Formal Verification of Simulink/Stateflow Diagrams

Liang Zou[1], Naijun Zhan[1], Shuling Wang[1](✉), and Martin Fränzle[2]

[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
wangsl@ios.ac.cn
[2] Department of Computing Science, C. V. Ossietzky Universität Oldenburg, Oldenburg, Germany

Abstract. Simulink is an industrial de-facto standard for building executable models of control systems and their environments. Stateflow is a toolbox used to model reactive systems via hierarchical statecharts within a Simulink model, extending Simulink's scope to event-driven and hybrid forms of embedded control. In safety-critical control systems, exhaustive coverage of system dynamics by formal verification would be desirable, being based on a formal semantics of combined Simulink/Stateflow graphical models. In our previous work, we addressed the problem of formal verification of pure Simulink diagrams via an encoding into HCSP, a formal modelling language encoding hybrid system dynamics by means of an extension of CSP. In this paper, we extend the approach to cover Simulink models containing Stateflow components also. The transformation from Simulink/Stateflow to HCSP is fully automatic, and the formal verification of the encoding is supported by a Hybrid Hoare Logic (HHL) prover based on Isabelle/HOL. We demonstrate our approach by a real world case study originating from the Chinese High-speed Train Control System (CTCS-3).

Keywords: Simulink/Stateflow · Formal verification · Hybrid CSP · Hybrid Hoare Logic

1 Introduction

Simulink [1] is an environment for the model-based analysis and design of signal-processing systems. Being based on a large palette of individually simple function blocks and 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. 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 have become a de-facto standard in the embedded systems industry, as the circuit analogy of Simulink and the hierarchy mechanisms of Stateflow, which include sequential

B. Finkbeiner et al. (Eds.): ATVA 2015, LNCS 9364, pp. 464–481, 2015.
DOI: 10.1007/978-3-319-24953-7_33

and concurrent state composition, help to comprehend the massively concurrent mixed-signal dynamics arising from the multiple data paths and state-rich reactive components present in modern embedded control systems.

System analysis and design validation within Simulink/Stateflow are based on numerical simulation, rendering it prone to incomplete coverage of open systems and possible unsoundness of analysis results due to numerical error. There are various partial remedies to this problem: Statistical model checking (SMC, cf. e.g. [6]) tries to save the virtues of simulation-based analysis, namely low computational complexity and moderate constraints on the shape of models, while addressing the coverage problem through a rigorous statistical interpretation of simulation results. Yet being a simulation-based procedure that employs the standard simulators, it still is subject to the soundness issue of numerical simulation. Set-based numerical simulation computing enclosures of system trajectories, as pioneered already in the sixties [14], could in principle resolve this problem, but variants handling non-smooth derivatives [16], as omnipresent in Simulink/Stateflow models, do not scale to the system sizes encountered in industrial models. The same applies to state-exploratory methods either computing (an overapproximation of) the reachable state space or (semi-)deciding a symbolic reachability criterion. These require to translate the Simulink model into the input language of a formal verification tool for hybrid discrete-continuous systems, be it an automaton-based language [3] or a symbolic description [9], 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. Fully automatic analysis of Simulink/Stateflow models is, however, hampered by Simulink's modeling paradigm of connecting numerous concurrently executing small blocks via continuous-time synchronous dataflow, yielding tightly coupled, fine-granular concurrency in the translated models, which is detrimental to state-exploratory analysis.

In this paper, we do therefore investigate the translation of Simulink/Stateflow into a process calculus with its richer set of composition primitives. We present an encoding of the semantics in terms of HCSP [11], a formal modelling language encoding hybrid system dynamics by means of an extension of CSP [10], thus providing the formal basis for an automatic translation from Simulink/Stateflow to HCSP. As analysis of HCSP models is supported by an interactive Hybrid Hoare Logic (HHL) prover based on Isabelle/HOL, this provides a gateway to mechanized verification, which we demonstrate on a real world case study originating from the Chinese High-speed Train Control System. This research extends our previous work [23] addressing Simulink only.

Related Work. There has been a range of work on translating Simulink into modelling formalisms supported by analysis and verification tools. Tripakis *et al.* [19] present 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.* [4] put forth a semantics for discrete-time Simulink diagrams using Circus, a combination of Z and CSP. Meenakshi *et al.* [12] propose an algorithm translating a Simulink subset into the input language of the finite-state model checker NuSMV. Chen *et al.* [5] present a translation of Simulink models to the real-time specification language Timed Interval Calculus (TIC), which can accommodate continuous Simulink diagrams directly, and they validate 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. In contrast, our previous work [23] can handle all continuous blocks by using the notions of differential equations and invariants in the HCSP encodings, and the use of a process calculus based language facilitates compositionality in both construction and verification.

Beyond the pure Simulink models considered in the above approaches, models comprising reactive components triggered by and affecting Simulink's dataflow have also been studied recently. Hamon *et al.* [7] propose an operational semantics of Stateflow, which serves as a foundation for developing tools for formal analysis of Stateflow designs. Scaife *et al.* [17] translate a subset of Stateflow into Lustre for formal analysis. Tiwari [18] defines a formal semantics of Simulink/Stateflow using guarded pushdown automata, in which continuous dynamical systems modeled by Simulink are discretized, and he discusses how to verify a guarded sequence via type checking, model checking and theorem proving. Agrawal *et al.* [3] propose 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 model checkers for hybrid systems support only very restricted differential equations. Miller *et al.* [13] propose a method to translate discrete Simulink/Stateflow models into Lustre for formal analysis. In our approach, we relax these constraints and consider how to define a formal semantics for the combination of Simulink/Stateflow in general, including advanced features like *early return logic*, *history junction*, *nontermination* of Stateflow. We gain this flexibility by resorting to interactive theorem proving rather than automatic model checking for discharging the proof obligations.

2 Simulink/Stateflow and Hybrid CSP

In this section, we briefly introduce the source language Simulink/Stateflow, the target language Hybrid CSP (HCSP), the specification language Hybrid Hoare Logic and its prover. We expand only on the most relevant features of Simulink/Stateflow, and the reader is referred to [1,2] for more details.

2.1 Simulink

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

Figure 1 gives a Simulink model of train movement, comprising four blocks, including continuous integrator blocks v and p and discrete blocks c and acc. Block v outputs the velocity of the train by integrating acceleration acc over time; similarly, p outputs the distance of the train computed by timewise integration of velocity v, and acc computes acceleration from the constant provided by c and distance p.

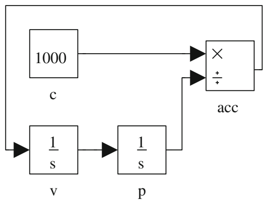


Fig. 1. A simple control system

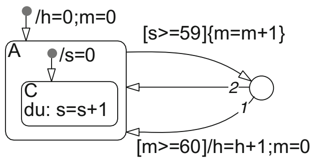


Fig. 2. A timer

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, fed with Simulink inputs and producing 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 can be active. In the following, we will explain the main ingredients of Stateflow.

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 the diagram, or may be local to it. A variable may also be set as input, output, or local, and moreover, it can be associated with an initial value if necessary.

States: States can be hierarchical, containing another Stateflow diagram. 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 such that there is no non-determinism in transition selection. A state may be associated with three optional types of actions: An *entry action* executed when the state is activated; a *during action* executed when no transition is enabled; and an *exit action* executed imemdaitely before a transition leaves the state. The actions of Stateflow may be either assignments, or emissions of events, etc.

States in an AND diagram are assigned priorities and their actions are actually executed in sequential order according to that priority.

Transitions: A transition is a connection between states. In Stateflow, it may be a complex transition network, consisting of several transitions joined by junctions. Default transitions with no source states or source junctions are allowed for OR diagrams, and they are used to select the active state when the OR diagram is activated. Each transition 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 action. All these components are optional. $cAct$ will be executed immediately when event E is triggered and condition C holds, while $tAct$ will is put in a queue first and executed after the corresponding transition is taken.

These syntactic components form the basis of the follwing execution semantics.

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

Broadcasting and Executing Transition: Each Stateflow diagram is activated at sample times periodically or by trigger events, depending on the diagram's settings. For the second case, as soon as one of the trigger event arrives, called *current event*, it 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; while for an OR diagram, it will find the active state of the diagram and be broadcasted 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.

Figure 2 gives an example of Stateflow. States A and C are activated initially, with variables h, m, and s being set to 0. A has a transition network to itself, enabled when s equals to 59. Once the network is enabled, the outgoing transition is executed, and thus m is increased by 1; then it will execute transition 1 due to its higher priority, increasing h by 1 and resetting m to 0 if m equals to 60. If transition 1 is not enabled, 2 is taken.

The combination of Simulink and Stateflow is exemplified by 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, updating the train's acceleration every minute to become $1000/p + m/100$. We then replace blocks acc and c by the modified diagram.

2.3 Hybrid CSP (HCSP)

HCSP [8,20,21] is a formal language for describing hybrid systems. It is an extension of Hoare's Communicating Sequential Processes (CSP) by timing constructs, interrupts, and differential equations modelling continuous evolution. Data exchange among processes is confined to instantaneous synchronous communication, avoiding shared variables as present in Simulink. For a comprehensive introduction to HCSP refer to [20].

The syntax of HCSP is given as follows:

$$\begin{aligned} P ::= {} & \mathrm{skip} \mid x := e \mid ch?x \mid ch!e \mid P;Q \mid B \to P \mid P \sqcup Q \mid X \\ & \mid \mu X.P \mid \langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \mid \langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \unrhd []_{i \in I}(io_i \to Q_i) \\ S ::= {} & P \mid S \| S \end{aligned}$$

Here, io_i stands for a communication event, i.e., $ch?x$ or $ch!e$, P, Q, Q_i for HCSP processes, x and s for variables, X for process variables, and ch for channel names. B and e are Boolean and arithmetic expressions. The intended meaning of the individual constructs is as follows:

- skip, $x := e$ (assignment), $ch?x$ (input), $ch!e$ (output), X, $P;Q$ (sequential composition), $B \to P$ (conditional statement), $P \sqcup Q$ (internal choice), $\mu X.P$ (recursion) and $P \| Q$ (parallel composition) are understood as in CSP.
- $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle$ is the continuous evolution statement, where s represents a vector of real variables and $\dot{s}$ their derivatives. It forces s to continuously evolve according to the differential equation $\mathcal{F}(s, \dot{s}) = 0$ as long as condition B holds, and terminates immediately when B turns false.
- $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \unrhd []_{i \in I}(io_i \to Q_i)$ behaves like $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle$, except that the continuous evolution is preempted as soon as one of the communications io_i occurs, then being followed by the respective Q_i. If the continuous evolution terminates before any preemption then the overall process terminates directly.

2.4 Hybrid Hoare Logic (HHL)

In [11], we have extended Hoare Logic to hybrid systems by adding history formulas to describe continuous properties that hold throughout the whole execution of HCSP processes. The history formulas are defined within Duration Calculus (DC), which is an extension of Interval Temporal Logic (ITL) [15] for specifying and reasoning about real-time systems. In HHL, each specification for a sequential process P takes the form $\{Pre\}\,P\,\{Post; HF\}$, where $Pre, Post$ are traditional pre-/post-conditions on variables expressed in first-order logic, and HF is a history formula in DC recording the execution history of P, including real-time and continuous properties. The specification for a parallel process is then obtained by assigning to each sequential component an HHL condition $\{Pre_1, Pre_2\}\,P_1\|P_2\,\{Post_1, Post_2; HF_1, HF_2\}$. Each HCSP construct is axiomatized within HHL by a set of axioms and inference rules. A more comprehensive explanation of HHL can be found at [11,20]. For tool support, we have implemented an interactive theorem prover for HHL based on Isabelle/HOL [20,22]. The tool can be downloaded from https://github.com/submission/Sim2HCSP.

3 From Simulink/Stateflow to HCSP

In previous work [23], we have translated a subset of Simulink into HCSP processes and then applied the HHL prover to the target processes for formal verification. In this section, we will first discuss the translation of Stateflow models into HCSP and then show how this translation can be integrated with our previous work, so that we get a translation from Simulink/Stateflow graphical models into HCSP. We first consider each of the main ingredients of Stateflow, including transition networks, broadcasting, states etc., and then provide a general template for translating a Stateflow diagram.

3.1 Transition Networks

For translating a transition, two main issues should be considered: first, a transition network may consist of several transitions joined by junctions, thus we need to traverse the transition network to find each valid transition path; second, a transition may have super-structure, i.e. the source and target states can be at different levels, so an appropriate ordered sequence of exit and entry actions has to be executed upo transitioning.

Algorithm 1 presents the main recursive algorithm for translating a transition network with source state S and triggering event E to a HCSP process. The boolean variable *done* indicates whether a valid transition that connects S and a destination state has been executed in current transition network. **TTN** adopts depth-first search. In order to deal with possible loops in a transition network N, we introduce a process variable P_J for each junction J in N, and P_{set} to store the process equation corresponding to P_J defined by recursively calling **TTN** starting from $J.\,target$ and $cPath$ represent current location and current

Algorithm 1. Translating Transition Networks ($\mathbf{TTN}(S, E, done)$)

```
Require: A source state $S$, a triggering event $E$, and a boolean variable $done$
Ensure: A HCSP process $proc$ representing the transition network starting from $S$
    and a set of HCSP processes $P_{set}$ that assist the definition of $proc$
    Vars $cPath$, $target$, $proc =$ "$ACT_{LOC} = ACT$", $P_{set} = \emptyset$;
 1: for ($cPath = 1$; $cPath \leq S.tSize$; $cPath = cPath + 1$) do
 2:   $t = S.getTrans(cPath)$;  $target = t.getTarget()$;
 3:   $proc = proc ⨟$ "($E == t.getEvent()$ & $t.getCond()$ & $\neg done$) $\rightarrow$
            ($t.getCAction()$; $ACT := (ACT_{LOC}; t.getTAction())$);";
 4:   if $target.isState()$ then
 5:     Find common father for states $S$, $target$, and all junctions along the intermediate path;
 6:     $proc = proc ⨟$ "$P_{exit}$; $ACT$; $P_{entry}$; $done := True$);";
 7:   else
 8:     if $\neg P_{set}.contains(P_{target})$ then
 9:       $P_{set}.put(P_{target}, \mathbf{TTN}(target, E, done))$;
10:     end if
11:     $proc = proc ⨟ P_{target} ⨟$ ");";
12:   end if
13: end for
```

transition being traversed respectively. ACT is a process variable to accumulate the transition actions that have been traversed.

proc is a string that represents the constructed HCSP process, which is put within double quotes in the algorithm. Operator ⨟ is used to concatenate two strings. At the beginning, *proc* is initialized by "$ACT := skip;$", then it is extended by concatenating the process constructed by calling **TTN** $(S, E, done)$. Finally, by solving equations in P_{set}, each occurrence of P_J in *proc* can be replaced by the corresponding solution.

We explain Algorithm 1 now. It starts by initializing ACT_{LOC} by ACT, and P_{set} by $\emptyset$. The **for** loop then translates the transitions of S according to their order:

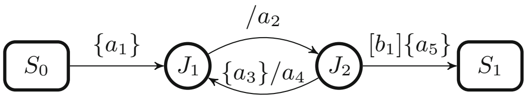
1. line 2 gets the transition according to the path number $cPath$, assigns it to t, and stores its target in $target$;
2. at line 3, if the event of t is same as E, the condition of t is satisfied, and $done$ is $False$, then the condition action is taken, meanwhile the transition action is put in ACT. Whenever the previous branch of current source state or junction failed, the corresponding accumulated actions in ACT will be discarded by recovering ACT to the value stored in ACT_{LOC} at the first entry of this source state or junction. Note that ACT plays the role of a stack here.
3. at lines 4–6, if $target$ is a state (rather than a junction), indicating that a valid transition is found, then find the common father of S, $target$, and all junctions along the intermediate path, exit from S step by step (by process P_{exit}), execute the transition actions ACT that are accumulated for S, enter to $target$ step by step (by P_{entry}), and set $done$ to be $True$, meaning that

no further transition is needed, and *proc* is closed in this loop with right a bracket.

4. otherwise (line 8–11), if *target* is a junction, not traversed yet (i.e., not in P_{set}), then call **TTN** to get the definition of P_{target} (i.e. the HCSP process constructed from $\mathbf{TTN}(target, E, done)$) and store definition equation in P_{set}, finally P_{target} and a right bracket is added at the end of *proc*.
5. at line 12, the current iteration of the **for** loop completes, and the next iteration will start to consider the next transition of S.

Notice that for some transitions, the triggering event (or condition) may not exist. For such case, the guard $E = t.getEvent()$ (or $t.getCond()$) in line 3 will set to $True$. Since the number of junctions is finite, **TTN** is called finitely often. Hence, Algorithm 1 terminates for any finite transition networks, no matter whether it contains loop or not.

Example 1. Figure 3 presents an example of a transition network with a loop. By applying the algorithm, we obtain the process listed below the statechart. In the process, we set a state S to active or inactive by assigning the corresponding Boolean variable a_S to 1 or 0 respectively.



$$\begin{tikzcd} S_0 \arrow[r, "\{a_1\}"] & J_1 \arrow[r, bend left, "/a_2"] & J_2 \arrow[l, bend left, "\{a_3\}/a_4"] \arrow[r, "{[b_1]\{a_5\}}"] & S_1 \end{tikzcd}$$

proc $= ACT := skip; ACT_{S_0} = ACT; \neg done \to (a_1; P_{J_1})$

The definition of P_{J_1} can be obtained by solving the equations in P_{set}:

$P_{J_1} = ACT_{J_1} = ACT; \neg done \to (ACT := (ACT_{J_1}; a_2); P_{J_2})$

$P_{J_2} = ACT_{J_2} = ACT; ((b_1 \wedge \neg done) \to (a_5; a_{S_0} := 0; ACT; a_{S_1} := 1; done := True));$
$\quad \neg done \to (a_3; ACT := (ACT_{J_2}; a_4); P_{J_1})$

Fig. 3. A transition network with a circle

3.2 Broadcasting and Monitor Process

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 broadcasting, a new local event may be emitted inside some sub-diagram, interrupting the current execution due to the local event. After the completion of processing the local event, the interrupted execution will be resumed.

We define a monitor process $\mathcal{M}$ in terms of HCSP to coordinate the execution of broadcasted events. The constant n stands for the number of parallel states of the current diagram, E for the 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.

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

As shown above, $\mathcal{M}$ initializes num to 0, and then repeats $\mathcal{M}_{main}$, which specifies how to trigger current diagram by current event as follows:

***num*=0:** The monitor waits for a triggering event E from outside first, modelled by P_{tri}, then receives the input data on $iVar$. Afterwards updates num to be 1, initializes both EL and NL to be empty, and then pushes E to stack EL and 1 to stack NL indicating the first sub-diagram to be triggered by E.

$1 \leq$ ***num*** $\leq n$: The monitor broadcasts current event to the num-th sub-diagram with the following three options:

1. either it broadcasts current event along channel BC_{num} to the num-th sub-diagram and sends the shared data along channel $VOut_{num}$ to it also;
2. or it receives a local event from the num-th sub-diagram along channel BR_{num}, which will interrupt the broadcasting of current event, then the local event will be broadcasted immediately to the first sub-diagram;
3. or it receives an acknowledgment from the num-th sub-diagram along channel BO_{num} to indicate that the broadcasting of E has been completed and at the same time receives the new shared data along VIn_{num}. Afterwards, it will broadcast the event to the $(num + 1)$-th sub-diagram if it exists.

num $= n + 1$**:** This indicates that the broadcasting of the current event has been finished. Consequently, the event and the corresponding sub-diagram are removed from the respective stack. If the resulting EL is empty, which indicates that the broadcast on current event E has completed, then sets num to become 0, outputs variable $oVar$, and waits for another triggering event from outside of the diagram. Otherwise, there still is some interrupted broadcast, and for such case, it will retrieve the previous event and its corresponding sub-diagram by reading the new top values from EL and NL and resume the interrupted broadcast event.

Note that if a diagram is triggered by sample time, then P_{tri} is simply defined as a wait statement of HCSP, i.e. **wait** T for sample time T.

3.3 Stateflow Diagrams

Finally, a Stateflow diagram can be defined 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. Especially, 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 no (entry/during/exit) action nor transitions associated to it.

$\mathcal{S}_i$ 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. 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 by calling Algorithm 1; 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, the monitor is notified about completion of the broadcasting and outputs the shared data.

Likewise, each sub-diagram (represented by P_{diag}) may be AND or OR diagram. Different from an outermost AND diagram, for simplicity, we define an AND sub-diagram in terms of 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.

Note that in the above, **TTN** returns an HCSP process corresponding to both outgoing and ingoing transitions of $\mathcal{S}_i$. Local events may be emitted during executing transition or state actions. In such a case, the current execution of the diagram needs to be interrupted for broadcasting the local event, and will be resumed after processing the broadcast is completed. For modeling this kind of preemption, we use general recursion of the form $\mu X.(P_1; X; P_2)$, X referring to the place where the local event is emitted.

$$\begin{aligned}
\mathcal{D} &\mathrel{\widehat{=}} \mathcal{M} \| \mathcal{S}_1 \| \cdots \| \mathcal{S}_n, \\
\mathcal{S}_i &\mathrel{\widehat{=}} P_{init}; P_{entry}; (BC_i?E;\ VOut_i?sv_i; \mathcal{S}_{du}; BO_i!;\quad VIn_i!sv_i)^*, \\
\mathcal{S}_{du} &\mathrel{\widehat{=}} done = False; \mathbf{TTN}(\mathcal{S}_i, E, done); \neg done \rightarrow (dur; P_{diag}), \\
P_{diag} &\mathrel{\widehat{=}} P_{and} \mid P_{or}, \quad P_{and} \mathrel{\widehat{=}} \mathcal{S}_{1_{du}}; \cdots; \mathcal{S}_{m_{du}}, \\
P_{or} &\mathrel{\widehat{=}} (a_{S_1} == 1 \rightarrow \mathcal{S}_{1_{du}}); \cdots; (a_{S_k} == 1 \rightarrow \mathcal{S}_{k_{du}}).
\end{aligned}$$

3.4 Handling Advanced Features of Stateflow

In this section, we discuss how to handle special features of Stateflow in the translation, including *early return logic*, *history junctiona*, *nontermination*, and so on. As discussed above, a Stateflow digram is translated into $\mathcal{D} \mathrel{\widehat{=}} \mathcal{M} \| \mathcal{S}_1 \| \cdots \| \mathcal{S}_n$. In the following discussion, we only consider how to revise the translation of S_i accordingly in order to address these advanced features, as the coordinated processes stay the same.

Early return logic prevents execution of an interrupt from returning to the interrupted action, thus discarding the rest of the execution of the interrupted action. E.g., in Fig. 4, activity in $A1$ is interrupted by broadcasting the local event e, which results in the control shifting to B. Thus, $A1$ becomes inactive. After the execution of B, the control will be shifted to A rather than $A1$, discarding $A2$.

We use the example in Fig. 4 to demonstrate how to deal with early return logic in our setting. By using activity control variables a_A, a_{A1}, a_{A2} and a_B, we revise the translation for the example diagram as follows:[1]

$$
\begin{aligned}
\mathrm{S}_1 = {} & a_A := 0; a_B := 0; a_A := 1; a_{A1} := 1; \\
& \mu X.\, (BC_1?E1; done_1 := \mathit{False}; (a_A == 1) \to ((\neg done_1 \wedge E1 == e) \to (\\
& (a_{A1} == 1) \to a_{A1} := 0; (a_{A2} == 1) \to a_{A2} := 0; a_A := 0; a_B := 1; done_1 := \mathit{True}); \\
& (\neg done_1) \to ((a_{A1} == 1) \to ((\neg done1) \to (BR_1!e; X; (a_{A1} == 1) \to (\\
& \qquad a_{A1} := 0; a_{A2} := 1; done_1 := \mathit{True}))); (a_{A2} == 1) \to skip)); \\
& (a_B == 1) \to skip; BO_1!)
\end{aligned}
$$

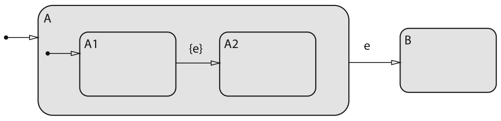


Fig. 4. Early return logic

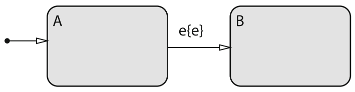


Fig. 5. Nontermination

A **history junction** memoizes control flow in an OR diagram for later reactivation. With a history junction in an OR diagram, the state marked by the default transition is only relevant for the first activation. Upon subsequent activation, control on te OR diagram returns to where the diagram was last left. To translate a *history junction*, we revise the definition of P_{entry}: We introduce a variable *history* to record the active state interrupted in the previous execution, and initialize it to the state pointed to by the default transition. P_{entry} is implemented simply by setting the active state to *history*.

Nontermination of Stateflow transitions is difficult to formalize and often not considered, like in [19]. Consider Fig. 5, where the statechart will exhibit nontermination whenever process S_1 receives an event e. In our approach, nontermination can easily be coped with. Regarding Fig. 5, we can translate it into an HCSP process as:

$$
\begin{aligned}
\mathrm{S}_1 = {} & a_A := 0; a_B := 0; a_A := 1; \\
& \mu X.\, (BC_1?E1; done1 := \mathit{False}; (a_A == 1) \to ((\neg done1 \wedge E1 == e) \to (BR_1!e; X; \\
& \qquad (a_A == 1) \to (a_A := 0; a_B := 1; done1 := \mathit{True}))); (a_B == 1) \to skip; BO_1!)
\end{aligned}
$$

[1] Please note that the example is an OR-diagram.

3.5 Combination of Simulink and Stateflow

Given a Simulink/Stateflow model, its Stateflow parts are translated into separate processes by the above algorithms. These processes are put in parallel with the processes obtained from the Simulink part, together forming the complete model of the system. The Simulink and Stateflow diagrams in parallel transmit data or events via channels. The communications between them are categorized into the following cases:

– As defined in Sect. 3.2, 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;
– as defined in Sect. 3.3, 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 the Simulink processes;
– the input/output variables and events inside Simulink part are handled as in our previous work [23].

3.6 Implementation

We had already implemented a tool **S2H** translating Simulink diagrams into HCSP processes as part of our previous work [23]. We have now extended **S2H** to support the translation of Stateflow diagrams, and we have named the new tool **S**im2**H**CSP. For each Stateflow diagram, it generates three files that correspond to the definitions of variables, processes, and assertions respectively. To use **S**im2**H**CSP, the user has to install the Java Runtime Environment and has to set two environment variables to point to the paths of Isabelle and HHL prover, respectively.

4 Case Study: Revisiting the Combined Scenario of CTCS-3

In our previous work [23], we modeled a combined operational scenario of Chinese Train Control System level 3 (CTCS-3) as a pure Simulink model, in which all the control behaviors of Radio Block Center (RBC), Train Control Center (TCC), Driver and so on are abstracted away as assumptions, as it is impossible to model such event-driven control behavior using Simulink. In this paper, we will revisit this example by modeling the event-driven control behaviour as Stateflow diagrams, and therefore give a complete Simulink/Stateflow model of this scenario by dropping the assumptions. In addition, we formally prove the Simulink/Stateflow model against the System Requiement Specification (SRS) by HHL Prover.

According to the SRS, a train needs to apply for movement authority (MA) from RBC under CTCS-3 or TCC under the backup system CTCS-2. If it succeeds, the train gets permission to move within the geometric extent of its MA. An MA is composed of a sequence of segments, each of which is associated with two kinds of speed limits $v_1 > v_2$, respectively for emergency brake intervention and service brake intervention, the end point e of the segment, and the operating mode *mode* of the segment.

Given an MA, as shown in Fig. 6, the static speed profile (solid line) is the region formed by the two speed limits, i.e., $v \leq seg.v_1$ and $v \leq seg.v_2$, where *seg* represents the current segment the train is running on; while the dynamic speed profile (dotted line) is calculated according to $v^2 + 2b\,s \leq next(seg).v_i^2 + 2b\,seg.e$ for $i = 1, 2$, where b represents the train's maximum deceleration and $next(seg)$ the next segment ahead of the train. At any time, the train must move within the static and dynamic speed profiles. For simplicity, in our modeling, we assume that each MA contains 3 segments, and whenever there is only one segment left, the train must apply for another MA extension.

The *combined scenario* is shown in Fig. 7: the train has got the MA till x_3; the control system is CTCS-2 before x_2, while CTCS-3 after x_2; and meanwhile, the moving mode is Full Supervision (FS) before x_2, while Calling On (CO) after x_2. So, in this scenario, the level transition and mode conversion will occur at x_2 simultaneously. By SRS, the train starts to apply for level transition from RBC at location ST . If RBC approves the request, the train will start the level upgrade from x_1, and when it reaches x_2, its control system is upgraded to CTCS-3, and the level transition completes. It should be noticed that, when the train moves between x_1 and x_2, it will be co-supervised by CTCS-2 and CTCS-3, thus it must follow the speed profiles of both control systems. In addition, under CTCS-3, a train is required to fully stop before starting a CO segment, and ask for the confirmation of the driver in order to enter the CO segment. As a result, both the speed limits at x_2, the starting point of a CO segment, will be 0 initially. Under CTCS-2, in contrast, the train will convert the mode to CO directly at x_2.

4.1 Modeling in Simulink/Stateflow

The top-level view of the Simulink model for the combined scenario is shown in Fig. 8. It consists of two sub-systems: the *plant* sub-system models the movement of the train by using the differential equation $\dot{s} = v$ and $\dot{v} = a$, with input a from the control sub-system; the *control* sub-system reads s and v from the plant periodically (every 0.125 s in our setting), based on which it computes the new acceleration a and sends it back to the plant; they together constitute a closed loop.

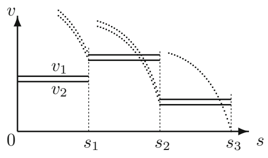


Fig. 6. Static and dynamic speed profiles

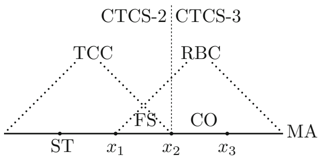


Fig. 7. Level and mode transition

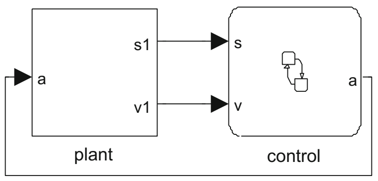


Fig. 8. The top-level view of Simulink model

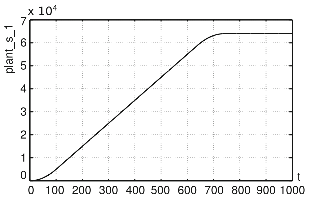


Fig. 9. The result of simulation

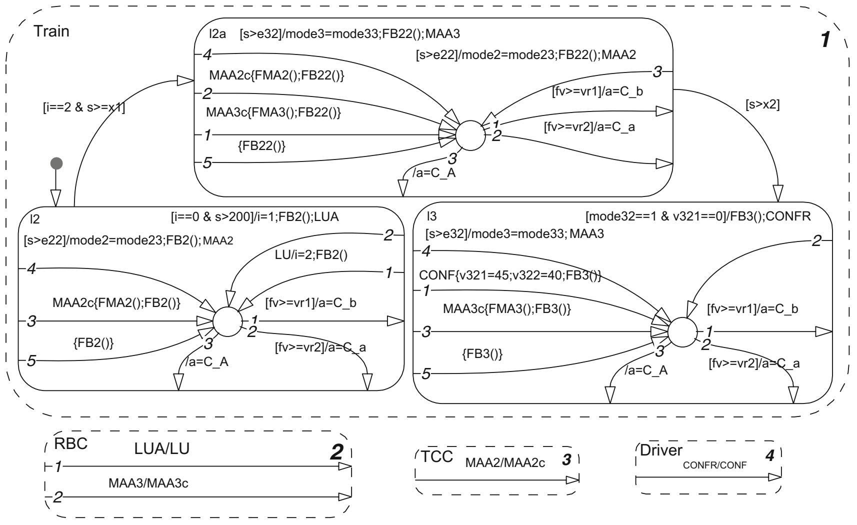


Fig. 10. The control model in Stateflow

The control sub-system is modeled as a Stateflow model, shown in Fig. 10. It is an AND diagram with four parallel states modeling the four components (i.e. *Train*, *RBC*, *TCC*, and *Driver*) involved in the scenario. A set of events, variables, and functions are introduced, for which we assume a naming convention that the first and second numbers always refer to the level and the MA segment on which the train is moving, resp., and the third to the type of speed limits, if existent. E.g., $e22$ represents the end of the second segment at level 2 (i.e. location x_2), etc. The *Train* state, featuring highest priority 1, contains three sub-states: $l2$ corresponding to the cases when the train is under CTCS-2, $l3$ to the case when the train is under CTCS-3, and $l2a$ the case when the train is co-supervised by CTCS-2 and CTCS-3. By performing the default transition, $l2$ becomes active first, and will check the following transitions in sequence:

– The outgoing transition goes to $l2a$, and becomes enabled when the train reaches x_1 and gets permission from RBC to start level transition (i.e., $i == 2$);

- The ingoing transitions 1 and 2 represent the application for level transition from RBC. Obviously 2 will be enabled first, when the train is approaching ST. It starts to apply level transition by setting i to 1 and sending LUA to RBC. Accordingly, RBC will be triggered and then approve the request by emitting LU back. As a result, transition 1 in $l2$ will be enabled and then set i to 2. This will enable the outgoing transition to $l2a$. The action $FB2()$ models the supervision of CTCS-2.
- The transitions 3 and 4 represent the application for MA extension from TCC. Transition 4 will be enabled first, when the train reaches e_{22}, and it will first update the mode to CO and then start to apply MA extension by sending $MAA2$ to TCC. Accordingly, TCC will be triggered and then agree on the request by emitting $MAA2C$ back. As a result, transition 3 in $l2$ will happen and extend the MA correspondingly. $FMA2$ models MA extension under CTCS-2.
- The transition 5 represents the supervision of CTCS-2 and is always enabled.
- When reaching the junction, there are three outgoing transitions, which by comparison with the speed limits $vr1$ and $vr2$, update the acceleration a correspondingly. Here C_b, C_a and C_A represent the maximum deceleration, a random deceleration, and a random acceleration respectively.

As the train moves, the outgoing transition of $l2$ will be taken, and state $l2a$ will become active. $l2a$ has one outgoing transition targeting $l3$, which is enabled when the train reaches beyond $x2$. $l2a$ has a similar structure to $l2$, except that it is not involved in the level transition, but enriched with transitions 1 and 4 representing the application for MA extension from RBC in the co-supervised area. $l3$ also has a similar structure to $l2$, except that it will ask for confirmation of mode conversion from the driver, via transitions 1 and 2.

Results. The simulation result is shown in Fig. 9, which indicates that the train will stop at location x_2 (thus state $l3$ is never reachable). Applying the tool **S**im2**H**CSP to translate the Simulink/Stateflow model, it generates seven files, corresponding to the variable, processes, and assertion definitions of the Simulink and Stateflow models resp., and the *goal* to be verified. Together, they contain 1351 lines of code in total. We use `P` to denote the resulting HCSP model for the combined scenario. Using HHL prover, we prove the following goal for P as a theorem (`T` stands for `True`):

```
lemma goal :"{T,T,T,T,T,T} P {plant_s_1<=64000, T,T,T,T,T;
              (l = 0) | (high (plant_s_1<=64000)), T,T,T,T,T}"
```

Obviously, the postcondition together with the history formula indicate that the train never moves across location x_2, i.e., `64000` here.

All the files related to the case study including the translation and formal interactive proof can be found at https://github.com/submission/Sim2HCSP.

5 Conclusion and Future Work

The combination of Simulink and Stateflow provides a seamless integration of hierarchical and parallel state machines into a control-oriented block-diagram formalism involving both discrete and continuous behaviors. Extending our previous work on translating pure Simulink models into the hybrid-state process calculus HCSP, this paper presents the translation of Simulink/Stateflow to HCSP. Based on this translation, full formal verification of Simulink/Stateflow diagrams can be achieved using the Isabell/HOL-based interactive verifier HHL Prover. We demonstrate our approach on the case study of a combined scenario from the Chinese Train Control System CTCS-3. In comparison to our previous work, more complex discrete control is involved, and as a consequence more complex interactions between the controllers and continuous plant are covered by the proof certificate obtained.

Acknowledgements. This paper is supported partly by "973 Program" under grant No. 2014CB340701, by NSFC under grants 91118007 and 91418204, by CDZ project CAP (GZ 1023), by the CAS/SAFEA International Partnership Program for Creative Research Teams, and by Deutsche Forschungsgemeinschaft as part of the Transregional Collaborative Research Center SFB/TR 14 AVACS.

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_ug.pdf
3. 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)
4. 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)
5. Chen, C., Dong, J.S., Sun, J.: A formal framework for modeling and validating Simulink diagrams. Formal Asp. Comput. **21**(5), 451–483 (2009)
6. Clarke, E.M., Zuliani, P.: Statistical model checking for cyber-physical systems. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 1–12. Springer, Heidelberg (2011)
7. Hamon, G., Rushby, J.: An operational semantics for Stateflow. Int. J. Softw. Tools Technol. Transf. **9**(5), 447–456 (2007)
8. He, J.: From CSP to hybrid systems. In: A Classical Mind, Essays in Honour of C.A.R. Hoare, pp. 171–189. Prentice Hall International (UK) Ltd (1994)
9. Herde, C., Eggers, A., Fränzle, M., Teige, T.: Analysis of hybrid systems using HySAT. In: ICONS 2008, pp. 196–201 (2008)
10. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall Intl, Upper Saddle River (1985)

11. Liu, J., Lv, J., Quan, Z., Zhan, N., Zhao, H., Zhou, C., Zou, L.: A calculus for hybrid CSP. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 1–15. Springer, Heidelberg (2010)
12. 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)
13. Miller, S.P., Whalen, M.W., Cofer, D.D.: Software model checking takes off. Commun. ACM **53**(2), 58–64 (2010)
14. Moore, R.E.: Interval Analysis. Prentice Hall, Upper Saddle River (1966)
15. Moszkowski, B., Manna, Z.: Reasoning in interval temporal logic. In: Engeler, E. (ed.) Logic of Programs 1979. LNCS, vol. 125. Springer, Heidelberg (1981)
16. Rauh, A., Sibert, C., Aschemann, H.: Verified simulation and optimization of dynimc systems with friction and hysteresis. In: Proceedings of ENOC 2011 (2011)
17. 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 (2004)
18. Tiwari, A.: Formal semantics and analysis methods for Simulink/Stateflow models. Technical report, SRI International (2002)
19. Tripakis, S., Sofronis, C., Caspi, P., Curic, A.: Translating discrete-time Simulink to Lustre. ACM Trans. Embedded Comput. Syst. **4**(4), 779–818 (2005)
20. 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)
21. Zhou, C., Wang, J., Ravn, A.P.: A formal description of hybrid systems. In: Alur, R., Sontag, E.D., Henzinger, T.A. (eds.) HS 1995. LNCS, vol. 1066, pp. 511–530. Springer, Heidelberg (1996)
22. Zou, L., Lv, J., Wang, S., Zhan, N., Tang, T., Yuan, L., Liu, Y.: Verifying chinese train control system under a combined scenario by theorem proving. In: Cohen, E., Rybalchenko, A. (eds.) VSTTE 2013. LNCS, vol. 8164, pp. 262–280. Springer, Heidelberg (2014)
23. Zou, L., Zhan, N., Wang, S., Fränzle, M., Qin, S.: Verifying simulink diagrams via a hybrid Hoare Logic prover. In: EMSOFT 2013 (2013)