An Improved HHL Prover: An Interactive Theorem Prover for Hybrid Systems

Shuling Wang(✉), Naijun Zhan, and Liang Zou

State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences, Beijing, China
wangsl@ios.ac.cn

Abstract. Hybrid systems are integrations of discrete computation and continuous physical evolution. To guarantee the correctness of hybrid systems, formal techniques on modelling and verification of hybrid systems have been proposed. Hybrid CSP (HCSP) is an extension of CSP with differential equations and some forms of interruptions for modelling hybrid systems, and Hybrid Hoare logic (HHL) is an extension of Hoare logic for specifying and verifying hybrid systems that are modelled using HCSP. In this paper, we report an improved *HHL prover*, which is an interactive theorem prover based on Isabelle/HOL for verifying HCSP models. Compared with the prototypical release in [22], the new HHL prover realises the proof system of HHL as a shallow embedding in Isabelle/HOL, rather than deep embedding in [22]. In order to contrast the new HHL prover in shallow embedding and the old one in deep embedding, we demonstrate the use of both variants on the safety verification of a lunar lander case study.

1 Introduction

Hybrid systems are fusions of discrete dynamic systems and continuous dynamic systems, many of which are safety-critical, e.g., transportation, healthcare, spacecrafts, *etc.* In order to ensure the correct functioning of hybrid systems, formal techniques on modelling and verification have been proposed. Among them, the most popular model is hybrid automata [2,10], with the subsequent temporal logic based specification languages and model checkers [1,8,13]. However, due to the undecidable reachability problem of hybrid systems, various abstractions or (numeric) approximations for hybrid automata are required [3,4]. This leads to incomplete coverage of the system dynamics or loss of precision of proof results.

Alternatively, the deductive approach has been proposed, which verifies systems by proofs rather than state space exploration in model checking. This approach asks for a formal modelling language with (de-)compositionality and meanwhile a specification logic for verifying the corresponding models. Following this research line, we extended Hoare Logic to hybrid systems and established Hybrid Hoare Logic (HHL) [11]. In HHL, a hybrid system is modeled by Hybrid CSP (HCSP) process. HCSP is a formal modeling language for hybrid systems,

M. Butler et al. (Eds.): ICFEM 2015, LNCS 9407, pp. 382–399, 2015.
DOI: 10.1007/978-3-319-25423-4_25

due to He, Zhou *et al.* [9,21], which is an extension of CSP by introducing differential equations for representing continuous evolution. HCSP inherits from CSP the compositional process algebra constructs including communication-based synchronization and concurrency, thus it is expressive enough for describing distributed components and the interactions between them. Moreover, it extends CSP with several forms of interrupts to continuous evolution for realizing communication-based discrete control. To capture both discrete and continuous behavior of HCSP, the assertion languages of HHL include two parts: one is first-order logic (FOL), used for specifying properties of discrete processes, and the other is a subset of Duration Calculus (DC) [19,20], called history formulas, for specifying the execution history for continuous processes. A proof system for HHL was provided in [11]. In particular, the notion of differential invariant [12,14] is used to characterize the behavior of differential equations.

In [22], a prototypical implementation for HHL verification framework in proof assistant Isabelle/HOL, called *HHL prover*, was reported. In the HHL prover of [22], HHL was realised in a deep embedding style, i.e., the assertion languages of HHL including FOL and DC are defined as new datatypes of Isabelle/HOL. Since then, the HHL prover has been successfully applied to the verification of some real-world hybrid systems, e.g., Chinese train control systems [23,24] and the GNC control program of a lunar lander [18].

The disadvantage of the prototypical HHL prover is very obvious: due to the deep embedding of the HHL assertions, the proof of FOL and DC formulas needs to be conducted by the user completely, to apply the deductive rules for FOL and DC manually, thus the proof effort is very high. The main contribution of this paper is to implement the proof system for HHL in shadow embedding [1]. In addition, to demonstrate the efficiency of the improved HHL prover, we apply the prover in both embeddings to the safety verification of the slow descent guidance control program of a lunar lander, which is a closed-loop dynamic system composed of a physical plant and an embedded control program. We make a comparison between the proof results obtained from both embeddings, which indicates that the shallow embedding has better performance than the deep embedding.

Related Work. There are some tools on formal modelling and verification of hybrid systems. The tool d/dt [5] provides reachability analysis and safety verification of hybrid systems with linear continuous dynamics and uncertain bounded input. iSAT-ODE [7] is a numerical SMT solver based on interval arithmetic that can conduct bounded model checking for hybrid systems. Flow* [6] computes over-approximations of the reachable sets of continuous dynamical and hybrid systems in a bounded time. However, due to the undecidable reachability problem of hybrid systems, the above tools based on model checking are incomplete. Based on the alternative deductive approach, the theorem prover KeYmaera [15] is proposed to verify hybrid systems specified using differential dynamic logic.

[1] The HHL prover in both embeddings, plus the corresponding models and proofs related to the case study, can be found at https://github.com/wangslyl/hhlprover.

Compared to our work, it supports a simple set of hybrid constructs that do not cover communications and parallel composition.

Organization. The rest of the paper is organized as follows: Sects. 2 and 3 introduce briefly the proof assistant Isabelle/HOL, and the modelling language HCSP and its specification logic HHL, respectively; Sects. 4 and 5 present the HHL prover in shallow and deep embeddings respectively; Sect. 6 presents the lunar lander case study; and Sect. 7 concludes the paper.

2 Isabelle/HOL

In this section, we give a brief introduction of Isabelle/HOL, based on which the modelling and verification framework of hybrid systems is mechanized. Isabelle/HOL is a proof assistant for Higher-Order Logic (HOL). It supports functional modeling of systems by providing datatypes, functions, terms and formulas; and meanwhile, it enables proof of properties by construction by providing a set of built-in inference rules and proof tactics. Except for basic types such as *bool*, `nat`, etc., Isabelle/HOL provides the way to define a recursive datatype, for instance, a list of elements of type `'a` can be defined by `'a list`:

datatype 'a list = Nil | Cons 'a 'a list

where `Nil`, referring to the empty list, and `Cons`, adding an element to the front of a list, are the two constructors respectively. A type can also be constructed from existing ones by using `types`, e.g. `types nlist = nat list`. With the existence of types, functions are used to describe the relations between values of different types. A recursive function can be defined with respect to the constructors for the involved datatypes, e.g., the function `len` returns the length of a list:

primrec len :: 'a list ⇒ nat **where**
len Nil = 0 |
len Cons x xs = 1 + len xs

Non-recursive functions can be defined with the `definition` command, and more general cases for both recursive and non-recursive functions can be defined by means of `fun`. By applying functions to arguments, terms are formed, and in particular, a special class of terms with type *bool* are formulas. The compound formulas can be formed by applying logical connectives, such as ∧, ¬, ∀, *etc.*

Isabelle can prove facts directly based on induction and simplications. Besides, it supports more complicated verification by applying HOL inference rules for classical reasoning, e.g. the introduction or elimination rules for conjunction ∧ or disjunction ∨, *etc.* It provides a set of methods to automate classical reasoning, such as `blast`, `auto`, `arith` and so on. Isabelle also includes some high-level proof tactics. In particular, the tool `sledgehammer` is a certified integration of third-party automated theorem provers and SMT solvers including Alt-Ergo, Z3, CVC3, and so on, and `nitpick` is a counterexample generator. For HHL prover in shallow embedding, all the proof obligations are reduced to HOL formulas at the end, for which `sledgehammer` can be used to search the proofs automatically.

3 Hybrid CSP

This section will give a brief introduction of Hybrid CSP (HCSP) and the specification logic Hybrid Hoare Logic for reasoning about HCSP processes. They constitute the theoretical basis on the modelling language and the safety logic of HHL prover.

3.1 Syntax

Hybrid CSP [9,21] is an extension of CSP by introducing differential equations for representing continuous evolution and several forms of interruptions to continuous evolution. The syntax of a subset of HCSP is given as follows:

$$\begin{aligned} P &::= \text{skip} \mid x := e \mid ch?x \mid ch!e \mid P; Q \mid B \to P \mid P \sqcup Q \mid P^* \\ &\quad \mid \langle \dot{s} = e \& B \rangle \mid \langle \dot{s} = e \& B \rangle \trianglerighteq [\!]_{i \in I} (io_i \to Q_i) \\ S &::= P \mid S \| S \end{aligned}$$

where P, Q, Q_i, S are HCSP processes, x and s stand for process variables, ch for channel name, io_i for a communication event (either input $ch?x$ or output $ch!e$), B and e for Boolean and arithmetic expressions, and I for a non-empty set of indices of communications, respectively. A whole HCSP model S is defined as a sequential process or a parallel composition of several sequential processes at the top level.

The intuitive meaning of the individual constructs is explained as follows:

- skip, $x := e$ are defined as usual.
- The input $ch?x$ receives a value along channel ch and assigns it to x, and output $ch!e$ sends the value of e along ch. A communication takes place as soon as both the sending party (i.e. $ch!$) and the receiving party (i.e. $ch?$) are ready, and may cause one side to wait.
- The sequential composition $P; Q$ behaves as P first, and if it terminates, as Q afterwards.
- The conditional $B \to P$ behaves as P if B is true, otherwise it terminates immediately.
- The internal choice $P \sqcup Q$ behaves as either P or Q, and the non-deterministic choice is made by the system itself.
- The repetition P^* executes P for some finite number of times.
- $\langle \dot{s} = e \& B \rangle$ is the continuous evolution statement, where s represents a vector of real variables and $\dot{s}$ the first-order derivative of s. It forces s to evolve continuously according to the differential equation $\dot{s} = e$ as long as B, which defines the *domain of* s, holds, and terminates when B turns false. $\langle \dot{s} = e \& B \rangle$ is a boundary interruption.
- The communication interruption $\langle \dot{s} = e \& B \rangle \trianglerighteq [\!]_{i \in I} (io_i \to Q_i)$ behaves like the continuous $\langle \dot{s} = e \& B \rangle$, except that it is preempted as soon as one of the communications io_i takes place, and then is followed by the respective Q_i.
- $S_1 \| S_2$ behaves as if S_1 and S_2 run independently except that all communications along the common channels connecting S_1 and S_2 are to be synchronized. S_1 and S_2 in parallel can neither share variables, nor input or output channels.

Some commonly used constructs of HCSP in [9,21] are derivable from the above syntax, e.g.,

$$\begin{array}{ll} \textbf{wait } d & \widehat{=}\ t := 0; \langle \dot{t} = 1 \& t < d \rangle \\ \langle \dot{s} = e \& B \rangle \trianglerighteq_d Q & \widehat{=}\ t := 0; \langle \dot{s} = e \wedge \dot{t} = 1 \& t < d \wedge B \rangle; t \geq d \rightarrow Q \end{array}$$

Especially the timeout $\langle \dot{s} = e \& B \rangle \trianglerighteq_d Q$ executes according to the continuous evolution $\langle \dot{s} = e \& B \rangle$ for the first d time units, and Q afterwards.

Example 1. The following presents a simple HCSP description of a continuously evolving plant with discrete control:

$$\langle \dot{x} = f(x,u) \rangle \trianglerighteq sensor!x \rightarrow actuator?u)^* \| (\textbf{wait } d; sensor?s; actuator!Comp(s))^*$$

The plant evolves according to the dynamics $\dot{x} = f(x,u)$ that depends on a control parameter u. Every d time units, the controller samples the state of the plant x via channel *sensor*, and computes the new control parameter by *Comp*, and sends it back to the plant for the next cycle via channel *actuator*.

3.2 Operational Semantics

Let *Real* be the set of reals. A state, ranging over σ, σ', is a function that assigns a value to each variable. For simplicity, given a state σ and an expression e, we also use $\sigma(e)$ to return the value of e under σ. A flow, ranging over h, h', is a function that assigns a state to each real time point in *Real*. Each transition relation has the form $(P, now, h) \xrightarrow{a} (P', now', h')$, where P, P' are processes, a is an event, now, now' are real time, and h, h' are flows, respectively. It represents that, starting from the initial time now with the initial flow h (notice that the initial state is exactly $h(now)$), P performs event a and evolves to P' at time now' with the flow h'. The events a here can be a discrete internal event, like skip, assignment, evaluation of Boolean conditions, *etc.*, or a communication event, or a time delay. For the sake of embedding in HHL prover, we define the flows h, h' to be total on the whole real domain. For the above transition, by confining flow h' on the interval $[now, now']$, we can obtain the execution history from P to P'.

For page limit, we only present the semantics of continuous evolution here.[2] Given an initial flow h and initial time now, assume $S(t)$ is a solution of $\dot{s} = e$ defined over $[0, d]$ for some duration $d > 0$, satisfying that $S(0) = h(now)(s)$. We define flow $h\langle now, d, S\rangle$ same to h except that for all $t \in (now, now + d]$, $h\langle now, d, S\rangle(t) = h(t)[s \rightarrow S(t - now)]$, i.e., the value of s is overriden by the solution $S(t)$ over the execution interval $(now, now + d]$. The semantics of continuous evolution is then defined by the following rules:

[2] The full version of both the operational semantics of HCSP and the specification logic HHL to be introduced next can be found at [17].

$$\frac{\forall t \in [0,d).h\langle now,d,S\rangle(now+t)(B)=true}{(\langle \dot{s}=e\&B\rangle,now,h)\xrightarrow{d}(\langle \dot{s}=e\&B\rangle,now+d,h\langle now,d,S\rangle)}$$

$$\frac{h(now)(B)=false \text{ or}\\ (h(now)(B)=true\wedge\forall t\in(0,d).h\langle now,d,S\rangle(now+t)(B)=false)}{(\langle \dot{s}=e\&B\rangle,now,h)\xrightarrow{\tau}(skip,now,h)}$$

The first rule indicates that, it evolves for d time units according to $\dot{s}=e$ if B evaluates to true within period $[now, now+d)$ (the right end exclusive). Otherwise, indicated by the second rule, the continuous evolution terminates at now if B evaluates to false at now, or if B evaluates to false at a positive open interval right to now (depending on whether B is open or close).

The transition closure $(P_0,now_0,h_0)\xrightarrow{a_1\cdots a_k}(P_k,now_k,h_k)$ for some $k>0$ is defined, iff there exists a sequence of transitions

$$(P_0,now_0,h_0)\xrightarrow{a_1}(P_1,now_1,h_1),\cdots,(P_{k-1},now_{k-1},h_{k-1})\xrightarrow{a_k}(P_k,now_k,h_k)$$

When $P_k=skip$, we call the sequence of the transitions a *complete execution* of P_0, and for simplicity write it as $(P_0,now_0,h_0)\rightsquigarrow(now_k,h_k)$ by omitting the labels and the terminating process $skip$.

3.3 Hybrid Hoare Logic

In order to verify HCSP, Hybrid Hoare Logic (HHL) [11] is defined. As an extension of Hoare logic to hybrid systems, it considers both discrete and continuous properties, that correspond to an isolated time point and a time interval resp.

History Formulas. In order to describe the interval-related properties, we introduce history formulas, that are defined by duration calculus (DC) [19,20]. DC is a first-order interval-based real-time logic with one binary modality known as chop $^\frown$, but is extended with a special structure of temporal variable, i.e. state durations. We define history formulas HF by the following subset of DC:

$$HF ::= \ell\circ T \mid \lceil S\rceil \mid HF_1{}^\frown HF_2 \mid \neg HF \mid HF_1\vee HF_2$$

where ℓ is a temporal variable denoting the length of the considered interval, $\circ\in\{<,=\}$ is a relation, T a non-negative real, and S a first-order state formula over process variables. HF can be interpreted over flows and intervals. Let the judgement $h,[a,b]\models HF$ represent that HF holds under h and $[a,b]$, then we have

$$h,[a,b]\models \ell\circ T \text{ iff } (b-a)\circ T \qquad h,[a,b]\models\lceil S\rceil \text{ iff } b>a\wedge\int_a^b h(t)(S)=b-a$$
$$h,[a,b]\models HF_1{}^\frown HF_2 \text{ iff } \exists c.a\le c\le b\wedge h,[a,c]\models HF_1\wedge h,[c,b]\models HF_2$$

As defined above, ℓ indicates the length of the considered interval. $\lceil S\rceil$ asserts that the duration of state S on interval $[a,b]$ is $b-a$, i.e. S holds almost everywhere in the considered non-point interval. Thus, based on $\lceil S\rceil$, an invariant property related to an interval can be specified. Later, we will write $\lceil S\rceil^<$ as an

abbreviation for $\lceil S \rceil \vee \ell = 0$ to include the point case. Lastly, $HF_1 \frown HF_2$ asserts that the interval can be divided into two sub-intervals such that HF_1 holds for the first and HF_2 for the second. The first-order connectives $\neg$ and $\vee$ can be explained as usual.

Specification and Inference Rules. The specification for a sequential HCSP process P takes the form $\{Pre\}P\{Post; HF\}$, where the pre-/post-condition Pre and $Post$, defined by first-order logic (FOL), specify properties of variables that hold at the beginning and termination of the execution of P respectively, and the history formula HF, specifies properties of variables that hold throughout the execution interval of P. The effect of discrete processes will be specified by the pre-/post-conditions, but not recorded in the history. The specification for a parallel process $P_1 \| P_2$ is then defined by assigning to each sequential component of it the respective pre-/post-conditions and the history formula, shown as below:

$$\{Pre_1, Pre_2\}P_1 \| P_2\{Post_1, Post_2; HF_1, HF_2\}$$

In HHL, HCSP constructs are axiomatized by a set of axioms and inference rules, which constitutes a basis for implementing the verification condition generator for verifying HCSP specifications in HHL prover. We will give a more detailed explanation of HHL in next section.

4 HHL Prover: Shallow Embedding

HHL prover aims to verify whether a HCSP process conforms to a HHL specification in a machine-checkable way. The implementation of HHL prover requires to embed the whole HHL verification framework in Isabelle/HOL. There are two different ways for the embedding: shallow or deep. The shallow embedding defines the assertions of HHL (i.e. FOL and DC formulas) by HOL predicates on process states or flows, while in deep embedding, it defines the assertions as new datatypes. In this section, we will present HHL prover in shallow embedding in detail, and in next section the prover in deep embedding.

4.1 HCSP

In both embeddings, we start from encoding the bottom construct, i.e. expressions, that are represented as a datatype `exp`[3]:

datatype exp = Con Val | RVar *string* | SVar *string* | BVar *string*
| exp [+] exp | exp [−] exp | exp [∗] exp | exp [/] exp

An expression can be a constant `Con v`, where `v` is of type `Val` for representing constants, e.g. `Con Real n`; a variable, that can be `RVar x`, `SVar x` and `BVar x`

[3] To distinguish from HOL, we wrap the arithmetic operators and FOL connectives with `[]`, and DC connectives with `[[]]` outside. For example, $\wedge, [\wedge], [[\wedge]]$ are HOL, FOL and DC conjunctions respectively.

for real, string, and boolean variables, respectively; an arithmetic expression constructed from arithmetic operators.

Based on `exp`, we represent HCSP processes by a datatype `proc`. Each construct of HCSP defined in Sect. 3 is encoded as a corresponding constructor in datatype `proc`. For examples, $B \to P$ is encoded as `IF B P`; the continuous evolution $\langle \dot{s} = e \& B \rangle$ is encoded as `<s:e&&Inv&b>`, with the addition of the differential invariant `Inv` of the differential equation $\dot{s} = e$ for the purpose of verification; for the same reason, the repetition P^* is encoded as `P*&&Inv`, where the loop invariant `Inv` is annotated. The invariants are unknown beforehand and will be solved in the proof process by calling an external invariant generator from HHL prover.

To encode the semantics of HCSP, we first define two types, `state` and `flow`, to model states and flows respectively. Then, given a process `P` of type `proc`, time `now`, `now'` of type *real*, and `h`, `h'` of type`flow`, the inductive function `semB P now h now' h'` returns true iff $(P, now, h) \rightsquigarrow (now', h')$ is a complete execution of P.

4.2 Assertion Languages

Two types of assertion logics are used in defining the specifications of HHL: FOL and DC. The FOL formulas are defined as predicates on states,

type_synonym fform = state ⇒*bool*

We can then write arbitrary Isabelle functions from `state` to *bool* to describe states. Especially, the FOL constructs can be derived as syntax flavours, like,

definition [True] :: fform **where**
[True] ≡ λ s. True
definition fImp :: fform ⇒ fform ⇒ fform (**infixl** "[→]" 65) **where**
p [→] q ≡λ s. p s → q s

The DC formulas are represented as predicates on flows and intervals,

type_synonym dform = flow ⇒*real* ⇒*real* ⇒*bool*

The history formulas presented in Sect. 3 can be defined correspondingly,

definition elE :: *real* ⇒ dform **where**
elE T ≡ λ h n m. (m−n) = T
definition almost :: fform ⇒ dform **where**
almost p ≡ λ h n m. (m>n) ∧ (∀a≥n. ∀ b≤m. a < b → (∃t. t>a ∧ t<b ∧ p(h(t))))
definition chop :: dform ⇒dform ⇒dform ("_[^]_" 80) **where**
H [^] M ≡λ h n m. (∃ nm. (nm ≥ n ∧ nm ≤ m ∧ H h n nm ∧ M h nm m))
definition dAnd :: dform ⇒dform ⇒dform ("[[&]]" 79) **where**
P [[&]] Q ≡ λ h n m. P h n m ∧ Q h n m

`elE T` implements $\ell = T$; `almost p` implements $\lceil p \rceil$, i.e. the duration of p is $m - n$ under flow h and interval $[n, m]$ satisfying $m > n$, iff for any positive open interval inside $[n, m]$, there always exists a point in it such that P is held; `H [^] M` implements $H^\frown M$.

As a consequence, the formulas can be interpreted directly, `(s |= p)` $\equiv$ `p s`, and `(h, [n, m] |= H)` $\equiv$ `H h n m`. Moreover, the proof of FOL and DC formulas is reduced to the proof of HOL formulas, which is supported by the built-in proof tactics of Isabelle/HOL. We have proved some of the lemmas for DC stated in [19] in Isabelle/HOL, e.g.,

$$True \Leftrightarrow \ell \geq 0 \quad \lceil S \rceil^\frown \lceil S \rceil \Leftrightarrow \lceil S \rceil \quad HF^\frown \ell = 0 \Leftrightarrow HF$$
$$\lceil S_1 \rceil \Rightarrow \lceil S_2 \rceil \text{ if } S_1 \Rightarrow S_2 \text{ is valid in FOL}$$

4.3 Specification and Inference Rules

With the definitions of HCSP and the assertion languages, we implement a function `ValidS` to represent a valid specification,

definition ValidS :: fform ⇒ proc ⇒ fform ⇒ dform ⇒ *bool* (“{ _}_{_; _}”)
where ValidS p c q H ≡ ∀ now h now' h' .semB c now h now' h'→ h(now) |= p
→ (h'(now') |= q ∧ h', [now, now'] |= H)

stating that, `{p} c {q;H}` is valid, iff starting from flow `h` and time `now`, if `c` terminates with flow `h'` and time `now'`, then the precondition `P` holds under `h` and `now` implies the postcondition `q` and the history formula `H` hold under `h'` and `now'`. Below we list some of the lemmas that correspond to the valid inference rules of HHL.

Assignment. Lemma `AssignRRule` presents the rule for assignment to a real variable, which indicates that `{p}` (RVar x := f) {q; H} holds, if `p` implies the weakest precondition `substF ([(RVar x, f)], q)` and `H` is implied by the strongest history formula `elE 0`.

lemma AssignRRule: (∀ s. (p [→]substF ([(RVar x, f)], q)) s) ∧
(∀ h now now'. (elE 0 [[→]] H) h now now')
⇒ {p} RVar x := f {q; H}

Here `substF ([(RVar x, f)], q)` is defined in the semantic level, i.e. after substituting `f` for `RVar x`, `q` holds,

substF ([(RVar x, f)], q) $\equiv$ λ s. q (λv. **if** v=(x, R) **then** evalE f s **else** s v))

in which `evalE f s` returns the value of `f` under state `s`.

Continuous Evolution. Lemma `ContinuousRule` states the rule for continuous evolutio. Function `cl`$(\cdot)$ extends the domain defined by the corresponding formula to include the boundary, e.g. `cl` $(x > 2) = x \geq 2$.

lemma ContinuousRule :∀ s.((p [→] Inv) [∧]
(exeFlow <v:E&&Inv&b> Inv [→] Inv) [∧] (Inv [∧] cl([¬]b) [→] q)) s
⇒ ∀ h now now'. ((elE 0[[∨]] almost (Inv [&] b)) [[→]] H) h now now'
⇒ {p} <v:E&&Inv&b> {q; H}

Consider the hypothesis, the FOL formula in the first two lines indicates that `Inv` is indeed a sufficiently strong invariant, i.e. it is satisfied by the initial state, preserved by the execution of the continuous evolution, and strong enough to guarantee the postcondition; the DC formula in the third line indicates that the evolution terminates immediately (specified by `elE 0`), or otherwise, if the evolution takes more than zero time, then the invariant `Inv` and the domain `b` hold almost everywhere throughout the whole execution. The lemma is proved valid, as a consequence, the proof of the specification for the continuous evolution is reduced to an equivalent differential invariant generation problem: if the `Inv` exists such that it satisfies the conditions in the hypothesis, then the original specification is proved. HHL prover will call an external invariant generator to solve the invariant generation problem.

Sequential Composition. As shown by Lemma `SequentialRule`, the postcondition of `P; Q` (i.e. `q`) is equivalent to the one of `Q`, and the history formula (i.e. `M`) is implied by the concatenation of the ones of `P` and `Q`. By recursively applying the inference rules of HHL, the two sub-specifications corresponding to `P` and `Q` can be transformed eventually to logical formulas. Notice that the intermediate formulas consisting of the postcondition of `P` (i.e. `w`), the history formula of `P` (i.e. `H`), and the hisotry formula of `Q` (i.e. `G`), are not contained in the final specification for `P; Q`. As a result, we need to instantiate these formulas when applying this rule in the proof process.

lemma SequentialRule : {p} P {w; H} ⇒{w} Q {q; G} ⇒
∀ h m n. (H [^] G [[→]] M) h m n ⇒{p} P; Q {q; M}

Communication and Parallel Composition. HHL [11] is not compositional with respect to parallel composition, due to the communications between processes and the complex interactions between discrete computation and continuous evolution. The HHL classifies parallel composition into three cases, which are specified by the following three rules respectively.

lemma Parallel1Rule : chanset P = {} ∧ chanset Q = {} ⇒{pp} P {qp; Hp}
⇒ {pq} Q {qq; Hq} ⇒{pp, pq} P||Q {qp, qq; Hp, Hq}

Lemma `Parallel1Rule` says that, when there is no communication event in both `P` and `Q`, the specification of `P||Q` can be copied from the ones of `P` and `Q` accordingly.

lemma CommunicationRule : {px, py} (P || Q) {qx, qy; Hx, Hy}
⇒ ∀ s. ((qx [→] substF ([(RVar x, e)], rx)) [∧] (qy [→] ry)) s
⇒ ∀ h n m. ((Hx[^](elE 0 [[]] almost qx) [[→]] Gx)
[[∧]] (Hy[^](elE 0 [[]]almost qy) [[→]] Gy)) h n m
⇒{px, py} P;Cm (ch??RVar x)||Q; Cm (ch!!e) {rx, ry; Gx, Gy}

where `Cm ch??RVar x` and `Cm ch!!e` implement $ch?x$ and $ch!e$ in HCSP respectively. Lemma `CommunicationRule` defines the case when a communication follows, no matter whether `P` or `Q` contains communication events or not. For such case, we need to synchronize the execution time till the occurrence of `ch??RVar x` and

`ch!!e`. For example, indicated by line 3, if `P` terminates before `Q`, then the input event needs to wait till `Q` terminates, and during the waiting time, the postcondition of `P`, i.e. q^x, always holds. Notice that `elE 0` is included for the case when `P` and `Q` terminate simultaneously. As soon as both parties of the communication are ready, the communication completes like an assignment assigning `e` to real variable `RVar x` (indicated by line 2).

lemma Parallel2Rule : {pp, pq} P||Q {qp, qq; Hp, Hq} ⇒
chanset P ≠ {} ∧ chanset Q ≠ {} ∧ chanset U = {} ∧ chanset V = {}
⇒ {qp} U {qu; Hu} ⇒{qq} V {qv; Hv}
⇒ {pp, pq} P; U||Q; V {qu, qv; Hp [^] Hu, Hq [^] Hv}

Lemma `Parallel2Rule` defines the remaining case when processes containing no communication event follow, provided that `P` and `Q` contain communications (the contrary case when `P` and `Q` do not contain communications can be reduced to the first rule). Indicated by this rule, the parallel composition is equal to executing `U` and `V` immediately from the terminating states of `P` and `Q` respectively.

Repetition. As shown by Lemma `RepetitionRule`, `Inv` is a loop invariant for P^*: the precondition `P` implies `Inv`, `Inv` gurantees the postcondition `q`, and `Inv` is preserved by one round execution of `P` (line 1); and `H` is idempotent with respect to chop (line 2). The final specification for P^* is reduced to an invariant generation problem, similar to continuous evolution.

lemma RepetitionRule: ∀ s. ((p [→] Inv) [∧] (Inv [→] q)) s ⇒{Inv} P {Inv; H}
⇒ ∀h n m. (H[^]H [[→]] H) h n m ⇒{p} P*&&Inv {q; H}

The general rules that are applicable for all HCSP constructs, like the consequene rule, the case analysis rule, and so on, can be defined as in traditional Hoare Logic. Here we will not list them all.

At the end, all the lemmas corresponding to the inference rules of HHL together constitute a verification condition generator of HHL prover for proving HCSP specifications. The proof is performed according to the following process: first, by applying the lemmas of HHL, a HCSP specification is transformed step by step to a set of HOL formulas, i.e. *verification conditions*; and then, by applying proof tactics and rules of HOL, the validity of verification conditions, that is equivalent to the correctness of the original HCSP specification, is proved.

However, when the specification to be proved contains unknown differential invariants or loop invariants, some verification conditions related to the invariants cannot be proved using HOL rules. In order to solve invariant-related constraints, we have implemented an invariant generator based on the techniques proposed in [12]. By defining an oracle `inv_oracle` in Isabelle/HOL to call the external invariant generator, HHL prover is able to prove the remaining invariant-related verification conditions. By now, the modelling and verification in HHL prover is completed.

5 HHL Prover: Deep Embedding

Different from the shallow embedding, the deep embedding defines the DC and FOL formulas by new datatypes and the meanings of them by the corresponding deductive rules. We present HHL prover in the deep embedding next.

5.1 Assertion Languages

FOL. The deep embedding of FOL includes the definitions of the syntax and the deductive system. FOL formulas are constructed from expressions by using relational operators for atomic cases, and inductively from sub-formulas by using logical connectives for the compound cases. In syntax, the formulas can be represented by the following datatype `fform`:

datatype fform = [False] | exp [=] exp | exp [<] exp
| [¬] fform | fform [∨] fform | [∀] *string* fform

The other logical connectives including [∧], [→], and [∃] can be derived as traditional. As seen from type `exp`, a string may correspond to three different variables, depending on the actual type construct (that can be `RVar`, `SVar`, or `BVar`). For quantified formula [∀]*string* `fform`, we assume by default that the name represented by a string `s` corresponds to the real variable occurring in `fform`, i.e. `RVar s`. Thus, we only consider the quantification over *real* variables here, but this restriction can be loosen by considering quantified variables of the other two types (i.e. *string* and *bool*) without any essential difficulty.

The semantics of FOL formulas is defined by induction on the constructs. Given a state `s` and a formula `P` of type `fform`, function `evalF(s, p)` is defined to return the truth value of `P` under state `s`. We then have `s |= p` iff `evalF (s, p)`.

We define the deductive system for `fform` in sequent calculus style. The Isabelle library includes the pre-defined theory `LK0` for a sequent calculus system of classical FOL with equation. For instances, the following two axioms define the introduction/elimination rules for conjunction in sequent calculus style:

conjR: [| $H⊢ $E, P, $F; $H⊢ $E, Q, $ |] ⇒$H⊢ $E, P [&] Q, $F
conjL: $H, P, Q, $G ⊢$E ⇒$H, P [&] Q, $G ⊢$E

where we represent a sequence of FOL formulas by putting a $ symbol before a capital letter, e.g. `$H`. We define the sequent calculus for `fform` (denoted by `DLK0`) based on `LK0` directly, but `LK0` is not complete because it does not include the rules for the arithmetic formulas for reals in `fform`, e.g. the arithmetic laws. In order to solve this problem, we combine deep embedding of defining explicit formulas in syntax and shallow embedding of applying the arithmetic proof tactics of Isabelle. The main step is to define an equivalent conversion between the validity of formulas of `fform` and HOL formulas, `formT (f :,: fform)`⇔⊢`f`, where the recursive function `formT` transforms a formula of type `fform` to a corresponding HOL formula. For instance, to prove the commutative law of `[*]`, we first apply `formT` to the corresponding formula,

formT (RVar x [∗] Rvar y [=] RVar y [∗] Rvar x)
= (rvar (x) ∗ rvar (y) = rvar (y) ∗ rvar (x))

where `rvar` is a constant function that maps a real variable expression to a real value. The HOL formula obtained after the conversion can be proved automatically by applying `auto` directly.

As shown above, when we prove a `fform` formula involving arithmetic, we will convert it equivalently to a HOL formula and then prove the HOL formula instead. However, to prove a `fform` formula without arithmetic occurring in it, two options are provided to users: applying FOL rules defined in `DLK0`, or converting the formula to HOL and applying HOL rules.

DC. To embed DC in deep style, we first define a datatype `dexp` to represent temporal expressions:

datatype dexp = ℓ | DR *real* | dexp [[+]] dexp | dexp [[−]] dexp | dexp [[∗]] dexp

`dexp` defines expressions that are interval-dependent, including the only temporal variable ℓ for representing the length of the considered interval, real constants, and arithmetic expressions. Then the datatype `dform` encodes the history formulas *HF*:

datatype dform = [[True]] | dexp[[=]]dexp | dexp[[<]]dexp
| almost fform | dform[^]dform [[¬]]dform | dform[[∨]]dform

The semantics of the history formulas is defined by induction on the constructs. Given a flow `f`, a timed interval `[c, d]`, and a temporal expression `te`, function `ievalE(f, te, c, d)` is defined to evaluate `te` under flow `f` and interval `[c, d]`; and based on this function, given a history formula `H`, function `ievalF(f, H, c, d)` is defined to return the truth value of `H` under flow `f` and interval `[c, d]`. Below show some examples:

ievalE(f, ℓ, c, d) = d−c
ievalF (f, almost S, c, d) = (c<d ∧ ∀ i, j. c≤ i < j ≤ d →
$\exists t$. i < t < j → evalF (f(t), S))
ievalF (f, H1[^]H2, c, d) = ∃ k.c<=k ∧ k<=d ∧ ievalF (f, H1, c, k)
∧ ievalF (f, H2, k, d)

We then have `h, [n, m] |= H` iff `ievalF (h, H, n, m)`.

To establish the sequent calculus style deductive system for `dform`, we first define the deductive system for the first-order connectives of `dform`, that is similar to the one built for `fform` above; then for ℓ, `[^]` and `almost`, we transform the deductive system of DC from [19] to sequent calculus style. For instance, the axiom $\lceil S\rceil^\frown\lceil S\rceil \leftrightarrow \lceil S\rceil$ is encoded by the following rules:

AlR : $H ⊢ (almost S[^]almost S), $E ⇒ $H ⊢ almost S, $E
AlL : $H, (almost S[^]almost S) ⊢ $E ⇒ $H, almost S ⊢ $E

where `$H`, `$E` represent arbitrary sequences of logical formulas of type `dform`. Other rules can be encoded similarly.

Inference Rules. We list the rule for assignment as an illustration. Because of the deep embedding of the assertions, the effect of assignment is expressed at the level of `fform` formulas by variable substitution. This is one main difference of using deep embedding from shallow embedding. For defining variable substitution, we implement a map as a list of pairs `(exp * exp) list`, and define two recursive functions: given a map `r` and an expression `e`, function `substE(r, e)` substitutes expressions occurring in `e` according to the map `r`; based on the definition of `substE`, given a formula `p` of type `fform`, function `substF(r, p)` substitutes expressions occurring in `p` according to the map `r`. Below we have the lemma for assignment `e:=f`:

lemma AssignRRule:
⊢ p [→]substF ([(RVar x, f)], q); ⊢elE 0 [[→]] H ⇒{p} RVar x :=f {q; H}

Other rules can be defined similarly. As we can see in deep embedding, the HHL specification is transformed into a set of explicit FOL and DC formulas, which can be proved by applying the corresponding deductive systems we have built.

Discussion. The general strengths and weakness of both embeddings can be found at [16], and here we will not list them again. We will make more specific comparison between the two embeddings in the case study section.

In both embeddings, the proof in HHL prover cannot be automated due to the following reasons: first, the intermediate assertions occurring in SequentialRule, CommunicationRule, *etc*, need to be instantiated in the proof process by the user manually; second, the constraints related to unknown differential invariants and loop invariants need to be gathered manually so that they are solved by the external invariant generator as a whole; finally, in shallow embedding, because of the limitation of SMT solvers, the HOL verification conditions containing quantifiers usually cannot be proved automatically; while in deep embedding, the FOL and DC verification conditions are proved by applying their deductive rules manually.

But on the contrary, compared to other automated provers, HHL prover is capable of modelling and verifying more complex hybrid systems, because of the expressiveness of both HCSP and HHL.

6 A Case Study

We demonstrate the use of HHL prover on proving the safety of the slow descent guidance control program of a lunar lander, which provides a specific sampled-data control system composed of the physical plant and the control program.

6.1 Description of the Control Program

The lunar lander's dynamics is mathematically represented by

$$\begin{cases} \dot{r} &= v \\ \dot{v} &= \frac{F_c}{m} - gM \\ \dot{m} &= -\frac{F_c}{Isp_1} \end{cases} \quad , \text{ where} \tag{1}$$

- r, v and m denote the altitude relative to lunar surface, vertical velocity, and mass of the lunar lander, respectively;
- F_c is the thrust imposed on the lander, which is a constant in each sampling period of length 0.128 s;
- $gM = 1.622\,\mathrm{m/s^2}$ is the magnitude of the gravitational acceleration on the moon;
- Isp is the *specific impulse* of the lander's thrust engine. When F_c lies in $[1500, 3000]$, $Isp = 2548\mathrm{N{\cdot}s/kg}$, and when F_c lies in $(3000, 5000]$, $Isp = 2842\mathrm{N{\cdot}s/kg}$. Thus the lander's dynamics comprises two different forms depending on the values of Isp.

The sample time of the guidance control program is fixed as 0.128s. In every period, the guidance program gets the values of the altitude r and the velocity v via the sensor, and then updates mass m, calculates acceleration aIC, and calculates thrust F_c in sequence. Especially, F_c is calculated according to

$$F_c := -0.01 \cdot (F_c - m \cdot gM) - 0.6 \cdot (v - vslw) \cdot m + m \cdot gM \tag{2}$$

where F_c on the right is the thrust of last period, and m is the updated mass in this period. The new thrust F_c will then be used for the next sampling cycle.

The safety property we want to prove for the guidance program is

(SP) $|v - vslw| \leq \varepsilon$, where $\varepsilon = 0.05\,\mathrm{m/s}$ is the tolerance of fluctuation of v around the target $vslw = -2\,\mathrm{m/s}$.

6.2 Verification in HHL Prover

First, we construct the HCSP model for the control program manually, denoted by `LL`, which is

definition P :: proc **where**
LL ≡ PC_Init; PD_Init; t:=(Con Real 0);(PC_Difff; t:=(Con Real 0); PD_Rep)∗

where `PC_Init` and `PD_Init` are initialization procedures for the continuous dynamics and the guidance program respectively; `PC_Diff` models the continuous dynamics given by (1) within a period of 0.128s; `PD_Rep` calculates thrust F_c according to (2) for the next sampling cycle; variable `t` denotes the elapsed time in each sampling cycle. Hence, process `LL` is initialized at the beginning by `PC_Init` and `PD_Init`, and behaves as a repetition of dynamics `PC_Diff` and computation `PD_Rep` afterwards.

Proof Result. By applying HHL prover (either in shallow or deep embedding), we have proved the following specification for process `LL`:

lemma goal: {fTrue} LL {safeProp; (elE 0 [[|]] almost safeProp)}

where `safeProp` of type `fform` encodes the safety property (**SP**). Lemma `goal` indicates that, starting from any state, the control program satisfies the safety property almost everywhere during the whole execution.

Comparison in Different Approaches. In both embeddings, the proof for lemma `goal` is composed of a sequence of rule applications of Isabelle/HOL. But the length of the proof in shallow embedding is about one half of the one in deep embedding. In detail,

- For shallow embedding, the rules applied mainly comprise of two kinds: the inference rules of HHL and the rules for unfolding the HOL predicates of FOL and DC formulas. Fortunately, many of the rules applied are found by the built-in tool `sledgehammer` of Isabelle/HOL automatically. This alleviates users' proof burden to a big extent.
- For deep embedding, the rules applied also comprise of two kinds: the deductive rules of FOL and DC. The verification conditions generated (in the form of FOL and DC) have a much smaller size than the ones (in the form of HOL) in shallow embedding, because they are not unfolded. But meanwhile, they need to be conducted by the user completely, to apply the deductive rules of both logic manually.

7 Conclusion

HHL prover can be used for verifying hybrid systems, that combine discrete computation, continuous dynamics, communications, and parallel composition, *etc.* As an interactive theorem prover, it formalizes HCSP for modelling hybrid systems and realises the Hybrid Hoare Logic (HHL) for verifying safety of HCSP models in Isabelle/HOL. The old HHL prover implemented HHL in deep embedding, but with great proof burden. This paper presents an improved HHL prover that implements HHL in shallow embedding. In addition, to compare the two different embedding styles, we demonstrated the use of both variants on a real-life example, i.e. the slow descent control program of a lunar lander. It can be seen from the proof results that the shallow embedding has better performance in the proof size and automation than deep embedding.

Acknowledgements. This paper is supported partly by "973 Program" under grant No. 2014CB340701, by NSFC under grants 91118007 and 91418204, by CDZ project CAP (GZ 1023), and by the CAS/SAFEA International Partnership Program for Creative Research Teams.

References

1. Alur, R.: Formal verification of hybrid systems. In: EMSOFT 2011, pp. 273–278 (2011)
2. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.-H.: Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In: Grossman, R.L., Ravn, A.P., Rischel, H., Nerode, A. (eds.) HS 1991 and HS 1992. LNCS, vol. 736, pp. 209–229. Springer, Heidelberg (1993)
3. Alur, R., Dang, T., Ivančić, F.: Predicate abstraction for reachability analysis of hybrid systems. ACM Trans. Embed. Comput. Syst. **5**(1), 152–199 (2006)

4. Asarin, E., Bournez, O., Dang, T., Maler, O.: Approximate reachability analysis of piecewise-linear dynamical systems. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, p. 20. Springer, Heidelberg (2000)
5. Asarin, E., Dang, T., Maler, O.: The d/dt tool for verification of hybrid systems. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, p. 365. Springer, Heidelberg (2002)
6. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: an analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 258–263. Springer, Heidelberg (2013)
7. Eggers, A., Ramdani, N., Nedialkov, N., Fränzle, M.: Improving SAT modulo ODE for hybrid systems analysis by combining different enclosure methods. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 172–187. Springer, Heidelberg (2011)
8. Frehse, G.: PHAVer: algorithmic verification of hybrid systems past hytech. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 258–273. Springer, Heidelberg (2005)
9. He, J. : From CSP to hybrid systems. In: A Classical Mind, Essays in Honour of C.A.R. Hoare, pp. 171–189. Prentice Hall International (UK) Ltd. (1994)
10. Henzinger, T.A.: The theory of hybrid automata. In: Inan, M.K., Kurshan, R.P. (eds.) LICS'1996. NATO ASI Series, vol. 170, pp. 278–292. Springer, Heidelberg (1996)
11. Liu, J., Lv, J., Quan, Z., Zhan, N., Zhao, H., Zhou, C., Zou, L.: A calculus for hybrid CSP. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 1–15. Springer, Heidelberg (2010)
12. Liu, J., Zhan, N., Zhao, H.: Computing semi-algebraic invariants for polynomial dynamical systems. In: EMSOFT 2011, pp. 97–106 (2011)
13. Manna, Z., Pnueli, A.: Verifying hybrid systems. In: Grossman, R.L., Ravn, A.P., Rischel, H., Nerode, A. (eds.) HS 1991 and HS 1992. LNCS, vol. 736, pp. 4–35. Springer, Heidelberg (1993)
14. Platzer, A., Clarke, E.M.: Computing differential invariants of hybrid systems as fixedpoints. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 176–189. Springer, Heidelberg (2008)
15. Platzer, A., Quesel, J.-D.: KeYmaera: a hybrid theorem prover for hybrid systems (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 171–178. Springer, Heidelberg (2008)
16. Wildmoser, M., Nipkow, T.: Certifying machine code safety: shallow versus deep embedding. In: Slind, K., Bunker, A., Gopalakrishnan, G.C. (eds.) TPHOLs 2004. LNCS, vol. 3223, pp. 305–320. Springer, Heidelberg (2004)
17. Zhan, N., Wang, S., Zhao, H.: Formal modelling, analysis and verification of hybrid systems. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Unifying Theories of Programming and Formal Engineering Methods. LNCS, vol. 8050, pp. 207–281. Springer, Heidelberg (2013)
18. Zhao, H., Yang, M., Zhan, N., Gu, B., Zou, L., Chen, Y.: Formal verification of a descent guidance control program of a lunar lander. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 733–748. Springer, Heidelberg (2014)
19. Zhou, C., Hansen, M.R.: Duration Calculus – A Formal Approach to Real-Time Systems. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2004)
20. Zhou, C., Hoare, C.A.R., Ravn, A.P.: A calculus of durations. Inf. Process. Lett. **40**(5), 269–276 (1991)

21. Chaochen, Z., Ji, W., Ravn, A.P.: A formal description of hybrid systems. In: Alur, R., Sontag, E.D., Henzinger, T.A. (eds.) HS 1995. LNCS, vol. 1066, pp. 511–530. Springer, Heidelberg (1996)
22. Zou, L., Lv, J., Wang, S., Zhan, N., Tang, T., Yuan, L., Liu, Y.: Verifying Chinese train control system under a combined scenario by theorem proving. In: Cohen, E., Rybalchenko, A. (eds.) VSTTE 2013. LNCS, vol. 8164, pp. 262–280. Springer, Heidelberg (2014)
23. Zou, L., Zhan, N., Wang, S., Fränzle, M.: Formal verification of simulink/stateflow diagrams. In: ATVA 2015 (2015) (to appear)
24. Zou, L., Zhan, N., Wang, S., Fränzle, M., Qin, S.: Verifying simulink diagrams via a hybrid hoare logic prover. In: EMSOFT 2013, pp. 1–10 (2013)