Verifying Simulink Diagrams Via A Hybrid Hoare Logic Prover*

Liang Zou, Naijun Zhan† and Shuling Wang
State Key Lab. of Comp. Sci.
Institute of Software
Chinese Academy of Sciences
{zoul,znj,wangsl}@ios.ac.cn

Martin Fränzle
Department of Information
Oldenburg University
fraenzle@informatik.uni-oldenburg.de

Shengchao Qin
Teesside University and
Beijing University of Technology
s.qin@tees.ac.uk

ABSTRACT

Simulink is an industrial de-facto standard for building executable models of embedded systems and their environments, facilitating validation by simulation. Due to the inherent incompleteness of this form of system validation, complementing simulation by formal verification would be desirable. A prerequisite for such an approach is a formal semantics of Simulink's graphical models. In this paper, we show how to encode Simulink diagrams into Hybrid CSP (HCSP), a formal modelling language encoding hybrid system dynamics by means of an extension of CSP. The translation from Simulink to HCSP is fully automatic. We furthermore discuss how to utilize a Hybrid Hoare Logic Prover to verify the translated HCSP models. We demonstrate our approach on a combined scenario originating from the Chinese High-speed Train Control System at Level 3 (CTCS-3).

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis

General Terms

Formal methods, Language transformation, Verification

Keywords

Simulink models, formal methods, Hybrid CSP, theorem proving

*Liang Zou, Naijun Zhan and Shuling Wang are funded partly by NSFC-91118007, NSFC-6110006, and National Science and Technology Major Project of China (2012ZX01039-004), Shuling Wang and Martin Fräzle are supported in part by the DFG-funded Transregional Collaborative Research Center SFB-TR 14 AVACS, and Shengchao Qin was funded in part by the EPSRC project EP/G042322.
†The corresponding author

1. INTRODUCTION

Simulink (`www.mathworks.com/products/simulink/`) is an environment for the model-based analysis and design of signal-processing systems and embedded control systems. Being based on a large palette of individually simple function blocks and on their composition by continuous-time synchronous data-flow, it offers an intuitive graphical modeling language reminiscent of circuit diagrams and thus appealing to the practicing engineer. As the circuit analogy renders reasoning about large numbers of concurrently operating, intertwined signal paths relatively easy —compared to, e.g., programming-like notations—, Simulink-based modeling, analysis, and design has become a de-facto standard in the embedded systems industry. Abstracting from the necessary iterations for bug fixing, a prototypical design flow here involves (1.) building the model of environment and embedded system (ES) functionality over abstract (mostly continuous) time, (2.) simulating the model, (3.) analyzing the simulation results, (4.) refining the time model of the ES by adding sampling times, phase delays, (de-)activation conditions, etc., (5.) redoing simulation and interpretation of simulation results, (6.) code generation and deployment, involving decisions on the actual amount of concurrency to be employed.

As analysis and validation within this flow is primarily based on simulation, Simulink offers a comprehensive range of numerical simulation algorithms, reflecting different compromises between system classes tackled, accuracy, and performance. Nevertheless, all of them are prone to the limitations of validation by (classical, i.e. unverified) numerical simulation, which are the intrinsically incomplete coverage of open systems and the possible unsoundness of analysis results due to numerical error. Statistical model checking (SMC, cf. e.g. [5]) deals with the first problem by means of a rigorous statistical interpretation of the simulation results without, however, addressing the soundness issue of numerical simulation. The latter could in principle be resolved by set-based verified simulation, but methods dealing with non-smooth derivatives, as frequently encountered in Simulink models, are only emerging [14] and far from able to cover realistic Simulink models. Furthermore, their performance, both wrt. runtime of simulation and conclusiveness of the conservatively overapproximate, set-valued results obtained, are likely to become prohibitive in an SMC context, leaving the issue of open systems unresolved.

The only technique currently being able to reconcile the above two issues is formal verification. The usual approach

978-1-4799-1443-2/13/$31.00 ©2013 IEEE

here is to translate Simulink into the input language of a formal verification tool for hybrid discrete-continuous systems, be it an automaton-based language [2] or a symbolic description [8], and employ the corresponding verification engines. These engines pursue an exhaustive search of the state space, thus providing certificates which cover all input stimuli possible in an open system, and do increasingly apply verified arithmetic, rendering them resistant against numerical error. Given Simulink's modeling paradigm of connecting numerous concurrently executing small blocks via continuous-time synchronous dataflow, the translated models are generally characterized by tightly coupled, fine-granular concurrency, which unfortunately is detrimental to analyzability.

In this paper, we do therefore investigate translation of Simulink into a process calculus with its richer set of composition primitives. We present an encoding of Simulink's semantics in terms of HCSP [10], a formal modelling language encoding hybrid system dynamics by means of an extension of CSP [9]. As analysis of HCSP models is supported by the interactive verification tool Hybrid Hoare Logic Prover, this provides a gateway to mechanized verification, which we demonstrate on a combined scenario originating from the Chinese High-speed Control System at Level 3 (CTCS-3).

Related Work.

There have been a range of work on translating Simulink into other modelling formalisms, for which analysis and verification tools are being developed. Tripakis *et al* [16] present an algorithm of translating discrete-time Simulink models to Lustre, a synchronous language developed with formal semantics and a number of tools for validation and analysis. Cavalcanti *et al* [3] present a semantics for discrete-time Simulink diagrams using Circus, a combination of Z and CSP. Meenakshi *et al* [12] present an algorithm that translates a subset of Simulink into input language of model checker NuSMV. Among all the works mentioned above, continuous time models of Simulink are not considered. The most related work by Chen *et al* [4], translates Simulink models to a real-time specification language Timed Interval Calculus (TIC), which can directly represent and analyze continuous Simulink diagrams, and to the end, validates TIC models by a theorem prover. However, the translation is limited as it can only handle continuous blocks whose outputs can be represented explicitly by a mathematical relation on inputs. Our approach can handle all continuous blocks by using the notion of differential equations and invariants; and on the other hand, our target language is a process language, with more intuitivity and compositionality in both construction and verification.

A Simulink model can include a stateflow block, which reacts to events triggered in the Simulink model and changes the states accordingly. The formalization of such Simulink/Stateflow models has also been studied recently. Hamon *et al* [6] propose an operational semantics of Stateflow, which serves as a foundation for developing tools for formal analysis of Stateflow designs. As mentioned previously, Agrawal *et al* [2] propose a method to translated Simulink/Stateflow models in to hybrid automata using graph translation. The target models represented by hybrid automata can then be submitted to related model checkers for formal analysis and verification, and thus get involved with the state space exploration. Tiwari [15] describes the formal semantics of Stateflow using communicating pushdown automata, for which differential equations for representing continuous are discretized by difference equations.

The rest of this paper is organized as follows: Sec. 2 introduces Simulink, HCSP, and Hybrid Hoare Logic with its prover; Sec. 3 and Sec. 4 present our algorithms for translating Simulink to HCSP, and the implementation respectively; Sec. 5 illustrates our approach by modelling and verifying a combined scenario of CTCS-3; Finally, Sec. 6 draws the conclusion and the future work.

2. SIMULINK AND HYBRID CSP

In this section, we introduce Simulink, HCSP, Hybrid Hoare Logic (HHL) and its prover. For Simulink, we highlight those features that are relevant to this work, and for more details please refer to [1].

2.1 Simulink

A Simulink model contains a set of blocks, subsystems, and wires, where blocks and subsystems cooperate by sending messages through the wires between them. Figure 1 gives a Simulink model of train movement, where rounded rectangles a_in, v_out, and s_out are in-ports and out-ports for sub-systems, and represent the acceleration, velocity, and trajectory of the train respectively. The two rectangular blocks v and s are *integrator* blocks of the Simulink library, each of which contains a parameter to represent the initial value of the output. An integrator block outputs its initial value at the beginning and the integration of the input signal afterwards. Hence, the block v outputs the velocity of the train, which is the integration of the input acceleration a_in; and on the other hand, the block s outputs the trajectory of the train, which is the integration of the input velocity v.

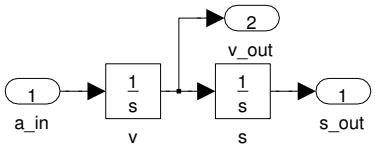

Figure 1: The plant of a train control system

An elementary block gets input signals and computes the output signals. However, to make Simulink more useful, almost every block in Simulink contains some user-defined parameters to alter its functionalities. One typical parameter is *sample time* which defines how frequently the computation is taken. Two special values, 0 and -1, may be set for sample time, where the sample time 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. The case when sample time is -1 will be discussed in Sec. 3.2.1, and therefore, from now on, we assume that the sample time is not equal to -1. Thus, blocks are classified into two categories, i.e., *continuous* and *discrete*, according to their sample times.

Blocks and subsystems in a Simulink model receive inputs and compute outputs in parallel, and wires specify the data flow between blocks. Meanwhile, computation conducted in a block takes no time and the computed output is delivered immediately to its receiver. Therefore, if we consider each block as a predicate relating inputs to outputs, the behavior

of the whole diagram is simply specified by the conjunction of the predicates of all the blocks in the diagram. To make this idea clear, several basic rules of Simulink need to be given.

- Logical loops among discrete blocks (except for delay blocks that output the past value of a signal) are not allowed. This rule prevents the **zeno** phenomena, i.e. a sequence of infinite many computations that take no time.
- The outputs of a block purely depend on the inputs and parameters set by users. Hence, the outputs of a block are never used to determine its own outputs. For instance, $x := x + 1$ is not implementable in Simulink. However, $y := x + 1$ can be easily implemented in Simulink. (This rule can be seen as a special case of the previous one.)

Because of the rules mentioned above, each *signal* for a wire in a diagram can be specified by a *timed trace* which is a function from time domain (modeled as non-negative reals $\mathbb{R}^{\geq 0}$) to the respective values.

The behavior of each block can be divided into a set of sub-behaviors, each of which is guarded by a condition. Moreover, these guards are exclusive and complete, i.e., the conjunction of any two of these guards is unsatisfiable and the disjunction of all of them is valid. So, each sub-behavior can be further specified as a predicate over input and output signals. For example, a kind of blocks named 'Switch' in Simulink has three input signals received from in-ports 1, 2, 3 respectively and one output signal. The input signal at in-port 2 is a boolean condition, and when it is true, the output signal will be equal to the input signal at in-port 1, otherwise the one at in-port 3. Hence, blocks can be interpreted by the following semantic function $Seman_B$:

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

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, $out(0)$ is the value of out at the 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. Obviously, $\bigvee_{k=1}^{m} B_k(ps, in) \Leftrightarrow True$, and $B_i(ps, in) \wedge B_j(ps, in) \Leftrightarrow False$ for any $i \neq j$, always hold.

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 $(t - (t \mod st))$ where st represents the sample time of the block.

So, the semantics of a Simulink diagram is defined by

$$Seman_D \;\widehat{=}\; \bigwedge_{j=1}^{n} Seman_B(init_j, ps_j), \quad (2)$$

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.

2.2 Hybrid CSP (HCSP)

HCSP [7, 19, 17] is a formal language for describing hybrid systems, which is an extension of CSP by introducing timing constructs, interrupts, and differential equations for representing continuous evolution. Exchanging data among processes is described solely by communications, no shared variable is allowed between different processes in parallel, so each program variable is local to the respective sequential component. We write $\mathcal{V}$ and Σ for the sets of variables and channel names, respectively. The syntax of HCSP is given as follows:

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

Here io_i stands for a communication event, i.e., either $ch?x$ or $ch!e$, P, Q, Q_i are HCSP processes, x and s stand for variables, and ch for channel name. B and e are Boolean and arithmetic expressions and d is a non-negative real constant.

The intended meaning of the individual constructs is as follows:

- skip, $x := e$, $ch?x$, $ch!e$ and $P; Q$ can be understood in a standard way.
- $B \to P$ behaves as P if B is true, otherwise it terminates immediately.
- $P \sqcup Q$ denotes internal choice. It behaves as either P or Q, and the choice is made by the process.
- The repetition P^* executes P for some finite number of times.
- $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle$ is the continuous evolution statement (hereafter shortly *continuous*). It forces the vector s of real variables to continuously evolve according to the differential equations $\mathcal{F}$ [1] as long as the boolean expression B, which defines the *domain of* s, holds, and terminates whenever B turns false.
- $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \trianglerighteq [\!]_{i \in I}(io_i \to Q_i)$ behaves like $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle$, except that the continuous is preempted as soon as one of the communications io_i occurs. That is followed by the respective Q_i. However, if the continuous part terminates before a communication from among $\{io_i\}_{i \in I}$ occurs, then the process terminates directly.
- $P \| Q$ behaves as if P and Q run independently except that all communications along the common channels connecting P and Q are to be synchronized. The processes P and Q in parallel can share neither variables, nor input nor output channels.

The other constructs of HCSP in [7, 19] are derivable from the above syntax, e.g., wait $d \stackrel{\text{def}}{=} t := 0; \langle \dot{t} = 1 \& t < d \rangle$, $stop \stackrel{\text{def}}{=} t := 0; \langle \dot{t} = 1 \& true \rangle$, $[\!]_{i \in I}(io_i \to Q_i) \stackrel{\text{def}}{=} stop \trianglerighteq [\!]_{i \in I}(io_i \to Q_i)$, and the timeout $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \trianglerighteq_d Q$ can be defined by $t := 0; \langle F(\dot{s}, s) = 0 \wedge \dot{t} = 1 \& t < d \wedge B \rangle; t \geq d \to Q$.

[1] The differential equations are written in sequence and separated by comma.

2.3 Hybrid Hoare Logic (HHL)

We extend Hoare Logic to hybrid systems, by adding history formulas to describe continuous properties held throughout the whole execution interval of HCSP processes. The history formulas are defined by duration calculus (DC), which is a real extension of Interval Temporal Logic (ITL) [13] for specifying and reasoning about real-time systems. Like ITL, the only modality in DC is the chop $^\frown$ to divide a considered interval into two consecutive sub-intervals such that its first operand is satisfied on the first sub-interval, while the second operand is satisfied on the second sub-interval. Besides, DC extends ITL by introducing durations of state expressions $\int S$, and the temporal variable ℓ to denote the length of the considered interval, i.e. $\int 1$. Here, we will adopt the notion of point formula introduced in [18], denoted by $\lceil S \rceil^0$, to mean that S holds at the considered point interval. Then the formula $\lceil S \rceil$ is defined as $\ell > 0 \wedge \neg(\ell > 0 ^\frown \lceil \neg S \rceil^0 ^\frown \ell > 0)$, meaning that the state expression S holds at each point of the reference interval which is not a point interval.

In [10], we have defined an extension of Hoare logic for specifying and reasoning about HCSP processes, i.e. HHL. Each specification for a sequential process P takes the form $\{Pre\}\ P\ \{Post; HF\}$, where *Pre*, *Post* represent pre-/post-conditions, expressed by first-order logic, to specify properties of variables held at starting and termination of the execution of P; and *HF* history formula, expressed by DC, to record the execution history of P, including real-time and continuous properties. The effect of discrete processes will be specified by the pre-/post-conditions, but not be recorded in the history. The specification for a parallel process is then defined by assigning to each sequential component of it the respective pre-/post-conditions and history formula, that is

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

In HHL, each of HCSP constructs is axiomatized by a set of axioms and inference rules, for example, the *continuous* is axiomatized by the following rule:

$$\frac{Init\ \Rightarrow\ Inv}{\{Init \wedge Pre\}\langle F(\dot{s}, s) = 0\ \&B\rangle\{Pre \wedge \mathbf{Cl}(Inv) \wedge \mathbf{Cl}(\neg B);\ (l = 0) \vee \lceil Inv \wedge Pre \wedge B \rceil\}}$$

where *Init* stands for the initial condition on continuous variables, *Inv* is a continuous invariant corresponding to the dynamical system[2], $\mathbf{Cl}(G)$ stands for the *closure* of G, e.g. when G is constructed by polynomial inequalities through $\wedge$ and $\vee$, $\mathbf{Cl}(G)$ can be obtained from G by replacing $<$ (and $>$) with $\leq$ (and $\geq$) in G. *Pre* does not contain s, and $l = 0$ in the history is to record the behaviour when the initial values satisfy $\neg B$ at very beginning. The full explanation of HHL can be found in [10, 17].

2.3.1 HHL Prover

For tool support, we have mechanized the verification architecture of HCSP specifications based on deductive reasoning, and implemented a theorem prover for HHL [20] in proof assistant Isabelle/HOL. By applying the prover, we can verify if an HCSP process is correct with respect to a specification written in HHL in a machine checkable way.

The mechanization in a proof assistant adopts the general approach: first, encode the process language (i.e. HCSP), assertion languages (i.e. first-order logic and DC), and then the specification logic (i.e. HHL) in Isabelle/HOL; second, based on the proof system of HHL, design a verification condition generator, which reduces an annotated HCSP process to a verification condition, a combination of first-order formula and a DC formula, that is valid if the original HCSP process conforms to the annotated assertions, i.e. the specification; finally, check the validity of the verification condition, which can be proved either by interactive theorem proving in Isabelle/HOL, or by automated theorem proving for its decidable fragments. The current version of the HHL prover mainly focuses on the interactive approach.

We illustrate how to use the HHL prover by verifying a simple process that models train movement. Let s be trajectory, v velocity, a acceleration, and $v_{\max}$ the speed limit of the train respectively. As modelled by the following process

$$\langle \dot{s} = v, \dot{v} = a \& v <= v_{\max} \rangle$$

the train moves with velocity v and acceleration a, but with constraint $v <= v_{\max}$ always holds. Denote the encoding of this process in HHL prover by `train_part`, we can prove easily a lemma, which claims that starting with initial velocity 0 and positive acceleration a, the train will always move with velocity within $[0, v_{\max}]$:

lemma: {v = 0 ∧ a > 0} train_part {0<v<=$v_{\max}$, high (0<=v<=$v_{\max}$)}
by (continuous_theorem)
by (*FOL* axiom)*
by (*DC* axiom)*
done

where `high (S)` encodes $\lceil S \rceil$. The proof can be divided to three steps: first, apply the axiom `continuous_theorem` that corresponds to the application of *continuous* rule presented in the previous part, and as a result, the specification to be proved is reduced to a first-order formula and a DC formula; then secondly and thirdly, prove the validity of the two logical formulas by applying a sequence of axioms of first-order logic and DC respectively, which we have already encoded in HHL prover.

3. FROM SIMULINK TO HCSP

In this section, we consider how to define a formal semantics for Simulink by encoding it into HCSP.

3.1 Translating Blocks

From the semantics we discussed in Sec.2.1, Simulink blocks behave as a formula of the form (1), but the definitions B_k and P_k for continuous and discrete blocks are different. Therefore, we encode continuous and discrete blocks into two different process patterns.

Regarding 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:

[2]Please refer to [11] for the details of continuous invariants of dynamical systems.

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

Given a continuous block *cb* with the above structure, we define *getInit(cb)* and *getComms(cb)* to represent the initial process $out := init$ and the communications $\{io_i\}_{i \in I}$ respectively; and *getDiffs(cb)* to represent the set of differential equations occurring in *cb*.

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:

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

Given a discrete block *db* with the above structure, we define *getInit(db)* similar as continuous; and *getCin*, *getCout* to represent the input *cin?in* and output *cout!out* respectively; and *db.st*, *db.comp* to represent the sample time *st* and computation process respectively.

3.2 Translating Diagrams

This section presents our algorithms for translating Simulink diagrams, including discrete diagrams, continuous diagrams, and arbitrary diagrams constructed out of them. Before giving the translation, some notations and basic pre-processing of diagrams are introduced first.

3.2.1 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.

Algorithm 1 Computing inherited sample times

Require: A diagram *diag*
Ensure: Calculate all determined sample times in *diag*

```
1: for (flag ← true; flag; skip) do
2:   flag ← false;
3:   for all b in diag do
4:     if !known(b) ∧ allKnown(b.srcBlocks()) then
5:       b.st ← GCD(b.srcBlocks());
6:       flag ← true;
7:     end if
8:   end for
9: end for
```

Algorithm 1 calculates the sample times of all the blocks of a given diagram recursively and terminates when all of the determined ones are known. Specially, function *known* checks whether the sample time of a block is known or not, and *allKonwn* checks whether the sample times of all input signals of the block in consideration are known or not. From now on, we only consider diagram for which the sample time of any block has been them is computed and thus known.

3.2.2 Translating Wires

In general, wires in Simulink diagrams can be considered as a special form of signals, and thus can be represented as variables. On the other hand, as seen from below, when a diagram is partitioned into a set of sub-diagrams, we will model each wire between any two sub-diagrams as a pair of input and output channels for transmitting values.

3.2.3 Separating Diagrams

We introduce the most important step in the translation of a Simulink diagram: separating the diagram to a set of connected sub-diagrams. We classify wires to three different kinds and then partition a diagram 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. The general case in which the user options for control are allowed will be discussed in subsection 3.2.7.

Algorithm 2 Separating diagrams

Require: A diagram *diag*
Ensure: Return a partition *partition* of the diagram

```
 1: new partition
 2: for all block in diag do
 3:   if !visited(block) then
 4:     new scc; scc.add(block); setVisited(block);
 5:     new bs; bs.add(block);
 6:     while !bs.empty() do
 7:       b ← bs.top();
 8:       for all cb in b.getConBlocks() do
 9:         if isShared(b, cb)∧!scc.contains(cb) then
10:           bs.add(cb); scc.add(cb); setVisited(cb);
11:         end if
12:       end for
13:       bs.remove();
14:     end while
15:     partition.add(scc);
16:   end if
17: end for
```

Algorithm 2 is given according to the strategy presented above. At the beginning, an empty list *partition* is allocated. Then the algorithm applies depth-first search to find out all connected components with the same type as a partition recursively. Thus, for each block *block* that has not been visited yet, a list *scc* is allocated to store the names of blocks of the whole connected sub-diagram that is reachable from

block. It first adds *block* to a local list *bs* and sets this block visited. If *bs* is not empty, then pops the top element from *bs* to *b*; then for any unvisited block *cb* that is connected to *b*, if the wires between *cb* and *b* are shared variables, denoted by *isShared*, then according to our strategy, we put *cb* to the same partition as *b*, and set *cb* as visited; after all the connected blocks of *b* are visited, *b* is removed from *bs* and the new top element of *bs* will be considered recursively, till *bs* becomes empty. Finally, *scc* is added into *partition*, and another unvisited block starts recursively, till all the blocks of the diagram are visited.

On termination, each element of *partition* is a list of names of connected blocks with the same type.

3.2.4 Translating Continuous Diagrams

Algorithm 3 Translating continuous diagrams

Require: A continuous diagram *diag*
Ensure: Return an HCSP process *proc*
1: $init.setEmpty()$; $comms.setEmpty()$
2: **for all** *block* **in** *diag* **do**
3: $init \leftarrow init \fatsemi getInit(block)$
4: $comms \leftarrow comms \cup getComms(block)$
5: **end for**
6: $procR.setEmpty()$
7: $newDiffs \leftarrow Cartesian(diag.getDiffs())$
8: **for all** (dq, b) **in** *newDiffs* **do**
9: $procR \leftarrow procR \fatsemi \langle dq \& b \rangle \trianglerighteq [\!]_{io \in comms} io \rightarrow \text{skip}$
10: **end for**
11: $proc \leftarrow init \fatsemi (procR)^*$

Algorithm 3 translates a continuous diagram to an HCSP process, which is represented by a string here. We introduce two variables *init* and *comms* to represent the initial process and the interrupting communications of the final HCSP process respectively. At the beginning, both of them are set to be empty strings. For any continuous *block* of this diagram, we extract its initial process and interrupting communications by using *getInit*() and *getComms*() defined in Sec. 3.1 respectively. Because the variables and input/output channels of different continuous blocks are disjoint, the initial process and communications of the whole process can be defined by sequential composition and the union of those of all blocks, both represented in lines 3 and 4. To distinguish from the sequential composition of HCSP, we write $\fatsemi$ to stand for sequential composition in algorithms.

We then construct the process *procR* for repetition. The function *getDiffs*(*diag*) returns the tuple consisting of the sets of differential equations of all blocks in *diag*. The set of differential equations for the diagram can then be defined as the Cartesian of the tuple. For instances, assume there are two blocks and the sets of differential equations of them are $\{f1\&b1, f2\&b2\}$ and $\{g1\&c1, g2\&c2\}$ resp., then

$$Cartesian(\{f1\&b1, f2\&b2\}, \{g1\&c1, g2\&c2\}) \stackrel{\text{def}}{=} \{f1g1\&b1 \wedge c1, f1g2\&b1 \wedge c2, f2g1\&b2 \wedge c1, f2g2\&b2 \wedge c2\}$$

For each differential equation in the resulting Cartesian set, we add the interrupting communications to it in line 9. The process *procR* can finally be defined as the sequential composition of all differential equations with interrupting communications.

3.2.5 Translating Discrete Diagrams

Algorithm 4 Translating discrete diagrams

Require: A discrete diagram *diag*
Ensure: Return an HCSP process *proc*
1: $odiag \leftarrow order(diag)$; $init.setEmpty()$;
2: $cin.setEmpty()$; $cout.setEmpty()$;
3: **for all** *block* **in** *odiag* **do**
4: $init \leftarrow init \fatsemi getInit(block)$
5: $cin \leftarrow cin \fatsemi getCin(block)$
6: $cout \leftarrow cout \fatsemi getCout(block)$
7: **end for**
8: $bc \leftarrow GCD(odiag.getst())$
9: $procR.setEmpty()$
10: **for all** *block* **in** *odiag* **do**
11: $procR \leftarrow (procR \fatsemi (block.st \mid t \rightarrow block.comp))$
12: **end for**
13: $procR \leftarrow cin \fatsemi procR \fatsemi cout$
14: $procR \leftarrow procR \fatsemi (temp := t) \fatsemi \langle \dot{t} = 1 \& t < temp + bc \rangle$
15: $proc \leftarrow init \fatsemi (t := 0) \fatsemi (procR)^*$

Algorithm 4 translates a discrete diagram to an HCSP process. Based on the restriction that there is no logic loop of discrete blocks, thus we can find an order of discrete blocks of the diagram such that a block can only rely on the blocks prior to it. The ordered blocks are denoted by *odiag*. We introduce *init*, *cin* and *cout* to represent the initial process, the input and output of the final HCSP process respectively. All of them are initialized to be empty, and calculated as the sequential composition of those of the blocks in *odiag* respectively in line 3-7. We use *bc* to represent the sample time of the final process, and is defined as the greatest common divisor of the sample times of all blocks.

We then construct the process *procR* by repetition, which will be taken every *bc* time units. Starting from time *temp* (whose value in the first period is 0), for all blocks *block*, if the time *t* is divided by the sample time of *block*, represented by $(block.st \mid t)$, then the computation of *block*, i.e. *block.comp* will be performed in this period. Finally, we reach the process *procR* by adding input and output processes before and after the computation respectively.

3.2.6 Translating Subsystems

So far we have presented the translation of Simulink diagrams. In this section, we will define how to translate Simulink subsystems, that are composed of a set of blocks, diagrams, and other subsystems. The subsystems can be classified to three different types: normal subsystems, triggered subsystems, and enabled subsystems. For simplifying the presentation, we consider the un-nested subsystems, i.e. the ones which do not contain other subsystems inside. The nested subsystems can be translated by applying the following methods recursively.

A. Normal subsystems.

We call subsystems that contain neither triggered nor enabled blocks inside as normal subsystems. 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

translated to a diagram, the translation of which has been presented above.

B. Triggered subsystems.

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 event*s: 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 Algorithm 4 to translate the resulting diagram, but with a little modification to line 13 as shown below, and finally reach the process for the triggered subsystem:

$$procR \leftarrow tri? \fatsemi cin \fatsemi procR \fatsemi cout$$

where *tri* represents the input triggering signal, indicating that the computation of the subsystem will be activated by signal *tri*? from outside, not periodically as normal since $bc = -1$ here.

The next step is to define the outside block that outputs the triggering signal. When the block is discrete, firstly, we do not consider the wire connecting the block and the triggered subsystem, i.e. the one transmitting the triggering signal, then according to our method proposed in Sec. 3.1, we can construct the process for the block and suppose that it has the structure as in Sec. 3.1; secondly, we define a new computation to be the following one:

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

That means, 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 a computation of the block 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,

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

When the block that outputs the triggering signal is continuous, similar to the discrete case, we construct the process for the continuous output block and suppose it has the structure as defined in Sec. 3.1. Then, we modify the differential equation part of the process as follows:

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

where B_{tri} defines the condition for occurring a trigger 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.

C. Enabled subsystems.

An enabled subsystem 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. In the following, to avoid unnecessary complication or even unexpected behavior of enabled subsystems, we assume that all normal blocks inside the enabled subsystem and the input signals of it have the same sample time with the enabling signal. In the literature, e.g. [16] and [4], the same restriction is assumed.

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 according to Algorithm 3.

On the other hand, 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 according to Algorithm 4.

3.2.7 User Options for Translation

A. Options in Separating the Diagram.

As defined in Sec. 3.2.3, when separating a diagram, we assume that the control represented by discrete blocks is centralized, thus we put all connected discrete blocks into one partition. We loosen the restriction here and allow users to decide whether the control is centralized or distributed. When the control represented by two connected discrete blocks is distributed, then we need to separate the two blocks and put them to two different partitions. We say a control option made by user is valid, if and only if after separating the diagram, the two ends of wires connecting any two distributed discrete blocks must reside in two different sub-diagrams.

Now we define how to translate two distributed discrete blocks. For ease of presentation below, according to the

direction of transmitting signals, we call one of them source block and the other target block. Assume the sample times of the source and target blocks are p and q respectively. When p is not equal to q, we need to negotiate about the values output from source and input to target. Assume the set of wires between these two blocks is I, and for each wire $i \in I$, the signal transmitted along it is s_i. For constructing the process for the distributed blocks, we first cut all the wires in I and define a buffer *Buf* in between,

$$Buf \widehat{=} \|_{i \in I} \ (fun(ch_{ini}?s_i, ch_{outi}!s_i, p, q))^* \ ,$$

where ch_{in_i}, ch_{out_i} are added to represent the input and output channels transmitting signal s_i from source block and to the target block respectively, and *fun* is defined for negotiation. The definition of *fun* makes sure that for each input signal, the target block always gets its newest value in the last nearest period of the source block. To achieve this, we need to record the occurrence of input and output processes $ch_{ini}?s_i, ch_{outi}!s_i$ in the time interval equal to the least common multiple of sample times p and q. For example,

$$fun(ch_{in}?s, ch_{out}!s, 2, 3) \widehat{=} ch_{in}?s; ch_{out}!s; ch_{in}?s; ch_{out}!s; ch_{in}?s$$

After *Buf* is defined, we translate the source and target blocks according to our method presented in Sec.3.1, and then by putting the resulting processes in parallel composition with *Buf*, we reach the process for the distributed discrete blocks finally.

On the contrary, to be more flexible, for any two blocks that are not reachable from each other, we provide user an option to add a wire between them, which will not affect the semantics but only for translation purpose. With the addition of the wire, we can partition these two blocks into one group, and do the translation accordingly.

B. Options in Abstraction.

Details may help us to figure out what actually happens, however sometimes too much of them may stop us to find the truth. We provide options for users to abstract away the details of Simulink blocks when the translation is being done. Instead of translating directly the actual block, we ask user to define the semantics of the block manually and use the semantics for translation. The format of the semantics must conform to two rules: for a continuous block, a list of differential equations with domains must be provided; for a discrete block, the sample time and computation must be provided.

4. IMPLEMENTATION

We implement the above translation algorithms in a protypical tool, called S2H, [3] in Java. The tool S2H takes a Simulink model from MATLAB (in xml format) as input and generates an annotated HCSP model as output, which is written in the input syntax of HHL prover. The generated HCSP model consists of four files for variable definitions, process definitions, assertion definitions, and a goal to be proved, respectively. Before verifying the model in HHL prover, we need to refine the definitions of assertions and goal, to make them more precise to reflect the actual requirement we expect to prove. But it should be pointed out that modification to the files for variable and process definitions are not allowed. Finally, the refined HCSP model, that is in the form of HHL specifications, is verified by HHL prover. The whole verification architecture is shown in Fig. 2.

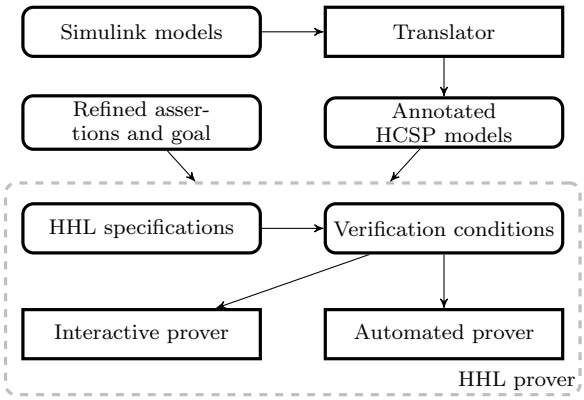

Figure 2: Verification Architecture of Simulink Models

For using the tool S2H, user should install Java Runtime Environment on the machine first, and then set two environmental variables for the paths of Isabelle and HHL prover respectively.

5. CASE STUDY

In this section, we illustrate our approach by modelling and verifying a combined operational scenario of Chinese Train Control System level 3 (CTCS-3) with respect to its System Requiement Specification (SRS). The operating behavior of CTCS-3 is specified by 14 basic scenarios, all of which cooperate with each other to constitute normal functionality of train control system. The combined scenario considered here integrates movement authority, level transition, and mode transition of CTCS-3.

The movement authority (MA) scenario is the basis to guarantee trains not to collide with each other. In this scenario, a train applies for MA from Radio Block Center (RBC) in CTCS-3 or Train Control Center in the backup system CTCS-2, and if the application succeeds, the train gets the permission to move but only within the MA it owns. An MA is composed of a sequence of segments. For each segment *seg*, we use $seg.v_1$ and $seg.v_2$ to represent the speed limits for which the train must implement emergency brake and normal brake (thus $v_1 \geq v_2$), *seg.e* the end point of the segment, and *seg.mode* the operating mode of the train in the segment. Two modes are considered in this case study: Full Supervision (FS), for which a train knows the complete information including its MA, line and train data, thus it can move with normal speed; Calling On (CO), for which a train cannot confirm cleared routes, thus it must move with limited speed. Especially, for safety consideration, for a train under CTCS-3, RBC can only grant it the MAs before a CO segment, and the train needs to ask the permission of the driver before moving into the CO segment. Thus, both the speed limits for CO segments are initialized to be 0 in CTCS-3.

Given an MA, we define for each segment *seg* the static

[3]The tool and the proof of the case study in next section can be found at
`https://www.github.com/submission/emsoft2013`

speed profile as the region formed by the two speed limits of the segment, i.e. $v \leq seg.v_1$ and $v \leq seg.v_2$, and the dynamic speed profile as the one by calculating down to the higher speed limit of next segment (i.e. $next(seg)$) taking into account the train's maximum deceleration (i.e. b), i.e. $v^2 + 2b\,s \leq next(seg).v_1^2 + 2b\,seg.e$. It is required that the train must move within the static and dynamic speed profiles.

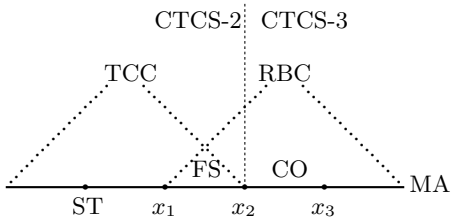

Figure 3: Level and mode transition

The Combined Scenario.

The combined scenario is shown in Fig. 3, which occurs under the following situation: the train has got enough MA to complete the combined scenario; there are two adjacent segments in the MA, divided by location x_2, and at x_2, the level transition from CTCS-2 to CTCS-3, and the mode transition from FS to CO, will occur simultaneously; the train stops initially at location x_1, and has an agreement from RBC to start level transition at x_1 and complete the level transition at x_2.

Under these conditions, the level transition scenario occurs as follows: when the train moves between location x_1 and x_2, it will be co-supervised by CTCS-2 and CTCS-3 (for this case we will model the level of the train as 2.5). One consequence is that the train must conform to the speed profiles for both control systems. Thus, x_2 as the starting point of the CO segment, both the speed limits for it will be 0; and then when the train reaches location x_2, the level of the train is set to 3 and the level transition completes. The mode transition scenario behaves as follows: if the train moves under level 2, it can update the mode to CO directly at location x_2; otherwise if it moves under level 3, it needs to ask the permission from the driver to enter the CO segment. If the driver says true, the speed limits of the CO segment will be reset to be $40km/h$ and $45km/h$ respectively, and as a consequence, the train updates its mode to be CO and passes x_2 with a positive speed.

5.1 Modeling in Simulink

Fig. 4 shows the top-level view of the Simulink model for the combined scenario, with the following explanations:

- A set of constants and variables are defined and initialized. Constants v_{11} and v_{12} represent the speed limits for FS segment, with values 105 and 100, and x_2 and *eoa* locations for transition and the end of MA, with values 10 and 20; variables $s, v, a, level$ and $mode$ represent the trajectory, velocity, acceleration, level, and mode of the train respectively, and v_{21} and v_{22} the speed limits for CO segment. In blocks IC1-IC4, variables $level, mode, v_{21}$ and v_{22} are initialized to 2.5, 0 (for representing FS), 0, 0 respectively.

- The continuous plant is defined by a sub-system in Simulink, and it models the movement of train using differential equation $\langle \dot{s} = v, \dot{v} = a \rangle$, with initial values 0, 0 for s, v respectively. a is initialized to 1 by block IC5.

- Blocks $B0 - B5$ act like sensors, and observes the states of the train periodically (we set the period to be 0.125s). $b0 - b5$ are logical formulas for monitoring movement of the train, and are modelled using MATLAB functions. For instances, the train is required to move forward (with non-negative speed), and move within the static profile, the dynamic profile of the segment, and as soon as they are violated, $b0, b1, b2$ will be activated respectively; moreover, when the train completes level and mode transitions, $b3, b4$ will be activated respectively; and when the train gets the permission from the driver to enter CO segment, $b5$ will be activated.

- The controller is also defined by a sub-system in Simulink, and it models the discrete computation that must be taken to the train whenever $b0 - b5$ become true. For instances, if the train reaches the speed profile, a needs to be reset to be negative; when the train performs mode transition in CTCS-3, if it gets permission from the driver to enter the CO segment, v_{21} and v_{22} will be reset to 40 and 45 respectively; if the level transition and mode transition succeed, $mode$ and $level$ will be reset to 1 (for representing CO) and 3 respectively.

  After the related variables for the train are reset by the controller, they are translated to the train plant as inputs, and a new period starts.

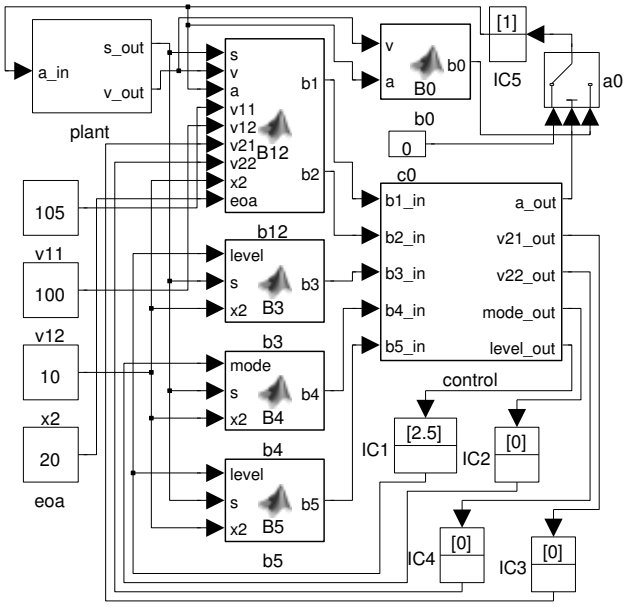

Figure 4: The top-level view of Simulink model

In all, the model for the combined scenario contains 16 blocks, and two sub-systems for the train and controller, which in turn are composed of 26 and 5 blocks respectively.

Results.

After setting the constants and initial values for variables, we get the simulation result, shown in Fig. 5, which indicates that the train will stop at location x_2. Following this, we translate the Simulink model for the combined scenario into HCSP model using tool S2H, which in detail consists of four files *varDef*, *procDef*, *assertDef* and *goal*, 488 lines of code in all We refine the assertions in *assertDef* and the goal in *goal* according to the property to be proved, and then verify the refined model in HHL prover. The verification is more general than simulation: the simulation asks for the exact (initial) values of the constants and variables, e.g. normal acceleration and brake acceleration; however, during verification, we abstract the values away by the method introduced in Sec. 3.2.7, i.e. from 1 and -0.1 to intervals $[-1, 1]$ and $[-1, 0)$ respectively.

Using HHL prover, we finally prove the following goal as a lemma:

```
lemma goal : "{True,True} P {plant_s_1 <=(Real 10),
True; (l = Real 0) | (high (plant_s_1 <=(Real 10)))
,True}"
```

where the postcondition together with the history formula indicate that the train never moves across location x_2, i.e. `10` here.

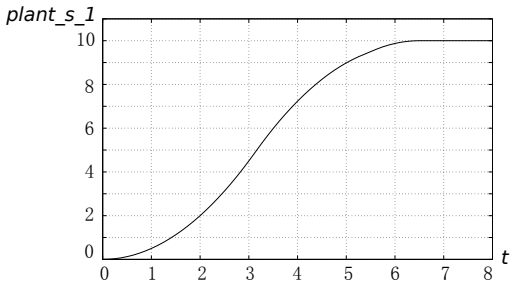

Figure 5: The result of simulation

6. CONCLUSION AND FUTURE WORK

This paper presents an automatic translation from Simulink models to HCSP processes. The resulting HCSP processes can not only provide a formal semantics for Simulink models, but also they can be verified by an HHL prover we have implemented in proof assistant Isabelle/HOL. We demonstrate our approach by considering a case study on a combined scenario of Chinese Train Control System at Level 3. Both the simulation result in Simulink and the verification result in HHL prover indicate a design error in CTCS-3.

For future work, we will investigate the use of HCSP to formalize Simulink/Stateflow models with flexible control flow diagrams; and on the other hand, to apply our approach to more practical hybrid systems, especially the other scenarios of CTCS-3 and their combinations, for safety analysis and verification.

7. REFERENCES

[1] *Simulink User's Guide*, 2013. `http://www.mathworks.com/help/pdf_doc/simulink/sl_using.pdf`.

[2] A. Agrawal, G. Simon, and G. Karsai. Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations. *Electr. Notes Theor. Comput. Sci.*, 109:43–56, 2004.

[3] A. Cavalcanti, P. Clayton, and C. O'Halloran. Control law diagrams in circus. In *FM'05*, volume 3582 of *LNCS*, pages 253–268, 2005.

[4] C. Chen, J. S. Dong, and J. Sun. A formal framework for modeling and validating Simulink diagrams. *Formal Asp. Comput.*, 21(5):451–483, 2009.

[5] E. Clarke and P. Zuliani. Statistical model checking for cyber-physical systems. In *ATVA'11*, volume 6996 of *LNCS*, pages 1–12. 2011.

[6] G. Hamon and J. Rushby. An operational semantics for Stateflow. *Int. J. Softw. Tools Technol. Transf.*, 9(5):447–456, 2007.

[7] J. He. From CSP to hybrid systems. In *A Classical Mind, Essays in Honour of C.A.R. Hoare*, pages 171–189. Prentice Hall International (UK) Ltd., 1994.

[8] C. Herde, A. Eggers, M. Fränzle, and T. Teige. Analysis of hybrid systems using HySAT. In *ICONS 2008*, pages 196–201. IEEE Computer Society, 2008.

[9] C. A. R. Hoare. *Communicating Sequential Processes*. PrenticeH, 1985.

[10] J. Liu, J. Lv, Z. Quan, N. Zhan, H. Zhao, C. Zhou, and L. Zou. A calculus for hybrid CSP. In *APLAS'10*, pages 1–15, 2010.

[11] J. Liu, N. Zhan, and H. Zhao. Computing semi-algebraic invariants for polynomial dynamical systems. In *EMSOFT '11*, pages 97–106. ACM, 2011.

[12] B. Meenakshi, A. Bhatnagar, and S. Roy. Tool for translating Simulink models into input language of a model checker. In *ICFEM'06*, volume 4260 of *LNCS*, pages 606–620, 2006.

[13] B. C. Moszkowski and Z. Manna. Reasoning in interval temporal logic. In *Logic of Programs*, pages 371–382. Springer, 1983.

[14] A. Rauh, C. Sibert, and H. Aschemann. Verified simulation and optimization of dynimc systems with friction and hysteresis. In *Proc. of ENOC 2011*, Rome, 2011.

[15] A. Tiwari. Formal semantics and analysis methods for Simulink Stateflow models. Technical report, SRI International, 2002.

[16] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time Simulink to Lustre. *ACM Trans. Embedded Comput. Syst.*, 4(4):779–818, 2005.

[17] N. Zhan, S. Wang, and H. Zhao. Formal modelling, analysis, verification of hybrid systems. Invited by ICTAC-School 2013, published as a book chapter in LNCS 8050, Springer-Verlag, 2013.

[18] C. Zhou and X. Li. A mean-value duration calculus. In *A Classical Mind, Essays in Honour of C.A.R. Hoare*, pages 432–451. Prentice-Hall International, 1994.

[19] C. Zhou, J. Wang, and A. P. Ravn. A formal description of hybrid systems. In *Hybrid systems*, pages 511–530. Springer, 1996.

[20] L. Zou, J. Lv, S. Wang, N. Zhan, T. Tang, L. Yuan, and Y. Liu. Verifying chinese train control system under a combined scenario by theorem proving. VSTTE 2013.