MARS: A Toolchain for Modelling, Analysis and Verification of Hybrid Systems*

Mingshuai Chen, Xiao Han, Tao Tang, Shuling Wang, Mengfei Yang, Naijun Zhan, Hengjun Zhao and Liang Zou

Abstract We introduce a toolchain MARS for Modelling, Analyzing and veRifying hybrid Systems we developed in the past years. Using MARS, we build executable models of hybrid systems using the industrial standard environment Simulink/Stateflow, which facilitates analysis by simulation. To complement simulation, formal verification of Simulink/Stateflow models is conducted in the toolchain via the following steps: first, we translate Simulink/Stateflow diagrams to Hybrid CSP (HCSP) processes by an automatic translator Sim2HCSP, where HCSP is an extension of CSP for formally modelling hybrid systems; second, to justify the translation, another automatic translator HCSP2Sim that translates from HCSP to Simulink is provided, so that the consistency between the original Simulink/Stateflow model and the translated HCSP formal model can be checked by co-simulation; then, the HCSP processes obtained in the first step are verified by an interactive Hybrid Hoare Logic (HHL) prover; during the verification, an invariant generator independent of the theorem prover for synthesizing invariants for differential equations and loops is needed. We will demonstrate the toolchain by analysis and verification of a descent guidance control program of a lunar lander, which is a real-world industry example.

Mingshuai Chen, Shuling Wang, Naijun Zhan, and Liang Zou
State Key Lab. of Computer Science, Institute of Software, Chinese Academy of Sciences e-mail: {chenms,wangsl,znj,zoul}@ios.ac.cn

Xiao Han and Tao Tang
State Key Lab. of Rail Traffic Control and Safety, Beijing Jiaotong University

Mengfei Yang
Chinese Academy of Space Technology

Hengjun Zhao
School of Computer and Information Science, Southwest University

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

1 Introduction

Hybrid systems combine discrete controllers and continuous plants, and occur ubiquitously in safety-critical application areas such as transportation and avionics. To guarantee the correctness, formal techniques on modelling and verification of hybrid sytems have been proposed [3, 21, 27, 29]. Besides, as a complementary activity to verification, several approaches have also been proposed for testing such systems [2, 4, 10]. However, the deep interactions between discrete and continuous components, and in addition, the complex continuous dynamics described by (nonlinear) differential equations, make the formal analysis and verification of hybrid sytems extremely difficult. Most existing work mentioned above can only deal with restricted systems, e.g., [3, 21] deal with dynamic and hybrid systems with a decidable reachability problem; [27] considered how to verify hybrid systems using simulation semantics, which cannot guarantee the correctness of hybrid systems in general because of the inherent incompleteness of simulation; while it is difficult to handle communication and parallelism using the approach in [29].

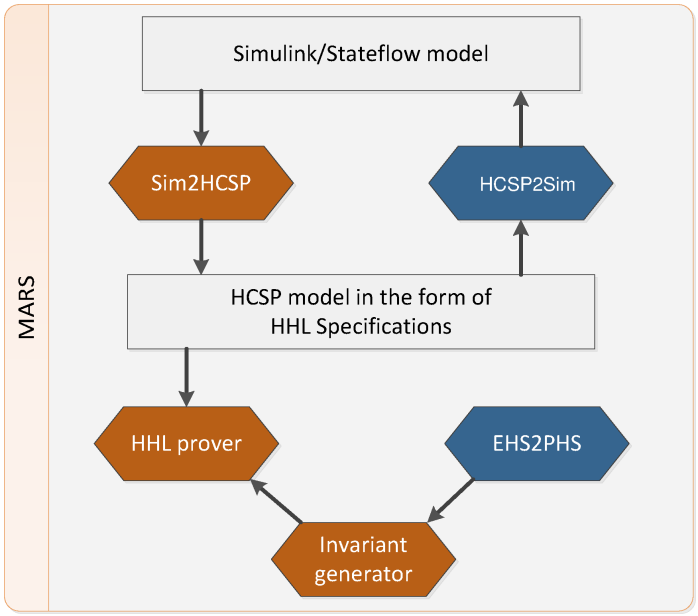

Fig. 1 Verification architecture

To develop reliable complicated hybrid systems, we propose the toolchain MARS for Modelling, Analyzing and veRifing hybrid systems. As shown in Fig. 1, the architecture of MARS is composed of three parts: a translator Sim2HCSP, an HHL prover, and an invariant generator. At the top level, we build executable models

of hybrid systems in the graphical environment Simulink/Stateflow. As an industrial de-facto standard for designing embedded systems, Simulink/Stateflow facilitates the building of an executable model for a complicated system. Specifically, analysis and validation of a Simulink/Stateflow model can be conducted by simulation. However, simulation is inherently incomplete in coverage of system test cases and unsound due to numerical error. As a remedy, it deserves to further verify Simulink/Stateflow models in a formal verification tool.

In our approach, the translator Sim2HCSP is designed to translate Simulink/Stateflow models to HCSP [18, 39]. By extending CSP with differential equations, HCSP is a formal specification language for modelling hybrid systems, and meanwhile, it is the input language of the interactive HHL prover. By applying Sim2HCSP, the translation from Simulink/Stateflow to HCSP is fully automatic. Complementary to Sim2HCSP, an automatic inverse translator HCSP2Sim is implemented to justify its correctness. We use HCSP2Sim to translate the HCSP model resulting from Sim2HCSP back to Simulink, and check the consistency between the output Simulink/Stateflow model and the original Simulink/Stateflow model by co-simulation.

The HHL prover is then applied to verify the above HCSP models obtained from Sim2HCSP. The HHL prover is a theorem prover for Hybrid Hoare Logic (HHL) [22, 35]. As the input of the HHL prover, the HCSP models are written in the form of HHL specifications. Each HHL specification consists of an HCSP process, a pre-/post-condition that specifies the initial and terminal states of the process, and a history formula that records the whole execution history of the process, respectively. HHL defines a set of axioms and inference rules to deduce such a specification. Finally, by applying the HHL prover, the specification to be proved will be transformed into an equivalent set of logical formulas, which will be proved by applying axioms of corresponding logics in an interactive or automatic way.

To handle differential equations, we use the concept of *differential invariants* to characterize their properties without solving them [23, 30]. For computing differential invariants, we have implemented an independent invariant generator, which will be called during the verification in the HHL prover. The invariant generator integrates both the quantifier elimination and SOS (sum-of-squares) based methods for computing differential invariants of polynomial equations, and can also deal with non-polynomial systems by transformation techniques we proposed in [24], which is implemented as EHS2PHS in Fig. 1.

To evaluate MARS, we report our experience in using MARS on a case study in real industry, i.e. a descent guidance control program of a lunar lander, which is a closed-loop control system with non-linear differential equations[1].

In our previous work [36], we studied the same example and verified it by combining several different verification techniques including simulation, bounded model checking and theorem proving. In this paper, we mainly focus on the tool implementation and integration, rather than on the case study itself as in [36]. The new contribution of this paper is threefold:

[1] The toolchain MARS and the verification of the lunar lander example can be found at `http://lcs.ios.ac.cn/~znj/tools/MARS_v1.1.zip`

- Firstly, we implement the reverse translator HCSP2Sim from HCSP to Simulink, to justify the correctness of the translation tool Sim2HCSP from Simulink to HCSP by co-simulation. This is not considered in the original version of Sim2HCSP presented in [42];
- Secondly, based on the invariant generation techniques proposed in [23, 24], we implement an invariant generator for differential equations and integrate it into the HHL prover. In [36], the invariants of related dynamics are synthesized manually. Besides, the tool EHS2PHS that abstracts a non-polynomial hybrid system by a polynomial one based on the technique in [24] is integrated to the invariant generator;
- Finally, we provide a seamless integration of all the tools on modelling, analysis and verification of hybrid systems as a toolchain MARS.

1.1 Related Work

There are some work on tools for formal verification of Simulink/Stateflow diagrams addressing both discrete and continuous blocks. In [6] Chen *et al.* proposed an approach that translates Simulink models to a real-time specification language and then validated the models via a generic theorem prover. However, their approach can only handle a special class of differential equations with closed form solutions, and cannot handle Stateflow diagrams. Tools based on numerical simulation or approximation are proposed. STRONG [13] performs bounded time reachability and safety verification for linear hybrid systems based on robust test generation and coverage. Breach [14] uses sensitivity analysis to compute approximate reachable sets and analyzes properties in the form of MITL based on numerical simulation. C2E2 [15] analyzes the discrete-continuous Stateflow models annotated with discrepancy functions by transforming them to hybrid automata, and then checks bounded time invariant properties of the models based on simulation.

There are some tools for verifying hybrid systems modelled by formal specification languages. The tool d/dt [5] provides reachability analysis and safety verification of hybrid systems with linear continuous dynamics and uncertain bounded input. iSAT-ODE [16] is a numerical SMT solver based on interval arithmetic that can conduct bounded model checking for hybrid systems. Flow* [8] computes overapproximations of the reachable sets of continuous dynamical and hybrid systems in a bounded time. Both iSAT-ODE and Flow* are able to handle non-polynomial ODEs (ordinary differential equations). Based on deductive method, the interactive theorem prover KeYmaera [31] (and its newly developed version KeYmaera X [17]) verifies hybrid systems specified using differential dynamic logic. These tools, however, are not directly applicable to Simulink/Stateflow models.

Organization. The rest of the paper is organized as follows: Section 2 introduces the tool Sim2HCSP for translating Simulink/Stateflow models, as well as its inverse HCSP2Sim. Section 3 and Section 4 introduce the HHL prover for verifying HCSP

models and the invariant generator respectively. In each of the sections, the corresponding tool is demonstrated by the descent guidance control program of a lunar lander. Section 5 concludes the paper.

2 Sim2HCSP Translator

In this section, we demonstrate a fully automatic translator *Sim2HCSP* [41,42] that encodes Simulink/Stateflow diagrams into HCSP processes.

Simulink/Stateflow As an industrial de-facto standard, Simulink [1] is extensively used for modelling, simulating and analyzing multidomain dynamic and embedded systems. It provides a graphical block diagramming tool and a customizable set of block libraries for building executable models of embedded systems and their environments. A Simulink model contains a set of blocks, subsystems, and wires, where blocks and subsystems cooperate by sending messages through the wires between them. For an elementary block, it basically gets input signals and computes the output signals assisted by a set of user-defined parameters to alter its functionalities. One typical parameter is the *sample time*, which defines how frequently the computation is taken. Two special values, 0 and -1, may be set for sample time, where 0 indicates that the block is used for simulating the physical environment and hence computes continuously, and -1 signifies that the sample time of the block is not determined yet, which will be determined by the sample times of the in-coming wires to the block. Thus, blocks are classified into two categories, i.e. *continuous* and *discrete*, according to their sample times.

As a toolbox integrated into Simulink, Stateflow offers the modelling capabilities of statecharts for reactive systems. It can be used to construct Simulink blocks, fed with Simulink inputs and produces Simulink outputs. A Stateflow diagram has a hierarchical structure, which can be an *AND diagram*, for which states are arranged in parallel and all of them become active whenever the diagram is activated; or an *OR diagram*, for which states are connected with transitions and only one of them becomes active when the diagram is activated. A Stateflow diagram consists of an alphabet of events and variables, a finite set of states, and transition networks.

Hybrid CSP Hybrid CSP (HCSP) [18, 39] is a formal modelling language for hybrid systems which extends CSP [19] by introducing differential equations, time constructs, and interrupts. In HCSP, exchanging data among processes is solely described by communications, and no shared variable is allowed between different processes in parallel. We denote by *dVar* and *cVar* the countable set of discrete and continuous variables respectively, and by *Chan* ranged over ch, ch_1, ..., the countable set of channels. The syntax of HCSP is given as follows:

$$\begin{aligned} P &\;\widehat{=}\; \text{skip} \mid x := e \mid ch?x \mid ch!e \mid P;Q \mid B \to P \mid P \sqcup Q \mid P^* \\ &\quad \mid \langle \mathscr{F}(\dot{s},s) = 0 \& B \rangle \mid \langle \mathscr{F}(\dot{s},s) = 0 \& B \rangle \trianglerighteq []_{i \in I}(io_i \to Q_i) \\ S &\;\widehat{=}\; P \mid S \| S \end{aligned}$$

Here $ch, ch_i \in Chan$, io_i stands for a communication event, i.e. either $ch_i?x$ or $ch_i!e$, $x \in dVar \cup cVar$, $s \in cVar$, B and e are Boolean and arithmetic expressions respectively, P, Q, Q_i are sequential processes, and S stands for a system, i.e. an HCSP process.

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

- skip terminates immediately having no effect on variables; and $x := e$ assigns the value of expression e to x and then terminates.
- $ch?x$ receives a value along channel ch and assigns it to x, and $ch!e$ sends the value of e along ch. A communication takes place as soon as both the sending and the receiving parties 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, and otherwise it terminates immediately.
- The internal choice $P \sqcup Q$ behaves as either P or Q, and the choice is made randomly by the system.
- The repetition P^* executes P for some finite number of times.
- $\langle \mathscr{F}(\dot{s}, s) = 0 \& B \rangle$ is the continuous evolution statement. It forces the vector s of real variables to evolve continuously according to the differential equations $\mathscr{F}$ as long as the Boolean expression B, which defines the *domain of* s, holds, and terminates when B turns false. For hybrid automata, non-determinism occurs when both the domain of the continuous evolution and the jump condition are satisfied, i.e. it can choose to stay in the continuous evolution, or leave it by making a discrete transition. In HCSP, there is no such non-determinism.
- $\langle \mathscr{F}(\dot{s}, s) = 0 \& B \rangle \trianglerighteq [\!]_{i \in I}(io_i \to Q_i)$ behaves like the continuous $\langle \mathscr{F}(\dot{s}, s) = 0 \& B \rangle$, except that it is preempted as soon as one of the communications io_i takes place. That is followed by the respective Q_i. Notice that, if the continuous terminates before a communication among $\{io_i\}_{i \in I}$ occurs, then the process terminates immediately without waiting for communication. When multiple communications from $\{io_i\}_{i \in I}$ get ready simultaneously before the others, an internal choice among these ready communications occur.
- $S_1 \| S_2$ 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.

Sim2HCSP Translator Given a Simulink/Stateflow model, Sim2HCSP translates its Simulink and Stateflow parts separately. With the approach in [42], the Simulink part is translated into a set of HCSP processes, while using the approach in [41], the Stateflow part is translated into another set of HCSP processes. Then, these HCSP processes are composed in parallel to form the whole model of the system. The Simulink and Stateflow diagrams in parallel transmit data or events via communications. Please refer to [41, 42] for details. Sim2HCSP takes Simulink/Stateflow models (in xml format, which is generated by a Matlab script) as input, and outputs several files as the definitions for the corresponding HCSP processes, which contain three files for defining variables, processes, and assertions for the Simulink part, and the same three files for each Stateflow diagram within the Stateflow part.

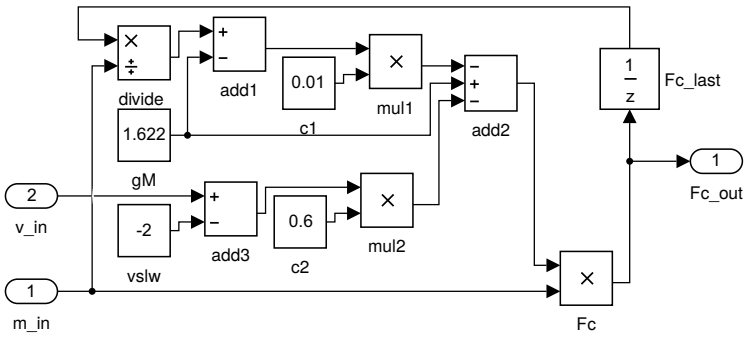

Fig. 2 Simulink diagram of the guidance program for the slow descent phase

We demonstrate the translation approach by a scenario originating from the descent guidance control program of a lunar lander, which actually provides a specific sampled-data control system composed of the physical plant and the embedded control program.

Example 1 (running example). The guidance control program is built as a Simulink diagram in Fig. 2, which includes three parts: updating mass m, calculating acceleration aIC, and calculating thrust F_c. The sample time of all blocks is fixed as 0.128s, i.e. the period of the guidance program. In Fig. 2, block m_in reads mass m from the continuous plant (modelled as the Simulink diagram in Fig.3) periodically, block Fc is used to calculate thrust F_c, and the rest are used to calculate acceleration aIC. In particular, there are two inputs for block Fc: the first is the acceleration aIC, which is defined as

$$-0.01(F_c/m - gM) - 0.6(v - vslw) + gM$$

as shown in the diagram; the second is the mass m, and F_c is then defined as the product of aIC and m. The details of the guidance program can be found in [36].

The lander's dynamics is mathematically represented by

$$\begin{cases} \dot{r} &= v \\ \dot{v} &= \frac{F_c}{m} - gM \\ \dot{m} &= -\frac{F_c}{Isp} \\ \dot{F_c} &= 0 \end{cases} \tag{1}$$

where

- 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;
- $gM = 1.622$ m/s^2 is the magnitude of the gravitational acceleration on the moon;

– *Isp* denotes the *specific impulse*[2] of the lander's thrust engine. It has two possible values depending on the values of F_c. When F_c is less or equal than 3000 N, $Isp = 2548$ N s/kg, and otherwise, $Isp = 2842$ N s/kg. For simplicity, we use Isp_1 and Isp_2 to represent the two values of the impulse, and meanwhile, use ODE_1 and ODE_2 to represent the two differential equations corresponding to Isp_1 and Isp_2 as defined by (1) respectively.

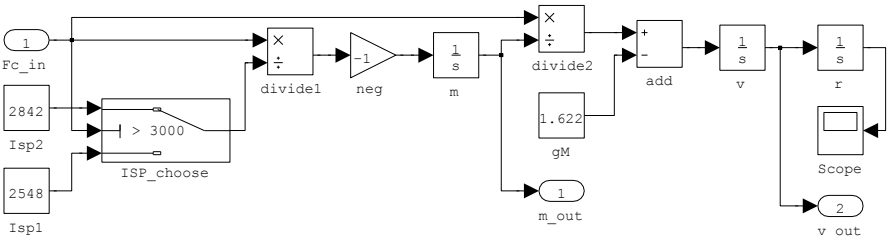

Fig. 3 The Simulink diagram of the dynamics for the slow descent phase

The physical dynamics in (1) is modelled by the diagram shown in Fig. 3, where the threshold of block ISP_choose is 3000, meaning that it outputs 2842 as the value of Isp when F_c is greater than 3000 and 2548 otherwise. The initial values of m, v, and r ($m = 1250$ kg, $r = 30$ m, $v = -2$ m/s) are specified as initial values of the *integrator* blocks m, v, and r respectively. Specifically, an integrator block outputs its initial value at the beginning and the integration of the input signal afterwards.

The safety property we want to prove for the lunar lander system is

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

The simulation result w.r.t the velocity v is illustrated in Fig. 4. It is shown that the velocity of the lander is kept between -2 m/s and -1.9999 m/s, which corresponds to the safety property we proposed above.

Then the manually constructed Simulink model is translated into annotated HCSP using the tool Sim2HCSP, which employs the HCSP pattern

```
definition P :: proc where
"P == PC_Init; PD_Init; t:=0; (PC_Diff;t:=0;PD_Rep)*"
```

In process P, 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 0.128 s; PD_Rep calculates thrust F_c according to

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

[2] Specific impulse is a physical quantity describing the efficiency of rocket engines. It equals the thrust produced per unit mass of propellant burned per second.

for the next sampling cycle; variable t denotes the elapsed time in each sampling cycle. Hence, process P 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.

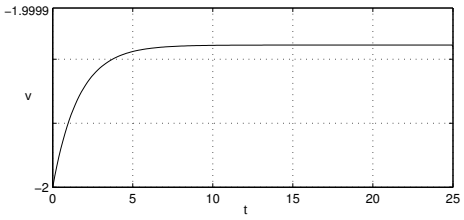

Fig. 4 The original simulation result

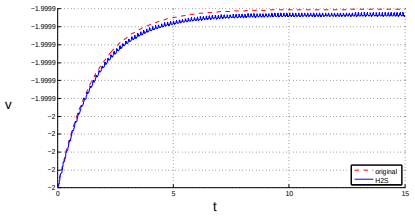

Fig. 5 The co-simulation result

Consistency Checking by Co-simulation To justify the correctness of the translation above, we provide a method to check the consistency between the original Simulink model and the generated HCSP formal model. This is done with the help of a tool called *HCSP2Sim* [7], an inverse decoding from HCSP back into Simulink. The translator HCSP2Sim takes as input an HCSP process transformed directly from the HCSP model generated by Sim2HCSP, and generates a Simulink graphical model in the mdl format automatically as output. Figure 5 illustrates the co-simulation result, where the evolution of the lander's velocity v in the original Simulink model is shown as the red dash line[3] and the one for the inversely translated Simulink model as the blue line. The co-simulation result shows that the translation loop keeps the behaviour of the system consistently. However, as also shown by the result, there exists a gap between the red and blue lines. This is the inevitable consequence of introducing some necessary *delay* blocks in the translation from HCSP to Simulink, to prevent the *zeno*[4] phenomena while keeping the well-composed translation architecture. Nevertheless, absolute magnitude of the gap can be reduced by means of narrowing the simulation time step to an acceptable slot. In such way, a more precise co-simulation can be conducted. As an additional byproduct, the inverse translation also provides people with the ability to simulate an abstract formal model and see how the system behaves immediately and intuitively.

3 HHL Prover

This section presents the HHL prover for reasoning about HCSP models, and before that, gives a brief introduction of the Hybrid Hoare Logic (HHL) based on which the prover is implemented.

[3] Identical to the line in Fig. 4.

[4] A sequence of infinitely many computations that take finite time.

Hybrid Hoare Logic For verifying the behavior of HCSP processes, a deductive calculus called Hybrid Hoare Logic (HHL) is proposed in [22]. Given a process P, the specification $\{Pre\}P\{Post;\ HF\}$ is defined, where Pre and $Post$ are first-order logic (FOL) formulas for specifying the pre-/post-conditions holding at the beginning and termination of P, and HF is a duration calculus (DC) [37, 38] formula for specifying the history throughout the whole execution of P. Here DC is an interval logic for describing real-time systems. In particular, as used below in the paper, ℓ is a temporal variable denoting the length of the considered interval, and $\lceil S \rceil$ for some FOL formula S means that S holds everywhere in the considered interval.

In HHL, for each HCSP construct, a set of inference rules are given for deducing its specifications. Below we explain the rule for the continuous evolution $\langle \mathscr{F}(\dot{s},s) = 0 \& B \rangle$. Instead of explicit solutions, the concept of *differential invariant* [23, 30] is used to characterize the behavior of the corresponding differential equations. As shown by the following rule, a differential invariant Inv needs to be annotated in the specification:

$$\frac{\begin{array}{c} Init \to Inv \quad (Inv, \mathscr{F}) \to Inv \quad p \wedge close(Inv) \wedge close(\neg B) \to q \\ l = 0 \vee \lceil close(Inv) \wedge p \wedge close(B) \rceil \to G \end{array}}{\{Init \wedge p\}\ \langle \mathscr{F}(\dot{s},s) = 0 \& Inv \& B \rangle\ \{q; G\}}$$

where $Init$ specifies the initial state for s, p for other variables rather than s (thus will not change during the evolution), and function $close(\cdot)$ extends the domain by the corresponding formula to include the boundary; $(Inv, \mathscr{F})$ represents the formula describing the post-states of $\mathscr{F}$ executing from a state satisfying Inv. Consider the hypothesis, the FOL formula in the first line indicates that Inv is indeed a sufficiently strong invariant, i.e. it is satisfied by the initial state, preserved by the continuous evolution, and strong enough to guarantee the postcondition; the DC formula in the second line indicates that the evolution terminates immediately (specified by $l = 0$), or otherwise, if the evolution takes more than zero time, then the closure of invariant Inv, the precondition p (related to discrete variables) and the closure of domain B hold everywhere throughout the whole execution. We have proved the soundness of the rule, and thus the proof of the specification of the continuous evolution will be reduced to an equivalent differential invariant generation problem: if Inv exists such that it satisfies the conditions in the hypothesis, then the original specification is proved.

The HHL Prover The interactive theorem prover *HHL prover*, as illustrated by Fig. 1, is implemented in Isabelle/HOL to mechanize the HHL framework and has been applied for verifying practical hybrid systems [36, 40]. The prover encodes the HHL framework in a deep style: the HCSP processes and the two assertion languages (i.e. FOL and DC) are defined by respective new datatypes, and in consequence, the inference system of HCSP (i.e. HHL), the deductive systems of FOL and DC are defined as new axioms, of Isabelle/HOL respectively. In the HHL prover, a set of verification conditions for HHL specifications are generated first by applying HHL inference rules, and then these conditions are proved by applying the FOL and DC deductive rules. Most of the proofs are done interactively. To improve this,

we define a conversion function from our FOL formulas to HOL formulas and thus the existing proof tactics of Isabelle/HOL are applicable. For example, the powerful *sledgehammer* that integrates third-party SMT solvers such as Z3 [12] can be applied to prove FOL formulas in the HHL prover.

When the specification to be proved contains unknown differential invariants, some verification conditions related to the invariants remain unproved in HHL prover. For such cases, the prover needs to call external provers, e.g. the invariant generator in MARS, for solving the invariants. This will be explained in detail in the next section.

Example 2 (running example). In Sect. 2, by applying Sim2HCSP, we get the HCSP process P for the lunar lander example. In order to meet the design requirement of the control program, we need to prove the following specification for it:

```
{True} P {|v-vlsw|<=0.05; (l=0)|high(|v-vlsw|<=0.05)}
```

where *high* corresponds to the $\lceil\ \rceil$ operator in DC. The specification indicates that the slow descent phase satisfies the safety property, i.e., the difference between the velocity v and the target velocity *vlsw* is always at most 0.05. By applying HHL prover, the specification is finally reduced to the following five unsolved constraints for the differential invariants of P:

```
lemma cons1: "(t<=0.128) & (t>=0) & Inv |- |v-vlsw|<=0.05"
lemma cons2: "(v=-2) & (m=1250) & (Fc=2027.5)
  & (t=0) |- Inv"
lemma cons3: "(t= 0.128) & Inv
  |- substF([(t,0)], substF([(Fc,
     -0.01*(Fc-1.622*m) - 0.6*(v+2)*m + 1.622*m)],Inv))"
lemma cons4: "exeFlow(''v, m, r, t'',
  ''(Fc/m) - 1.622, -(Fc/2548), v, 1'',t < 0.128,Inv) |- Inv"
lemma cons5: "exeFlow(''v, m, r, t'',
  ''(Fc/m) - 1.622, -(Fc/2842), v, 1'',t < 0.128,Inv) |- Inv"
```

The intuitive explanation of the constraints is: during each period of length 0.128 s, the invariant *Inv* is sufficiently strong to deduce the safety property (*cons*1), the initial state satisfies *Inv* (*cons*2), the computation, and the continuous evolution governed by the two differential equations of P, preserve *Inv* respectively (*cons*3, *cons*4 and *cons*5). In the above constraints, function $exeFlow(ode, f)$ for given equation *ode* and precondition f returns the postcondition after executing the continuous flow represented by *ode* from a state satisfying f. In the next section, we will show how to apply an external invariant generator to handle these constraints.

4 Invariant Generator

To prove the invariant related subgoals during the verification in the HHL prover, we need to call an external *invariant generator* from the HHL prover. The invariant generator of MARS provides two approaches to synthesizing invariants, i.e., quantifier

elimination (QE) based and SOS based. Before introducing the invariant generator, we explain how to invoke an external prover in Isabelle.

4.1 Isabelle Oracle

Isabelle provides the oracle mechanism to use new decision procedures not based on its inference kernel. Listing 1 defines the oracle to decide invariant related constraints. Function *trans_allCons* translates an invariant constraint in the form of FOL formulas into the string representation expected by the solver. The core function *decide* takes a string representation of the invariant constraints and passes it to the script program implementing the invariant generator, and then returns true if an invariant exists such that the constraints are satisfied, or false otherwise. These two functions are then combined into the oracle *inv_oracle*, which verifies an input invariant constraint using *decide*, and outputs it as a theorem of Isabelle without any change if it is certified. Finally, to be used for Isabelle proofs, the oracle *inv_oracle* is wrapped into a tactic *inv_oracle_tac* and then a new method *inv_oracle* is created based on this tactic.

```
ML{*
fun trans_allCons t = ...
fun decide p = "$InvGen/script.sh "^"\""^p^"\""
  |> Isabelle_System.bash_output
  |> fst
  |> isTrue;*}
oracle inv_oracle = {* fn ct =>
  if decide (trans_allCons (Thm.term_of ct))
  then ct
  else error "Proof failed."*}
ML{*
val inv_oracle_tac =
  CSUBGOAL (fn (goal, i) =>
  (case try inv_oracle goal of
    NONE => no_tac
  | SOME thm => rtac thm i))*}
method_setup inv_oracle = {*
  Scan.succeed (K (Method.SIMPLE_METHOD' inv_oracle_tac))*}
```

Listing 1 The Oracle for deciding differential invariants

Depending on the different methods for computing differential invariants, we have implemented two oracles: *inv_oracle_qe* based on quantifier elimination, and *inv_oracle_sos* based on the SOS method. We will explain these methods in more detail in Sects. 4.2–4.5.

Example 3 (running example). By applying the oracle *inv_oracle_sos*, we have proved the conjunction of the unsolved five constraints presented in Example 2 as a lemma:

```
lemma allCons: "|- cons1 [&] cons2 [&] cons3 [&] cons4 [&] cons5"
apply (simp: add consi_def for all i)
apply inv_oracle_sos
done
```

At this state, by applying MARS, the verification of the safety for the lunar lander example thus is completed. Specifically, the manual proof script consists of approximately 300 lines and the verification is done within one minute on a 32-bit Linux computer with a 1.60GHz Intel Core-i5 processor and 4GB of RAM.

Next we present the invariant generator in detail.

4.2 Differential Invariant Generation

The basic idea of differential invariant generation is by using templates and constraint solving. For simplicity, we illustrate the idea on systems with a single ODE and no jumps. For such systems, the unresolved constraints as in Example 2 and Example 3 would roughly be as follows:

a) $\phi_{\text{pre}} \longrightarrow \phi_{\text{inv}}$;
b) $\phi_{\text{inv}} \longrightarrow [\dot{x} = f]\phi_{\text{inv}}$;
c) $\phi_{\text{inv}} \longrightarrow \phi_{\text{post}}$,

where

- a) means that a certain precondition ϕ_{pre} implies the required invariant ϕ_{inv} ;
- b) means that any trajectory of the ODE $\dot{x} = f$ starting from ϕ_{inv} will always satisfy ϕ_{inv}, that is, ϕ_{inv} is a differential invariant of $\dot{x} = f$;
- c) means that the differential invariant ϕ_{inv} implies a certain postcondition ϕ_{post} .

For systems with different modes and jumps betweens these modes, as well as reset functions related to the jumps, additional constraints will be imposed, which are omitted here.

Example 4. In a more readable way, the five unresolved lemmas in Example 2 and Example 3 impose the following constraints:

(C1) $t \leq 0.128 \wedge t \geq 0 \wedge Inv \longrightarrow |v - vslw| \leq 0.05$;
(C2) $v = -2 \wedge m = 1250 \wedge F_c = 2027.5 \wedge t = 0 \longrightarrow Inv$;
(C3) $t = 0.128 \wedge Inv \longrightarrow Inv(t \mapsto 0; F_c \mapsto F_c')$, with F_c' defined in (2);
(C4) *Inv* is the differential invariant of the constrained dynamical system

$$\langle ODE_1; 0 \leq t \leq 0.128 \wedge F_c \leq 3000 \rangle$$

(C5) *Inv* is also the differential invariant of the constrained dynamical system

$$\langle ODE_2; 0 \leq t \leq 0.128 \wedge F_c > 3000 \rangle$$

where ODE_1 and ODE_2 are the dynamics in (1) corresponding to Isp_1 and Isp_2 respectively.

If ϕ_{pre} and ϕ_{post} are polynomial formulas, and f is a polynomial vector field, then we can try to generate ϕ_{inv} by defining a polynomial template, i.e. a polynomial formula with undetermined parameters as an invariant candidate and then solving certain constraints to get the parameters. We have the following two approaches for generating constraints from a)-c) and getting the parameters:

1) **QE-Based**: transform a), b) and c) into first-order polynomial formulas as proposed in [23] and then apply quantifier-elimination (QE) [9] to the quantified conjunction of the transformed formulas to see if the parameters have solutions;
2) **SOS-Based**: transform a), b) and c) into sum-of-squares (SOS) constraints as proposed in [20] and then use an SDP (semi-definite programming) solver to solve the constraints to get the values of parameters.

The QE-approach is exact and more general, and in particular, the transformation of [23] is sound and complete, while the SOS approach is more efficient due to the use of numerical computation. We have implemented invariant generators based on both QE and SOS, and integrated them into the MARS tool chain. We will give more details about the two generators in Sects. 4.4 and 4.5 respectively.

When ϕ_{pre} and ϕ_{post} are non-polynomial formulas, or f is a non-polynomial vector field, we will use the abstraction approach proposed in the next subsection.

4.3 Abstraction of Elementary Hybrid Systems by Variable Transformation

In practice, HSs (hybrid systems) may contain elementary functions such as exp, ln, sin, cos, etc., called *Elementary Hybrid Systems* (EHSs). Due to the non-polynomial expressions which lead to undecidable arithmetic, verification of EHSs is very hard. Existing approaches based on partition of the state space or overapproximation of reachable sets suffer from state space explosion or inflation of numerical errors. In [24], we proposed a symbolic abstraction approach that reduces EHSs to polynomial hybrid systems (PHSs), by replacing all non-polynomial terms with newly introduced variables. Thus the verification of EHSs is reduced to the one of PHSs, enabling us to apply all the well-established verification techniques and tools for PHSs to EHSs. In this way, it is possible to avoid the limitations of many existing methods. We have implemented the above abstraction procedure as a tool EHS2PHS.

For example, the dynamics of the lunar lander involves non-polynomial expression, $\dot{v} = \frac{F_c}{m} - gM$, which is abstracted by the tool EHS2PHS based on a rule of variable transformation, i.e. $a = \frac{F_c}{m}$, where a happens to be the instant acceleration produced by the thrust F_c of the lander. The equivalently transformed polynomial system will then be delivered to the invariant generator.

4.4 QE-Based Invariant Generator

The invariant generator based on quantifier elimination is implemented in Mathematica as a Wolfram script. It can be accessed in Isabelle through the method *inv_oracle_qe* using command *apply inv_oracle_qe*. The generator takes two parameters as input: constraints ϕ_{allCons} to be solved from the Isabelle function *trans_allCons* as shown in Listing 1, as well as a positive integer n through the user interface. The parameter n is the order of polynomials which will be used to generate a parameterized polynomial invariant template based on variables X extracted from ϕ_{allCons}. The parameters in the invariant template is denoted as U and there is a user interface to set certain parameters in U to 0 in order to reduce the difficulty of quantifier elimination. There is a placeholder *inv* in ϕ_{allCons}, which will then be replaced by the generated invariant template.

Now ϕ_{allCons} is a conjunction of constraints like those shown in Example 4. Then constraints like (C4) are translated into polynomial formulas using the technique proposed in [23], and accordingly, ϕ_{allCons} is transformed into a conjunction of polynomial formulas, denoted by ϕ_{poly}. Use the default quantifier elimination function *Resolve* in Mathematica to eliminate all the quantifiers in $\exists U \forall X : \phi_{\text{poly}}$, and a result *True* or *False* will be returned. The invariant generator will then pass this result to Isabelle.

4.5 SOS-Based Invariant Generator

In order to avoid the high complexity of quantifier elimination algorithms, which takes doubly exponential time on real closed fields [11], an alternative is provided to synthesize invariants based on sum-of-squares (*SOS*) relaxation approach in the study of polynomial hybrid systems [20]. Given a bunch of unproven constraints derived from Isabelle, the SOS-based invariant generator first transforms them into a sequence of SOS-constraints w.r.t the user-defined invariant template, and then invokes semidefinite programming (*SDP*) [28, 34] to solve the parameterized polynomial invariant.

We continue the lunar lander example to demonstrate the use of the generator. Like the QE method, the SOS-based invariant generator can be triggered in Isabelle by an oracle called *inv_oracle_sos*, in which a terminal window is initially popped-up for the user to specify the upper bound of the polynomial degree d (we assume that the undetermined invariant *Inv* is a semialgebraic set of the form $PInv \leq 0$, where *PInv* is a parameterized polynomial with degree d); and then a Mathematica script *ScriptGenerator* is executed to generate an SOS-constraint model *sosInv.m* written as a script of the Matlab-based optimization tool Yalmip [25, 26]. For instance, the safety constraint (C1) which is equivalent to

$$t \geq 0 \wedge t \leq 0.128 \wedge (v < -2.05 \vee v > -1.95) \rightarrow PInv > 0,$$

is transformed to an SOS-constraint:

$$SOS(PInv - s_1 * t * (0.128 - t) - s_2 * (v + 1.95) * (v + 2.05) - eps)$$

where $SOS(f)$ indicates that the function f is a sum-of-squares polynomial, s_1 and s_2 are both SOS polynomials, and eps is a given positive constant denoting a margin introduced to avoid the errors of numerical computation in Matlab; to determine the parameters in *PInv*, s_1, and s_2, as well as parameters in the other constraints, the Yalmip script *sosInv.m* is then executed in Matlab and invokes the solver SDPT-3 [32, 33] to solve all the SOS-constraints; finally, another Mathematica script *InvChecker* is called to check and return the solving result back to Isabelle, namely *True* if the problem is successfully solved, or *False* otherwise. With $d = 6$, we get a result of *True* associated with the invariant shown in Fig. 6 (left part), and complete the proof of lemma *allCons* in Example 3 eventually.

In addition, once the SOS-based invariant generator is triggered by applying oracle *inv_oracle_sos* in Isabelle, all the procedures described above, except for the pop-up terminal, are transparent to users, i.e. no Matlab desktop or Mathematica frontend can be observed. Therefore in order to give an intuitive observation of the invariant, we provide an additional notebook file *InvChecker.nb* that can be executed in a Mathematica frontend to plot a graphical region of the generated invariant as depicted by Fig. 6 (right part). Besides, to avoid synthesizing a false invariant due to numerical computation errors, we can also integrate symbolic posterior checking of the generated invariants in *InvChecker.nb*, based on the symbolic computation packages provided in Mathematica.

$$\begin{aligned}
&2.716877217 + 0.2881 * t + 1.6781 * v - 0.3244 * a + 0.1974 * t \\
&- 0.0274 * t * v + 0.1110 * v^2 + 0.0133 * t * a + 0.4345 * v * a \\
&+ 0.5502 * a^2 - 0.1210 * t^3 - 0.0575 * t^2 * v - 0.1659 * t * v^2 \\
&- 0.0169 * v^3 - 0.1182 * t^2 * a - 0.5511 * t * v * a + 0.1171 * v^2 * \\
&- 0.7916 * t * a^2 + 0.1479 * v * a^2 - 0.0728 * a^3 + 0.0659 * t^4 \\
&- 0.0552 * t^3 * v + 0.1924 * t^2 * v^2 + 0.2271 * t * v^3 + 0.0623 * v^4 \\
&+ 0.0517 * t^3 * a + 0.1108 * t^2 * v * a + 0.2281 * t * v^2 * a \\
&+ 0.0464 * v^3 * a + 0.5376 * t^2 * a^2 - 0.1645 * t * v * a^2 \\
&+ 0.0220 * v^2 * a^2 + 0.0033 * t * a^3 - 0.0107 * v * a^3 + 0.0230 * a \\
&- 0.3817 * t^5 + 0.2199 * t^4 * v + 0.0200 * t^3 * v^2 + 0.2136 * t^2 * v \\
&- 0.0824 * t * v^4 - 0.1764 * t^4 * a - 0.1554 * t^3 * v * a \\
&- 0.4660 * t^2 * v^2 * a - 0.2303 * t * v^3 * a - 0.0869 * t^3 * a^2 \\
&- 0.1280 * t^2 * v * a^2 - 0.1325 * t * v^2 * a^2 - 0.0484 * t^2 * a^3 \\
&- 0.0240 * t * v * a^3 - 0.0619 * t * a^4 + 0.3226 * t^6 + 0.0936 * t^5 \\
&+ 0.2142 * t^4 * v^2 + 0.0581 * t^3 * v^3 + 0.1452 * t^2 * v^4 \leq 0
\end{aligned}$$

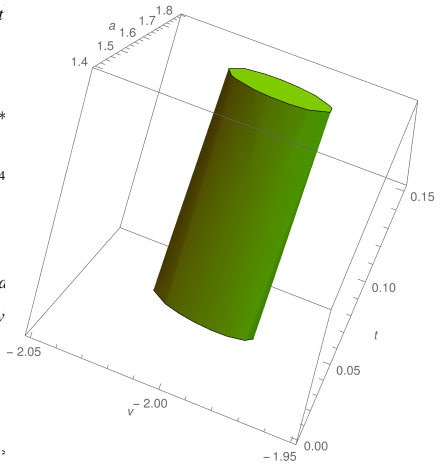

Fig. 6 The invariant generated by SOS relaxation with $d = 6$

5 Conclusion and Future Work

We presented a toolchain named MARS that links the modelling, analysis and verification of hybrid systems. The workflow of using MARS consists of the following phases: firstly, hybrid systems are modelled in the Simulink/Stateflow environment, which also facilitates model validation through numerical simulation; secondly, to overcome the limitations of simulation, the informal Simulink/Stateflow models are automatically transformed through the Sim2HCSP translator into formal models in the HCSP language; meanwhile, by an inverse translation from HCSP to Simulink models using the tool HCSP2Sim, and performing co-simulation, the consistency between the informal and formal models are justified; finally, the HCSP models can be verified preserving the given properties using the interactive HHL Prover, in which different schemes for automatic differential invariant generation are integrated, possibly with the support of EHS2PHS to abstract an EHS to a PHS first. We have discussed the details of the implementation of all components of MARS, and demonstrated how to use it through a real-life example of the slow descent control of a lunar lander.

As future work, we plan to improve MARS in the following aspects: the HHL prover needs improving its HHL verification framework and also its encoding in Isabelle/HOL so that more automation can be achieved for the proofs; the external invariant generators need to be enhanced with more efficient symbolic or hybrid numeric-symbolic computation techniques; the toolchain will be applied to other real-world case studies such as the modelling and verification of Chinese High-Speed Train Control System (CTCS); various component tools of MARS need to be more tightly integrated with a friendly user interface provided; and so on.

References

1. Simulink User's Guide (2013). `http://www.mathworks.com/help/pdf_doc/simulink/sl_using.pdf`
2. Aerts, A., Mousavi, M.R., Reniers, M.: A tool prototype for model-based testing of cyber-physical systems. In: M. Leucker, C. Rueda, D.F. Valencia (eds.) ICTAC 2015, pp. 563–572. Springer International Publishing (2015)
3. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.H.: Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In: R.L. Grossman, A. Nerode, A.P. Ravn, H. Rischel (eds.) Hybrid Systems, *Lecture Notes in Computer Science*, vol. 736, pp. 209–229. Springer Berlin Heidelberg (1993)
4. Annpureddy, Y., Liu, C., Fainekos, G., Sankaranarayanan, S.: S-TaLiRo: A tool for temporal logic falsification for hybrid systems. In: P.A. Abdulla, K.R.M. Leino (eds.) TACAS 2011, pp. 254–257. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
5. Asarin, E., Dang, T., Maler, O.: The d/dt tool for verification of hybrid systems. In: CAV 2002, *Lecture Notes in Computer Science*, vol. 2404, pp. 365–370 (2002)
6. Chen, C., Dong, J.S., Sun, J.: A formal framework for modelling and validating Simulink diagrams. Formal Aspects of Computing **21**(5), 451–483 (2009)
7. Chen, M., Ravn, A., Yang, M., Zhan, N., Zou, L.: A two-way path between formal and informal design of embedded systems. In: Proc. UTP 2016 (2016). To appear

8. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: CAV 2013, *Lecture Notes in Computer Science*, vol. 8044, pp. 258–263 (2013)
9. Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decompostion. In: H. Brakhage (ed.) Automata Theory and Formal Languages, *Lecture Notes in Computer Science*, vol. 33, pp. 134–183. Springer Berlin Heidelberg (1975)
10. Dang, T., Nahhal, T.: Coverage-guided test generation for continuous and hybrid systems. Formal Methods in System Design **34**(2), 183–213 (2009)
11. Davenport, J.H., Heintz, J.: Real quantifier elimination is doubly exponential. J. Symb. Comput. **5**(1-2), 29–35 (1988)
12. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS 2008, *Lecture Notes in Computer Science*, vol. 4963, pp. 337–340. Springer (2008)
13. Deng, Y., Rajhans, A., Julius, A.A.: STRONG: A trajectory-based verification toolbox for hybrid systems. In: QEST 2013, *Lecture Notes in Computer Science*, vol. 8054, pp. 165–168 (2013)
14. Donzé, A.: Breach, a toolbox for verification and parameter synthesis of hybrid systems. In: CAV 2010, *Lecture Notes in Computer Science*, vol. 6174, pp. 167–170 (2010)
15. Duggirala, P.S., Mitra, S., Viswanathan, M., Potok, M.: C2E2: A verification tool for annotated Stateflow models. In: TACAS 2015, *Lecture Notes in Computer Science*, vol. 9035, pp. 68–82 (2015)
16. Eggers, A., Ramdani, N., Nedialkov, N., Fränzle, M.: Improving SAT modulo ODE for hybrid systems analysis by combining different enclosure methods. In: SEFM 2011, pp. 172–187. Springer-Verlag (2011)
17. Fulton, N., Mitsch, S., Quesel, J., Völp, M., Platzer, A.: KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. In: CADE 2015, pp. 527–538 (2015)
18. 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)
19. Hoare, C.: Communicating sequential processes, vol. 178. Prentice-hall Englewood Cliffs (1985)
20. Kong, H., He, F., Song, X., Hung, W.N., Gu, M.: Exponential-condition-based barrier certificate generation for safety verification of hybrid systems. In: N. Sharygina, H. Veith (eds.) CAV 2013, *Lecture Notes in Computer Science*, vol. 8044, pp. 242–257. Springer Berlin Heidelberg (2013)
21. Lafferriere, G., Pappas, G.J., Yovine, S.: Symbolic reachability computation for families of linear vector fields. Journal of Symbolic Computation **32**(3), 231–253 (2001)
22. Liu, J., Lv, J., Quan, Z., Zhan, N., Zhao, H., Zhou, C., Zou, L.: A calculus for hybrid CSP. In: K. Ueda (ed.) APLAS 2010, *Lecture Notes in Computer Science*, vol. 6461, pp. 1–15. Springer Berlin Heidelberg (2010)
23. Liu, J., Zhan, N., Zhao, H.: Computing semi-algebraic invariants for polynomial dynamical systems. In: EMSOFT 2011, pp. 97–106. ACM, New York, NY, USA (2011)
24. Liu, J., Zhan, N., Zhao, H., Zou, L.: Abstraction of elementary hybrid systems by variable transformation. In: FM 2015, *Lecture Notes in Computer Science*, vol. 9109, pp. 360–377 (2015)
25. Löfberg, J.: YALMIP : A toolbox for modeling and optimization in MATLAB. In: Proc. of the CACSD Conference. Taipei, Taiwan (2004). `http://users.isy.liu.se/johanl/yalmip`
26. Löfberg, J.: Pre- and post-processing sum-of-squares programs in practice. IEEE Transactions on Automatic Control **54**(5), 1007–1011 (2009)
27. Manna, Z., Pnueli, A.: Verifying hybrid systems. In: R.L. Grossman, A. Nerode, A.P. Ravn, H. Rischel (eds.) Hybrid Systems, *Lecture Notes in Computer Science*, vol. 736, pp. 4–35. Springer Berlin Heidelberg (1993)
28. Parrilo, P.A.: Semidefinite programming relaxations for semialgebraic problems. Mathematical programming **96**(2), 293–320 (2003)
29. Platzer, A.: Differential-algebraic dynamic logic for differential-algebraic programs. Journal of Logic and Computation **20**(1), 309–352 (2010)

30. Platzer, A., Clarke, E.M.: Computing differential invariants of hybrid systems as fixedpoints. In: A. Gupta, S. Malik (eds.) CAV 2008, *Lecture Notes in Computer Science*, vol. 5123, pp. 176–189. Springer Berlin Heidelberg (2008)
31. Platzer, A., Quesel, J.D.: KeYmaera: A hybrid theorem prover for hybrid systems. In: IJCAR 2008, *Lecture Notes in Computer Science*, vol. 5195, pp. 171–178. Springer (2008)
32. Toh, K.C., Todd, M., Tütüncü, R.H.: SDPT3 — a MATLAB software package for semidefinite programming. Optimization Methods and Software **11**, 545–581 (1999)
33. Tütüncü, R.H., Toh, K.C., Todd, M.J.: Solving semidefinite-quadratic-linear programs using SDPT3. Mathematical programming **95**(2), 189–217 (2003)
34. Vandenberghe, L., Boyd, S.: Semidefinite programming. SIAM Review **38**(1), 49–95 (1996)
35. Wang, S., Zhan, N., Zou, L.: An improved HHL prover: An interactive theorem prover for hybrid systems. In: ICFEM 2015, *Lecture Notes in Computer Science*, vol. 9407, pp. 382–399 (2015)
36. 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: FM 2014, *Lecture Notes in Computer Science*, vol. 8442, pp. 733–748 (2014)
37. Zhou, C., Hansen, M.R.: Duration Calculus — A Formal Approach to Real-Time Systems. Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag Berlin Heidelberg (2004)
38. Zhou, C., Hoare, C., Ravn, A.P.: A calculus of durations. Information Processing Letters **40**(5), 269–276 (1991)
39. Zhou, C., Wang, J., Ravn, A.P.: A formal description of hybrid systems. In: R. Alur, T.A. Henzinger, E.D. Sontag (eds.) Hybrid Systems III, *Lecture Notes in Computer Science*, vol. 1066, pp. 511–530. Springer Berlin Heidelberg (1996)
40. 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: E. Cohen, A. Rybalchenko (eds.) VSTTE 2013, *Lecture Notes in Computer Science*, vol. 8164, pp. 262–280. Springer Berlin Heidelberg (2014)
41. Zou, L., Zhan, N., Wang, S., Fränzle, M.: Formal verification of Simulink/Stateflow diagrams. In: ATVA 2015, *Lecture Notes in Computer Science*, vol. 9346, pp. 464–481 (2015)
42. 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)