HHL Prover : An Improved Interactive Theorem Prover for Hybrid Systems

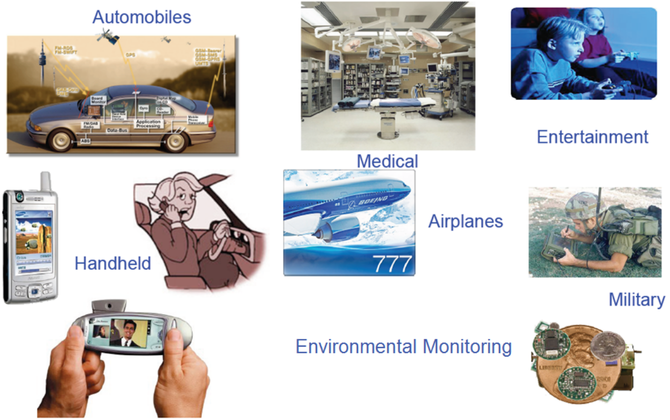
Shuling Wang, Naijun Zhan, and Liang Zou

State Key Lab. of Computer Science, Institute of Software, Chinese Academy of Sciences

Paris, November 2015

Hybrid Systems

Hybrid systems exhibit combinations of discrete jumps and continuous evolution, many of which are safety-critical.



Verification Approaches

- Reachability analysis and model checking
  - Hybrid automata + temporal logic based specification languages + model checkers ;
  - Abstractions or numerical approximations.

Verification Approaches

- Reachability analysis and model checking
  - Hybrid automata + temporal logic based specification languages + model checkers ;
  - Abstractions or numerical approximations.
- Deductive verification
  - A formal modelling language with (de-)compositionality and a specification logic for verifying the corresponding models ;
  - The differential invariants are the key for verifying differential equations.

Related Work

- Reachability analysis and model checking
  - d/dt [Asarin, Bournez, *et al.*], reachability analysis of hybrid systems with linear continuous dynamics and uncertain bounded input ;
  - iSAT-ODE [Eggers, Ramdani, *et al.*], a numerical SMT solver based on interval arithmetic that conducts bounded model checking ;
  - Flow* [Chen, Ábrahám, *et al.*], computing the over-approximations of the reachable sets of hybrid systems in a bounded time ;
  - ...

Related Work

- Reachability analysis and model checking
  - d/dt [Asarin, Bournez, *et al.*], reachability analysis of hybrid systems with linear continuous dynamics and uncertain bounded input ;
  - iSAT-ODE [Eggers, Ramdani, *et al.*], a numerical SMT solver based on interval arithmetic that conducts bounded model checking ;
  - Flow* [Chen, Ábrahám, *et al.*], computing the over-approximations of the reachable sets of hybrid systems in a bounded time ;
  - ...
- Deductive verification
  - KeYmaera [Platzer, Quesel, *et al.*], a theorem prover of hybrid systems using differential dynamic logic.

Contributions of this Work

An interactive theorem prover, called HHL prover, for verifying hybrid systems that are

- modelled by hybrid CSP (HCSP), an extension of CSP [Hoare] with differential equations for describing hybrid systems, and
- specified by hybrid Hoare Logic (HHL), an extension of Hoare logic with history formulas for reasoning about HCSP models.

Contributions of this Work

An interactive theorem prover, called HHL prover, for verifying hybrid systems that are

- modelled by hybrid CSP (HCSP), an extension of CSP [Hoare] with differential equations for describing hybrid systems, and
- specified by hybrid Hoare Logic (HHL), an extension of Hoare logic with history formulas for reasoning about HCSP models.

HHL prover is based on the mechanization of HCSP and HHL in the proof assistant Isabelle/HOL.

Contributions of this Work

This is an improved HHL prover of a previous version [Zou, Wang, Zhan, 2013] :

- HHL verification framework : shallow embedding in Isabelle/HOL, reducing the proof effort to a big extent ;
- We re-verify a real-world example : the slow descent guidance control program of a lunar lander, as an illustration.

Outline

Hybrid Systems : Hybrid CSP

Hybrid CSP (HCSP) [He&Zhou, 1994] is an extension of CSP by introducing timing constructs, continuous evolution and interrupts.

Hybrid Systems : Hybrid CSP

Hybrid CSP (HCSP) [He&Zhou, 1994] is an extension of CSP by introducing timing constructs, continuous evolution and interrupts.

HCSP inherits from CSP the communication-based parallelism :

- Message-passing by $ch!e$ and $ch?x$.
- A communication is a synchronisation of $ch!e$ and $ch?x$.
- The parallel composition $P||Q$ behaves as if P and Q run independently except that the communications along the common channels connecting P and Q are to be synchronized.

Hybrid Systems : Hybrid CSP

Continuous evolution $\langle \dot{s} = e \& B \rangle$:

- It evolves according to $\dot{s} = e$ as long as B holds, and terminates when B turns false.
- $\mathrm{wait}\ d \ \widehat{=}\ t := 0; \langle \dot{t} = 1 \& t \leq d \rangle$.

Hybrid Systems : Hybrid CSP

Communication interruption $\langle \dot{s} = e \& B \rangle \trianglerighteq []_{i \in I}(io_i \rightarrow Q_i)$:

- io_i - a set of communication events (i.e. $ch!e$ or $ch?x$) ;
- It initially proceeds like the continuous evolution $\dot{s} = e$, and is interrupted on some communication in io_i, and then proceeds like Q_i.

Hybrid Systems : Hybrid CSP

Communication interruption $\langle \dot{s} = e \& B \rangle \trianglerighteq []_{i \in I}(io_i \to Q_i)$:

- io_i - a set of communication events (i.e. $ch!e$ or $ch?x$) ;
- It initially proceeds like the continuous evolution $\dot{s} = e$, and is interrupted on some communication in io_i, and then proceeds like Q_i.

Composite constructs are taken originally from CSP :

- $P; Q$, sequential composition ;
- $B \to P$, conditional ;
- P^*, repetition ;
- $P \sqcup Q$, non-deterministic choice.

HCSP : An Example

A description of a continuously evolving plant with discrete control :

$$
\begin{array}{ll}
 & (\langle \dot{x} = f(x,u) \& True \rangle \unrhd [] (sensor!x \rightarrow actuator?u))^* \\
\| & (\mathrm{wait}\ d; sensor?s; actuator!Comp(s))^*
\end{array}
$$

Hybrid Hoare Logic (HHL)

Hybrid Hoare Logic (HHL) for deductive verification of HCSP.

- A specification for a process P: $\{Pre\}\ P\ \{Post;\ HF\}$
  - *Pre*, *Post* - pre-/post-conditions, in first-order logic (FOL) ;
  - *HF* - history formula, in the interval-based Duration Calculus (DC).

Hybrid Hoare Logic (HHL)

Hybrid Hoare Logic (HHL) for deductive verification of HCSP.

- A specification for a process P : $\{Pre\}\ P\ \{Post;\ HF\}$
  - Pre, $Post$ - pre-/post-conditions, in first-order logic (FOL) ;
  - HF - history formula, in the interval-based Duration Calculus (DC).
- The HCSP constructs are axiomatized in HHL by a set of inference rules.
  - The inference system constitutes the basis for the verification condition generator of HHL prover.

HHL prover

Two ways to embed the whole HHL verification framework in Isabelle/HOL :

- Shallow embedding
  - It defines the assertions of HHL (i.e. FOL and DC formulas) by HOL predicates on process states ;
- Deep embedding [Zou, Wang, and Zhan, 2013]
  - It defines the FOL and DC assertions as new datatypes, and defines the meanings of the datatypes by the deductive rules (of FOL and DC resp.).

In this paper, the first approach is adopted.

HHL prover : HCSP Syntax Encoding

HCSP expressions by datatype `exp`, and HCSP processes by datatype `proc`.

HHL prover : HCSP Syntax Encoding

HCSP expressions by datatype `exp`, and HCSP processes by datatype `proc`.

Each construct of `proc` corresponds to the one in HCSP syntax :

HHL prover : HCSP Syntax Encoding

HCSP expressions by datatype `exp`, and HCSP processes by datatype `proc`.

Each construct of `proc` corresponds to the one in HCSP syntax :

- $B \rightarrow P$ is encoded as `IF B P`;

HHL prover : HCSP Syntax Encoding

HCSP expressions by datatype `exp`, and HCSP processes by datatype `proc`.

Each construct of `proc` corresponds to the one in HCSP syntax :

- $B \rightarrow P$ is encoded as `IF B P`;
- $\langle \dot{s} = e \& B \rangle$ is encoded as `<s :e&&Inv&B>`, where `Inv` is the differential invariant of the differential equation $\dot{s} = e$;

HHL prover : HCSP Syntax Encoding

HCSP expressions by datatype `exp`, and HCSP processes by datatype `proc`.

Each construct of `proc` corresponds to the one in HCSP syntax :

- $B \to P$ is encoded as `IF B P`;
- $\langle \dot{s} = e \& B \rangle$ is encoded as `<s :e&&Inv&B>`, where `Inv` is the differential invariant of the differential equation $\dot{s} = e$;
- P^* is encoded as `P*&&Inv`, where `Inv` is the loop invariant;

HHL prover : HCSP Syntax Encoding

HCSP expressions by datatype `exp`, and HCSP processes by datatype `proc`.

Each construct of `proc` corresponds to the one in HCSP syntax :

- $B \to P$ is encoded as `IF B P` ;
- $\langle \dot{s} = e \& B \rangle$ is encoded as `<s :e&&Inv&B>`, where `Inv` is the differential invariant of the differential equation $\dot{s} = e$;
- P^* is encoded as `P*&&Inv`, where `Inv` is the loop invariant ;
  - Both invariants are annotated for the purpose of verification.

HHL prover : HCSP Syntax Encoding

HCSP expressions by datatype `exp`, and HCSP processes by datatype `proc`.

Each construct of `proc` corresponds to the one in HCSP syntax :

- $B \rightarrow P$ is encoded as `IF B P`;
- $\langle \dot{s} = e \& B \rangle$ is encoded as `<s :e&&Inv&B>`, where `Inv` is the differential invariant of the differential equation $\dot{s} = e$;
- P^* is encoded as `P*&&Inv`, where `Inv` is the loop invariant;
  - Both invariants are annotated for the purpose of verification.
- $\cdots$

Assertion Languages : FOL

Semantic functions :

- `state` - functions from variables to values, and
- `flow` - functions from time to states (recording the execution interval)

Assertion Languages : FOL

Semantic functions :

- `state` - functions from variables to values, and
- `flow` - functions from time to states (recording the execution interval)

The FOL formulas `fform` are defined as predicates on states,

type_synonym fform = state $\Rightarrow$ *bool*

Assertion Languages : FOL

Semantic functions :

- `state` - functions from variables to values, and
- `flow` - functions from time to states (recording the execution interval)

The FOL formulas `fform` are defined as predicates on states,

```
type_synonym fform = state ⇒ bool
```

The FOL constructs can be derived as special Isabelle functions of type `fform`, e.g.

```
definition fEqual :: "exp ⇒ exp ⇒ fform"  ("[=]")  where
 e [=] f ≡ λs. evalE e s = evalE f s
```

Assertion Languages : DC

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

type_synonym dform = flow $\Rightarrow$ *real* $\Rightarrow$ *real* $\Rightarrow$ *bool*

Assertion Languages : DC

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

type_synonym dform = flow $\Rightarrow$ *real* $\Rightarrow$ *real* $\Rightarrow$ *bool*

The main operators of DC include :

- `elE T` : the length of the interval is T,

  definition elE :: *real* $\Rightarrow$ dform **where**
  elE T $\equiv \lambda$ h n m. (m $-$ n) = T

Assertion Languages : DC

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

type_synonym dform = flow $\Rightarrow$ *real* $\Rightarrow$ *real* $\Rightarrow$ *bool*

The main operators of DC include :

- `elE T` : the length of the interval is T,

  definition elE :: *real* $\Rightarrow$ dform **where**
  elE T $\equiv$ λ h n m. (m $-$ n) = T

- `almost p` : p holds almost everywhere in the interval,

  definition almost :: fform $\Rightarrow$ dform **where**
  almost p $\equiv$ λ h n m. (m>n) $\wedge$ ($\forall$a$\geq$n. $\forall$ b$\leq$m. a < b $\rightarrow$
  $\exists$t. t>a $\wedge$ t<b $\wedge$ p(h(t))))

Assertion Languages : DC

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

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

The main operators of DC include :

- elE T : the length of the interval is T,

  definition elE :: *real* ⇒ dform **where**
  elE T ≡ λ h n m. (m − n) = T

- almost p : p holds almost everywhere in the interval,

  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))))

- H[^]M : the interval can be separated into two sub-intervals such that H and M hold on them resp,

  definition chop :: dform ⇒ dform ⇒ dform ("_[^]_") **where**
  H[^]M ≡ λ h n m. (∃ nm. (nm ≥ n ∧ nm ≤ m ∧ H h n nm ∧ M h nm m))

Assertion Languages : FOL and DC in Deep Emb.

In deep embedding, both FOL and DC formulas are constructed in the syntax level step by step from the bottom-most expressions.

The datatype `fform` encodes the FOL formulas :

datatype fform = [False] | exp $[=]$ exp | exp $[<]$ exp
| $[\neg]$ fform | fform $[\vee]$ fform | $[\forall]$ *string* fform

The datatype `dform` encodes the DC formulas :

datatype dform = [[True]] | dexp$[[=]]$dexp | dexp$[[<]]$dexp
| almost fform | dform[^]dform $[[\neg]]$dform | dform$[[\vee]]$dform

As new datatypes, the deductive systems of FOL and DC are defined as axioms for the reasoning of FOL and DC formulas.

Specification and Inference Rules

ValidS p c q H, represents a valid specification $\{p\}\, c\, \{q; H\}$:

ValidS p c q H $\equiv$ $\forall$ now h now' h' . semB c now h now' h' $\rightarrow$ h(now) |= p
$\rightarrow$ (h'(now') |= q $\wedge$ h', [now, now'] |= H)

- semB c now h now' h', representing the big-step semantics, c starts execution from the initial flow h and time now, and terminates with flow h' and time now' ;
- the precondition p holds under h and now, implies that the postcondition q and the history formula H hold under h' and now'.

Inference Rule : Continuous Evolution

In Isabelle/HOL, we have proved a set of lemmas stating that all the inference rules of HHL are valid.

Inference Rule : Continuous Evolution

In Isabelle/HOL, we have proved a set of lemmas stating that all the inference rules of HHL are valid.

```
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}
```

- Inv is indeed a sufficiently strong invariant (lines 1-3).

Inference Rule : Continuous Evolution

In Isabelle/HOL, we have proved a set of lemmas stating that all the inference rules of HHL are valid.

```
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}
```

- Inv is indeed a sufficiently strong invariant (lines 1-3).
- H is implied by the strongest history formula (line 4).

Inference Rule : Continuous Evolution

- Eventually, the proof of the continuous evolution is reduced to an equivalent differential invariant generation problem (a set of constraints w.r.t. `Inv`).

Inference Rule : Continuous Evolution

- Eventually, the proof of the continuous evolution is reduced to an equivalent differential invariant generation problem (a set of constraints w.r.t. `Inv`).
- HHL prover will call an external invariant generator, an oracle `inv_oracle_SOS` in Isabelle/HOL, to solve the invariant generation problem.

Proof Process

All the inference rules of HHL together constitute a verification condition generator of HHL prover for proving HCSP specifications.

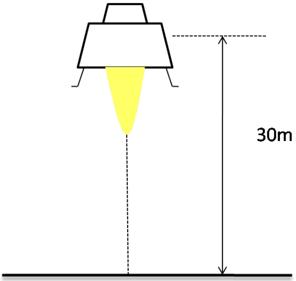
Proof Process

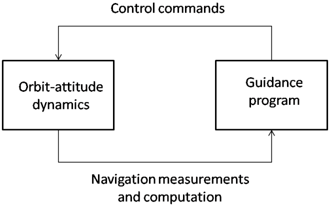
All the inference rules of HHL together constitute a verification condition generator of HHL prover for proving HCSP specifications.

Proof Process

- first, by applying the verification condition generator, an HCSP specification is transformed step by step to a set of logical formulas (FOL and DC formulas) ;
- then, by applying the proof tactics and rules of HOL, the validity of these logical formulas, which is equivalent to the correctness of the original HCSP specification, is proved.

Case Study : a Lunar Lander





Description of the Dynamics

The lunar lander's dynamics is mathematically represented by

$$\begin{cases} \dot{r} & = & v \\ \dot{v} & = & \frac{F_c}{m} - gM \\ \dot{m} & = & -\frac{F_c}{Isp_i} \end{cases}$$

Description of the Control Program

- Sample time : 0.128s.
- In every period, the guidance program
  - reads r and v via the sensor,
  - updates m, and calculates F_c.

The new thrust F_c will then be used for the next sampling cycle.

Description of the Control Program

- Sample time : 0.128s.
- In every period, the guidance program
  - reads r and v via the sensor,
  - updates m, and calculates F_c.

  The new thrust F_c will then be used for the next sampling cycle.

The safety property to be proved is :

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

Modelling and Verification in HHL Prover

First, the HCSP model for the control program is constructed :

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

Modelling and Verification in HHL Prover

First, the HCSP model for the control program is constructed :

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

By applying HHL prover, we have proved the following lemma :

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

which indicates that, starting from any state, the control program satisfies the safety property almost everywhere during the whole execution.

Comparison : Shallow vs. Deep

- Both the proofs for the case study are composed of a sequence of rule applications of Isabelle/HOL.

1. a certified integration of third-party automated theorem provers and SMT solvers including Alt-Ergo, Z3, CVC3, and etc..

Comparison : Shallow vs. Deep

- Both the proofs for the case study are composed of a sequence of rule applications of Isabelle/HOL.
- The proof length in shallow embedding (about 200 lines) is about one half of the one in deep embedding.

1. a certified integration of third-party automated theorem provers and SMT solvers including Alt-Ergo, Z3, CVC3, and etc..

Comparison : Shallow vs. Deep

- Both the proofs for the case study are composed of a sequence of rule applications of Isabelle/HOL.
- The proof length in shallow embedding (about 200 lines) is about one half of the one in deep embedding.
- Shallow embedding
  - The rules applied mainly comprise of two kinds : the inference rules of HHL, and the rules for unfolding the HOL predicates defining FOL and DC formulas.
  - Most proofs for deciding validity of formulas can be found by the built-in tool `sledgehammer` [1] of Isabelle/HOL automatically.
- Deep embedding
  - The rules applied comprise of two kinds : the inference rules of HHL, and the deductive rules of FOL and DC.
  - The proof needs to be conducted by the user completely, to apply the deductive rules of both logic manually.

1. a certified integration of third-party automated theorem provers and SMT solvers including Alt-Ergo, Z3, CVC3, and etc..

Remarks on Automation

In both embeddings, the proof in HHL prover cannot be fully automatic :

Remarks on Automation

In both embeddings, the proof in HHL prover cannot be fully automatic :

- The intermediate assertions in the `SequentialRule`, `CommunicationRule`, etc, need to be instantiated in the proof process by the user manually ;

Remarks on Automation

In both embeddings, the proof in HHL prover cannot be fully automatic :

- The intermediate assertions in the `SequentialRule`, `CommunicationRule`, etc, need to be instantiated in the proof process by the user manually ;
- The constraints related to unknown differential and loop invariants need to be gathered manually so that they are solved by the external invariant generator as a whole ;

Remarks on Automation

In both embeddings, the proof in HHL prover cannot be fully automatic :

- The intermediate assertions in the `SequentialRule`, `CommunicationRule`, etc, need to be instantiated in the proof process by the user manually ;
- The constraints related to unknown differential and loop invariants need to be gathered manually so that they are solved by the external invariant generator as a whole ;
- In shallow embedding, due to the limitation of SMT solvers, the HOL verification conditions containing quantifiers usually cannot be proved automatically ;

Remarks on Automation

In both embeddings, the proof in HHL prover cannot be fully automatic :

- The intermediate assertions in the `SequentialRule`, `CommunicationRule`, etc, need to be instantiated in the proof process by the user manually ;
- The constraints related to unknown differential and loop invariants need to be gathered manually so that they are solved by the external invariant generator as a whole ;
- In shallow embedding, due to the limitation of SMT solvers, the HOL verification conditions containing quantifiers usually cannot be proved automatically ;
- In deep embedding, the FOL and DC verification conditions are proved by applying their deductive rules manually.

Remarks on Automation

In both embeddings, the proof in HHL prover cannot be fully automatic :

- The intermediate assertions in the `SequentialRule`, `CommunicationRule`, etc, need to be instantiated in the proof process by the user manually ;
- The constraints related to unknown differential and loop invariants need to be gathered manually so that they are solved by the external invariant generator as a whole ;
- In shallow embedding, due to the limitation of SMT solvers, the HOL verification conditions containing quantifiers usually cannot be proved automatically ;
- In deep embedding, the FOL and DC verification conditions are proved by applying their deductive rules manually.

HHL prover is capable of modelling and verifying more complex hybrid systems, because of the expressiveness of both HCSP and HHL.

Concluding Remarks

- We present an improved HHL prover as a shallow embedding of Isabelle/HOL for verifying hybrid systems.

Concluding Remarks

- We present an improved HHL prover as a shallow embedding of Isabelle/HOL for verifying hybrid systems.
- We apply the HHL prover on safety verification of a real-life example, i.e. the slow descent control program of a lunar lander.

Concluding Remarks

- We present an improved HHL prover as a shallow embedding of Isabelle/HOL for verifying hybrid systems.
- We apply the HHL prover on safety verification of a real-life example, i.e. the slow descent control program of a lunar lander.
- The proof results show that the shallow embedding has better performances in the proof size and automation than deep embedding.