket \langle \dot{x} = x, \dot{y} = -\frac{y}{2}, \dot{t} = 1 \rangle \llbracket xy^2 = 1 \rrbracket$$

We get the VC $t = 10 \wedge xy^2 = 1 \rightarrow x > 0$ and the precondition $(t < 10 \rightarrow \exists y. xy^2 = 1) \wedge (\neg t < 10 \rightarrow x > 0)$.

Proof (of the (dG) rule). Given starting state \mathbf{x}_0 , we distinguish two cases based on whether \mathbf{x}_0 satisfies D .

1. If \mathbf{x}_0 satisfies D , then there exists a solution $\gamma : [0, T] \rightarrow \mathbb{R}^n$ of the ODE $\langle \dot{\mathbf{x}} = \mathbf{e} \rangle$ with initial condition $\mathbf{x}(0) = \mathbf{x}_0$, such that $\gamma(t)$ satisfies D for $t \in [0, T)$ and $\gamma(T)$ satisfies $\neg D$. We wish to show that $\gamma(T)$ satisfies Q . Since $D \rightarrow \exists \mathbf{y}. I$ holds in the precondition and \mathbf{x}_0 satisfies D , using \mathbf{r} to denote a witness of \mathbf{y} , we get that $(\mathbf{x}_0, \mathbf{r})$ satisfies I . With $\mathbf{f}(\mathbf{x}, \mathbf{y})$ satisfying Lipschitz condition, let $\psi : [0, T] \rightarrow \mathbb{R}$ denote the solution of ODE $\langle \dot{\mathbf{y}} = \mathbf{f}(\mathbf{x}, \mathbf{y}) \rangle$ with initial value \mathbf{r} . Given that \mathbf{y} are fresh variables and do not limit the duration of the solutions of \mathbf{x} , we get that (γ, ψ) is the solution of the ODE $\langle \dot{\mathbf{x}} = \mathbf{e}, \dot{\mathbf{y}} = \mathbf{f}(\mathbf{x}, \mathbf{y}) \rangle$ with initial condition $(\mathbf{x}_0, \mathbf{r})$. Then from $\llbracket \overline{D} \rrbracket \langle \dot{\mathbf{x}} = \mathbf{e}, \dot{\mathbf{y}} = \mathbf{f}(\mathbf{x}, \mathbf{y}) \rrbracket \llbracket I \rrbracket$, we get that $(\gamma(t), \psi(t))$ satisfies I for $t \in [0, T]$. From $\partial D \wedge I \rightarrow Q$ and $(\gamma(T), \psi(T))$ satisfies I and ∂D , we get that $(\gamma(T), \psi(T))$ satisfies Q . Since \mathbf{y} does not occur in Q , we get that $\gamma(T)$ satisfies Q , as desired.
2. If \mathbf{x}_0 does not satisfy D , the proof is similar to the one for the (dW) rule.

A.6 Example and Proof for the Barrier Certificate Rule

Example 11. In this example, we use the (bc) rule to prove the following invariant triple:

$$\llbracket t \leq 10 \rrbracket \langle \dot{x} = x^3 + x^4 \rangle \llbracket x^3 > 5 \rrbracket$$

Then the premise of the (bc) rule is

$$t \leq 10 \wedge x^3 = 5 \rightarrow 3x^2(x^3 + x^4) > 0$$

Proof (of the (bc) rule). The proof is by contradiction. Consider a solution $\gamma : [0, T] \rightarrow \mathbb{R}^n$ of the ODE, satisfying the condition that $\gamma(t)$ satisfies P for all $t \in [0, T]$, and $\gamma(0)$ satisfies $f \geq 0$. Assume there exists $t' \in [0, T]$ such that $f(\gamma(t')) < 0$. According to the proof of strict barrier certificate [27] in [32], we get that there exists $t_{sup} \in [0, t')$ satisfying $f(\gamma(t_{sup})) = 0$ and the Lie derivative of f at t_{sup} is non-positive. However, according to the premise, $\dot{f} > 0$ when $f = 0$, which leads to a contradiction. Therefore, there cannot exist $t' \in [0, T]$ such that $f(\gamma(t')) < 0$, i.e. for all $t \in [0, T]$, $f(\gamma(t)) \geq 0$.

The proof is similar when considering the case where the invariant is $f > 0$.

A.7 Example and Proof for the Darboux Rule

Example 12. In this example, we use the (dbx₌) rule. The invariant triple is:

$$\llbracket t \leq 1 \rrbracket \langle \dot{x} = 5x^2 + 3x, \dot{z} = 5zx + 3z, \dot{t} = 1 \rangle \llbracket x + z = 0 \rrbracket$$

Then, using the (dbx₌) rule with $g = 5x + 3$, we obtain the following premise of the (dbx₌) rule:

$$t \leq 1 \rightarrow 5x^2 + 3x + 5zx + 3z = (5x + 3)(x + z)$$

Example 13. In this example, we use (dbx_≠) rule. The invariant triple is:

$$\llbracket t \leq 1 \rrbracket \langle \dot{x} = -x + 1, \dot{t} = 1 \rangle \llbracket x > 0 \rrbracket$$

Then, by using the (dbx_≠) rule with which $g = -1$, we obtain the following premise of the (dbx_≠) rule:

$$t \leq 1 \rightarrow -x + 1 \geq -x$$

Proof (of Darboux rule). We consider each rule in turn.

Rule (dbx₌) Consider a solution $\gamma : [0, T] \rightarrow \mathbb{R}^n$ to the ODE, satisfying the condition that $\gamma(t)$ satisfies P for all $t \in [0, T]$, and $\gamma(0)$ satisfies $f = 0$. We wish to show that $f(\gamma(t)) = 0$ for all $t \in [0, T]$. According to [25], given $\dot{f} = gf$, $f = 0$ stays invariant along the (analytic) solution to the ODE. Therefore, $\gamma(t)$ satisfies $f = 0$ for all $t \in [0, T]$.

Rule (dbx_≠) This rule can be derived by (dI), (dC), (dG) rule according to [25].

A.8 Example and Proof for the Solution Rule

Example 14. We illustrate rule (sln) with the following example:

$$\{x = 0 \wedge y = 0\} \langle \dot{x} = 1, \dot{y} = x \ \& \ x < 2 \rangle \{y = 2\}$$

The solution $\mathbf{u}(\tau, x_0, y_0)$ is given by $(\tau + x_0, \frac{1}{2}\tau^2 + x_0\tau + y_0)$. Hence $D(\mathbf{u}(\tau, x_0, y_0))$ is $\tau + x_0 < 2$, and $P'(x)$ is

$$\forall t > 0. (\forall 0 \leq \tau < t. \tau + x < 2) \wedge \neg(t + x < 2) \rightarrow \frac{1}{2}t^2 + xt + y = 2$$

The two premises $\forall 0 \leq \tau < t. \tau + x < 2$ and $\neg(t + x < 2)$ of the implication are together equivalent to $t + x = 2$. Hence, under the assumption of $x < 2$ (so that $t > 0 \wedge t + x = 2$ has a unique solution), the predicate P' reduces to

$$\frac{1}{2}(2 - x)^2 + x(2 - x) + y = 2$$

So the precondition we derive from the solution rule is $(x < 2 \rightarrow \frac{1}{2}(2 - x)^2 + x(2 - x) + y = 2) \wedge (\neg x < 2 \rightarrow y = 2)$. Since $x = 0 \wedge y = 0$ implies this condition, we prove the Hoare triple.

Proof (of solution rule). Given starting state \mathbf{x}_0 , we distinguish two cases based on whether \mathbf{x}_0 satisfies domain D .

1. If \mathbf{x}_0 satisfies D , then there exists a solution $\gamma : [0, T] \rightarrow \mathbb{R}^n$ of the ODE $\dot{\mathbf{x}} = e$ with initial state \mathbf{x}_0 , such that $\gamma(\tau)$ satisfies D for $\tau \in [0, T)$ and $\gamma(T)$ satisfies $\neg D$, and we wish to show that $\gamma(T)$ satisfies Q . Since $D \rightarrow P'$ holds in the precondition, we get that \mathbf{x}_0 satisfies P' :

$$\forall t > 0. (\forall 0 \leq \tau < t. D(\mathbf{u}(\tau, \mathbf{x}_0))) \wedge \neg D(\mathbf{u}(t, \mathbf{x}_0)) \rightarrow Q(\mathbf{u}(t, \mathbf{x}_0)).$$

Then let $t = T$. We get:

$$\forall 0 \leq \tau < T. D(\mathbf{u}(\tau, \mathbf{x}_0)) \wedge \neg D(\mathbf{u}(T, \mathbf{x}_0)) \rightarrow Q(\mathbf{u}(T, \mathbf{x}_0)).$$

Next, since \mathbf{u} is the unique solution to the differential equation $\dot{x} = e$, we get that $\gamma(\tau) = \mathbf{u}(\tau, \mathbf{x}_0)$ for all $0 \leq \tau \leq T$. Then we get:

$$\forall 0 \leq \tau < T. D(\gamma(\tau)) \wedge \neg D(\gamma(T)) \rightarrow Q(\gamma(T)).$$

From the fact that $\gamma(\tau)$ satisfies D for $\tau \in [0, T)$ and $\gamma(T)$ satisfies $\neg D$, we get $\gamma(T)$ satisfies Q , as desired.

2. If \mathbf{x} does not satisfy D , then the ODE is not executed, and we wish to show that \mathbf{x} satisfies Q . Since $\neg D \rightarrow Q$ holds in the precondition, \mathbf{x} satisfies Q , as desired.

B Proof of Soundness of the Verification Condition Generation

Theorem 1. *A Hoare triple $\{P_1 \wedge \dots \wedge P_m\} \mathcal{T} \{Q_1 \wedge \dots \wedge Q_n\}$ holds if all conditions in $\text{VC}(\{P_1 \wedge \dots \wedge P_m\} \mathcal{T} \{Q_1 \wedge \dots \wedge Q_n\})$ hold.*

Proof. We proceed by structural induction on \mathcal{T} . We assume an initial state in which P_1, \dots, P_m hold. We must show that after executing \mathcal{T} on this state, if \mathcal{T} terminates, Q_1, \dots, Q_n hold.

The predicates $\tilde{P}_1, \dots, \tilde{P}_m$ in (vc) are the subset of the preconditions P_1, \dots, P_m whose variables are never reassigned in \mathcal{T} . They hold in the initial state by assumption and since their variables never change, they hold in all states in the course of the program \mathcal{T} . Thus, for the purposes of this proof, we can assume that $\tilde{P}_1, \dots, \tilde{P}_m$ always hold. Therefore, we can also assume that the elements of $\text{vc}(\mathcal{T}, \{Q_1, \dots, Q_n\})$ always hold.

If T is any command except loop or ODE, the correctness of the Hoare triple is easy to see using the induction hypotheses and the VCs emerging via (pre) from (pre-skip), (pre-assn), (pre-seq), (pre-if), (pre-else), (pre-choice), and (pre-nassn).

If \mathcal{T} is a loop \mathcal{S}^* with invariants I_1, \dots, I_n , we proceed as follows. Since $\text{pre}(\mathcal{T}, Q_1, \dots, Q_n) = \{I_1, \dots, I_n\}$ by (pre-loop), we have $P_1 \wedge \dots \wedge P_m \rightarrow I_i \in \text{VC}(\{P_1, \dots, P_m\} \mathcal{T} \{Q_1, \dots, Q_n\})$ (pre) for each I_i . Thus, all I_i hold in the initial state. The induction hypothesis for \mathcal{S} , in conjunction with (vc-loop-body) and (vc-loop-maintain), imply $\{I_1, \dots, I_n\} \mathcal{S} \{I_1, \dots, I_n\}$. Hence, the I_i not only hold in the initial state, but still after executing \mathcal{S} arbitrarily often. Finally, by (vc-loop-exit), Q_1, \dots, Q_n hold in the final state.

If \mathcal{T} is a an ODE $\langle \dot{x} = e \ \& \ D \rangle$ with invariants I_1, \dots, I_k , we proceed as follows.

If the ODE is annotated to use the solution rule, we apply it using the VC stemming from the precondition (pre-sln), and we are done.

Otherwise, invariants have been specified, or the default invariant $I_1 = \text{true}$ has been set. If additionally ghost variables are specified, we employ the (dG)

rule; if no ghost variables are specified we employ the (dW) rule. The VCs stemming from (pre-dWG-skip) and (pre-dW-init) or (pre-dG-init) show that the rule (dW) or (dG) is applicable to our desired Hoare triple $\{P_1, \dots, P_m\} \mathcal{T} \{Q_1, \dots, Q_n\}$. The condition (vc-dWG-exec) discharges the right premise of the (dW) or (dG) rule. It remains the left premise

$$\llbracket \overline{D} \rrbracket \langle \dot{\mathbf{x}} = \mathbf{e}, \dot{\mathbf{y}} = \mathbf{f}(\mathbf{x}, \mathbf{y}) \rangle \llbracket I_1 \wedge \dots \wedge I_k \rrbracket$$

(possibly without \mathbf{y} in the case of the (dW) rule)

Next, we repeatedly apply the (dC) rule to isolate each invariant I_i . Then it remains to show

$$\llbracket \overline{D} \wedge I_1 \wedge \dots \wedge I_{i-1} \rrbracket \langle \dot{\mathbf{x}} = \mathbf{e}, \dot{\mathbf{y}} = \mathbf{f}(\mathbf{x}, \mathbf{y}) \rangle \llbracket I_i \rrbracket$$

We distinguish the different rules that could be in the annotation:

- If the user did not specify a rule and I_i is either `true` or `false`, what we need to show is obvious because `true` and `false` are always invariants.
- dI: We apply the (dI) rule. Its premise is justified by (vc-dI1), (vc-dI2), or (vc-dI3).
- dbx: We apply the (dbx) rule. Its premise is justified by (vc-dbx1) or (vc-dbx2).
- bc: We apply the (bc) rule. Its premise is justified by (vc-bc). □

C Simulink/Stateflow Models

C.1 Model for the Cruise Control System Example

Simulink models for the cruise control system are shown in Fig. 6, 7, and 8.

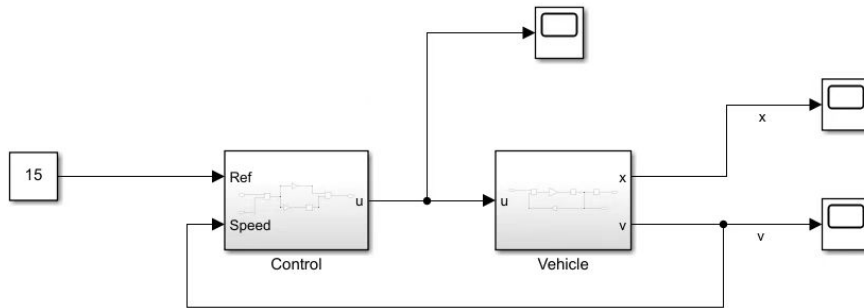


Fig. 6: Simulink Model of Cruise Control System.

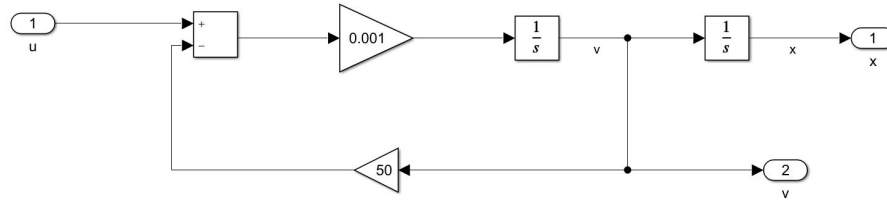


Fig. 7: Simulink Model of Vehicle Dynamics.

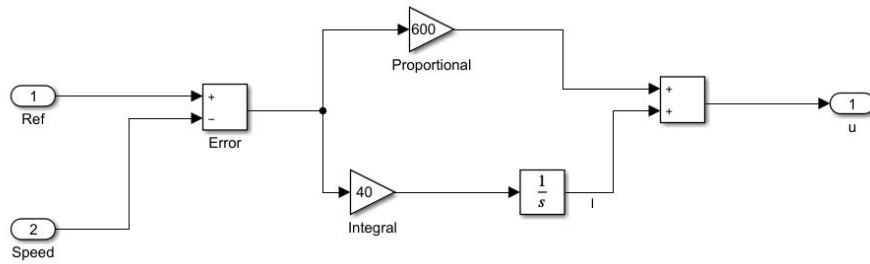


Fig. 8: Simulink Model of PI Controller.

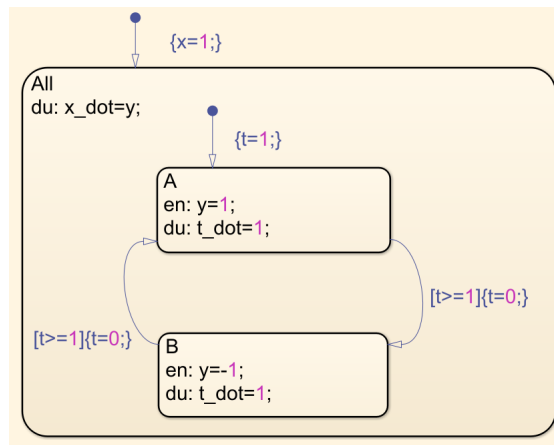


Fig. 9: Simulink/Stateflow Model of Sawtooth Wave.

C.2 Model and Code for the Sawtooth Wave Example

The Stateflow model of the sawtooth wave example is shown in Fig. 9.

Fig. 10 shows the HCSP code that is automatically translated from the Stateflow diagram for the sawtooth example, together with annotations of pre- and postconditions, invariants, and ODE proof rules used to verify the property that x remains in the interval $[0, 1]$ for this model.

```

1  pre [true];
2
3  Chart_All_A := 1;
4  Chart_All_B := 2;
5  Chart_All := 0;
6  x := 1;
7  Chart_st := Chart_All;
8  t := 1;
9  Chart_All_st := Chart_All_A;
10 y := 1;
11
12 {
13   Chart_ret := 0;
14   if (Chart_All_st == Chart_All_A) {
15     Chart_All_A_done := 0;
16     if (t >= 1) {
17       t := 0;
18       if (Chart_All_st == Chart_All_A) {
19         Chart_All_st := Chart_All_B;
20         y := -1;
21         Chart_All_A_done := 1;
22       }
23     }
24     Chart_ret := Chart_All_A_done;
25   } else if (Chart_All_st == Chart_All_B) {
26     Chart_All_B_done := 0;
27     if (t >= 1) {
28       t := 0;
29       if (Chart_All_st == Chart_All_B) {
30         Chart_All_st := Chart_All_A;
31         y := 1;
32         Chart_All_B_done := 1;
33       }
34     }
35     Chart_ret := Chart_All_B_done;
36   }
37   {x_dot = y, t_dot = 1 & t < 1} solution;
38 }*
39 invariant [Chart_All_st == Chart_All_A -> x == 1]
40           [Chart_All_st == Chart_All_B -> x == 0]
41           [Chart_All_st == Chart_All_A || Chart_All_st == Chart_All_B]
42           [t == 1];
43
44 post [x >= 0][x <= 1];

```

Fig. 10: HCSP code translated from the Stateflow diagram for the sawtooth example, with annotations.