Combining Formal and Informal Methods in the Design of Spacecrafts

Mengfei Yang[1] and Naijun Zhan[2](✉)

[1] Chinese Academy of Space Technology, Beijing, China
[2] State Key Laboratory of Computer Science,
Institute of Software, CAS, Beijing, China
znj@ios.ac.cn

Abstract. In this chapter, we summarize our experience on combing formal and informal methods together in the design of spacecrafts. With our approach, the designer can either build an executable model of a spacecraft using the industrial standard environment *Simulink/Stateflow*, which facilitates analysis by simulation, or construct a formal model using Hybrid CSP (HCSP), which is an extension of CSP for formally modeling hybrid systems. HCSP processes can be specified and reasoned about by Hybrid Hoare Logic (HHL), which is an extension of Hoare logic to hybrid systems. The connection between informal and formal methods is realized via an automatic translator from *Simulink/Stateflow* diagrams to HCSP and an inverse translator from HCSP to *Simulink*. The advantages of combining formal and informal methods in the design of spacecrafts include

- It enables formal verification as a complementation of simulation. As the inherent incompleteness of simulation, it has become an agreement in industry and academia to complement simulation with formal verification, but this issue still remains challenging although lots of attempts have been done (see the related work section);
- It provides an option to start the design of a hybrid system with an HCSP formal model, and simulate and/or test it using *Matlab* platform economically, without expensive formal verification if not necessary;
- The semantic preservation in shifting between formal and informal models is justified by co-simulation. Therefore, it provides the designer the flexibility using formal and informal methods according to the trade-off between efficiency and cost, and correctness and reliability.

We will demonstrate the above approach by analysis and verification of the descent guidance control program of a lunar lander, which is a real-world industry example.

Keywords: Spacecraft · Lunar lander · *Simulink/Stateflow* · formal methods · hybrid systems

This work is supported partly by "973 Program" under grant No. 2014CB340701, by NSFC under grants 91118007 and 91418204, and by the CAS/SAFEA International Partnership Program for Creative Research Teams.

Z. Liu and Z. Zhang (Eds.): SETSS 2014, LNCS 9506, pp. 290–323, 2016.
DOI: 10.1007/978-3-319-29628-9_6

1 Introduction

Spacecraft control systems like most digital controllers are, by definition, hybrid systems as they interact with and/or try to control some aspects of the physical world, also typical safety-critical as any fault could result in the failure of the whole mission. Detailed behavior modeling with rigorous specification, extensive analysis and formal verification, required for reliability prediction, is a great challenge for hybrid system designers. Spacecraft control systems further intensify this challenge with extensive interaction between computing units and their physical environment and their mutual dependence on each other. On the other hand, designing a spacecraft control system is a complex engineering process, and therefore it is unlikely to demand engineers to apply formal methods in the whole process of design because of efficiency and cost. So, it is extremely necessary to have a way to combine formal and informal methods in the design so that the engineers can flexibly shift between formal and informal methods.

In order to efficiently develop reliable safety-critical systems, model-based design (MBD) has become a major approach in the design of computer controlled systems. Using this approach at the very beginning, an abstract model of the system to be developed is defined. Extensive analysis and verification on the abstract model are then conducted so that errors can be identified and corrected at the very early stage. Then the higher-level abstract model is refined to a lower-level abstract model step by step, until it can be composed with existing components. There have been huge number of MBD approaches proposed and used in industry and academia, e.g., *Simulink/Stateflow* [1,2], Modelica [41], SysML [3], MARTE [40], Metropolis [9], Ptolemy [20], hybrid automata [25], CHARON [6], HCSP [24,51], Differential Dynamic Logic [36], Hybrid Hoare Logic [29], etc. These approaches can be classified into two paradigms according to whether with a solid theoretical foundation, i.e., formal as [6,9,20,24,25,29,36,51] and informal as [1–3,40,41].

It is commonly known that engineering informal methods for designing hybrid systems are very efficient and cheap, but cannot guarantee the correctness and reliability; in contrast, formal methods can guarantee the correctness and reliability of the system to be developed, but pay in low efficiency and high cost. Therefore it is desirable to provide the designer with the ability to choose between formal or informal analysis depending on the degree of confidence in the correctness of the design required by the application.

In this chapter, we report our experience on combing informal and formal methods together in the design of spacecrafts. The framework of our approach is as follows:

- We first build executable models of hybrid systems using the industrial standard environment *Simulink/Stateflow*, which facilitates analysis by simulation.
- Then, to complement simulation, formal verification of *Simulink/Stateflow* models is conducted via the following steps:
  1. first, we translate *Simulink/Stateflow* diagrams to Hybrid CSP (HCSP) processes by an automatic translator *Sim2HCSP*;

2. second, to justify the translation, another automatic translator H2S 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;
3. then, the obtained HCSP processes in the first step are verified by an interactive Hybrid Hoare Logic (HHL) prover;
4. during the verification, synthesizing invariants for differential equations and loops is needed.

- Of course, as an alternative, we can construct an HCSP formal model at the beginning of the design first, and then simulate and/or test the formal model economically if formal verification is not necessary.

Simulink [1] is an environment for the model-based analysis and design of embedded control systems, which offers an intuitive graphical modeling language reminiscent of circuit diagrams and thus appealing to the practising engineer. *Stateflow* [2] is a toolbox adding facilities for modeling and simulating reactive systems by means of hierarchical statecharts, extending *Simulink*'s scope to event-driven and hybrid forms of embedded control. Modeling, analysis, and design using *Simulink*/*Stateflow* (S/S) have become a de-facto standard in the embedded systems industry.

S/S relies on extensive simulation based on unverified numerical computation to validate system requirements, which is prone to incomplete coverage of open systems and possible unsoundness of analysis results due to numerical errors. As a result, existing errors in the model might not be discovered through simulation. If such incorrectly developed systems are deployed then any undetected errors can potentially cause a catastrophic failure. In safety-critical applications the risk of such failures is regarded as unacceptable. Reducing these risks by formal verification would be desirable, complementing simulation. Motivated by this, in our previous work [48,53,54], we presented a formal method for "closed-loop" verification of safety properties of S/S models. This is achieved by automatically translating S/S diagrams into HCSP [24,51], a formal modelling language for hybrid discrete-continuous systems. As formal analysis of HCSP models is supported by an interactive Hybrid Hoare Logic (HHL) prover based on Isabelle/HOL [29,47,52], this provides a gateway to mechanized verification of S/S models. To justify the translation from S/S to HCSP, in [14], we investigated how to translate HCSP formal models into *Simulink* graphical models, so that the consistency between the original *Simulink*/*Stateflow* model and the translated HCSP formal model can be checked by co-simulation.

In addition, in practice, people may start to build a formal model as a starting point of designing a system, based on which formal analysis and verification are conducted. However, a formal model is not easy to be understood by a domain expert or engineer, and therefore is not easy to be validated. In particular, the cost for formal verification of a formal model is quite expensive. In fact, many errors can be detected by testing and/or simulation in an economical way. Thus, it deserves to translate a formal model into a *Simulink* model, so that validation can be achieved by simulation; furthermore, detecting errors can be done with simulation in an economical way.

So, HCSP formal models can be simulated and/or tested using *Matlab* platform economically, without expensive formal verification when it is not necessary. Together with the work on translating S/S diagrams into HCSP, it provides the designer of embedded systems the flexibility using formal and informal methods according to the trade-off between efficiency and cost, and correctness and reliability.

We have implemented a toolchain called MARS [13] to support the above approach. MARS integrates a set of tools, including an automatic translator from S/S into HCSP, and an automatic translator from HCSP into *Simulink*, an HHL theorem prover, an invariant generator for hybrid systems which provides the options to synthesize an invariant with symbolic computation or numeric computation, and an abstractor to abstract an elementary hybrid system by a polynomial hybrid system.

The above approach and tool have been successfully applied in the design of spacecrafts, and we will demonstrate it by applying the approach to the design of a descent guidance control program of a lunar lander. A preliminary version of these results has been reported elsewhere [48].

1.1 Synopsis

This chapter first summarizes our experience on the design of spacecrafts before, most of which are joint work with other people. Then, we argue that combining formal and informal methods can provide the flexibility in the design of spacecrafts, but we do not provide any further technical contribution. The main results we used are listed as follows:

- HHL is a joint work with Chaochen Zhou, Shuling Wang, Dimitar Guelev, Jiang Liu, Jidong Lv, Zhao Quan, Hengjun Zhao and Liang Zou in [29,44,46], which extends classical Hoare logic to hybrid systems;
- Invariant generation of hybrid systems is a joint work with Jiang Liu and Hengjun Zhao in [30–32];
- The translation from S/S is based on the joint work with Martin Fränzle, Shengchao Qin, Shuling Wang and Liang Zou published in [53,54];
- The translation from HCSP to *Simulink* is based on the joint work in [14] with Anders P. Ravn, Mingshuai Chen, and Liang Zou;
- The tool implementation is based on the joint work [13,45,52] with Mingshuai Chen, Shuling Wang, Liang Zou, Tao Tang, Xiao Han and Hengjun Zhao;
- The case study part is based on the joint work in [48] with Hengjun Zhao, Bin Gu and Yao Chen.

Paper Organization. The rest of this paper is organized as follows. Section 2 briefly reviews *Simulink*/*Stateflow*, HCSP and HHL. Section 3 establishes a connection between S/S informal models and HCSP formal models. Section 4 focuses on the explanation of the toolchain MARS. In Sect. 5, we demonstrate our approach by analysis and verification of the GNC control program of a lunar lander. Section 6 introduces the related work. Section 7 draws a conclusion and discusses the future work.

2 *Simulink*/*Stateflow*, HCSP and HHL

In this section, we briefly introduce the industrial de-facto graphical modeling language *Simulink*/*Stateflow*, the formal modeling language Hybrid CSP (HCSP), the specification language Hybrid Hoare Logic and its prover. The reader is referred to [1,2,47] for more details.

2.1 Simulink

A *Simulink* model contains a set of blocks, subsystems, and wires, where blocks and subsystems cooperate by message transmission through the wires connecting them. An elementary block receives input signals and computes the output signals, and meanwhile, it contains some user-defined parameters to alter its functionality. One typical parameter is *sample time*, which defines how frequently the computation is performed. According to sample time, blocks are classified into two types: *continuous blocks* with sample time 0, and *discrete blocks* with sample time greater than 0. Blocks and subsystems in a *Simulink* model receive inputs and compute outputs in parallel, and wires specify the data flow between them.

Figure 1 gives a *Simulink* model of train movement, comprising four blocks, including continuous blocks v and p, that are *integrator* blocks of the *Simulink* library, and discrete blocks c and *acc*. The block v outputs the velocity of the train, which is the time integral of the input acceleration from *acc*; similarly, p outputs the distance of the train, which is the time integral of the input velocity from v, and *acc* outputs the acceleration computed according to the constant provided by c and the input distance from p.

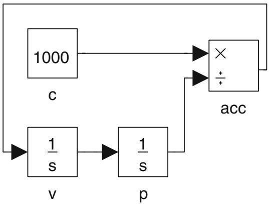

Fig. 1. A simple control system

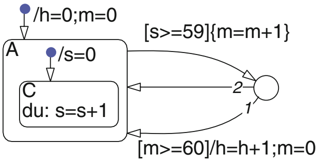

Fig. 2. A timer

2.2 Stateflow

As a toolbox integrated into *Simulink*, *Stateflow* offers the modeling capabilities of statecharts for reactive systems. It can be used to construct *Simulink* blocks, which can be fed with *Simulink* inputs to produce *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. In the following, we will explain the main ingredients of *Stateflow* and their intuitive meaning respectively.

Alphabet: The alphabet of a *Stateflow* diagram consists of a finite set of events and variables. An event can be an input or output of a diagram, which may be local to the diagram. A variable may also be set as input, output, or local, and moreover, it can be associated with an initial value if necessary.

States: A state describes an operating mode, possibly *active* or *inactive*. A state could be hierarchical, containing another *Stateflow* diagram inside. Because of hierarchy, transitions originating from a state are classified into two types depending on whether or not their target states are inside the same state: ingoing and outgoing transitions. All transitions are ordered by a strict priority so that there is no non-determinism in transition selection.

A state may be associated with three types of actions (all are optional): *entry action*, that is executed when the state is activated; *during action*, that is executed when no valid transition is enabled; and *exit action*, that is executed when a valid transition leaves from the state, and as a consequence the state becomes inactive. The actions of *Stateflow* may be either assignments, or emissions of events, etc.

States in an AND diagram must be specified with different priorities, that determine the order of their executions. The parallel states are actually executed in sequential order according to their priority.

Transitions: A complete transition is a path from source state to target state. In *Stateflow*, a complete transition may consists of several transition segments by joining connective junctions, which form a transition network from source state to target state. A connective junction is a graphical object to connect different transition segments, but itself can not be seen as a source state nor a target state of a complete transition. Each transition segment is of the form $E[C]\{cAct\}/tAct$, where E is an event, C is the guard condition, $cAct$ the condition action, and $tAct$ the transition action. All these components are optional. $cAct$ will be executed immediately when event E is triggered and condition C holds, while $tAct$ will be put in a queue first and be executed after the corresponding transition is taken.

Default transitions with no source states or source junctions are allowed for OR diagrams, and they are used to choose an active state when an OR diagram is activated.

Next we explain intuitively how a *Stateflow* diagram is executed.

Initialization: Initially, the whole system is activated: for an AND diagram, all the parallel states are activated according to the priority order; and for an OR diagram, one of the states is activated by performing the default transition.

Broadcasting and Executing Transition: Each *Stateflow* diagram is activated either by sampling time periodically or by triggering events, depending on the user-settings. For the second case, as soon as one of the triggering event arrives, called *current event*, the event will be broadcasted through the whole diagram. For an AND diagram, the event will be broadcasted sequentially to the parallel states inside the diagram according to the priority order over states; while for an OR diagram, it will find out the active state of the diagram (i.e. the one with the default transition) and broadcast the event to it. It will then check the outgoing transitions of the current active state according to the priority order, and if there is one valid transition that is able to reach a state, the transition will be taken; otherwise, check the ingoing transitions in the same way. If there is neither an outgoing nor an ingoing valid transition enabled, the during action of the state will be executed, and then the event is broadcasted recursively to the sub-diagram inside the state. The transition might connect states at different levels in the hierarchical diagram. When a transition connecting two states is taken, it will first find the common ancestor of the source and target states, i.e. the nearest state that contains both of them inside, then perform the following steps: exit from the source state (including its sub-diagram) step by step and at each step execute the exit action of the corresponding state and set it to be *inactive*, and then enter step by step to the target state (including its sub-diagram), and at each step, set the corresponding state to be *active* and execute the corresponding entry action.

Example 1. Fig. 2 gives an example of *Stateflow*. The states A and C are activated initially, so variables h, m, and s are set to 0. A has a transition network to itself, which becomes enabled when s equals to 59. Once the transition network is enabled, the outgoing transition is executed, and thus m is increased by 1; then it will execute transition 1 as it is with a higher priority by increasing h by 1 and resetting m to 0 if m equals to 60, otherwise, execute transition 2.

Note that s is reset to 0 whenever the transition network becomes enabled, as the sub-diagram of A is initialized again.

Combination of Simulink and Stateflow. How *Simulink* and *Stateflow* work together is exemplified by using the two examples in Figs. 1 and 2. In order to implement the block *acc* in Fig. 1, we revise the *Stateflow* diagram in Fig. 2 as follows: We add a condition action $[True]\{acc = 1000/p + m/100\}$ to transition 2 of the *Stateflow* diagram, meaning that the acceleration of the train is updated every minute and the new acceleration is calculated as $1000/p+m/100$. We then replace blocks *acc* and c by the modified stateflow diagram, which inputs p from the simulink diagram and then calculates and outputs the acceleration *acc* back to the simulink diagram.

2.3 Hybrid CSP (HCSP)

HCSP is an extension of Hoare's Communicating Sequential Processes for modeling hybrid systems [24,51]. In HCSP, differential equations are introduced to model continuous evolution of the physical environment along with interrupts. The set of variables is denoted by $\mathcal{V} = \{x, y, z, ...\}$ and the set of channels is denoted by $\mathcal{C} = \{ch_1, ch_2, ch_3, ...\}$. The processes of HCSP are constructed as follows:

$$\begin{aligned} P ::= & \text{skip} \mid x := e \mid\mid \text{wait } d \mid ch?x \mid ch!e \mid P; Q \mid B \rightarrow P \mid P \sqcup Q \mid X \mid \\ & \mu X.P \mid \langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \mid \langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \trianglerighteq [\!]_{i \in I}(io_i \rightarrow Q_i) \mid \\ & \langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \mid \langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \trianglerighteq_d Q \end{aligned}$$

$$S ::= P \mid S \| S$$

Here, P, Q, and Q_i represent sequential processes, whereas S stands for a (sub)system; $ch, ch_i \in \mathcal{C}$ are communication channels; while ch_i* is a communication event which can be either an input event $ch?x$ or an output event $ch!e$; B and e are the Boolean, and arithmetic expressions, respectively; and d is a non-negative real constant.

Process skip terminates immediately without updating variables, and process $x := e$ assigns the value of expression e to variable x and then terminates. Process *wait* d keeps idle for d time units without changing the variables. Interaction between processes is based on two types of communication events: $ch!e$ sends the value of e along channel ch, and $ch?x$ assigns the value received along channel ch to variable x. Communication takes place when both the source and the destination processes are ready.

A sequentially composed process $P; Q$ behaves as P first, and if it terminates, as Q afterward. The alternative process $B \rightarrow P$ behaves as P only if B is true and terminates otherwise. Internal choice between processes P and Q denoted as $P \sqcup Q$ is resolved by the process itself. Communication controlled external choice $[\!]_{i \in I}(ch_i* \rightarrow Q_i)$ specifies that as soon as one of the communications ch_i* takes place, the process starts behaving as process Q_i. The repetition P^* executes P for an arbitrary finite number of times, and the choice of the number of times is non-deterministic.

Continuous evolution is specified as $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle$. The real variable s evolves continuously according to differential equations $\mathcal{F}$ as long as the Boolean expression B is true. B defines the domain of s. Interruption of the continuous evolution due to B (as soon as it becomes false) is known as *Boundary Interrupt*. The continuous evolution can also be preempted due to the following interrupts:

- *Timeout Interrupt*: $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \trianglerighteq_d Q$ behaves like $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle$, if the continuous evolution terminates before d time units. Otherwise, after d time units of evolution according to $\mathcal{F}$, it behaves as Q.

- *Communication Interrupt*: $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \trianglerighteq []_{i \in I}(ch_i* \rightarrow Q_i)$ behaves like $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle$, except that the continuous evolution is preempted whenever one of the communications ch_i* takes place, which is followed by respective Q_i.

Finally, S defines an HCSP system on the top level. A parallel composition $S_1 \| S_2$ behaves as if S_1 and S_2 run independently, except that they need to synchronize along the common communication channels. The concurrent processes can only interact through communication, and no shared variables are allowed. A detailed explanation can be found in [47].

2.4 Hybrid Hoare Logic

In [29], classical Hoare Logic was extended to hybrid systems, called *Hybrid Hoare Logic* (HHL). In HHL, a hybrid system is modeled by HCSP process. 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) [49,50], called history formulas, for specifying the execution history for continuous processes. In HHL, a specification for a sequential process P is of the form $\{Pre\}\, P\, \{Post; HF\}$, where $Pre, Post$ represent precondition and postcondition, respectively, and are expressed by FOL to specify properties of variables held at starting and termination of the execution of P. HF is a history formula to record the execution history of P, including its real-time and continuous properties. The specification for a parallel process is then defined by assigning to each sequential component the respective precondition, postcondition, and history formula, that is

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

A proof system for HHL was provided in [29]. In particular, the notion of differential invariant [30,37] is used to characterize the behavior of differential equations.

HHL Prover. For tool support, we have implemented an interactive theorem prover for HHL based on Isabelle/HOL, please refer to [45,47,52] for more details.

3 Connection Between Informal and Formal Models

In this section, we show how to link informal and formal models via a translation from *Simulink/Stateflow* to HCSP and an inverse translation from HCSP to *Simulink*.

3.1 From *Simulink/Stateflow* to HCSP

3.1.1 Translating *Simulink*

The behavior of any block can be divided into a set of sub-behaviors, each of which is guarded by a condition. Moreover, these guards are mutually exclusive and complete, i.e., the conjunction of any two of them is unsatisfiable and the disjunction of them is valid. Hence, blocks can be interpreted by a transformation predicate over inputs and outputs as follows:

$$\mathit{Seman}_B(\mathit{init}, \mathit{ps}) \quad \widehat{=} \quad \mathit{out}(0) = \mathit{init} \wedge \bigwedge_{k=1}^{m} (B_k(\mathit{ps}, \mathit{in}) \Rightarrow P_k(\mathit{ps}, \mathit{in}, \mathit{out})), \quad (1)$$

where *init* stands for the initial output value set by user, *ps* are the user-set parameters that may change the function of the block, *in* and *out* are resp. the timed traces corresponding to input and output signals, $\mathit{out}(0)$ is the value of *out* at time 0. In the definition we assume that the block's behavior is split into m cases by B_k and in each case the behavior is specified by the corresponding predicate P_k. Additionaly, $\bigvee_{k=1}^{m} B_k(\mathit{ps}, \mathit{in})$ is valid, and $B_i(\mathit{ps}, \mathit{in}) \wedge B_j(\mathit{ps}, \mathit{in})$ is unsatisfiable for any $i \neq j$.

So, the semantics of a *Simulink* diagram is defined by

$$\mathit{Seman}_D \quad \widehat{=} \quad \bigwedge_{j=1}^{n} \mathit{Seman}_B(\mathit{init}_j, \mathit{ps}_j)\,, \quad (2)$$

where n is the number of blocks in the diagram, init_j and ps_j are the initial output value and parameters of the j-th block.

Notice that different types of blocks, i.e. continuous and discrete blocks, have different definitions for B_k and P_k because the input signals for discrete blocks only refer to the value of the closest sample time point, i.e. the value of input signals at time t should refer to the time $(t - (t \mod \mathit{st}))$ where st represents the sample time of the block.

Blocks

For a continuous block, its initialization is simply encoded as an assignment. A continuous block uses its B_ks as a partition of the whole state space, and continuously evolves following some differential equation $\mathcal{F}_k$ subject to the corresponding formula B_k. During the continuous evolution, the block is always ready for receiving new signals from in-ports, and sending the respective signals to out-ports (represented by io_i). Based on the continuous sample time, the blocks which receive signals from the continuous block via out-ports can always get the latest values. So, a continuous block can be encoded into the following process pattern:

$$\begin{aligned}
&\mathcal{PC}(\mathit{init}, \mathit{ps}) \;\widehat{=}\; \mathit{out} := \mathit{init}; P^* \\
&P \;\widehat{=}\; \langle \mathcal{F}_1(\dot{\mathit{out}}, \mathit{out}, \mathit{in}, \mathit{ps}) = 0 \& B_1(\mathit{in}, \mathit{ps})\rangle \trianglerighteq []_{i\in I}(io_i \rightarrow \text{skip}); \\
&\qquad \ldots; \\
&\qquad \langle \mathcal{F}_m(\dot{\mathit{out}}, \mathit{out}, \mathit{in}, \mathit{ps}) = 0 \& B_m(\mathit{in}, \mathit{ps})\rangle \trianglerighteq []_{i\in I}(io_i \rightarrow \text{skip})
\end{aligned}$$

For a discrete block, its initialization is also encoded as an assignment. However, a discrete block with sample time *st* only computes output signals at the time points whose values minus the initial time are divided by *st*, i.e. once every *st* time units. At the beginning of each period, it updates the input signal by receiving a new one from in-port, and after the computation, sends the new produced output signal to the out-port. Thus, the blocks which receive signals from the discrete block can always get the values of the last nearest period. Finally, a discrete block can be encoded as follows:

$$\begin{aligned}
\mathcal{PD}(init, ps, n) &\mathrel{\widehat{=}} out := init; P^* \\
P &\mathrel{\widehat{=}} cin?in; P_{comp}; cout!out; \text{wait}\ st \\
P_{comp} &\mathrel{\widehat{=}} B_1(in, ps) \rightarrow P_{comp_1}(in, out, ps); ...; \\
&\quad\ B_m(in, ps) \rightarrow P_{comp_m}(in, out, ps)
\end{aligned}$$

Diagrams

A diagram is translated into an HCSP process via the following steps:

Step 1: *Computing inherited sample times.* A *Simulink* diagram may contain blocks with unspecified sample time, which is called *inherited* and is indicated with value -1. An inherited sample time of a block is determined when the sample times of all the input signals of the block are known, and then it is computed as the greatest common divisor (GCD) of the sample times of these input signals.

Step 2: *Translating wires.* In general, wires in *Simulink* diagrams can be considered as a special form of signals, and thus can be represented as variables. In addition, when a diagram is partitioned into a set of sub-diagrams, we will model a wire between any two sub-diagrams as a pair of input and output channels for transmitting values.

Step 3: *Separating a diagram to a set of connected sub-diagrams.* We first classify wires to three categories: from continuous to continuous, from continuous to discrete (from discrete to continuous), and from discrete to discrete; and then partition a diagram to a set of largest connected blocks with the same type (that is either continuous or discrete) according to the following strategy:

(1) Wires between continuous blocks are modelled as shared variables, and hence, the two continuous blocks are put into one partition;

(2) Wires between a continuous block and a discrete block are modelled as channels, and thus, these two blocks are put into two disjoint partitions, and will transmit values via the channels;

(3) Wires between discrete blocks are hard to model because the control represented by the blocks may be centralized or distributed. In our approach, a control is assumed as centralized by default, and in this case, the wires between the discrete blocks are modelled as shared variables; and therefore, the two blocks are put in one partition. Please note that the general case in which the user options for control are allowed will be discussed later.

Step 4: *Translating each resulting continuous sub-diagram.* First, we collect all initialization parts of these continuous blocks in the continuous sub-diagram and put them in sequence as the initialization part; second, collect all communications happening in these continuous blocks and union them together as the communication part; third, cartesian the differential equations in these continuous blocks as the continuous evolution part, then construct a communication interruption by setting that the continuous evolution is interrupted by the communication part; finally, put the initialization part and the communication interruption in sequence.

Step 5: *Translating each resulting discrete sub-diagram.* As in the continuous case, we treat the sub-diagram as a discrete block. So, we first collect all initialization parts, inputs and outputs from the HCSP processes corresponding to these discrete blocks in the discrete sub-diagrams, and respectively put them in sequence according to the order of these blocks as the corresponding initialization, input and output in the final HCSP process for the sub-diagram; then we compute the greatest common divisor t of the sample times of these blocks as the sample time of the block; third, we update each computation part of these discrete block by letting it be computed every t time units, and then put all the updated computation parts in sequence together with the input and output to form the computation part of the block; finally, we introduce a timer to guarantee the computation part is executed periodically with period t.

Subsystems

A subsystem consists of a set of blocks, diagrams, and other subsystems. So, a system can be modeled hierarchically in *Simulink* with subsystems. In *Simulink*, there are three types of subsystems, i.e., *normal subsystems*, *triggered subsystems* and *enabled subsystems*. In the following, we show how to translate them into HCSP.

- A *normal subsystem* contains neither triggered nor enabled blocks inside. For this case, we flatten the subsystem directly by connecting the in-ports and out-ports attached to it to the corresponding in-ports and out-ports attached to the blocks inside it. The subsystem plus the outside blocks connected to it will then be reduced to a diagram, which can be translated as above.
- A *triggered subsystem* contains a triggered block inside it, and meanwhile, there is a corresponding input triggering signal targeting at the subsystem. The sample times of all the other input signals of the subsystem are equal to the one of the triggering signal. All the blocks except for the triggered block (called as normal blocks hereafter) inside the subsystem have unspecified sample time -1. They constitute a diagram, and will be activated by the trigger events. According to the change of the triggering signal, there are three types of *trigger events*: the rising, falling and changing of the sign of the triggering signal. Whenever a trigger event occurs, all the normal blocks inside the subsystem will be performed once. We flatten the rest of the triggered

subsystem except for the triggering signal and the triggered block, and then apply the above procedure to translate the resulting diagram. Taking the triggering signal into account, the computation part *procR* is revised by

$$procR \leftarrow tri?; cin; procR; cout,$$

where *tri* represents the input triggering signal, indicating that the computation of the subsystem will be activated by signal *tri*? from outside.
Meanwhile, we revise the translation of the outside block that outputs the triggering signal depending on its type as follows:

- *Discrete.* In this case, the computation part P_{comp} is replaced by the following process

$$osig := out_{tri}; P_{comp}; B_{tri}(osig, out_{tri}) \rightarrow tri!$$

  In which, we introduce a variable *osig* to record the output signal of last period at the beginning (here out_{tri} is used to represent the triggering signal); then after the computation part P_{comp} is performed, we compare the old signal *osig* and the new output signal out_{tri}. If they satisfy the condition B_{tri} for triggering an event, then a triggering event *tri*! occurs. The definition of B_{tri} depends on the triggering type, for instance, if the triggering signal is rising,

$$B_{tri}(osig, out_{tri}) \widehat{=} osig < 0 \wedge out_{tri} \geq 0 \ \vee osig \leq 0 \wedge out_{tri} > 0$$

- *Continuous.* In this case, the differential equation part in P_{comp} is replaced by the following process

$$\begin{aligned}
&\langle \mathcal{F}_1(\dot{out}, out) = 0 \& B_1 \wedge \neg B_{tri} \rangle \trianglerighteq \cdots ;\\
&\quad \ldots\\
&\langle \mathcal{F}_m(\dot{out}, out) = 0 \& B_m \wedge \neg B_{tri} \rangle \trianglerighteq \cdots ;\\
&\quad B_{tri} \rightarrow tri!;\\
&\langle \mathcal{F}_1(\dot{out}, out) = 0 \& B_1 \wedge B_{tri} \rangle \trianglerighteq \cdots ;\\
&\quad \ldots\\
&\langle \mathcal{F}_m(\dot{out}, out) = 0 \& B_m \wedge B_{tri} \rangle \trianglerighteq \cdots
\end{aligned}$$

  where B_{tri} defines the condition for occurring a triggered event, in particular for the rising case, it can be defined as $out_{tri} = 0 \wedge \dot{out}_{tri} > 0$, i.e. the value of the output signal is 0 and its first derivative is greater than 0. As soon as B_{tri} holds, the event *tri*! occurs, and then the process continuously evolves according to the differential equations of the block, till next time the trigger event occurs, when B_{tri} turns from false to true again.

– An *Enabled subsystem* P_{comp} contains an enabled block inside it, and meanwhile, there is a corresponding input enabling signal targeting at the subsystem. The blocks except for the enabled block (i.e. normal blocks) inside the enabled subsystem can be continuous or discrete, and whenever the input signal is greater than 0, they will be activated.

For both continuous and discrete cases, we model the wire connecting the block that outputs the enabling signal and the enabled subsystem as a shared variable *en*. When both the enabling signal and the enabled subsystem are continuous, first of all, for each normal block inside the subsystem, we add $en > 0$ as a conjunction with the domains of all its differential equations, and meanwhile, add an extra differential equation $\langle \dot{out} = 0 \& en \leq 0\rangle$ (meaning that the output is not changed when the signal is not enabled) to the block, thus the new domains for the block will be complete; then flatten the enabled subsystem, the resulting diagram plus the outside output block will constitute a new continuous diagram, which can be translated as above.

When both the enabling signal and the enabled subsystem are discrete and have the sample time, first of all, for each normal block inside the subsystem, we add the enabling condition $en > 0$ as a conjunction with the guards of the computation of the block; then flatten the enabled subsystem, the resulting diagram plus the outside output block will constitute a new discrete diagram, which can be translated as above.

The detail of the translation from *Simulink* to HCSP can be found in [54].

3.1.2 Translating *Stateflow*

A *Stateflow* diagram is translated as a process template $\mathcal{D}$, which is a parallel composition of the monitor process $\mathcal{M}$ and the parallel states $\mathcal{S}_1, \cdots, \mathcal{S}_n$ of the diagram, with the following form

$$\mathcal{D} \;\widehat{=}\; \mathcal{M} \| \mathcal{S}_1 \| \cdots \| \mathcal{S}_n.$$

The monitor process $\mathcal{M}$ is an HCSP process, which monitors the broadcasting of the event among the states $\mathcal{S}_i$. Each $\mathcal{S}_i$ is also an HCSP process, which is the encoding of the corresponding state in the *Stateflow* diagram. When the diagram is an OR diagram, n will be 1, and the only state $\mathcal{S}_1$ corresponds to the virtual state that contains the diagram, which has neither (entry/during/exit) action nor transition associated to it.

$\mathcal{S}_i$ is an HCSP process corresponding to the i-th state. $\mathcal{S}_i$ first initializes the local variables of the state and activates the state by executing the entry action, defined by P_{init} and P_{entry} respectively; then it is triggered whenever an event E is emitted by the monitor $\mathcal{M}$ possibly with the shared data, and performs the following actions: first, initializes *done* to *False* indicating that no valid transition has been executed yet, and searches for a valid transition starting from $\mathcal{S}_i$ by calling a depth-first algorithm **TTN**; if *done* is still false, then executes the during action *dur* and all of its sub-diagrams. Note that for an OR diagram, the execution of the virtual state is essentially to execute the sub-diagram directly; finally, notifies the monitor the completion of the broadcasting and outputs the shared data.

Likewise, each sub-diagram (represented by P_{diag}) may be AND or OR sub-diagram. Different from the AND diagram at the outermost, for simplicity, we define the AND sub-diagram as a sequential composition of its parallel states.

This is reasonable because there is no true concurrency in *Stateflow* and the parallel states are actually executed in sequence according to their priorities. The OR diagram is encoded as a sequential composition of the connecting states, guarded by a condition $a_{S_i} == 1$ indicating that the i-th state is active. In a word, $\mathcal{S}_i$ can be represented by the following HCSP process:

$$\begin{aligned}
\mathcal{S}_i &\mathrel{\widehat{=}} P_{init}; P_{entry}; (BC_i?E;\ VOut_i?sv_i; \mathcal{S}_{du}; BO_i!;\ VIn_i!sv_i)^*,\\
\mathcal{S}_{du} &\mathrel{\widehat{=}} done = False; \mathbf{TTN}(\mathcal{S}_i, E, done); \neg done \rightarrow (dur; P_{diag}),\\
P_{diag} &\mathrel{\widehat{=}} P_{and} \mid P_{or},\\
P_{and} &\mathrel{\widehat{=}} \mathcal{S}_{1_{du}}; \cdots; \mathcal{S}_{m_{du}},\\
P_{or} &\mathrel{\widehat{=}} (a_{S_1} == 1 \rightarrow \mathcal{S}_{1_{du}}); \cdots; (a_{S_k} == 1 \rightarrow \mathcal{S}_{k_{du}}).
\end{aligned}$$

Note that in the above, **TTN** returns an HCSP process corresponding to both outgoing and ingoing transitions from/to $\mathcal{S}_i$. In **TTN**, local events may be emitted, e.g. during executing actions of transitions or states. For such case, the current execution of the diagram needs to be interrupted by broadcasting the local event, and after the broadcasting is completed, the interrupted execution will be resumed.

The monitor process $\mathcal{M}$ in terms of HCSP coordinates the execution of broadcasted events. When an event is broadcasted, an OR diagram will broadcast the event to its active state, while an AND diagram will broadcast the event to each of its sub-diagrams according to the priority order. During the broadcasting, a new local event may be emitted inside some sub-diagram, and thus current execution will be interrupted by the local event. After the completion of the local event, the interrupted execution will be resumed. $\mathcal{M}$ can be defined by the following HCSP process:

$$\begin{aligned}
\mathcal{M} &\mathrel{\widehat{=}} num := 0; (\mathcal{M}_m)^*\\
\mathcal{M}_m &\mathrel{\widehat{=}} (num == 0) \rightarrow (P_{tri}; CH_{in}?iVar; num := 1; EL := [\,]; NL := [\,];\\
&\qquad\qquad\qquad \mathbf{push}(EL, E); \mathbf{push}(NL, 1));\\
&\quad (num == 1) \rightarrow (BC_1!E;\ VOut_1!sv[\!](BR_1?E; \mathbf{push}(EL, E); \mathbf{push}(NL, 1); num := 1)\\
&\qquad\qquad\qquad [\!](BO_1?;\ VIn_1?sv; num := num + 1; \mathbf{pop}(NL); \mathbf{push}(NL, num));\\
&\quad \ldots\\
&\quad (num == n) \rightarrow (BC_n!E;\ VOut_n!sv[\!](BR_n?E; \mathbf{push}(EL, E); \mathbf{push}(NL, 1); num := 1)\\
&\qquad\qquad\qquad [\!](BO_n?;\ VIn_n?sv; num := num + 1; \mathbf{pop}(NL);\ \mathbf{push}(NL, num));\\
&\quad num == n + 1 \rightarrow (\mathbf{pop}(EL); \mathbf{pop}(NL); \mathbf{isEmpty}(EL) \rightarrow (num := 0;\ CH_{out}!oVar);\\
&\quad \neg\mathbf{isEmpty}(EL) \rightarrow (E := \mathbf{top}(EL); num := \mathbf{top}(NL)))),
\end{aligned}$$

where P_{tri} stands for the process corresponding to the triggered event, $CH_{in}?iVar$ for receiving the input of the triggered event, $CH_{out}!oVar$ for sending out the update during broadcasting the event, n for the number of parallel states of current diagram, E for current event, num for the sub-diagram to which current event is broadcasted. EL and NL are two stacks respectively to store the broadcasted events and the corresponding sub-diagrams to which these events are broadcasted.

Advanced features of *Stateflow* can also be handled well by HCSP, please see [53] for the detail.

3.1.3 Translating Combination of *Simulink* and *Stateflow*

Given a *Simulink/Stateflow* model, its *Simulink* and *Stateflow* parts are translated by using procedures in Sects. 3.1.1 and 3.1.2 respectively, and then put the resulting HCSP processes in parallel to form the whole model of the system. The *Simulink* and *Stateflow* diagrams in parallel transmit data or events via communications. The communications between them are categorized into the following cases:

- The input (and output) variables from (and to) *Simulink* will be transmitted through the monitor process to (and from) *Stateflow*;
- The input events from *Simulink* will be passed via the monitor to *Stateflow*;
- The output events (i.e. the ones occurring in $\mathcal{S}_1, \cdots, \mathcal{S}_n$ in the *Stateflow* diagram) will be sent directly to *Simulink*;
- The input/output variables and events inside *Simulink* part are handled as in Sect. 3.1.1

Please see [53] for the detail.

3.2 From HCSP to *Simulink*

In [14], we present a translation from HCSP to *Simulink* as an inverse procedure of the translation from *Simulink/Stateflow* to HCSP. The basic idea is to define an operational semantics for HCSP using *Simulink*. This means that everything in an HSCP model must be represented in *Simulink*. The latter is constituted from subsystems and therefore even arithmetic or Boolean expressions which are incorporated in HSCP must be translated to a *Simulink* subsystem in a consistent manner. For example, it is a natural way to define the meaning of any arithmetic (Bolean) expression as a normal subsystem, for instance, Fig. 3 is a *Simulink* subsystem corresponding to $x - 1 + y * ((-2)/3.4)$.

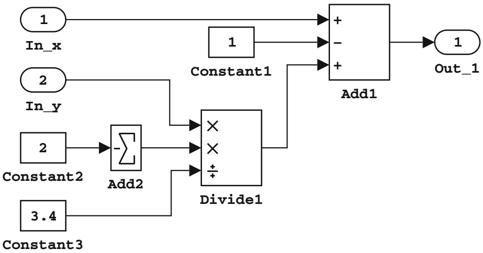

Fig. 3. $x - 1 + y * ((-2)/3.4)$

For modeling sequential composition, inspired by UTP [26], we therefore introduce a pair of Boolean signals *ok* and *ok*$'$ into each subsystem, which is translated from an HCSP process, to indicate the relevant initiation and termination. If *ok*$'$ is false, the process has not terminated and the final values of the process variables are unobservable. Similarly, if *ok* is false, the process has never started and even the initial values are unobservable. Additionally, *ok* and ok' are local to each subsystem corresponding to an HCSP process, which never occur in the process text. Furthermore, *ok* and ok' in a *Simulink* subsystem are constructed as an in-port signal named In_*ok* and an out-port signal named Out_*ok* respectively. For example, the semantics of skip is defined by a subsystem given in Fig. 4.

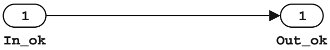

Fig. 4. Skip statement

The translation of a continuous evolution of HCSP is very involved, which is shown in Fig. 5, where the group of differential equations $\mathcal{F}$ and the Boolean condition B are encapsulated into a single subsystem respectively. The enabled subsystem F contains a set of integrator blocks corresponding to the vector s of continuous variables, and executes continuously whenever the value of the input signal, abbreviated as *en*, on enable-port is positive. Intuitively, subsystem B guards the evolution of subsystem F by taking the output signals of F as its inputs, i.e. $s_B = s'_F$, and partially controlling the enable signal of F via its output Boolean signal, denoted by B. As a consequence, an algebraic loop occurs between subsystem B and F which is not allowed in *Simulink*, and a plain solution is to insert an unit delay block with an initial value 1 insert after subsystem B.

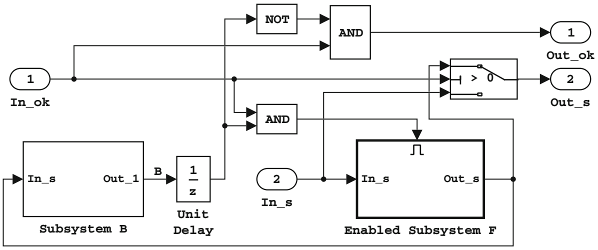

Fig. 5. Continuous Evolution

The full description and justification of the translation can be found in [14].

4 Tool Implementation

We have implemented all the above theories and integrated them as a toolchain named *MARS* for Modelling, Analyzing and veRifing spacecraft control

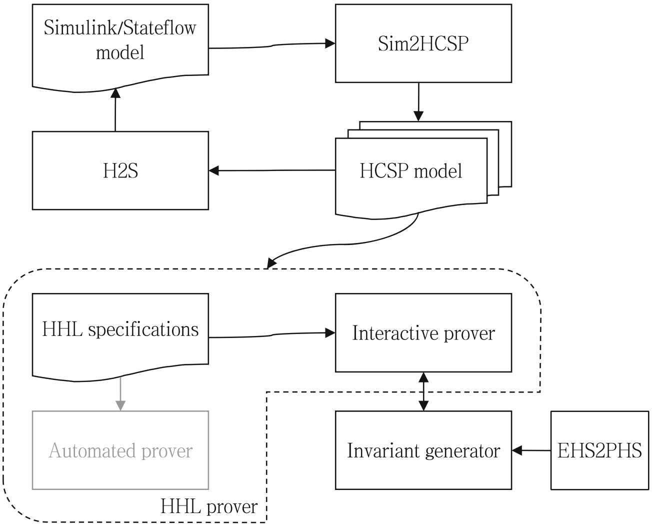

Fig. 6. Verification architecture

systems [13]. As shown in Fig. 6, the architecture of MARS is composed of three parts: a linking between informal and formal models, consisting of a translator Sim2HCSP from *Simulink/Stateflow* to HCSP and a translator from HCSP to *Simulink*, an HHL prover, and an invariant generator.

The translator Sim2HCSP is designed to translate *Simulink/Stateflow* models to HCSP. By applying Sim2HCSP, the translation from *Simulink/Stateflow* to HCSP is fully automatic, and to justify its correctness, another automatic inverse translator H2S is implemented. We use H2S 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. HHL prover is a theorem prover for Hybrid Hoare Logic (HHL) [29]. As the input of 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 to specify the initial and terminating states of the process, and a history formula to record the whole execution history of the process, respectively. HHL defines a set of axioms and inference rules to deduce such specifications. Finally, by applying 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 verify differential equations, we use the concept *differential invariants* to characterize their properties without solving them [30]. For computing differential invariants, we have implemented an independent invariant generator, which will be called during the verification in HHL prover. The invariant generator integrates both the quantifier elimination and SOS based methods for computing differential invariants of polynomial equations, and can also deal with non-polynomial systems by transformation techniques we proposed [32], which is implemented as EHS2PHS in Fig. 6.

5 A Case Study: Analysis and Verification of a Descent Guidance Control Program of a Lunar Lander

5.1 Description of the Verification Problem

At the end of 2013, China launched a lunar lander to achieve its first soft-landing and roving exploration on the moon. After launching, the lander first entered an Earth-Moon transfer orbit, then a 100 km-high circular lunar orbit, and then a 15 km × 100 km elliptic lunar orbit. At perilune of the elliptic orbit, the lander's variable thruster was fired to begin the powered descent process, which can be divided into 6 phases. As shown in Fig. 7, the terminal phase of powered descent is the slow descent phase, which should normally end several meters above the landing site, followed by a free fall to the lunar surface. One of the reasons to shut down the thruster before touchdown is to reduce the amount of stirred up dust that can damage onboard instruments.

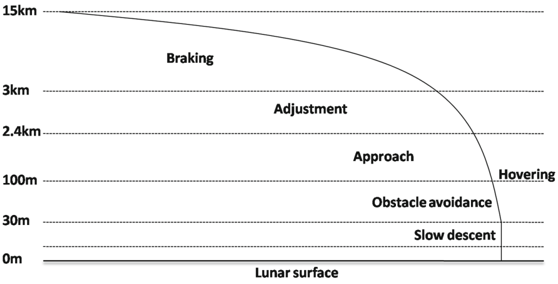

Fig. 7. The powered descent process of the lunar lander.

Powered descent is the most challenging task of the lunar lander mission because it is fully autonomous. Due to communication delay, it is impossible for stations on earth to track the rapidly moving lander, and remote control commands from earth cannot take effect immediately. The lander must rely on its own guidance, navigation and control (GNC) system to, in real time, acquire its current state, calculate control commands, and use the commands to adjust its attitude and engine thrust. Therefore the reliable functionality of the GNC system is the key to the success of soft-landing.

Clearly, the powered descent process of the lander gives a specific hybrid system (HS), i.e. a sampled-data control system composed of the physical plant and the embedded control program, which forms a closed-loop with the following prominent features: (1) the physical dynamics is modelled by ordinary differential equations (ODEs) with general elementary functions (rational, trigonometric, exponential functions etc.); (2) the program has complex branching conditions and numerical computations; (3) the physical process is frequently interrupted by control inputs from the program; (4) the system suffers from various uncertainties. Due to the high complexity, analysis and verification of such a system is very hard and beyond the capacity of many existing verification tools.

As a case study, we show how to apply the above approach to analysis and verification by focusing on one of the 6 phases, i.e. the slow descent phase, of the powered descent process. Through such verification, trustworthiness of the lunar lander's control program is enhanced. According to the framework of our approach, analysis and verification procedure can be outlined as follows:

(1) we first build a *Simulink*/*Stateflow* model of the closed-loop system and analyze its behaviour by simulation;
(2) then, with the tool Sim2HCSP [53,54], the *Simulink*/*Stateflow* graphical model is automatically translated to a formal model given by HCSP;
(3) subsequently, to justify the above translation, using the translator H2S [14], the resulted HCSP process is translated to a *Simulink* diagram inversely, so that the consistency between the original *Simulink*/*Stateflow* model and the translated HCSP model can be checked by co-simulation;
(4) finally, a formal verification of the system is conducted using HHL Prover [52]. During the verification, we need to call the tool EHS2PHS [32] first to abstract the considered elementary hybrid system to a polynomial hybrid system, and then exploit the tool invariant generator [30] to synthesize an invariant of the obtained polynomial HS.

All the above procedure is fully supported by the toolchain MARS [13].

5.1.1 Overview of the Slow Descent Phase

The slow descent phase begins at an altitude (relative to lunar surface) of approximately 30 m and terminates when the engine shutdown signal is received. The task of this phase is to ensure that the lander descends slowly and smoothly to the lunar surface, by nulling the horizontal velocity, maintaining a prescribed uniform vertical velocity, and keeping the lander at an upright position. The descent trajectory is nearly vertical w.r.t. the lunar surface (see Fig. 8).

The operational principle of the GNC system for the slow descent phase (and any other phases) can be illustrated by Fig. 9. The closed loop system is composed of the lander's dynamics and the guidance program for the present phase. The guidance program is executed periodically with a fixed sampling period. At each sampling point, the current state of the lander is measured by IMU (inertial measurement unit) or various sensors. Processed measurements are then input into the guidance program, which outputs control commands, e.g. the magnitude and direction of thrust, to be imposed on the lander's dynamics in the following sampling cycle.

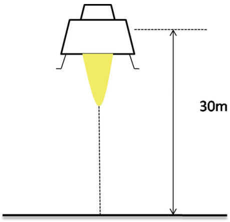

Fig. 8. The slow descent phase.

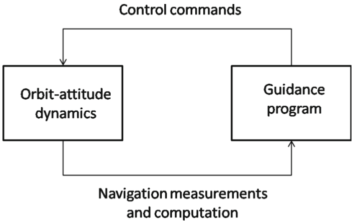

Fig. 9. A simplified configuration of GNC.

We next give a mathematical description of the lander's dynamics as well as the guidance program of the slow descent phase. For the purpose of showing the technical feasibility and effectiveness of formal methods in the verification of aerospace guidance programs, we neglect the attitude control as well as the orbit control in the horizontal plane, resulting in a one-dimensional (the vertical direction) orbit dynamics.

Dynamics. Let the upward direction be the positive direction of the one-dimensional axis. Then the lander's dynamics is given by

$$\begin{cases} \dot{r} &= v \\ \dot{v} &= \frac{F_c}{m} - gM \\ \dot{m} &= -\frac{F_c}{Isp_1} \\ \dot{F_c} &= 0 \\ F_c &\in [1500, 3000] \end{cases} \quad \text{and} \quad \begin{cases} \dot{r} &= v \\ \dot{v} &= \frac{F_c}{m} - gM \\ \dot{m} &= -\frac{F_c}{Isp_2} \\ \dot{F_c} &= 0 \\ F_c &\in (3000, 5000] \end{cases}, \quad \text{where} \tag{3}$$

- 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;
- gM is the magnitude of the gravitational acceleration on the moon, which varies with height r but is taken to be the constant $1.622\,\mathrm{m/s^2}$ in this paper, since the change of height $(0 \leq r \leq 30\mathrm{m})$ can be neglected compared to the radius of the moon;
- $Isp_1 = 2500\mathrm{N{\cdot}s/kg}$ and $Isp_2 = 2800\mathrm{N{\cdot}s/kg}$ are the two possible values that the *specific impulse*[1] of the lander's thrust engine can take, depending on whether the current F_c lies in $[1500, 3000]$ or $(3000, 5000]$, and thus the lander's dynamics comprises two different forms as shown in (3);
- note that the terms $\frac{F_c}{m}$ in (3) make the dynamics non-polynomial.

Guidance Program. The guidance program for the slow descent phase is executed once for every 0.128 s. The control flow of the program, containing 4 main blocks, is demonstrated by the left part of Fig. 10.

[1] 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.

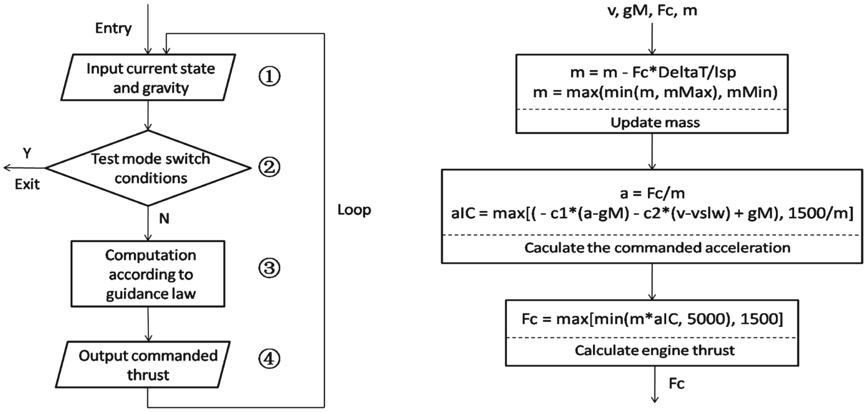
Entry
Input current state and gravity
①
Y
Exit
Test mode switch conditions
②
N
Computation according to guidance law
③
Output commanded thrust
④
Loop
v, gM, Fc, m
m = m - Fc*DeltaT/Isp
m = max(min(m, mMax), mMin)
Update mass
a = Fc/m
alC = max[(- c1*(a-gM) - c2*(v-vslw) + gM), 1500/m]
Caculate the commanded acceleration
Fc = max[min(m*alC, 5000), 1500]
Calculate engine thrust
Fc

Fig. 10. The guidance program for the slow descent phase.

The program first reads data given by navigation computation (block 1), and then decides whether to stay in the slow descent phase or switch to other phases by testing the following conditions (block 2):

(SW1) shutdown signal 1, which should normally be sent out by sensors at the height of 6m, is received, and the lander has stayed in slow descent phase for more than 10s;

(SW2) shutdown signal 2, which should normally be sent out by sensors at the height of 3m, is received, and the lander has stayed in slow descent phase for more than 10s;

(SW3) no shutdown signal is received and the lander has stayed in the slow descent phase for more than 20s.

If any of the above conditions is satisfied, then the GNC system switches from slow descent phase to no-control phase and a shutdown command is sent out to the thrust engine; otherwise the program will stay in the slow descent phase and do the guidance computation (block 3) as shown in the right part of Fig. 10, where

- v and gM are the vertical velocity and gravitational acceleration from navigation measurements or computation; note that we have assumed gM to be a constant;
- F_c and m are the computed thrust and mass estimation at last sampling point; they can be read from memory;
- $DeltaT = 0.128\,\mathrm{s}$ is the sampling period;
- Isp is the specific impulse which can take two different values, i.e. 2500 or 2800, depending on the current value of F_c;

- $mMin = 1100\,\mathrm{kg}$ and $mMax = 3000\,\mathrm{kg}$ are two constants used as the lower and upper bounds of mass estimation;
- $c_1 = 0.01$ and $c_2 = 0.6$ are two control coefficients in the guidance law;
- $vslw = -2\,\mathrm{m/s}$ is the target descent velocity of the slow descent phase;
- the output F_c (block 4) will be used to adjust engine thrust for the following sampling cycle; it can be deduced from the program that the commanded thrust F_c always lies in the range $[1500, 5000]$.

5.1.2 Verification Objectives

Together with the engineers participating in the lunar lander project, we propose the following properties to be verified regarding the closed-loop system of the slow descent phase and the subsequent free fall phase.

Firstly, suppose the lunar lander enters the slow descent phase at $r = 30\,\mathrm{m}$ with $v = -2\,\mathrm{m/s}$, $m = 1250\,\mathrm{kg}$ and $F_c = 2027.5\mathrm{N}$. Then

(P1) **Safety 1:** $|v - vslw| \le \varepsilon$ during the slow descent phase and before touchdown[2], where $\varepsilon = 0.05\,\mathrm{m/s}$ is the tolerance of fluctuation of v around the target $vslw = -2\,\mathrm{m/s}$;

(P2) **Safety 2:** $|v| < vMax$ at the time of touchdown, where $vMax = 5\,\mathrm{m/s}$ is the upper bound of $|v|$ to avoid the lander's crash when contacting the lunar surface;

(P3) **Reachability:** one of the switching conditions (SW1)-(SW3) will finally be satisfied so that the system will exit the slow descent phase.

Furthermore, by taking into account such factors as uncertainty of initial state, disturbance of dynamics, sensor errors, floating-point calculation errors etc., we give

(P4) **Stability and Robustness:** (P2) and (P3) still holds, and an analogous of (P1) is that v will be steered towards $vslw = -2\,\mathrm{m/s}$ after some time.

5.2 Analysis by Simulation

We first build a *Simulink*/*Stateflow* model of the closed-loop system for the slow descent phase. Then based on the model we analyze the system's behaviour by simulation.

The physical dynamics specified by (3) is modelled by the *Simulink* diagram shown in Fig. 11.

In Fig. 11, several blocks contain parameters that are not displayed:

[2] Note that if no shutdown signal is received, there exists possibility that the lander stays in the slow descent phase after landing.

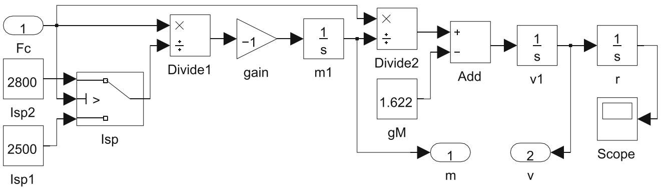

Fig. 11. The *Simulink* diagram of the dynamics for the slow descent phase.

- the threshold of Isp is 3000, which means Isp outputs 2800 when F_c is greater than 3000, and 2500 otherwise;
- the initial values of m, v and r ($m = 1250\,\mathrm{kg}$, $r = 30\,\mathrm{m}$, $v = -2\,\mathrm{m/s}$) are specified as initial values of blocks m1, v1 and r respectively.

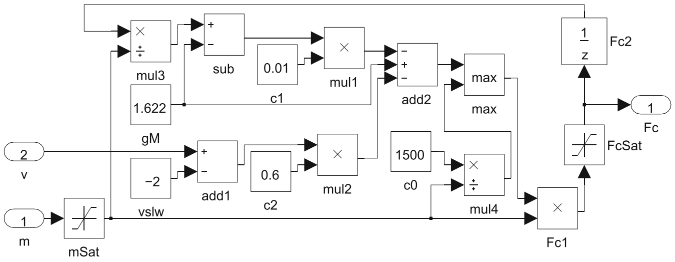

Fig. 12. The *Simulink* diagram of the guidance program for the slow descent phase.

As specified in Fig. 10, The guidance program includes three parts: updating mass m, calculating acceleration aIC, and calculating thrust F_c. The *Simulink* diagram for the guidance program is shown in Fig. 12, in which the sample time of all blocks are fixed as 0.128 s, i.e. the period of the guidance program. In Fig. 12, blocks m and mSat are used to update mass m, blocks Fc1 and FcSat are used to calculate thrust F_c, and the rest are used to calculate acceleration aIC. Blocks mSat and FcSat are saturation blocks from *Simulink* library which limit input signals to the upper and lower bounds of m and F_c respectively.

The simulation result is shown in Fig. 13. The left part shows that the velocity of the lander is between -2 and -1.9999, which corresponds to (P1); the right part shows that if shutdown signal 1 is sent out at 6 m and is successfully received by the lander, then (SW1) will be satisfied at time 12.032s, which corresponds to (P3).

5.3 From *Simulink/Stateflow* Model to HCSP Model

Given a *Simulink/Stateflow* model, Sim2HCSP translates its *Simulink* and *Stateflow* parts separately. With the approach in [54], the *Simulink* part is translated into HCSP processes, while using the approach in [53], the *Stateflow* part

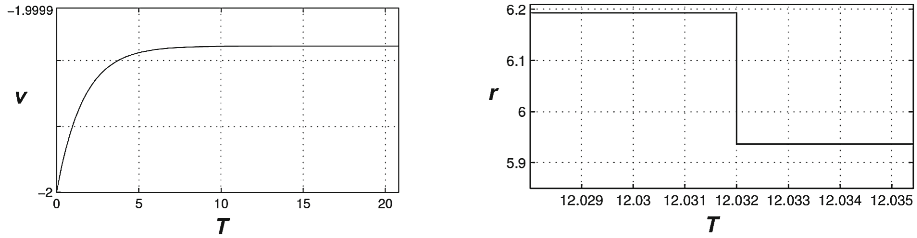

Fig. 13. The simulation result.

is translated into another HCSP processes. Then, these HCSP processes are put together 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 [53,54] 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.

Then the manually constructed *Simulink* model is translated into annotated HCSP using the tool Sim2HCSP, which is basically as

```
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 (3) within a period of 0.128 s; PD_Rep calculates thrust F_c according to

$$F_c' := -0.01 \cdot (F_c - m \cdot gM) - 0.6 \cdot (v - vslw) \cdot m + m \cdot gM \tag{4}$$

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.

5.4 Consistency Checking by Co-Simulation

To validate the above translated HCSP model, we translate it into a *Simulink* model using the tool H2S inversely, which consists of 63 nested subsystems. The top-level overview of the translated *Simulink* model is shown in Fig. 14, where a parallel pattern interprets the physical plant *PC* and the control program *PD*.

To validate the formal model, the translated *Simulink* model is simulated with a fixed simulation step of 0.0001 s, and the evolution of the lander is shown

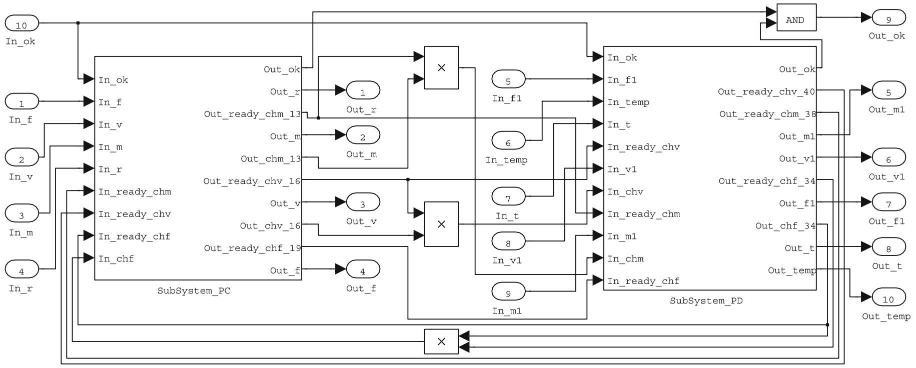

Fig. 14. The top-level overview of the translated *Simulink* model

as the solid curve in Fig. 15. For velocity, we also illustrate the corresponding results of the original *Simulink* model in the dash curve, showing that the translation loop well keeps the system behaviours consistently. Moreover, the left part shows that the velocity of the lander is between −2 and −1.9999 m/s, which corresponds to (R1); the right part shows that if shut-down signal is sent out at 6 m and is successfully received by the lander, then (R3) is satisfied at time 12.0569 s; and then with a subsequent free fall, (R2) is guaranteed.

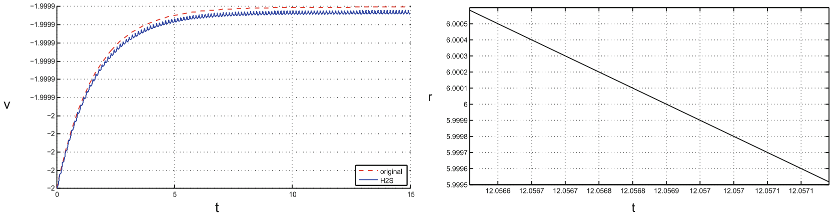

Fig. 15. The evolution in physical plant *PC*

By combining formal and informal approaches in validation and verification of the lunar lander, the reliability was indeed improved, and the domain experts and engineers were also convinced.

5.5 Verification

In this section, we formally verify the property (P1), and the proof for the other properties (P2)-(P4) can be found in [48].

In order to verify property (P1), we give the following proof goal in HHL Prover:

```
lemma goal : ‘‘{True} P {safeProp; (l=0 | (high safeProp))}’’
```

where safeProp stands for $|v - vslw| \leq \varepsilon$. The parts True and safeProp specify the pre- and post-conditions of P respectively. The part (l=0 — (high safeProp)) specifies a duration property, where l=0 means the duration is 0, and high means that the following state expression should hold everywhere on a considered interval.

After applying proof rules in HHL Prover with the above proof goal, the following three lemmas remain unresolved:

```
lemma constraint1: "(t<=0.128) & Inv |- safeProp"
lemma constraint2: "(v=-2) & (m=1250) & (Fc=2027.5)
  & (t=0) |- Inv"
lemma constraint3: "(t= 0.128) & Inv
  |- substF([(t,0)], substF([(Fc,
     -0.01*(Fc-1.622*m) - 0.6*(v+2)*m + 1.622*m)],Inv))"
```

In a more readable way, the three lemmas impose the following constraints:

(C1) $0 \leq t \leq 0.218 \wedge Inv \longrightarrow |v - vslw| \leq \varepsilon$;
(C2) $v = -2 \wedge m = 1250 \wedge F_c = 2027.5 \wedge t = 0 \longrightarrow Inv$;
(C3) $t = 0.128 \wedge Inv \longrightarrow Inv(0 \leftarrow t; F_c' \leftarrow F_c)$, with F_c' defined in (4);
(C4) Inv is the invariant of both constrained dynamical systems

$$\langle ODE_1; 0 \leq t \leq 0.128 \wedge F_c \leq 3000\rangle \text{ and } \langle ODE_2; 0 \leq t \leq 0.128 \wedge F_c > 3000\rangle\,,$$

where ODE_1 and ODE_2 are the two dynamics defined in (3).

Invariant Generation. Invariant generation for polynomial continuous/hybrid systems has been studied a lot [30]. To deal with systems with non-polynomial dynamics, we propose a method based on variable transformation. For this case study, we replace the non-polynomial terms $\frac{F_c}{m}$ in ODE_1 and ODE_2 by a new variable a. Then by simple computation of derivatives we get two transformed polynomial dynamics:

$$ODE_1' \mathrel{\widehat{=}} \begin{cases} \dot{r} = v \\ \dot{v} = a - 1.622 \\ \dot{a} = \frac{a^2}{2500} \end{cases} \text{ and } ODE_2' \mathrel{\widehat{=}} \begin{cases} \dot{r} = v \\ \dot{v} = a - 1.622 \\ \dot{a} = \frac{a^2}{2800} \end{cases}. \tag{5}$$

Furthermore, it is not difficult to see that the update of F_c as in (4) can be accordingly transformed to the update of a given by

$$a' \mathrel{\widehat{=}} -c_1 \cdot (a - gM) - c_2 \cdot (v - vslw) + gM. \tag{6}$$

As a result, if we assume Inv to be a formula over variables v, a, t, then (C2)-(C4) can be transformed to:

(C2') $v = -2 \wedge a = 1.622 \wedge t = 0 \longrightarrow Inv$;
(C3') $t = 0.128 \wedge Inv \longrightarrow Inv(0 \leftarrow t; a' \leftarrow a)$, with a' defined in (6);
(C4') Inv is the invariant of both constrained dynamical systems $\langle ODE_1'; 0 \leq t \leq 0.128\rangle$ and $\langle ODE_2'; 0 \leq t \leq 0.128\rangle$[3] with ODE_1' and ODE_2' defined in (5).

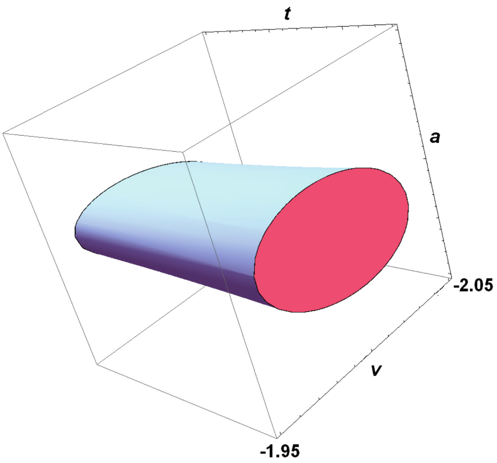

Fig. 16. The invariant for HHL Prover.

Note that the constraints (C1) and (C2')-(C4') are all polynomial. Then the invariant *Inv* can be synthesized using the SOS (sum-of-squares) relaxation approach in the study of polynomial hybrid systems [28]. With the *Matlab*-based tool YALMIP and SDPT-3, an invariant $p(v, a, t) \leq 0$ as depicted by Fig. 16 is generated. Furthermore, to avoid the errors of numerical computation in *Matlab*, we perform post-verification using the computer algebra tool RAGlib[4] to show that the synthesized $p(v, a, t) \leq 0$ is indeed an invariant. Thus we have successfully completed the proof of property (P1) by theorem proving. On the platform with Intel Q9400 2.66 GHz CPU and 4 GB RAM running Windows XP, the synthesis costs 2 s and 5 MB memory, while post-verification costs 10 min and 70 MB memory.

6 Related Work

6.1 Related Formalization of Sumulink/Stateflow

There has been a range of work on translating *Simulink* into modelling formalisms supported by analysis and verification tools. Tripakis *et al.* [43] presented an algorithm for translating discrete-time *Simulink* models to Lustre, a synchronous language featuring a formal semantics and a number of tools for validation and analysis. Cavalcanti *et al.* [11] put forth a semantics for discrete-time *Simulink* diagrams using Circus, a combination of Z and CSP. Meenakshi *et al.* [34] proposed an algorithm that translates a subset of *Simulink* into the input language of the finite-state model checker NuSMV. Chen *et al.* [12] presented an algorithm that translates *Simulink* models to the real-time specification language

[3] We have abstracted away the domain constraints on F_c.
[4] http://www-polsys.lip6.fr/safey/RAGLib/.

Timed Interval Calculus (TIC), which can accommodate continuous *Simulink* diagrams directly, and they validated TIC models using an interactive theorem prover. Their translation is confined to continuous blocks whose outputs can be represented explicitly by a closed-form mathematical relation on their inputs.

Beyond the pure *Simulink* models considered in the above approaches, models comprising reactive components triggered by and affecting the *Simulink* dataflow model have also been studied recently. Hamon *et al.* [23] proposed an operational semantics of *Stateflow*, which serves as a foundation for developing tools for formal analysis of *Stateflow* designs. Scaife *et al.* [39] translated a subset of *Stateflow* into Lustre for formal analysis. Tiwari [42] defines a formal semantics of *Simulink*/*Stateflow* using guarded pushdown automata, in which continuous dynamical systems modeled by *Simulink* are discretized, and he discussed how to verify a guarded sequence via type checking, model checking and theorem proving. Agrawal *et al.* [5] proposed a method to translate *Simulink*/*Stateflow* models into hybrid automata using graph flattening, and the target models represented by hybrid automata can then be formally analyzed and verified by model checkers for hybrid systems. Their approach induces certain limitations, both for the discrete-continuous interfaces in *Simulink*/*Stateflow* models, where the output signals of *Stateflow* blocks are required to be Boolean and to immediately connect to the selector input of an analog switch block, and for the forms of continuous dynamics, as most of current model checkers for hybrid systems support only very restricted differential equations. Miller *et al.* [35] proposed a method to translate discrete *Simulink*/*Stateflow* models into Lustre for formal analysis.

In contrast, the formal semantics for *Simulink*/*Stateflow* given here is based on the work of [53,54], in which the meanings of most of syntactic entities and features of *Simulink*/*Stateflow* are well handled by using HCSP. E.g., the meaning of all continuous blocks can be well defined by using the notions of differential equations and invariants in the HCSP encodings, advanced features like *early return logic*, *history junction*, *nontermination* of *Stateflow* can be easily handled by using the notion of *recursion* of HCSP, which are not addressed in most of the existing work. The payment is that we have to resort to interactive theorem proving instead of automatic model checking for discharging the proof obligations.

6.2 Related Verification of Embedded Systems

Verification of full feedback system combining the physical plant with the control program has been advocated by Cousot [16] and Goubault et al. [22]. There are some recent work in this trend which resembles our approach in this paper. In [10], Bouissou et al. presented a static analyzer named HybridFluctuat to analyze hybrid systems encompassing embedded software and continuous environment; subdivision is needed for HybridFluctuat to deal with large initial sets. In [33], Majumdar *et al.* also presented a static analyzer CLSE for closed-loop control systems, using symbolic execution and SMT solving techniques;

CLSE only handles linear continuous dynamics. In [7], Saha *et al.* verified stability of control software implementations; their approach requires expertise on analysis of mathematical models in control theory using such tools as Lyapunov functions.

6.3 Related Verification Tools

Some tools are available for formal verification of *Simulink*/*Stateflow* based on numerical simulation or approximation. STRONG [17] performs bounded time reachability and safety verification for linear hybrid systems based on robust test generation and coverage. Breach [18] uses sensitivity analysis to compute approximate reachable sets and analyzes properties in the form of MITL based on numerical simulation. C2E2 [19] 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 many tools developed for formal modelling and verification of hybrid systems. The tool d/dt [8] provides reachability analysis and safety verification of hybrid systems with linear continuous dynamics and uncertain bounded input. iSAT-ODE [4] is a numerical SMT solver based on interval arithmetic that can conduct bounded model checking for hybrid systems. Flow* [15] 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 [38] is proposed to verify hybrid systems specified using differential dynamic logic. Compared to our work, it supports a simple set of hybrid constructs that do not cover communications and parallel composition.

6.4 Related Industrial Case Studies

There are some recent work on application of formal methods in the aerospace industry. For example, in [27] Johnson *et al.* proved satellite rendezvous and conjunction avoidance by computing the reachable sets of nonlinear hybrid systems; in [21] Katoen *et al.* reported on their usage of formal modelling and analysis techniques in the software development for a European satellite.

7 Conclusions

In this paper, we summarize our experience on combining formal and informal methods in the design of spacecrafts. The ingredients of our approach include

- A translation from S/S to HCSP, implemented as Sim2HCSP;
- A translation from HCSP to *Simulink*, implemented as H2S;
- A deductive way to verify a translated S/S model via HHL prover;

- An abstraction of elementary hybrid systems by polynomial hybrid systems, implemented as EHS2PHS;
- Invariant generation of polynomial hybrid systems, implemented as invariant generator.

The advantages of our approach include

- It enables formal verification as a complementation of simulation. As the inherent incompleteness of simulation, it has become an agreement in industry and academia to complement simulation with formal verification, but this issue still remains challenging although lots of attempts have been done (see the related work section);
- It provides an option to start the design of a hybrid system with an HCSP formal model, and simulate and/or test it using *Matlab* platform economically, without expensive formal verification if not necessary.
- The semantic preservation in shifting between formal and informal models is justified by co-simulation. Therefore, it provides the designer the flexibility using formal and informal methods according to the trade-off between efficiency and cost, and correctness and reliability.

The effectiveness of our approach has been demonstrated in the successful analysis and verification of the descent guidance control program of a lunar lander.

Acknowledgements. We thank all of our collaborators with whom the joint work are reported in this chapter, including Prof. Chaochen Zhou, Prof. Martin Fränzle, Prof. Shengchao Qin, Prof. Anders P. Ravn, Prof. Tao Tang, Prof. Bin Gu, Dr. Jiang Liu, Dr. Jidong Lv, Dr. Shuling Wang, Dr. Hengjun Zhao, Dr. Liang Zou, Dr. Yao Chen, Mr. Mingshuai Chen and Mr. Zhao Quan.

References

1. Simulink User's Guide (2013). http://www.mathworks.com/help/pdf_doc/simulink/sl_using.pdf
2. Stateflow User's Guide (2013). http://www.mathworks.com/help/pdf_doc/stateflow/sf_using.pdf
3. SysML V 1.4 Beta Specification (2013). http://www.omg.org/spec/SysML
4. Eggers, A., Fränzle, M., Herde, C.: SAT modulo ODE: a direct SAT approach to hybrid systems. In: Cha, S.S., Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 171–185. Springer, Heidelberg (2008)
5. Agrawal, A., Simon, G., Karsai, G.: Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. Int. Workshop Graph Transform. Visual Model. Tech. **109**, 43–56 (2004)
6. Alur, R., Henzinger, T.A.: Modularity for timed and hybrid systems. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243. Springer, Heidelberg (1997)
7. Anta, A., Majumdar, R., Saha, I., Tabuada, P.: Automatic verification of control system implementations. In: EMSOFT 2010, pp. 9–18 (2010)

8. 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, pp. 365–370. Springer, Heidelberg (2002)
9. Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A.L.: Metropolis: an integrated electronic system design environment. IEEE Comput. **36**(4), 45–52 (2003)
10. Bouissou, O., Goubault, E., Putot, S., Tekkal, K., Vedrine, F.: HybridFluctuat: a static analyzer of numerical programs within a continuous environment. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 620–626. Springer, Heidelberg (2009)
11. Cavalcanti, A., Clayton, P., O'Halloran, C.: Control law diagrams in circus. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 253–268. Springer, Heidelberg (2005)
12. Chen, C., Dong, J.S., Sun, J.: A formal framework for modeling and validating simulink diagrams. Formal Asp. Comput. **21**(5), 451–483 (2009)
13. Chen, M., Han, X., Tang, T., Wang, S., Yang, M., Zhan, N., Zhao, H., Zou, L.: MARS: A toolchain for modeling, analysis and verification of spacecraft control systems. Technical Report ISCAS-SKLCS-15-04, State Key Laboratories of Computer Science, Institute of Software, CAS (2015)
14. Chen, M., Ravn, A., Yang, M., Zhan, N., Zou, L.: A two-way path between formal and informal design of embedded systems. Technical Report ISCAS-SKLCS-15-06, State Key Laboratories of Computer Science, Institute of Software, Chinese Academy of Sciences (2015)
15. 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)
16. Cousot, P.: Integrating physical systems in the static analysis of embedded control software. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 135–138. Springer, Heidelberg (2005)
17. Deng, Y., Rajhans, A., Julius, A.A.: STRONG: a trajectory-based verification toolbox for hybrid systems. In: Joshi, K., Siegle, M., Stoelinga, M., D'Argenio, P.R. (eds.) QEST 2013. LNCS, vol. 8054, pp. 165–168. Springer, Heidelberg (2013)
18. Donzé, A.: Breach, a toolbox for verification and parameter synthesis of hybrid systems. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 167–170. Springer, Heidelberg (2010)
19. Duggirala, P.S., Mitra, S., Viswanathan, M., Potok, M.: C2E2: a verification tool for stateflow models. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 68–82. Springer, Heidelberg (2015)
20. Eker, J., Janneck, J., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity - the ptolemy approach. Proc. IEEE **91**(1), 127–144 (2003)
21. Esteve, M.-A., Katoen, J.-P., Nguyen, V., Postma, B., Yushtein, Y.: Formal correctness, safety, dependability, and performance analysis of a satellite. In: ICSE 2012, pp. 1022–1031 (2012)
22. Goubault, E., Martel, M., Putot, S.: Some future challenges in the validation of control systems. In: ERTS 2006 (2006)
23. Hamon, G., Rushby, J.: An operational semantics for stateflow. Int. J. Softw. Tools Technol. Transf. **9**(5), 447–456 (2007)
24. He, J.: From CSP to hybrid systems. In: Roscoe, A.W. (ed.) A Classical Mind, Essays in Honour of C.A.R. Hoare, pp. 171–189. Prentice Hall International (UK) Ltd, Hertfordshire (1994)

25. Henzinger, T.A.: The theory of hybrid automata. In: LICS 1996, pp. 278–292, July 1996
26. Hoare, C.A.R., He, J.: Unifying Theories of Programming, vol. 14. Prentice Hall, Englewood Cliffs (1998)
27. Johnson, T.T., Green, J., Mitra, S., Dudley, R., Erwin, R.S.: Satellite rendezvous and conjunction avoidance: case studies in verification of nonlinear hybrid systems. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 252–266. Springer, Heidelberg (2012)
28. Kong, H., He, F., Song, X., Hung, W.N.N., Gu, M.: Exponential-condition-based barrier certificate generation for safety verification of hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 242–257. Springer, Heidelberg (2013)
29. 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)
30. Liu, J., Zhan, N., Zhao, H.: Computing semi-algebraic invariants for polynomial dynamical systems. In: EMSOFT 2011, pp. 97–106 (2011)
31. Liu, J., Zhan, N., Zhao, H.: Automatically discovering relaxed lyapunov functions for polynomial dynamical systems. Math. Comput. Sci. **6**(4), 395–408 (2012)
32. Liu, J., Zhan, N., Zhao, H., Zou, L.: Abstraction of elementary hybrid systems by variable transformation. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 360–377. Springer, Heidelberg (2015)
33. Majumdar, R., Saha, I., Shashidhar, K.C., Wang, Z.: CLSE: closed-loop symbolic execution. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 356–370. Springer, Heidelberg (2012)
34. Meenakshi, B., Bhatnagar, A., Roy, S.: Tool for translating simulink models into input language of a model checker. In: Liu, Z., Kleinberg, R.D. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 606–620. Springer, Heidelberg (2006)
35. Miller, S.P., Whalen, M.W., Cofer, D.D.: Software model checking takes off. Commun. ACM **53**(2), 58–64 (2010)
36. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reasoning **41**(2), 143–189 (2008)
37. 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)
38. 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)
39. Scaife, N., Sofronis, C., Caspi, P., Tripakis, S., Maraninchi, F.: Defining and translating a safe subset of simulink/stateflow into lustre. In: EMSOFT 2004, pp. 259–268. ACM (2004)
40. Selic, B., Gerard, S.: Modeling and Analysis or Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems. The MK/OMG Press, Burlington (2013)
41. Tiller, M.: Introduction to Physical Modeling with Modelica, vol. 615. Springer, New York (2001)
42. Tiwari, A.: Formal semantics and analysis methods for Simulink Stateflow models. Technical report, SRI International, (2002)
43. Tripakis, S., Sofronis, C., Caspi, P., Curic, A.: Translating discrete-time Simulink to Lustre. ACM Trans. Embed. Comput. Syst. **4**(4), 779–818 (2005)

44. Wang, S., Zhan, N., Guelev, D.: An assume/guarantee based compositional calculus for hybrid CSP. In: Agrawal, M., Cooper, S.B., Li, A. (eds.) TAMC 2012. LNCS, vol. 7287, pp. 72–83. Springer, Heidelberg (2012)
45. Wang, S., Zhan, N., Zou, L.: An improved hhl prover: an interactive theorem prover for hybrid systems. In: Butler, M., et al. (eds.) ICFEM 2015. LNCS, vol. 9407, pp. 382–399. Springer, Heidelberg (2015). doi:10.1007/978-3-319-25423-4_25
46. Zhan, N., Wang, S., Guelev, D.: Extending Hoare logic to hybrid systems. Technical report ISCAS-SKLCS-13-02, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences (2013)
47. 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)
48. 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)
49. 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)
50. Zhou, C., Hoare, C.A.R., Ravn, A.: A calculus of durations. Inf. Process. Lett. **40**(5), 269–276 (1991)
51. 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. Springer, Heidelberg (1996)
52. 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)
53. Zou, L., Zhan, N., Wang, S., Fränzle, M.: Formal verification of simulink/stateflow diagrams. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) ATVA 2015. LNCS, vol. 9364, pp. 464–481. Springer, Heidelberg (2015). doi:10.1007/978-3-319-24953-7_33
54. 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)