Inductive Verification of Hybrid Automata with Strongest Postcondition Calculus

Daisuke Ishii[1], Guillaume Melquiond[2], and Shin Nakajima[1]

[1] National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, Japan,
dsksh@acm.org, nkjm@nii.ac.jp
[2] INRIA Saclay–Île-de-France, LRI, bât 650, Université Paris Sud 11, Orsay, France,
guillaume.melquiond@inria.fr

Abstract. Safety verification of hybrid systems is a key technique in developing embedded systems that have a strong coupling with the physical environment. We propose an automated logical analytic method for verifying a class of hybrid automata. The problems are more general than those solved by the existing model checkers: our method can verify models with symbolic parameters and nonlinear equations as well. First, we encode the execution trace of a hybrid automaton as an imperative program. Its safety property is then translated into proof obligations by strongest postcondition calculus. Finally, these logic formulas are discharged by state-of-the-art arithmetic solvers (e.g., Mathematica). Our proposed algorithm efficiently performs inductive reasoning by unrolling the execution for some steps and generating loop invariants from verification failures. Our experimental results along with examples taken from the literature show that the proposed approach is feasible.

1 Introduction

Hybrid systems, transition systems with continuous dynamics, are a good model for embedded systems that have a strong coupling with the physical environment. Achieving the desired reliability levels of such systems has brought a challenging and important problem in formal methods research.

To date, verification of hybrid systems has been extensively studied with two prominent approaches: *model checking* and *logical analysis*. The model-checking approach has been successfully applied to practical examples with tools such as HyTech [12], PHAVer [8], and HybridSAL [22]. The approach is said algorithmic: tools numerically over-approximate a certain class of *hybrid automata* (HA) to have piecewise-linear systems, and apply model-checking methods [4]. The second approach is based on logical analysis [16]. While the theory of logical analysis has been studied extensively, there are few practical tools. A notable and successful exception is KeYmaera [18]. The logical analytic approach can be applied to the class of *hybrid programs* which generalize the automata handled by model checking. Indeed, this class includes systems with symbolic parameters and nonlinear dynamics. There is, however, a major drawback: the larger the class of systems is, the less automatic its verification becomes. Engineers thus

have to apply some proof strategies during the interactive verification process, which requires understanding the target model.

In this paper, we propose a partly automated tool for the logical analysis of HA that makes heavy use of state-of-the-art arithmetic solvers. Our goal is to prove safety properties. First, our method encodes executions of HA into straight-line imperative programs. This formalism allows us to construct a *lasso-shaped* structure based on induction: after exhibiting at most m steps of continuous evolution and discrete transition, any execution of the system forms a loop with a length of at most n steps between some specific regions of the state space. Then, the imperative program is transformed into a conjunction of verification conditions as a result of *strongest postcondition* (SP) calculus. The resulting logic formula involves real-arithmetic predicates and ordinary differential equations (ODEs). The generated conditions can be discharged using solvers such as Mathematica for some nonlinear HA.

The contribution of this work is as follows. The use of an imperative language and SP calculus gives a straightforward justification of the soundness of our method for generating the finite-length verification conditions from HA. The algorithm we propose realizes an automated verification process, although some user interactions are needed to determine efficiently (a) correct numbers m and n of steps to unroll the execution and (b) the loop invariant that represents the initial region of the loop. Computer algebra techniques, however, are employed to automate most of the work in generating loop invariants.

This paper is organized as follows. Section 2 introduces the class of hybrid automata. Section 3 describes a simple imperative language for simulating HA and the corresponding SP calculus. In Section 4, we present the concept of induction and loop unrolling, and describe an algorithm for automated verification. Section 5 describes an implementation using Mathematica. Section 6 reports how our implementation behaves on several examples and provides a comparison of the results with existing tools. Section 7 describes some related studies.

2 Hybrid Automata

In this paper, we model hybrid systems as hybrid automata (HA) [11].

Definition 1. *A* hybrid automaton *is a tuple* $HA = \langle L, V, Init, \mathcal{G}, \mathcal{R}, \mathcal{F}, \mathcal{I} \rangle$ *that consists of the following components:*

- *A finite set* $L = \{l_1, \ldots, l_p\}$ *of* locations.
- *A finite set* $V = \{x_1, \ldots, x_q\}$ *of real-valued* variables. $\mathbb{R}^V$ *is the set of all of the* valuations *of the system.*
- *An* initial condition *Init in* $L \times \mathbb{R}^V$ *that specifies the* initial states.
- *A family* $\mathcal{G} = \{G_{l,l'}\}_{l \in L, l' \in L}$ *of* guard conditions $G_{l,l'}$ *in* $\mathbb{R}^V$.
- *A family* $\mathcal{R} = \{R_{l,l'}\}_{l \in L, l' \in L}$ *of* reset functions $R_{l,l'} : \mathbb{R}^V \to \mathbb{R}^V$.
- *A family* $\mathcal{F} = \{F_l\}_{l \in L}$ *of* vector fields $F_l : \mathbb{R}^V \to \mathbb{R}^V$.
- *A family* $\mathcal{I} = \{I_l\}_{l \in L}$ *of* location invariants I_l *in* $\mathbb{R}^V$.

$$\frac{t>0 \quad \phi(0)=\nu \quad \forall \tilde{t}\in[0,t]\ \frac{d\phi}{dt}=F_l \wedge I_l[\phi(\tilde{t})]}{\langle l,\nu\rangle \xrightarrow{t} \langle l,\phi(t)\rangle} \qquad \frac{G_{l_1,l_2}[\nu_1] \quad \nu_2=R_{l_1,l_2}(\nu_1) \quad I_{l_2}[\nu_2]}{\langle l_1,\nu_1\rangle \xrightarrow{0} \langle l_2,\nu_2\rangle}$$

Fig. 1. Operational semantics of HA.

A (finite or infinite) execution *of HA is a sequence* $\sigma_0 \xrightarrow{t_1} \sigma_1 \xrightarrow{t_2} \cdots$, *for which* $\sigma_i \in L\times\mathbb{R}^V$ *and* $Init[\sigma_0]$ *holds, and* $\xrightarrow{*}$ *is either a* continuous evolution phase $\xrightarrow{t}$ *where* $t>0$ *or a* discrete transition phase $\xrightarrow{0}$ *and is given by the rules in Figure 1. In the first rule,* $\frac{d\phi}{dt}=F_l$ *is an abbreviation of* $\frac{d\phi(\tilde{t})}{dt}=F_l(\phi(\tilde{t}))$. *We say that an execution is length-k* canonical *when of the form* $\sigma_0 \xrightarrow{t_1} \sigma_1 \xrightarrow{0} \sigma_2 \xrightarrow{t_3} \cdots \xrightarrow{0} \sigma_{2k}$ *that alternates continuous and discrete phases.*

In this paper, we assume that multiple discrete transitions do not occur in an instant. We also assume that no discrete transition occurs initially. Thus, any execution can be expressed as a canonical execution. Verification of non-canonical executions can be considered as future work. Infinite-length canonical executions are supported; yet in presence of Zeno points (infinite number of transitions in finite time), HA executions are handled only up to the first point.

Example 1. Water-level monitor (WLM) [3,15]. A controlled water tank is modeled as a four-location constant-rate HA, as illustrated in Figure 2. It supplies water at a constant rate $rate_{out}$, whereby, in location *off* (and *sw-on*), the water level y decreases as $\frac{d\phi_y}{dt}=rate_{out}$. In location *on* (and *sw-off*), the system pumps water to refill the tank, which results in the water level changing as $\frac{d\phi_y}{dt}=rate_{in}$. A sensor observes y and switches between the locations *on* and *off* when the level reaches the thresholds *low* or *high*. However, it takes *delay* seconds for switching, hence the locations *sw-on* and *sw-off*. In this paper, we constrain the values for the constant parameters as follows:

$$\begin{aligned} &min \le low \ \wedge\ high \le max \ \wedge\ low < high \ \wedge\ delay > 0 \wedge \\ &max \ge high + rate_{in}\cdot delay \ \wedge\ min \le low + rate_{out}\cdot delay. \end{aligned} \tag{1}$$

Because the discrete transition edges in the automaton form a single cycle, the trace of locations that were reached is the same for all of the executions.

Definition 2. *A* safety property *(or an* inductive invariance*) is expressed by a formula* $\Box P$, *where* P *is a predicate on* $L\times\mathbb{R}^V$. $HA \models \Box P$ *denotes that HA satisfies* $\Box P$, *that is, predicate* P *holds initially and is preserved by every discrete transition and continuous evolution.*

Example 2. In the following sections, we will prove that the level stays between a lower and an upper limit, which is expressed by the following safety property:

$$\Box(min \le y \le max).$$

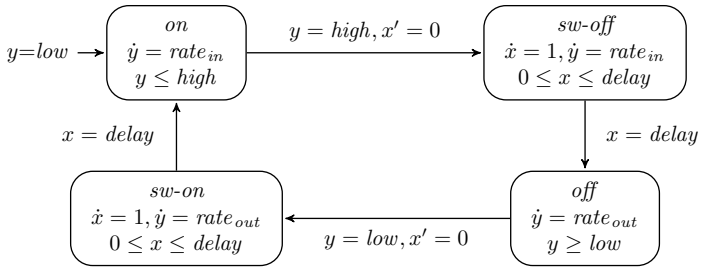


Fig. 2. Water-level monitor.

3 Modeling HA Executions with Programs

In this section, we introduce the theoretical foundation of our method. It analyzes finite and infinite executions of a HA by reusing traditional tools in program verification. We first introduce a simple imperative language in which the statements simply sketch the executions of the HA (Section 3.1). Then, we provide a notion of strongest postcondition for each program statement given a precondition, and we prove that this calculus derives the safety of the HA (Section 3.2).

3.1 Imperative Language

Given a *HA*, we define an untyped imperative language $\mathbf{Imp}_{HA}$. This language is basic, since it does not even provide loops. For the purpose of this work, it has only sequences and the commands `evolve` and `trans`. The command `evolve` expresses a continuous evolution of the *HA* for a given duration, while `trans` expresses a discrete transition.

Definition 3. *The language* $\mathbf{Imp}_{HA}$ *is given by the following syntax:*

$$s ::= \texttt{skip} \mid s;\ \ s \mid \texttt{evolve}\ t \mid \texttt{trans}$$

Definition 4. *A* program state *(denoted S or S_i) is a map from variable names to program values. A special variable x_s is associated to the "current" state ($\in L \times \mathbb{R}^V$) of the HA execution. For the sake of readability, pseudo-variables are introduced to access part of the HA state as follows: $x_s = \langle x_l, \cdot \rangle = \langle \cdot, x_v \rangle = \langle \cdot, (x_1, \ldots, x_i, \ldots, x_q) \rangle$. We assume this equivalence is always maintained automatically when a new value is assigned to a pseudo-variable.*

Figure 3 describes the operational semantics of the language. $[\![e]\!]_S$ denotes the term obtained by replacing each free variable of an expression e by its associated value in the program state S. $S\{x \mapsto v\}$ denotes the program state obtained by adding to S that variable x is associated to the value v. The rules for `skip` and sequence are the usual ones. The rules for `evolve` and `trans` are derived from the operational semantics of a HA execution. Note that we allow the statement `evolve` 0, so that the theorems presented in this paper have a simple way to check the safety property for the initial state or after a discrete transition.

$$\frac{}{S, (\texttt{skip};\ s) \rightsquigarrow S, s} \qquad \frac{S_1, s_1 \rightsquigarrow S_2, s_2}{S_1, (s_1;\ s_3) \rightsquigarrow S_2, (s_2;\ s_3)} \qquad \frac{}{S, \texttt{evolve}\ 0 \rightsquigarrow S, \texttt{skip}}$$

$$\frac{[\![x_s]\!]_S \xrightarrow{t} \sigma}{S, \texttt{evolve}\ t \rightsquigarrow S\{x_s \mapsto \sigma\}, \texttt{skip}} \qquad \frac{[\![x_s]\!]_S \xrightarrow{0} \sigma}{S, \texttt{trans} \rightsquigarrow S\{x_s \mapsto \sigma\}, \texttt{skip}}$$

Fig. 3. Operational semantics of $\mathbf{Imp}_{HA}$.

Lemma 1. *For any execution* $\sigma_0 \xrightarrow{t_1} \sigma_1 \xrightarrow{0} \cdots \xrightarrow{t_k} \sigma_{2k-1}$ *of the HA, assuming that* $\sigma = \sigma_0$ *holds for the initial program state, there is an execution of the following* $\mathbf{Imp}_{HA}$ *program that does not block (that is, it reduces to* `skip`*) and such that the final program state satisfies* $x_s = \sigma_{2k-1}$.

$$\texttt{evolve}\ t_1;\ \texttt{trans};\ \cdots;\ \texttt{evolve}\ t_k$$

Note that this program might also have either blocking executions or executions that end on a different HA state; the former are made irrelevant by our SP-based approach, while the latter are expected due to the non-deterministic nature of HA. For the programs above, the execution is canonical only for the first $k-1$ continuous steps; the last duration t_k can be arbitrarily short. It can also be arbitrarily large, if the HA stays infinitely long in that continuous evolution.

Since we can now express any partial execution of a HA as a program, we can state the safety property of the HA as a property that every non-blocking program must satisfy in its final state.

Lemma 2. *If, for all non-blocking programs of* $\mathbf{Imp}_{HA}$ *of the above form starting from an initial program state* $\sigma \in Init$*, property* P *holds in the final program state, then* P *is a safety property for the HA (up to the first Zeno point, if any).*

3.2 Strongest Postconditions

In this section, we instantiate the principles of program verification [13,7] with $\mathbf{Imp}_{HA}$. We are not interested in manual verification, so we will skip over the definition of Hoare triples and directly go to the topic of verification conditions (VCs). Moreover, since we are not dealing with reachability but only safety, we do not have to prove that programs are non-blocking, we can just assume they are. Therefore, weakest preconditions (WPs) and strongest postconditions (SPs) are dual from each other for our purpose. Should we have to perform backward reachability analysis, WP computation would be better suited. This is not the case though, so we choose SP, so as to follow the direction of time.

Lemma 3 (Soundness of SP). *For any program* s *in* $\mathbf{Imp}_{HA}$*, if the initial state satisfies a given property* P*, the final state satisfies* $SP(P, s)$ *(assuming* s

terminates) with SP inductively defined as follows. [3]

$$SP(P, \texttt{skip}) := P \qquad SP(P, s_1;\ s_2) := SP(SP(P, s_1), s_2)$$

$$SP(P, \texttt{evolve}\ t) := \exists \phi\ P[x_v \leftarrow \phi(0)] \wedge \phi(t) = x_v \wedge (\forall \tilde{t} \in [0, t]\ \tfrac{d\phi}{dt} = F_{x_l} \wedge I_{x_l}[\phi(\tilde{t})])$$

$$SP(P, \texttt{trans}) := \exists \langle l', x'_v \rangle\ P[x_s \leftarrow \langle l', x'_v \rangle] \wedge G_{l',x_l}[x'_v] \wedge x_v = R_{l',x_l}(x'_v) \wedge I_{x_l}[x_v]$$

Proof. Let us assume that there are S and S' such that $[\![P]\!]_S$ holds and $S, s \rightsquigarrow^* S', \texttt{skip}$. We just have to prove that $[\![SP(P, s)]\!]_{S'}$ holds. The proof is performed inductively on the structure of the statement s by checking that every case of SP is implied by the operational semantics of $\mathbf{Imp}_{HA}$. This is a consequence of the operational semantics of HA given on Figure 1. □

Example 3. Let us prove that, if the HA of Figure 2 is in a state satisfying $x_l = on \wedge y = low$, then any continuous evolution of duration t leads to a state satisfying $y \leq max$. By Lemmas 2 and 3, it is sufficient to prove that the following implication holds in any program state:

$$SP((x_l = on \wedge y = low), \texttt{evolve}\ t) \Rightarrow y \leq max.$$

Let us assume that we are in program state such the left-hand side holds, and we prove that $y \leq max$ holds. From the definition of SP, we know that there exists a function ϕ such that

$$(x_l = on \wedge \phi_y(0) = low) \wedge \phi(t) = (x, y) \wedge (\forall \tilde{t} \in [0, t]\ \tfrac{d\phi}{dt} = F_{x_l} \wedge I_{x_l}[\phi(\tilde{t})])$$

As a consequence, we have $y = low + rate_{in} \cdot t$ (by solving the ODE) and $y \leq high$ (by unfolding the location invariant I_{x_l}). The latter property, in conjunction with Constraint (1) of Example 1, proves the goal $y \leq max$ by linear arithmetic.

Remark 1. As we will later pass the verification conditions to automated tools, it is important to eliminate as many quantifiers as possible beforehand. For instance, $SP(P, \texttt{trans})$ has the form $\exists l'\ Q[l']$. This is equivalent to the disjunction $Q[l_1] \vee \ldots \vee Q[l_p]$ with $l_1, \ldots, l_p$ all the locations. In the case of $SP(P, \texttt{evolve}\ t)$, Example 3 shows how one can get rid of $\exists \phi$ if the ODE admits a closed form.

4 Inductive Verification Method

4.1 Induction Strategy

We now present an algorithm derived from Lemma 2 that performs safety verification of a HA. The statement of Lemma 2 is unpractical, as it requires verifying infinitely-many programs. This section describes how we can build weaker yet more practical variants of it, by only considering a bounded number of programs. The approach is as follows. Let us assume that there is a predicate P^+ such that $P^+ \Rightarrow P$ and

[3] $P[x \leftarrow e]$ denotes the substitution of all the occurrences of variable x in P with e.

– from an initial state, any execution of HA reaches a state satisfying P^+ after alternating at most m continuous evolutions and m discrete transitions,
– from any state satisfying P^+, any execution of HA reaches a state satisfying P^+ after alternating at most n continuous evolutions and n discrete transitions.

Verifying the safety property is therefore simple:

– For the initial m-step execution, we check that every intermediate state is safe and that the execution finally reaches the region represented by predicate P^+ (base case).
– For the n-step execution from the region P^+, we check that every intermediate state is safe and that the execution finally reaches the region P^+.

The success of our approach depends on whether we can exhibit some lengths m and n and some predicate P^+ for a given HA.

We first show the simplest case ($m = 0$ and $n = 1$): the base case is the verification of the initial states, and the induction is performed on a continuous phase followed by a discrete phase.

Theorem 1 (Simplest case). *Given a predicate P^+ such that $P^+ \Rightarrow P$ holds in any state, the following inference rule is correct:*

$$\frac{VC_0 : \mathit{Init} \Rightarrow P^+ \qquad \begin{array}{c} VC_1 : \forall t \geq 0\ SP(P^+, \texttt{evolve}\ t) \Rightarrow P \\ VC_{-1} : \forall t \geq 0\ SP(P^+, \texttt{evolve}\ t; \texttt{trans}) \Rightarrow P^+ \end{array}}{HA \models \Box P}$$

Proof. VC_0 checks that the initial states satisfy the property P^+. VC_{-1} inductively verifies that all of the possible two consecutive continuous and discrete phases $\sigma_i \xrightarrow{t_{i+1}} \sigma_{i+1} \xrightarrow{0} \sigma_{i+2}$ evolve for the arbitrary duration t_{i+1} from a state σ_i that satisfies P^+ to a state σ_{i+2} that again satisfies P^+. VC_1 ensures that the safety property was not broken during the continuous phase. □

We now extend the above induction to a more generic case.

Theorem 2 (Unrolled case).

$$\frac{\begin{array}{lll} SP_1 \equiv SP(\mathit{Init} \wedge \neg P^+, \texttt{evolve}\ t_1) & VC_1 : & \forall t_1 \geq 0\ SP_1 \Rightarrow P \\ SP_2 \equiv SP(SP(SP_1, \texttt{trans}) \wedge \neg P^+, \texttt{evolve}\ t_2) & & \\ & VC_2 : & \forall t_1, t_2 \geq 0\ SP_2 \Rightarrow P \\ \vdots & & \\ SP_m \equiv SP(SP(SP_{m-1}, \texttt{trans}) \wedge \neg P^+, \texttt{evolve}\ t_m) & & \\ & VC_m : & \forall t_1 \ldots t_m \geq 0\ SP_m \Rightarrow P \\ SP_0 \equiv SP(SP_m, \texttt{trans}) & VC_0 : & \forall t_1 \ldots t_m \geq 0\ SP_0 \Rightarrow P^+ \\ SP_{m+1} \equiv SP(P^+, \texttt{evolve}\ t_1) & VC_{m+1} : & \forall t_1 \geq 0\ SP_{m+1} \Rightarrow P \\ \vdots & & \\ SP_{m+n} \equiv SP(SP(SP_{m+n-1}, \texttt{trans}) \wedge \neg P^+, \texttt{evolve}\ t_n) & & \\ & VC_{m+n} : & \forall t_1 \ldots t_n \geq 0\ SP_{m+n} \Rightarrow P \\ SP_{-1} \equiv SP(SP_{m+n}, \texttt{trans}) & VC_{-1} : & \forall t_1 \ldots t_n \geq 0\ SP_{-1} \Rightarrow P^+ \end{array}}{HA \models \Box P}$$

Proof. This theorem is an extension of Theorem 1. It verifies that a state satisfying P^+ can be reached in at most m steps initially (from VC_1 to VC_0), and then inductively that P^+ can always be reached again in at most n steps (from VC_{m+1} to VC_{-1}). □

Remark 2. Only VC_0 and VC_{-1} check that P^+ holds after an execution; all the other VCs check the safety property P only. Moreover, except for VC_{m+1}, all these other conditions compute the SP by assuming that P^+ does not hold. Indeed, there might be less than n transitions before reaching again a state satisfying P^+ (or m transitions initially).

4.2 Verification Algorithm

Given a HA, a safety property $\Box P$, and the maximal numbers m_{max} and n_{max} of steps to unroll, the algorithm in Figure 4 tries to check that all the hypotheses of Theorem 2 hold, and thus that $HA \models \Box P$ holds too.[4] The algorithm performs the inductive verification with every $m \leq m_{max}$ and $n \leq n_{max}$ (line 1). It iteratively computes the base case (line 4) and then the induction step (line 7). Procedure Validate returns *true* if the given logic formula holds, *false* if it cannot conclude. The verification succeeds if all the verification conditions are successfully validated (line 10).

When the verification fails during the induction step, we strengthen the loop invariant (line 8) so that the failing condition holds, and we perform the verification anew. Possibly, procedure Learn strengthened the invariant so much that we detect it is now useless (line 3). In this case, we leave from the inner recursion and try the verification with another m and n, or otherwise return *false*.

Note that the algorithm does not specify how to enumerate m and n. Typically, we enumerate from $m = 0$ and $n = 1$, but for certain models, we can guess the values, e.g., from the size of a lasso-shaped automaton. In the algorithm, the verification of the base case (lines 4-6, named BaseCase) and the induction step (lines 7-11, named Induction) are independent, thus we can also reverse the order of the two verification processes.

4.3 Loop Invariant Generation

In the following, we present the loop invariant generation method implemented in procedure Learn. Let us assume that the verification of a condition $VC_i \equiv \forall t_1 \dots t_i \geq 0\ SP(P^+, s) \Rightarrow P$ has failed in the induction step. Then, Learn(VC_i) generates a lemma Q from the failed verification. Specifically, Learn generates a formula Q such that VC_i becomes valid after we update the loop invariant as $P^+ := P^+ \wedge Q$. Basically, Learn searches for Q such that $SP(Q, s) \Rightarrow VC_i$ holds by applying algebraic transformations to VC_i. Note that all the occurrences of variable x_s (the current state) in Q refer to the time P^+ holds, while the ones in VC_i refer to the state at the end of the execution of s. To fix this discrepancy,

[4] A failure of the algorithm does not imply that the safety property is invalid.

Input: HA; P; $m_{max} \in \mathbb{N}_{\geq 0}$; $n_{max} \in \mathbb{N}_{>0}$
Output: *true*: $HA \models \Box P$; *false*: cannot decide $\Box P$ within $m_{max} + n_{max}$ steps

for $m \in \{0, \ldots, m_{max}\}$; $n \in \{1, \cdots, n_{max}\}$ **do**
  $P^+ := P$
  while $P^+ \not\equiv$ *false* **do**
    if $\neg \forall i \in \{0, \ldots, m\}$ $\mathsf{Validate}(VC_i)$ **then**
      break
    end if
    if $\exists j \in \{m+1, \ldots, m+n, -1\}$ $\neg\mathsf{Validate}(VC_j)$ **then**
      $P^+ := P^+ \wedge \mathsf{Learn}(VC_j)$
    else
      return *true*
    end if
  end while
end for
return *false*

Fig. 4. Algorithm for inductive verification.

Learn computes Q by using a *quantifier elimination* (QE) method, such as the Resolve procedure of Mathematica:

$$Q := \mathsf{QE}(\forall x_s \forall t_1 \ldots t_i \ (SP((P^+ \wedge x_0 = x_s), s) \ \Rightarrow \ P))[x_0 \leftarrow x_s].$$

To simplify the loop invariant, the other local variables in VC_i, i.e., $\phi, \tilde{t}, l', x'_v$ introduced in the SP calculus in Lemma 3 and t_i introduced in Theorem 2, should also be removed. Unfortunately, QE with mixed quantifiers and function quantifiers is a hard problem in general. See Remark 1 and the next section for details on how we perform this simplification.

The formula computed for Q is often a large disjunctive formula that is unusable as a loop invariant. For instance, some sub-formulas of Q describe states that are never accepted by the HA. Such sub-formulas are not only useless but make the verification process expensive. So we strengthen Q according to the following strategies:

- *Lemma separation.* We split Q at the (top-most) disjunction operators and employ one (or several) of the resulting sub-formulas.
- *Location disabling.* When we remove a sub-formula of Q that is related to some location l, we insert the constraint $x_l \neq l$. The resulting loop invariant might be effective when combined with loop unrolling.

5 Implementation

We have implemented the method presented in the previous sections using Mathematica 8.0.4[5], which can perform the computations in a fully symbolic manner.

[5] `http://www.wolfram.com/mathematica/`

Note that the loop invariant generation by Learn (line 8) is not automatic but guided by the user so as to apply the strategies described in Section 4.3. Validate is implemented in three different ways by using the built-in procedures of Mathematica, FullSimplify, Reduce, and FindInstance. We also rely on Mathematica's DSolve to find closed form of ODEs whenever possible.

In the implementation of BaseCase and Induction, we optimize the computation in two ways. First, we do not validate each VC_i separately but try to reuse the common assumptions. When validating VC_i, the algorithm computes SP_i which axiomatizes the state after executing the corresponding program s_i, and then validates $SP_i \Rightarrow P/P^+$. If we perform the validation of VCs in ascending order, we can compute SP_i from SP_{i-1} efficiently. Second, we perform location-wise validation of VCs to avoid the inefficiency that occurs when the execution of program s spans multiple locations. So we replicate the SP and instantiate each copy with a different location (cf. Remark 1). Throughout the computation, we manage the set of the copies instead of the original SP. Although it causes Validate to be called more often, the computation is more efficient in general.

Example 4. We verify the safety property of Example 2 for the HA in Example 1 with this implementation. Following the main algorithm, we first compute with $m = 0$ and $n = 1$. We run BaseCase to check that *Init* entails $P^+ \equiv P$, and it returns *true*. Next, we simulate a continuous and discrete change by running Induction. It computes the SP separately for each of the locations, *on*, *sw-off*, *off*, and *sw-on*, and validates VCs. For VC_1, the validation for locations *on* and *off* succeeds but the validation for *sw-off* and *sw-on* fails. Procedure Learn generates the following lemmas for these two locations.

$$\begin{aligned} Q_{sw\text{-}off} \equiv\ & min + x \cdot rate_{in} \leq y + delay \cdot rate_{in} \leq max + x \cdot rate_{in} \vee \\ & x = delay \vee y + delay \cdot rate_{in} < low + x \cdot rate_{in}, \\ Q_{sw\text{-}on} \equiv\ & min + x \cdot rate_{out} \leq y + delay \cdot rate_{out} \leq max + x \cdot rate_{out} \vee \\ & x = delay \vee high + x \cdot rate_{out} < y + delay \cdot rate_{out}. \end{aligned}$$

Here, we can use either of the two presented strategies for improving the loop invariant. For instance, location disabling appends

$$Q_1 := x_l \neq sw\text{-}off \wedge x_l \neq sw\text{-}on$$

to P^+. The VCs are then successfully validated with $m = 0$ and $n = 2$.

The lemma-separation strategy makes use of the additional lemmas generated by Learn. Here, we divide each lemma into three parts at the top-most disjunction operator. Then, the first part of each lemma (denoted $Q_{sw\text{-}off,1}$ and $Q_{sw\text{-}on,1}$) makes the verification successful. More precisely, if we append

$$Q_2 := (x_l = sw\text{-}off \Rightarrow Q_{sw\text{-}off,1}) \wedge (x_l = sw\text{-}on \Rightarrow Q_{sw\text{-}on,1})$$

to P^+, the validation succeeds with $m = 0$ and $n = 1$.

Table 1. Experimental results.

example	locs	vars	unroll	lemmas	Mathematica	MC tool	KeYmaera
WLM (Ex. 1)	4	2	0/1	2	0.85s	–	1.8s
LGB	2	3	4/2	3	2.22s	0.004s (H)	–
temp. control	4	3	1/1	4	2.82s	0.012s (H)	–
bouncing ball	1	2	0/1	1	0.49s	–	0.9s
ETCS	2	3	0/1	1	4.48s	–	3.1s
highway 9	10	9	0/2	1	0.22s	0.22s (P)	–
highway 19	20	19	0/2	1	3.64s	–	–

6 Experiments

To confirm the feasibility of our method and to compare it with existing tools, we applied it to several examples taken from the literature. We also verified the examples using the existing tools, HyTech, PHAVer, and KeYmaera, for comparison. The encoded models for the implementation is available at `http://www.ueda.info.waseda.ac.jp/~ishii/pub/mathybrid/`. Table 1 reports the results of verifying the examples using our implementations. The columns are: the number of locations; the number of variables; the way loops are unrolled (i.e., m/n); how many times P^+ had to be improved by the main algorithm; the computational time taken by the BaseCase and Induction procedures implemented in Mathematica; the time taken by HyTech (version 1.04f, indicated by "H") or PHAVer (version 0.38, indicated by "P"); and the time taken by KeYmaera (version 3.0). The notation "–" means that the verification failed. The experiments were run on a 3.4GHz Intel Xeon processor with 4GB of RAM. Note that the computational time for our method only measures the process after we found the loop invariants, since their generation requires some human interaction.

6.1 Considered Examples

WLM. Example 1 could be verified with our proposed method in a reasonable time, as explained in Example 4. In [15], the same instance was handled by using a mathematical solver manually, whereas our Mathematica implementation verified the instance by simply following the algorithm. The model-checking (MC) tools could not handle this instance because of the nonlinear terms caused by the parameterized flow rate. KeYmaera verified this example but the model had to be given a loop invariant beforehand [17].

Leaking gas burner (LGB) [3]. Our implementation verified this rectangular HA consisting of two locations $L = \{\mathit{leaking}, \mathit{non\text{-}leaking}\}$ as follows: Induction failed in the verification of the first continuous evolution in the two locations. The lemma generated for *leaking* was successful. For *non-leaking* though, we had to resort to our location-disabling strategy. Then, the verification succeeded with $m = 4$ and $n = 2$. This model was verified efficiently by the MC tools. KeYmaera could not verify the model, even with the loop invariant.

Temperature control [3]. Our implementation verified this problem after some preliminary transformations. First, we verified that location `shutdown` of the HA is never reached. In order to get a loop invariant, we strengthened the safety property by appending the negation of the guard condition of the transition edge to `shutdown`. The failure of Induction led to a lemma of the form $Q_1 \vee Q_2 \vee Q_3$, but setting each sub-lemma as a loop invariant did not make the verification successful. After some trials, we found that the lemma $Q_1 \wedge (Q_2 \vee Q_3)$ was a necessary loop invariant. Finally, the verification succeeded for $m = 1$ and $n = 1$. This model was also verified efficiently by the MC tools. KeYmaera could not verify the model, even with the loop invariant we had found.

Bouncing ball. This simple nonlinear HA describes a ball with a constant acceleration. As exemplified in [16], we verified that the height of the ball never exceeds the initial energy level of the ball, assuming that the reflection coefficient is smaller than 1. We first attempted the verification under a simple constraint that specified only the sign of each parameter and generated a lemma equivalent to the energy consumption constraint in [16]. We succeeded by setting this lemma as the initial condition and the loop invariant. KeYmaera verified the model given the energy consumption constraint as the initial condition.

European train control system (ETCS) [16,10,2]. The simple model borrowed from [16] is about a train at a position z that should not exceed a limit m. The original model does not have guard conditions so we set them manually based on the analysis in [2]. We attempted to verify the safety property $\Box z < m$ by running the algorithm with $m = 0$ and $n = 1$. Verification succeeded after we obtained a loop invariant from the failure in the validation of VC_1. This model was also verified in [16,10] by using several strategies for the model transformation and loop invariant generation. MC tools could not verify the model because of the nonlinear constraints. KeYmaera verified the model by setting a specific parameter constraint as described in [16].

Highway [14]. This model concerns an autonomous highway with n vehicles. We solved instances for $n = 9$ and $n = 19$, which were also computed by the specific method in [14]. PHAVer verified the instance of $n = 9$ but the computation for $n = 19$ failed after consuming the available memory. KeYmaera could not verify this example.

6.2 Discussions

The MC tools verified three examples quite efficiently. However, our method was better for the other examples. First, it can handle uncertain parameters. Example 1 involves such parameters, as described in Equation (1). In the bouncing ball example, the initial height, velocity, and reflection coefficient are parameterized. Although HyTech and PHAVer verify the same problems with constant values given to the parameters, they cannot verify the instances that involve uncertain parameters. Second, our method scales better: for the highway example, PHAVer can handle only the instances up to $n = 15$ [14].

Although KeYmaera handles various hybrid programs automatically, it did not succeed on most hybrid programs that were translated from hybrid au-

tomata. Users often need to annotate models with a loop invariant that might be difficult to extract from the original problem [17]. Otherwise, users need to interact with the underlying theorem prover to investigate the correct derivation tree with various deduction rules. Our approach is limited in verification strategies, i.e., induction and loop unrolling, but the results show that the approach is effective for various examples in practice.

Although our method requires that the executions are lasso shaped (from the point of view of the loop invariant P^+), many examples in the literature can be handled. It, however, requires other verification strategies for the case of compositional and distributed hybrid automata.

7 Related Work

Various tools for the logical analysis of hybrid systems have been proposed. These methods translate hybrid systems into an underlying verification framework, such as STeP [15], PVS [1], SAL [9], Fluctuat [5], and Event-B [2,21]. However, neither the translation nor the verification is fully automated, because some invariants must be added manually, and the theorem provers require some interactions.

Another tool, KeYmaera [18,16], developed by Platzer et al., has been successful in recent years. This tool supports *hybrid programs* that are annotated using *differential (algebraic) dynamic logic*. A dedicated theorem prover verifies the programs by using a set of proof strategies [16]. With its imperative language, which is more expressive than HA, and its corresponding logic (which depends on 141 inference rules [18]), KeYmaera is able to perform various logical analysis through a variety of strategies, including induction, and can serve as a basis for a complete verification framework [16]. In contrast, our framework consists of a light imperative language that is sufficiently expressive to encode HA executions and a logical framework that is introduced to pursue automated verification with the induction strategy.

Recently, a logical analysis tool based on the framework of Hoare logic and relying on infinitesimal variables was proposed [10]. Although its verification scheme comes with several strategies and an invariant generation technique, its practical uses are still unclear.

There are techniques for hybrid systems that generate polynomial invariants by analyzing the executions of a HA via Gröbner basis manipulations [20,19]. These methods could be integrated in the Learn procedure of our framework.

The proposed method also relates to BMC methods. BMC of infinite executions based on induction has been proposed (e.g., [6]), but this approach is applied to discrete systems with continuous states. Most of the BMC tools for hybrid systems handle only finite executions. This is not the case for Hybrid SAL Relational Abstracter [22]. This tool is a translator from hybrid systems to discrete systems with a specific abstraction method. Our method directly handles HA without the abstraction.

8 Conclusions

This paper presents a tool for logical analysis of safety properties of HA, which is able to deal with a large class of linear and nonlinear HA, in contrast with the model-checking approach found in major existing tools.

Rather than introducing various derivation rules to automatically verify HA, we are using a simple process inspired from deductive program verification: strongest postcondition calculus. It allows us to compute logical formulas that, once proved, guarantee the safety of the HA. Our experiments show that our method succeeds in a reasonable time on some example HA from literature, including some that were not solvable with existing tools. The verification process amounts to finding loop invariants, as is the case for program verification. This search for sufficient invariants is guided by the responses from the decision procedures assisted by Mathematica.

A limitation of our approach is that the invariant generation process still requires some human interaction. Efficient automated search of invariant generations is the next challenge for us to tackle. Another direction for further research would be to explore the relation between our approach and some methods from model checking, e.g., verification of an over-approximated model [4].

Acknowledgments The authors are indebted to the anonymous referees for their helpful comments. This work was partially funded by JSPS (KAKENHI 23-3810).

References

1. Ábrahám-Mumm, E., Steffen, M., Hannemann, U.: Verification of hybrid systems: Formalization and proof rules in PVS. In: ICECCS. pp. 48–57 (2001)
2. Abrial, J.R., Su, W., Zhu, H.: Formalizing hybrid systems with Event-B. In: ABZ. pp. 178–193. *LNCS* 7316 (2012)
3. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138(1), 3–34 (1995)
4. Alur, R., Henzinger, T.A., Lafferriere, G., Pappas, G.J.: Discrete abstractions of hybrid systems. In: Proc. of the IEEE. vol. 88, pp. 971–984 (2000)
5. Bouissou, O., Goubault, E., Putot, S., Tekkal, K., Vedrine, F.: HybridFluctuat: A static analyzer of numerical programs within a continuous environment. In: CAV. pp. 620–626. *LNCS* 5643 (2009)
6. De Moura, L., Ruess, H., Sorea, M.: Bounded model checking and induction: From refutation to verification. In: CAV. pp. 14–26. *LNCS* 2725 (2003)
7. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18(8), 453–457 (1975)
8. Frehse, G.: PHAVer: Algorithmic verification of hybrid systems past HyTech. International Journal on Software Tools for Technology Transfer (STTT) 10(3), 263–279 (2008)
9. Ghosh, R., Tiwari, A., Tomlin, C.: Automated symbolic reachability analysis: with application to delta-notch signaling automata. In: HSCC. pp. 233–248. *LNCS* 2623 (2003)

10. Hasuo, I., Suenaga, K.: Exercises in nonstandard static analysis of hybrid systems. In: CAV. pp. 462–478. *LNCS* 7358 (2012)
11. Henzinger, T.A.: The theory of hybrid automata. Verification of Digital and Hybrid Systems (NATO ASI Series F: Computer and Systems Sciences) 170, 265–292 (2000)
12. Henzinger, T.A., Ho, P.H., Wong-Toi, H.: HyTech: A model checker for hybrid systems. STTT 1, 110–122 (1997)
13. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 and 583 (1969)
14. Jha, S.K., Krogh, B.H., Weimer, J.E., Clarke, E.M.: Reachability for linear hybrid automata using iterative relaxation abstraction. In: HSCC. pp. 287–300. *LNCS* 4416 (2007)
15. Manna, Z., Sipma, H.: Deductive verification of hybrid systems using STeP. In: HSCC. pp. 305–318. *LNCS* 1386 (1998)
16. Platzer, A.: Logical Analysis of Hybrid Systems. Springer (2010)
17. Platzer, A.: Guide for KeYmaera hybrid systems verification tool. http://symbolaris.com/info/KeYmaera-guide.html (2012), [Accessed January 1, 2013]
18. Platzer, A., Quesel, J.D.: KeYmaera: A hybrid theorem prover for hybrid systems. In: IJCAR. pp. 171–178. *LNCS* 5195 (2008)
19. Rodriguez-Carbonell, E., Tiwari, A.: Generating polynomial invariants for hybrid systems. In: HSCC. pp. 590–605. *LNCS* 3414 (2005)
20. Sankaranarayanan, S., Sipma, H., Manna, Z.: Constructing invariants for hybrid systems. In: HSCC. pp. 539–554. *LNCS* 2993 (2004)
21. Su, W., Abrial, J.R., Zhu, H.: Complementary methodologies for developing hybrid systems with Event-B. In: ICFEM. pp. 230–248. *LNCS* 7635 (2012)
22. Tiwari, A.: HybridSAL relational abstracter. In: CAV. pp. 725–731. *LNCS* 7358 (2012)