Simppaal - A Framework For Statistical Model Checking of Industrial Simulink Models

PREDRAG FILIPOVIKJ, NESREDIN MAHMUD, RALUCA MARINESCU, GUILLERMO RODRIGUEZ-NAVAS, and CRISTINA SECELEANU, Mälardalen University
OSCAR LJUNGKRANTZ and HENRIK LÖNN, Volvo Group Trucks Technology

The evolution of automotive systems has been rapid. Nowadays, electronic brains control dozens of functions in vehicles, like braking, cruising, etc. Model-based design approaches, in environments such as MATLAB Simulink, seem to help in addressing the ever-increasing need to enhance quality, and manage complexity, by supporting functional design from a set of block libraries, which can be simulated and analyzed for hidden errors, but also used for code generation. For this reason, providing assurance that Simulink models fulfill given functional and timing requirements is desirable. In this paper, we propose a pattern-based, execution-order preserving automatic transformation of atomic and composite Simulink blocks into stochastic timed automata that can then be formally analyzed with Uppaal Statistical Model Checker (Uppaal SMC). Our method is supported by the Simppaal tool, which we introduce and apply on an industrial prototype called the Brake-by-Wire system. This work enables the formal analysis of industrial Simulink models, by automatically generating stochastic timed automata counterpart.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Theory of computation → Verification by model checking**; • **Computer systems organization →** *Embedded software*;

Additional Key Words and Phrases: Simulink, Formal Analysis and Verification, Statistical Model Checking, Uppaal SMC, Stochastic Timed Automata

ACM Reference format:
Predrag Filipovikj, Nesredin Mahmud, Raluca Marinescu, Guillermo Rodriguez-Navas, Cristina Seceleanu, Oscar Ljungkrantz, and Henrik Lönn. 2017. Simppaal - A Framework For Statistical Model Checking of Industrial Simulink Models. 1, 1, Article 1 (September 2017), 22 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION AND MOTIVATION

The current trend in automotive systems is to replace mechanical components with electronic ones (e.g. in drive-by-wire). The resulting software-based control systems exhibit high complexity and are difficult to analyze/verify. Indeed, such systems are *safety-critical*, and need be developed under standards like ISO26262 [1], in order to provide guarantees (and evidence) that safe operation is assured.

To achieve some form of assurance with respect to safety-critical requirements, as well as valuable design insight, *model-based design* enables industry to create executable specifications in the form of Simulink [8] models that can be simulated and formally analyzed [5] to detect hidden design errors and requirements violations.

This work is supported by the Swedish Governmental Agency for Innovation Systems (VINNOVA) under project 2013-01299.

XXXX-XXXX/2017/9-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Analyzing Simulink models formally has been a research target for a while now. Existing work [4, 16, 17] provides solutions based on (stochastic) hybrid automata, extended finite automata etc., yet no integrated framework exists, which could serve as a basis for an automated tool support applicable to complex industrial Simulink models. To address this gap, in this paper, we introduce a pattern-based approach (Section 3) that captures formally the behaviors of Simulink blocks, as networks of stochastic timed automata, and report our experience from analyzing an industrial system, with UPPAAL SMC (Statistical Model Checker) [10] (Section 5). Our use case, that is, the *Brake-by-Wire* (BBW) prototype comes from Volvo Group Trucks Technology (VGTT), Sweden, a well-known truck manufacturer.

We classify the Simulink blocks into *atomic* (basic computational units) and *composite* (hierarchical structures which functionality is realized though set of atomic blocks) blocks. Further, we separate the former into *discrete-time* and *continuous-time* blocks, depending on whether a given sample time is used in the simulation or not. In order to be able to reason about such blocks, we propose a generic tuple definition.The definition captures the functionality of an atomic Simulink block as a `blockRoutine()` that updates the state variables, after which it produces an output observable at particular time instances defined as a multiple of the block's sample time (in case of discrete blocks), or continually observable in case of a continuous block. Next, we define the semantics of the Simulink blocks in terms of timed transition systems, and we provide a proof of soundness of the transformation into particular stochastic timed automata, by showing that the atomic Simulink block refines our proposed stochastic patterns, in the discrete-time case. In cases when the Simulink model contains continuous-time blocks, the soundness resorts to comparing simulation traces generated by simulating the Simulink and UPPAAL SMC models, respectively. A crucial aspect of the Simulink to stochastic timed automata transformation is preservation of the correct execution order of the Simulink blocks, both at the system and subsystem level. We do this by introducing a *flattening algorithm* that removes the hierarchy from Simulink models by capturing only the execution order of the blocks called *sorted order list*, as computed by the Simulink simulation engine at the beginning of the simulation.

The crux of our method is twofold: (i) the transformation relies on transformation patterns, which eases the modeling process while preserving the execution semantics of Simulink blocks, and (ii) verifying the encodings of the Simulink blocks behaviors as C routines in UPPAAL, with the program verifier `Dafny` [18]. To be able to apply our approach on the selected BBW industrial use case, we also provide a tool called SIMPPAAL that takes as input the Simulink model together with the sorted list and automatically generates the formal model to be statistically model checked. Our endeavor is justified by the industrial needs of ensuring correctness with respect to both functional and timing behaviors of the automotive embedded systems. Moreover, an initial investigation of verifying large Simulink models with the existing Simulink Design Verifier tool shows limitations in terms of scalability and coverage of all types of requirements. The application of our approach to the BBW Simulink model does not yet confirm the scalability of our approach, but it shows its feasibility. We show that we can automatically generate a network of stochastic timed automata corresponding to the Simulink blocks from the BBW model and their sorted order of execution, and analyze it via statistical model checking the complete transformed model, against probabilistic functional and timing requirements, with high accuracy. Applying exhaustive model checking on the 4-wheels architectural model of the BBW, integrated with formal semantics in terms of timed automata generates a very large state-space, unless reduction techniques are applied [20, 21]. It is then foreseeable that applying exact model checking on more complex industrial models would most likely run into the state-space explosion problem. This motivates our choice of SMC as the analysis solution for Simulink models, despite the fact that the method is not exact.

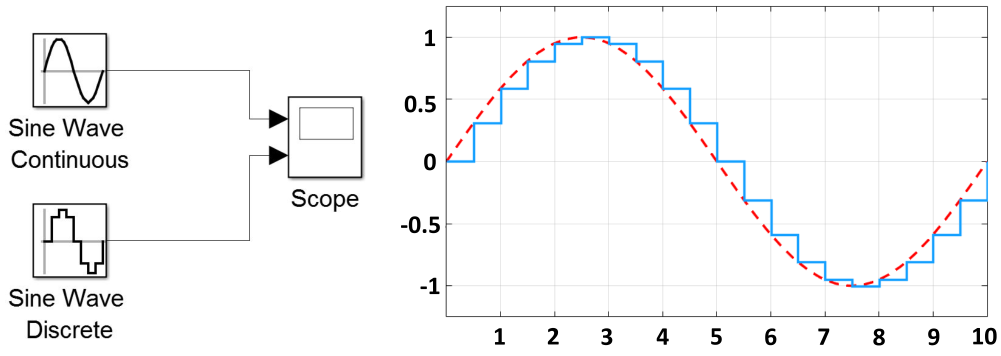
The remainder of the paper is organized as follows. In Section 2 we overview Simulink, and UPPAAL SMC, after which we present our Simulink to stochastic timed automata transformation approach in Section 3. The architecture of the SIMPPAAL tool is described in Section 4, and its validation by applying it on the BBW prototype is shown in Section 5. In Section 7 we compare to related work, before concluding the paper and outlining future lines of research in Section 8.

2 PRELIMINARIES

In this section, we present an overview of Simulink, stochastic (priced) timed automata and Uppaal SMC.

2.1 Simulink

Simulink [8] is a graphical programming environment for model-based design, simulation, verification, and code generation of multi-domain dynamic systems. The model-based design is achieved based on predefined set of atomic blocks, including *Sum*, *Product*, *Gain*, *Sine*, *Logical Operator*, *Relational Operator*, etc., organized in predefined libraries. Such blocks represent computational modules that produce an output based on a equation or another modeling concept either continuously (i.e., continuous-time blocks) or at specific points in time (i.e., discrete-time blocks). Discrete-time blocks have a specific feature of delaying the first output. The duration of the delay interval is called *offset*. Fig. 1 shows a visual representation of the continuous-time (red dashed line) and discrete-time (blue continuous line) behaviors of the *Sine* wave Simulink block. Simulink also supports the definition of custom blocks modeled as Stateflow diagrams and user-defined functions via the concepts of *s-function* written in Matlab, C, C++, or Fortran, and *Block Masks* that have user-defined interface, encapsulated logic, and hidden data from the user.

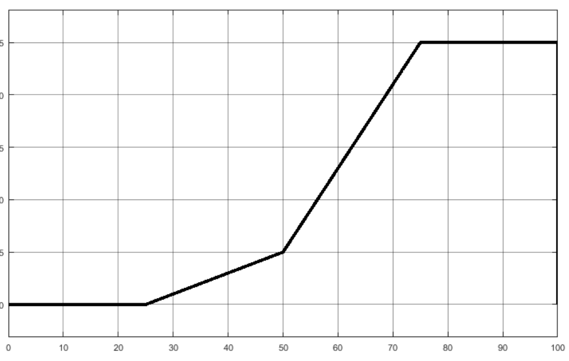


(a) (b)

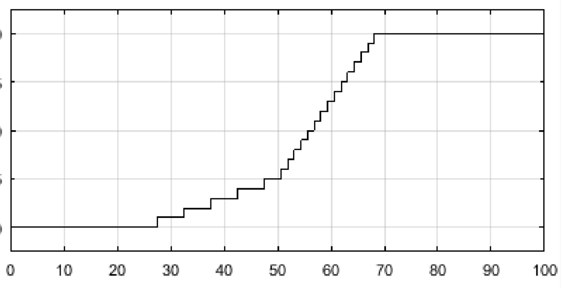
Fig. 1. Example (Sine Wave Block): (a) Simulink Diagram and (b) Simulation Result

A hierarchical model is achieved through the implementation of a *Subsystem*, a block that encapsulates a set of atomic blocks and possibly other subsystems. A subsystem can be either *virtual*, meaning that the encapsulated blocks are evaluated according to the overall system model, or *non-virtual*, where the encapsulated blocks are executed as a single unit that can be conditionally executed based on a predefined triggering, function call, or enabling input. Other blocks also aid the creation of a hierarchical diagram, like *Inport* and *Outport* block from the *Ports and Subsystems* library, and *Goto* and *From* block from the *Signal Routing* library.

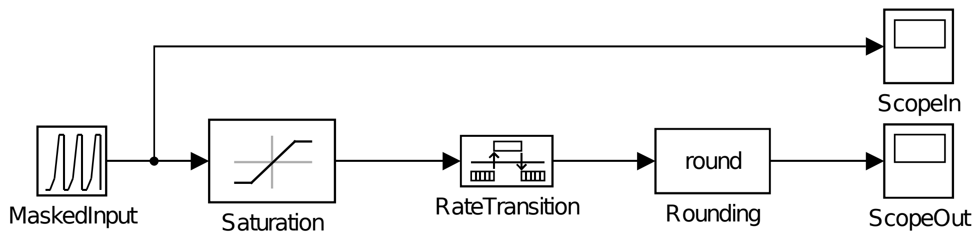
For example, in Fig. 2 we show a small Simulink model, which is an excerpt from the BBW model. The first block is a masked block called *MaskedInput* that produces the signal presented in Fig. 2a. The subsequent block called *Saturation* limits the value of the signal to a maximum value of 100. In cases when the input signal is below the maximum, the value remains unchanged. Finally, the signal is rounded to the closed integer by the *Rounding* block and represents the output signal of the system witch is displayed by Fig. 2b. Between the *Saturation* and the *Rounding* block, there is the *RateTransition* block, which ensures the correct data transfer between the two blocks. Scope blocks are used to visualize different signals.



(a)



(b)



(c)

Fig. 2. Example (Simulink model): (a) Simulink input signal (Scope1), (b) Simulink output signal (Scope2), and (c) The Simulink model.

Simulink models can be simulated and the results can be displayed as simulation runs. The order in which the blocks are invoked during simulation is called *sorted order* (*slist* for short) and is computed by the model compiler during runtime. The *slist* for a given Simulink model can be generated using the `sldebug` command from the `MATLAB` console while `MATLAB` environment is in debug mode. The *slist* consists of tuples, where each tuple corresponds to a Simulink block, be it composite or atomic, containing the following information: execution order number, unique identifier and type. The *slist* contains information about the hierarchical structure of the Simulink model for the non-virtual composite blocks. In the hierarchical structure, each of the non-virtual composite blocks creates local context of execution, which we referrer to as *nested level*. This means that when a non-virtual subsystem is to be executed during simulation, first all of the blocks inside the subsystem will be executed before the execution of the subsequent blocks on the same level is started. As the Simulink model supports modeling non-virtual subsystems inside non-virtual subsystems, theoretically it is possible to have unlimited levels of nesting inside a given Simulink model. The virtual composite blocks as well as the blocks that do not perform computation, such as: *Mux*, *Demux*, *Goto*, *From*, etc. are not part of the slist.

2.2 Uppaal SMC

Uppaal SMC [9] is a statistical model checker for system models represented as networks of stochastic priced timed automata. A stochastic priced timed automaton (SPTA) is defined as the following tuple:

$$SPTA = \langle L, l_0, X, \Sigma, E, R, I, \mu, \gamma \rangle, \tag{1}$$

where L is a finite set of locations, $l_0 \in L$ is the initial location, X is a finite set of continuous variables, $\Sigma = \Sigma_i \uplus \Sigma_o$ is a finite set of actions partitioned into inputs (Σ_i) and outputs (Σ_0), E is a finite set of edges of the form (l, g, a, φ, l'), where l and l' are locations, g is a predicate on $\mathbb{R}^X$, action label $a \in \Sigma$, and φ is a binary relation on $\mathbb{R}^X$, $R : L \longrightarrow \mathbb{N}^X$ that assigns a rate vector to each location, I assigns an invariant predicate $I(l)$ to any location l, μ is the set of all density delay functions $\mu_s \in L \times R^X$, which can be either uniform or exponential distribution, and γ is the set of all output probability functions γ_s over the Σ_o output edges of the automaton.

The semantics of the probabilistic SPTA is defined over a timed transition system, whose states are pairs $s = (l, v) \in L \times \mathbb{R}^X$, with $v \models I(l)$, and transitions defined as: (i) delay transitions ($(l, v) \xrightarrow{d} (l, v')$ with $d \in \mathbb{R}_{\geq 0}$ and $v' = v + d$), and (ii) discrete transitions ($(l, v) \xrightarrow{a} (l', v')$ if there is an edge (l, g, a, Y, l') such that $v \models g$ and $v' = v[Y]$, where $Y \subseteq X$, and $v[Y]$ is the valuation assigning 0 when $x \in Y$ and $v(x)$ otherwise). We write $(l, v) \rightsquigarrow (l', v')$, if there is a finite sequence of delay and discrete transitions from (l, v) to (l', v'). The delay density function μ_s over delays in $\mathbb{R}_{\geq 0}$ for each state a is either uniform or exponential distribution depending on the invariant of the location l. Let E_l denote the disjunction of guards such that $(l, g, o, -, -) \in E$ for some output o. $D(l, v) = \sup\{d \in \mathbb{R}_{\geq 0} : v + d \models I(l)\}$ denotes the supremum delay, whereas $d(l, v) = \inf\{d \in \mathbb{R}_{\geq 0} : v + d \models E_l\}$ denotes the infimum delay before enabling an output. If $D(l, v) < \infty$ then the delay density function μ_s for a given state s is a uniform distribution over the interval $[d(l, v), D(l, v)]$, otherwise it is an exponential distribution with a rate $P(l)$. For every state s, the output probability function γ_s over Σ_o is a uniform distribution over the set $\{o : (l, g, o, -, -) \in E \wedge v \models g\}$ whenever the set is non empty.

Under the assumption of input-enabledness, disjointedness of clock sets and output actions, a collection of composable SPTA can be defined as a network of SPTA (NSPTA) ($A1 \parallel A2 \parallel \ldots \parallel An$). The states of the NSPTA are defined as a tuple $s = \langle s_1, \ldots, s_n \rangle$, where s_j is a state of A_j of the form (l, v), where $l \in L^j$ and $v \in \mathbb{R}^{X^j}$, where different automata synchronize based on standard broadcast channels. The probabilistic semantics is based on the principle of independence between components. Each component decides on its own (based on a given delay density function and the output probability function) how much to delay before producing an output.

For encoding the patterns presented in this paper, we use SPTA with real-valued clocks that evolve with implicit rate 1. These automata are in fact timed automata with stochastic semantics, called stochastic timed automata (STA). A network of STA (NSTA) is a parallel composition of STA, defined in a similar way like NSPTA. The notion of SPTA is introduced due to the fact, that, for analysis we use monitor automata (composed in parallel with the actual system model) that implement the *stop-watch* mechanism, which renders the model an NSPTA.

Uppaal SMC uses a probabilistic extension of WMTL [6] to provide:

- *Hypothesis testing*: check if the probability to reach a state ϕ within cost $x \leq C$ is greater or equal to a certain threshold p ($Pr(\star_{x \leq C} \phi) \geq p$),
- *Probability evaluation*: calculate the probability $Pr(\star_{x \leq C} \phi)$ for some NSPTA,
- *Probability comparison*: is $P(\star_{x \leq C} \phi_1) > P(\star_{y \leq D} \phi_2)$?

where $\star$ stands for either *future* ($\Diamond$) or *globally* ($\Box$) temporal operator.

3 SIMULINK TO UPPAAL SMC: APPROACH

In this section, we present our general approach for transforming Simulink models into NTSA. The transformation handles a wide range of block types, including both atomic and composite blocks belonging to various categories such as: non-virtual, continuous, discrete, s-functions, etc. A study on the types of blocks used in various industrial

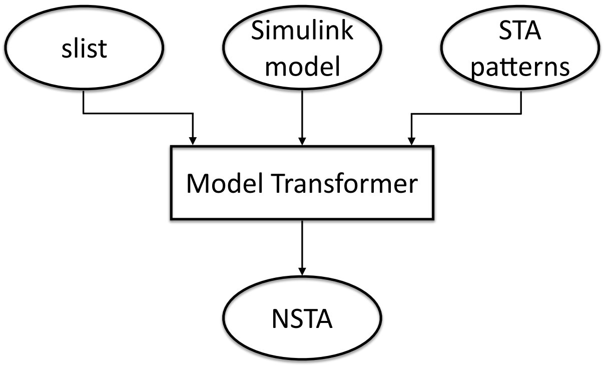


Fig. 3. Overall Approach of the Simulink to UPPAAL SMC Transformation

Simulink models motivates this choice. Figure 3 illustrates the high level process of the transformation. The approach relies on three artifacts: the Simulink model, the *slist*, and the set of STA Uppaal patterns, which are parsed and transformed by the *Model Transformer* process in order to generate the NSTA model as a resulting artifact. Our work consists of the following steps:

(i) We give a formal definition for the Simulink model and the atomic Simulink blocks (see Section 3.1);
(ii) We propose a flattening algorithm that transforms hierarchical Simulink models into flat models that do not contain composite Simulink blocks (see Section 3.3);
(iii) We transform the continuous-time and discrete-time atomic blocks into their respective STA using the *Transformation Patterns* (see Section 3.2);
(iv) We give a proof of transformation soundness for Simulink models consisting of discrete-time Simulink blocks only (see Section 3.4).

The transformation of the user defined atomic blocks (e.g., S-function, Masked and Custom blocks) is highly dependent on the documentation. Even though there is no limitation when it comes to identification and transformation of the execution behavior of such blocks, the transformation of the `blockRoutine` depends on the specification of the input-output function. This is due to the fact that the inner contents might be hidden (Masked block) or implemented in various programming languages (S-function block) for which we do not provide automatic translation into `C` function.

3.1 Formal definitions

Each Simulink block has an input-output function, input and output parameters, data types support and runtime characteristics, e.g., sample time. The description of each Simulink block is accessible online [24]. In the following, we introduce the formal syntax of a Simulink block and a Simulink model, respectively. These definitions are used to reason about the soundness of our transformation, by establishing connections between formal syntactic definitions and their corresponding semantics.

Definition 3.1 (Simulink block). An atomic Simulink block, denoted by B, is defined as the following tuple:

$$B = \langle s_n, V_{in}, V_{out}, V_D, \Delta, Init, blockRoutine \rangle \tag{2}$$

where:

(i) $s_n \in \mathbb{Z}$ - is the execution order number;
(ii) V_{in} - is a finite set of typed input real-valued variables;

(iii) V_{out} - is a finite set of typed output real-valued variables;
(iv) V_D - is a finite set of typed data (or state) real-valued variables;
(v) Δ - represents the set of time points at which output is produced, $\Delta = n * t_s + \mathit{offset}$, where: t_s, offset $\in \mathbb{R}_{\geq 0}$ are the sample time and the offset of the atomic Simulink block, respectively, and n $\in \mathbb{N}$. For continuous blocks Δ is infinitely small and *offset*=0, meaning that the output is produced at infinitely small intervals;
(vi) $Init()$ - is an initialization of the data variables;
(vii) $blockRoutine() = Update(); Output()$ - is the sequential composition of $Output()$ and $Update()$ functions. It captures the functionality of a Simulink block, where: $Output() : V_{in} \times V_D \mapsto V_{out}$ is the output function and $Update() : V_{in} \times V_D \mapsto V_D$ is the update function.

Based on the definition of a Simulink block, we propose a formal definition of a Simulink model.

Definition 3.2 (Simulink model). A Simulink model is formally defined as a sequential composition of n Simulink blocks, as follows:

$$S = B_1 \otimes B_2 \otimes B_3 \cdots \otimes B_n \tag{3}$$

where: $s_n^s = \bigcup_{i=1}^{n} s_n^i$ is an ordered list of execution with $\forall (i,j). i < j \Rightarrow s_i < s_j$, $V_{in}^S = \bigcup_{i=1}^{n} V_{in}^i$ is the set of input variables, $V_{out}^S = \bigcup_{i=1}^{n} V_{out}^i$ is the set of output variables, $V_D^S = \bigcup_{i=1}^{n} V_D^i$ is the set of internal state variables, $\Delta_S = \bigcup_{i=1}^{n} \Delta^i$ is the set of time points at which the respective data and output variables are updated, and $(Init; blockRoutine)_S \triangleq (Init_1; blockRoutine_1)\|_{=\Delta_1}; (Init_1; blockRoutine_2)\|_{=\Delta_2}; \ \ldots \ ; (Init_n; blockRoutine_n)\|_{=\Delta_n}$ is an ordered list of pairs of (Init, blockRoutine), which are executed atomically at given times Δ_i.

Semantics of Simulink blocks. Let us rewrite $\Delta = n * t_s + \mathit{offset}$ of Definition 3.1, as an integral multiple of Simulink's simulation step $\delta \in \mathbb{Q}_{\geq 0}$, that is, $\Delta = n * (m * \delta) + (r * \delta)$, $n, m, r \in \mathbb{N}$. Let us also assume that $x \in V_{in}$, $u \in V_D$, and $y \in V_{out}$ are input, data, and output variables, respectively. Then, we define the semantics of a Simulink block in terms of the following discrete-time transition system.

Definition 3.3 (Semantics of a Simulink block). Assume B is a Simulink block as given in Definition 3.1. The semantics of B is a timed transition system, as follows:

$$T_B = \langle q_0, Q, \mathcal{L}, \longrightarrow \rangle, \tag{4}$$

where $Q = \mathbb{R}^n$ is the state space: a state $q = y|_t = (y, t)$ is given by the values of all output variables y at a given time instance $t \in \mathbb{R}_{\geq 0}$, for given input at time t, that is, $x|_t$, and data at time t, that is, $u|_t$, $q_0 = y_0|_{t_0} = (y_0, t_0) \in Q$ is the initial state, $t_0 \in \mathbb{R}_{\geq 0}$, $\mathcal{L} = \mathcal{L}_a \cup \mathcal{L}_t$ is the set of labels, with $\mathcal{L}_a$ the set of action labels: $\mathcal{L}_a = \{Init, blockRoutine\}$, $\mathcal{L}_t$ the set of time labels: $\mathcal{L}_t = \{r * \delta, m * \delta\}$, and $\longrightarrow$ is the transition relation: $\longrightarrow \subseteq Q \times \mathcal{L}_a \times \mathcal{L}_t \times Q$ with two types of transitions:

$$\begin{aligned} q_0 \quad &\xrightarrow{Init, r*\delta} \quad q' \iff t' = t_0 + r * \delta, \text{ and } \exists\, y_0|_{t'} \text{ such that } y|_{t'} = y_0|_{t'} \\ q \quad &\xrightarrow{blockRoutine, m*\delta} \quad q' \iff t' = t + m * \delta, \text{ and } \exists\, u|_t, x|_{t'} \text{ such that} \\ & \qquad\qquad u|_{t'} = f(x|_{t'}, u|_t), \text{ and } y|_{t'} = f(x|_{t'}, u|_{t'}). \end{aligned}$$

The first transition is the *Init*-type transition, fired at the beginning of the block's execution, at t_0, and the second is the *Operation*-type corresponding to generating outputs for given inputs, at particular time points, $t' = t + m * \delta$. Note that state q' can be the same as q if the input does not change between two sample times. If the Simulink block is continuous, then $m = 1$, $r = 1$, meaning that transitions are fired "infinitely" often, that is, every δ. Note that Definition 3.3 assumes an unknown but constant simulation step δ during the entire simulation time, which is one of the possible cases in Simulink.

By the above definition, a finite run ϱ of the Simulink block can be defined as the following sequence of transitions:

$$q_0 \xrightarrow{Init, r*\delta} q_1 \xrightarrow{blockRoutine, m*\delta} \ldots \xrightarrow{blockRoutine, m*\delta} q_n$$

where q_n is the last (final) state.

We denote by $Runs(B, q_0)$ the set of finite runs of B from q_0. Assuming $s_1 < s_2 < \ldots < s_n$ the execution order numbers of the blocks in a Simulink model S described as in Definition 3.2, a run of S is defined as the sequence of *Init* and *Operation* transitions of each block, at each step $i \leq n$, in the corresponding order of execution.

3.2 STA Patterns

In order to facilitate the transformation of atomic Simulink blocks into their equivalent STA, we propose two transformation patterns for the continuous-time and discrete-time blocks. The transformation patterns are reusable and conform to the semantics of the tuple introduced in Equation 3. Figure 4 shows our transformation patterns encoded in the input language of Uppaal SMC.

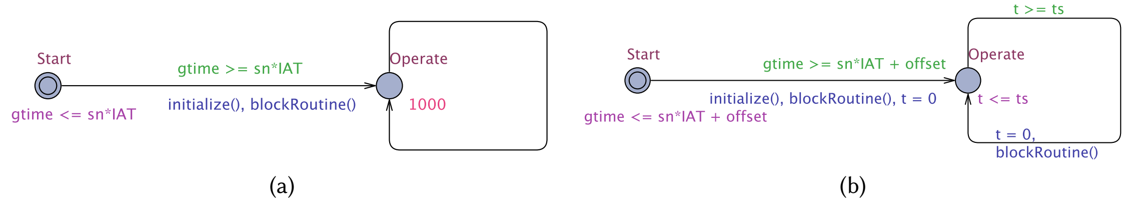


Fig. 4. STA Transformation Patterns: (a) Continuous-time and (b) Discrete-time Blocks

Each pattern has its own execution mechanism. The execution of continuous-time patterns presented in Figure 4a proceeds according to an exponential distribution for unbounded delays, whereas the execution of the discrete-time pattern presented in Figure 4b proceeds according to the uniform distribution for time-bounded delays modeled via the invariant. The elements of the STA patterns are as follows:

(i) Location `Start` - In the `Start` location the automaton waits for its release according to the order of execution given in the *slist* and, for the discrete-time blocks, the offset parameter;

(ii) Location `Operate` - The `Operate` location models the operational mode of a Simulink block. For the discrete-time pattern, this location is decorated with an invariant connected to the block's period. For the continuous-time pattern, the location is decorated with an exponential rate λ which determines the probability of the automaton to remain in this location at each simulation step, according to an exponential distribution;

(iii) Edge `(Start, Operate)` - The edge is enabled when the guard condition for releasing the block is satisfied. The time for release for continuous blocks (Figure 4a) is given as `gtime` $\geq$ `sn*IAT`, where: `gtime` is the the global clock, `sn` is the block's execution number and and the `IAT` is the constant inter-arrival time between two consecutive releases of automata. For the discrete blocks (Figure 4b) the release time depends on the `offset` parameter which denotes the delay of the first execution of the block. When the edge is traversed, two update actions (modeled as C functions) are executed: `initialize()`, which initializes the data variables, and `blockRoutine()` which updates the output.

(iv) The edge `(Operate, Operate)` - The edge is taken every time the automaton updates its output. As explained above, the edge is traversed according to exponential distribution for continuous-time patterns or at discrete time intervals for the discrete-time patterns.

To prove the correctness of the `blockRoutine` functions we use pre-/post-condition verification using the Dafny [18] program verifier. A set of pre-conditions is used to describe the input, output and state variables prior to the execution of the `blockRoutine`. Given that the pre-condition holds, after the execution of the blockRoutine, the set of post-conditions has to be established. We consider the `blockRoutine` to be correct if the specified set of postconditions is satisfied for all executions. For complex block routines that contain loops, we use loop invariants and termination conditions. The detailed description of verification procedure for the `blockRoutine` using `Dafny` has been omitted due to space limitation. For full details, we refer the reader to our technical report [13].

3.3 Flattening Algorithm for Preserving the Block Execution Order

In this section, we describe the procedure used to assign unique execution order for each block inside a Simulink model, called *flattening* procedure.

The flattening of a Simulink model is performed in two steps, as follows: (i) removing the non-virtual composite Simulink blocks from the model and replacing them with a set of atomic Simulink blocks, and (ii) assigning correct execution order number for the atomic blocks such that the original behavior of the model is preserved. The proposed flattening procedure is recursive which makes it suitable for flattening Simulink models with arbitrary many nested levels.

An intuitive, yet naive approach for flattening a Simulink model would be to apply the flattening procedure on the model itself, which includes traversing the complete model. Even though such approach is feasible with respect to step (i), it cannot satisfy step (ii), as the Simulink model itself does not contain information about the execution order of the contained blocks, since the execution order number of each Simulink block is determined at the beginning of the simulation.

Algorithm 1 Flattening algorithm for sorted order list.

```
1:  function FLATTEN(String currentBlockId, String currentBlockOrderNo, String parentBlockOrderNo)
2:      $orderedList \leftarrow emptyList$                                      ▷ Ordered list containing blocks IDs.
3:      if $isAtomicBlock(currentBlockId)$ then                                  ▷ The current block is atomic.
4:          $orderedList.append(parentBlockOrderNo.concat(currentBlockOrderNo))$
5:      else                                                                     ▷ The current block is a subsystem.
6:          $currentChildren \leftarrow getChildren(currentBlockId)$
7:          $concatenatedParentId \leftarrow parentBlockOrderNo.concat(currentBlockOrderNo)$
8:          for all $child$ in $currentChildren$ do
9:              $orderedList.append(flatten(child.id, child.orderNo, concatenatedParentId))$
10:     return $orderedList$
```

As explained in Section 2.1, the *slist* represents the hierarchical structure of the Simulink model as a collection of *contexts*. We refer to the Simulink model itself as the *root context*, while each of the non-virtual subsystem blocks inside the model are refereed to as *local contexts*. For the virtual and atomic subsystem blocks, no contexts are created, as such blocks are flattened by Simulink automatically. Each context creates a *nested level*. The blocks residing directly on the root level have a global execution number. Blocks residing inside a local context have a local execution number, which is assigned relative to the local context. Given such structure, the procedure for flattening a Simulink model is reduced to a procedure of assigning global execution numbers to all atomic Simulink blocks contained inside all the local contexts inside the model. By doing that, we perform an implicit flattening of the Simulink model, as the correct order of execution of the atomic Simulink blocks from the model relative to the root context is determined. The pseudo code of the algorithm which is used for assigning global execution order number of the atomic Simulink blocks nested arbitrary deep inside a given Simulink model is given in Algorithm 1. The algorithm produces new *slist*, which contains a subset of the original tuples corresponding to the atomic blocks only, sorted according to their global execution number from first to last, which when executed as such give the overall behavior of the original Simulink model. In the newly generated *slist* all the non-virtual subsystems (local contexts) are replaced with the respective set of atomic blocks.

The flattening of a Simulink model is fully automated, meaning no user interaction is required. The procedure takes as an input the original slist generated in the `MATLAB` console saved as text file.

3.4 Proof of Transformation Soundness

Assume a Simulink model, as described by Definition 3.2, consisting of discrete-time blocks only, which has to be analyzed against properties in the category “for all paths” (e.g., invariance/safety, inevitability etc.). In order to show the soundness of our approach, we can show that the set of runs of the resulting NSTA, obtained by using the semantic pattern given in Figure 4b, is *refined* by the set of runs of the Simulink model, under the discrete-time blocks only assumption. We have that a Simulink model A' is a refinement of the NSTA model A if and only if $Runs'_A \subseteq Runs_A$, meaning that if the model A satisfies a safety or inevitability property p, and A' refines A, it then follows that A' also satisfies p.

We show the refinement at the discrete-time block level first and then we explain how the result extends to Simulink models with discrete-time blocks. For this, we use the results on decision problems for timed automata, overviewed by Alur and Madhusudan [3]. The authors show that reachability is decidable for the discrete-time or sampled semantics of timed automata, assuming an unknown non-negative rational sample time. If we consider the STA pattern of Figure 4b, we notice that the output probabilities over edges outgoing from locations `Start` and `Offset`, according to the uniform distribution γ, are 1, since there is only one outgoing edge from each location, respectively. Similarly, the delay density function μ gives probability 1 of delaying in location `Operate` for t_s time units, due to the disjointness of the invariant $t \leq t_s$ and the guard $t \geq t_s$. Basically, the automaton of Figure 4b is a deterministic closed timed automaton (since clock constraints are of the form $x \bowtie c$, with $\bowtie \in \{\leq, \geq\}$).

Refinement also equates to the problem of language inclusion between timed automata, which is an undecidable problem in general. An important class of timed automata for which the inclusion problem is decidable involves the notion of *digitization* [3]. A timed language L is said to be *closed under digitization* if discretizing a timed word (a string of symbols tagged with occurrence times) in the language, by approximating the events of the timed word to the closest tick of a discrete clock results in a word that is also in L. It is a proven fact that closed timed automata are closed under digitization. This means that constructing a sampled version of the STA automaton of Figure 4b yields an automaton that is a refinement of the original pattern, since $L_{A_d} \subseteq L_A$, where A is an automaton conforming to our STA pattern, and A_d is its discretized version.

Let us consider a digitization of the transformation pattern automaton of Figure 4b, as follows:

Definition 3.4 (Sampled semantics of STA of Figure 4b). Given a timed automaton A as in Figure 4b, and the sampling rate $\delta \in \mathbb{Q}$ (equal to the simulation step of the Simulink block), we define an automaton A_δ with the states, initial states and final states the same as the states, initial states, and final states of A, and the transitions of A_δ labeled with either action $a \in \Sigma \cup \{\epsilon\}$, where ϵ is not in Σ, with $m * \delta$, $m \in \mathbb{N}$, or with $r * \delta$, $r \in \mathbb{N}$. We call A_δ a sampled (digitized) timed automaton.

Note that in any reachable state of A_δ, the values of clocks are integral multiple of δ. A run of A_δ with initial state s_0, over a finite timed trace $\zeta = (t_0, a_0)(t_1, a_1)(t_2, a_2) \dots (t_n, a_n)$ is a sequence of transitions:

$$s_0 \xrightarrow{0, Initialize} s_1 \xrightarrow{r*\delta, blockRoutine} s_2 \xrightarrow{m*\delta, blockRoutine} \dots \xrightarrow{m*\delta, blockRoutine} s_n.$$

Theorem 1. *Let us assume a discrete-time Simulink block B defined by Definition 3.1, and a discrete transformation pattern described by a timed automaton with sampled semantics A_δ, as in Definition 3.4. Then, we have that B refines A_δ.*

Proof: There is a direct mapping between a location l of A_δ and the value of the output variable y of B, meaning that in locations `Offset` and `Operate` the variable y is observable (is assigned over the corresponding discrete transitions, respectively). By Definition 3.1 and Definition 3.4, all transition sequences possible in B are also possible in A_δ. Therefore, given q_0 the initial state of B and s_0 the initial state of A_δ, it follows that $Runs(B, q_0) \subseteq Runs(A_\delta, s_0)$, which equates to the fact that B is a refinement of A_δ. Q.E.D.

Given the fact that A_δ refines A, the STA pattern automaton of Figure 4b, it follows by transitivity of the refinement relation that the Simulink block B refines the STA pattern automaton A, assuming the discrete-time behavior.

This result extends to a Simulink model defined by Definition 3.2, if the former contains only discrete-time blocks. Given the execution order of each block, only one block at a time can be enabled and executed in the Simulink model, therefore the probability distributions of the network of STA that represents the Simulink model's transformation render transitions with probability one, so the parallel composition of STA is in fact a parallel composition of deterministic timed automata, which due to the enforced execution order is in fact a sequential composition of deterministic timed automata that can be further sampled. Given the result of Theorem 1, it follows that a discrete-time Simulink model described as in Definition 3.2 refines the network of STA in which it is transformed.

In case the Simulink model contains one or more continuous-time blocks that are being transformed by instantiating the transformation pattern of Figure 4a, the resulting network of STA uses the exponential distribution to compute the delay of each continuous-time STA, and the uniform distribution to chose the STA that is going to broadcast its output within the network. Therefore, in such cases, to have an indication on the correctness of transformation, we compare the simulations of the Simulink model, generated by Simulink, with the simulations of its STA counterpart generated by Uppaal SMC. If they are identical, we can conclude that the behaviors of the Simulink model and its translation are similar, to the extent provided by simulation.

4 SIMPPAAL TOOL

In this section, we present our tool called Simppaal (SIMulink to uPPAAL), which automates the process of transforming Simulink models into NSTA suitable for analysis using the Uppaal SMC tool, as described in Section 3. In the following subsections, we describe the architecture and the functionality of the Simppaal tool.

4.1 Simppaal Architecture

The architecture of the Simppaal tool is based on a modular design, where the overall functionality is achieved through a set of communicating (software) modules, via well-defined Application Programming Interfaces (APIs). The architectural design is given in Figure 5, and is based on the following concepts: *artifacts*, which can be input or output represented as circles, and the *modules* represented as squares. In the following, we use the term module and *software engine* interchangeably. The tool is implemented in the JAVA programming language, with limited usage of third party libraries. The core module of the tool is the *Simulink Transformation Engine (STE)*, whereas the other two modules *UPPAAL File Parser Engine* (UPE) and the *SList Parser Engine* (SPE) have supportive roles, which include serialization and de-serialization of the specific artifacts.

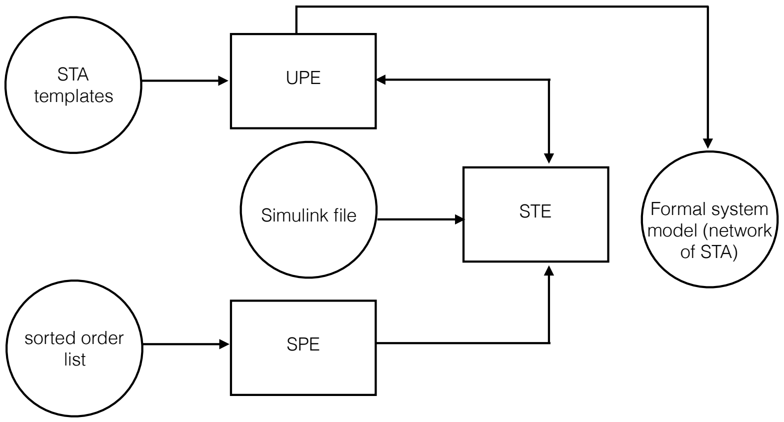


Fig. 5. Simppaal tool architecture.

As shown in Figure 5, the transformation process is based on three different input artifacts: the Simulink model file, the *slist* given as a textual file, and an Uppaal .xml file that contains the templates for the continuous- and discrete-time blocks. Each module that is handling a given artifact is responsible for: i) reading and parsing that input in a format such that it can be consumed by other modules, and ii) writing back to that file if required. Given that, the STE module is responsible for parsing the Simulink files, the *Uppaal Parser Engine (UPE)* handles the reading from and writing to Uppaal specific .xml files, while the sorted order list is handled by the *SList Parser Engine (SPE)*.

The STE module. This is the core module of Simppaal. Its main responsibility is to transform a Simulink model file into a NSTA suitable for analysis using Uppaal SMC. This is by no means a trivial procedure, and for that purpose, the module itself is further decomposed into submodules as follows: a submodule for reading and manipulating Simulink models, and another one for transforming Simulink blocks into STA. The role of the first submodule is to read a model file from the disk and store it as a memory object. It delivers functionalities such as: retrieving a Simulink block by its unique identification, navigating through the structure of the model, identifying the predecessors and successors of a given Simulink block, etc. The implementation of this submodule is based on the ConQAT library[1], which provides an API that eases the model's traversal and block manipulation. The library, however, exhibits limitations when it comes to traversing referenced subsystem blocks, as the contents of the referenced subsystem resides in a different context saved in a separate file. To mitigate this problem, we have developed a *"context-switching"* procedure that enables the tool to switch contexts, that is, go from one file and back, without information loss.

The second STE submodule is responsible for transforming an atomic Simulink block into a corresponding STA, by mapping Simulink parameters into STA specific constructs, such as: sample time, execution order number and the block routine. The submodule also generates `Dafny` verification expressions for each instance of block routine.

The UPE module. This module is used for reading the Uppaal file that contains the patterns for the continuous- and discrete-time blocks, as well as for writing the result Uppaal model into a new file. The module provides an API that allows manipulation of Uppaal files, including operations such as: deserializing a Uppaal file into a Uppaal model, adding and retrieving elements from the Uppaal model (automaton, location, edge) and serializing

[1]https://www.cqse.eu/en/products/simulink-library-for-java/overview/

the UPPAAL memory model back into a file that is used as an input to the UPPAAL tool. The UPE can be used as a stand-alone library or as part of any other tool for manipulating UPPAAL models.

The SPE module. A module that implements the flattening algorithm discussed in Section 3.3. It reads the *slist* provided as a textual file and applies the flattening algorithm. The result is a new *slist* of atomic Simulink blocks according to the execution order number that is then passed on to the STE. Unlike the UPE, the SPE module is bound to a specific purpose and cannot be reused outside the initially-intended context.

4.2 SIMPPAAL work flow

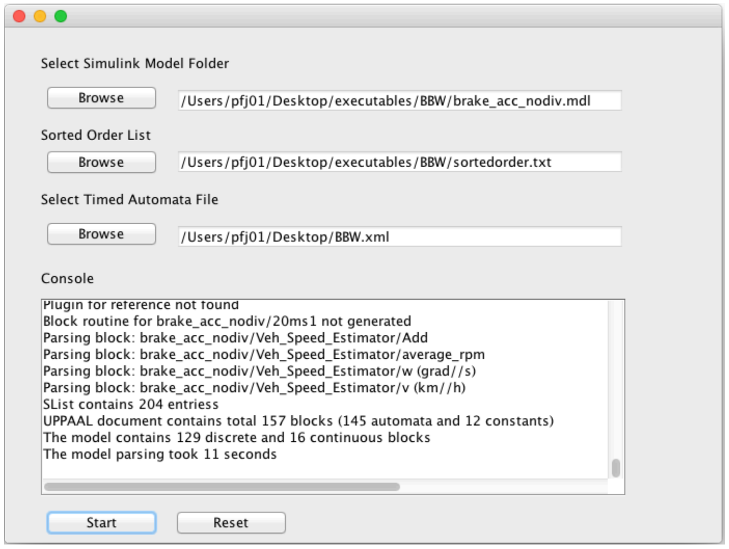
The transformation of Simulink models into networks of STA, as implemented in SIMPPAAL, is performed in the following steps:

(1) Flattening of the sorted order list
(2) Collecting information about the atomic Simulink blocks
  (a) Finding a block in the model and retrieving block parameters
  (b) Populating the lists of predecessors and successors
(3) Transforming atomic Simulink blocks into STA
  (a) Mapping Simulink block information into STA constructs
  (b) Determining the output signal type
  (c) Instantiation of the STA inside the UPPAAL model
(4) Saving the generated UPPAAL model in a new file

The transformation process of a Simulink model into a NSTA starts by loading and flattening the slist such that the global execution number to each atomic block in the model is assigned (Step 1).

The flattened *slist* is then used as a primary input of STE, for transforming the Simulink model into a NSTA. Basically, the transformation of the Simulink model is a process of iterating through the list, collecting information about each block (Step 2), and mapping that information onto an adequate STA pattern (Step 3). The process of collecting information about each block is performed as a set of simpler actions that include Step 2.a: getting an entry from the list and locating the Simulink block inside the model by its unique identifier. The procedure locates the given block no matter how deeply it is nested inside the model, even if it resides in another file. This is enabled by our context-switching technique. Once the block has been identified, the STE tries to identify all of its *predecessors*, which is a list of non-virtual atomic Simulink blocks, performed in Step 2.b. In our implementation, the following blocks are considered as virtual: `Mux`, `Demux`, `Inport`, `Goto`, `From` and `Outport`. Additionally, some non-virtual blocks that do not perform computational routines, such as `Scope` and `RateTransition` are added to the list of virtual blocks. In other words, a predecessor is an atomic block whose output is consumed by the block that is currently being transformed. In a similar way, the STE identifies the list of *successors*, which is a list of non-virtual atomic Simulink blocks, which uses the output of the given Simulink block as an input for producing an output.

Once all the transformation-relevant information for the block is gathered, the STE transforms the Simulink block into an STA in Step 3. The transformation is done in several steps: first, in Step 3.a, the STE calls UPE to provide the list of patterns. Once the patterns are loaded, the STE determines the execution type of the block (continuous-time or discrete-time) based on the existence of sample time, and assigns the appropriate pattern from the list. Then, the block details, such as execution order number, sample time (if discrete) and the inter-arrival time are mapped onto the pattern. Next, based on the block type, the STE tries to load the plug-in that generates the block routine as a `C`-function and a `Dafny` verification objective. If there exists no plug-in for the given block type, an empty block routine is generated. With the generation of the block routine, all the template constructs have been instantiated with block-specific ones. With this, the block transformation is complete and the pattern becomes an instantiation of an STA.



Select Simulink Model Folder

Browse /Users/pfj01/Desktop/executables/BBW/brake_acc_nodiv.mdl

Sorted Order List

Browse /Users/pfj01/Desktop/executables/BBW/sortedorder.txt

Select Timed Automata File

Browse /Users/pfj01/Desktop/BBW.xml

Console

```
Plugin for reference not found
Block routine for brake_acc_nodiv/20ms1 not generated
Parsing block: brake_acc_nodiv/Veh_Speed_Estimator/Add
Parsing block: brake_acc_nodiv/Veh_Speed_Estimator/average_rpm
Parsing block: brake_acc_nodiv/Veh_Speed_Estimator/w (grad//s)
Parsing block: brake_acc_nodiv/Veh_Speed_Estimator/v (km//h)
SList contains 204 entriess
UPPAAL document contains total 157 blocks (145 automata and 12 constants)
The model contains 129 discrete and 16 continuous blocks
The model parsing took 11 seconds
```

Fig. 6. Simppaal GUI.

Once the automaton is obtained, in Step 3.b, the STE determines the type of the output produced by the block, which can be either scalar or vector of type `Boolean` or `Double`. Even though the type of the output in general is defined for each block type, sometimes it can be determined by other factors, such as: the type and format of the input (ex: a Gain block that has input scalar can produce either scalar, vector or matrix, but if the input is vector it cannot produce scalar).

The transformation is completed in Step 3.c, and the STA that corresponds to the given block is added to the Uppaal model. The operation includes adding the automaton into the list of automata, and instantiating a global shared variable with a type and name determined in Step 3.b, which represents the output channel. Finally, in Step 4, the generated Uppaal model is saved into the file system, in XML format, which can then be used as an input to the Uppaal SMC tool.

In the current version, Simppaal is a standalone tool that can be used via the simple interface, as presented in Figure 6. To create a Uppaal model file, the user has to select the root Simulink model, slist, and the destination file where the result Uppaal model is saved. Once all parameters have been selected, the transformation can be started by pressing the `Start` button. During the transformation, the Simppaal tool logs important messages in the console part. After the transformation is complete, the user can save the console as a log file for analyzing the output, or for debugging purposes.

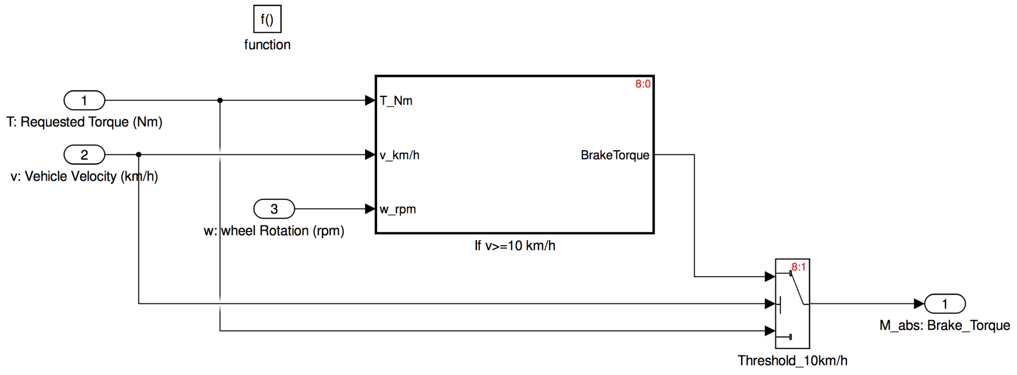
4.3 Scope of Application

The current version of the tool has a limited scope and can be used for a certain subset of Simulink models only. This is mostly due to the fact that currently, we have implemented a set of plug-ins for the automatic generation of Simulink blocks that are present in the BBW model.

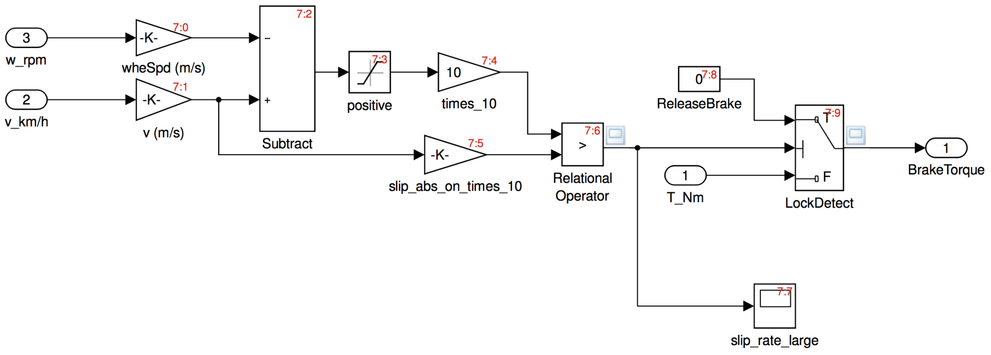
The Simppaal tool cannot properly handle model referencing in cases when a parent model references directly a model instead of a library. This is due to the fact that the structures of the referenced models and the referenced libraries are different. The referenced libraries always start with a subsystem block that has the same ‘in’ and ‘out’ ports as the subsystem that is referencing the library in the parent model. In contrast, the referenced models are loaded as such, meaning that the contents are not necessarily wrapped inside a subsystem block. We are already working on addressing this technical limitation, which is expected to be fixed in the next release of the Simppaal tool.

5 APPLICATION ON AN INDUSTRIAL USE CASE: THE BRAKE-BY-WIRE SYSTEM

In this section, we introduce the BBW use case, which we transform into a NSTA with the Simppaal tool and the verified with Uppaal.



(a)



(b)

Fig. 7. The ABS function: (a) The ABS subsystem (b) The "If v>=10km/h" subsystem

System description. BBW is a prototype implementation of a braking system equipped with an anti-lock braking (ABS) function, and without any mechanical connection between the brake pedal and the four brake actuators. The dedicated pedal sensor reads its current position, which is used to compute the desired brake torque that is distributed to each wheel. The four wheel sensors measure the rotational speeds, which are used to compute the velocity of the car. If the velocity exceeds 10km/h, the ABS function is enabled to avoid the wheels locking or skidding. The friction coefficient has a nonlinear relationship with the slip rate. When the slip rate starts increasing, the friction coefficient also increases. After a certain threshold, an increase in the slip rate reduces the friction coefficient of the wheel. For this reason, the ABS decides if the requested brake torque should be applied or if the brake actuator is released based on the value of the slip rate.

Despite being a system prototype, the BBW Simulink model is a faithful representation of a realistic industrial system. It contains 320 blocks connected in a hierarchical model with four levels of nesting. For exemplification purposes, we present the ABS function in Figure 7.

At system level, the BBW system has a set of 13 functional and 4 timing requirements that need to be verified. In this parer, we present six of them, as follows:

R1$_{BBW}$ The time needed for a brake request to be computed shell not exceed 10 ms.

R2$_{BBW}$ The time needed for a brake request to propagate from the pedal sensor to the wheel actuator shell not exceed 50 ms.

R3$_{BBW}$ The difference between the time needed for a brake request to propagate to two different wheel actuators shell not exceed 4 ms.

R4$_{BBW}$ If the brake request is 0, then the ABS sell set the torque to 0.

R5$_{BBW}$ If the vehicle speed > 10 km/h and the slip rate > threshold, then the ABS shall set the torque to 0.

R6$_{BBW}$ If the vehicle speed ≤ 10 km/h or the slip rate ≤ threshold, then the ABS shall apply the requested torque.

Transformation. The Simppaal tool produces a network of 145 automata corresponding to the computational blocks in the Simulink model (e.g., gain, sum, rounding), while the 175 non-computational blocks have been removed during the flattening and transformation phases. In this network, 129 STA are created using the discrete-time pattern and 16 STA are created using the continuous-time pattern. Simppaal also generates 129 block routines automatically, while 16 blocks routines are left to be implemented manually (e.g., S-functions, Sateflow blocks, masked blocks). The BBW model contains four identical simple flow charts represented as Sateflow blocks, which are modeled manually based on a 1-to-1 mapping (i.e., flow chart conditions are mapped to guards, and flow chart actions are mapped to updates). Due to the fore-mentioned missing block routines, the automatically generated model cannot be used as such for analysis. However, these shortcomings can be fixed with minimal effort, which makes the model suitable for analysis using the Uppaal SMC tool. A short tutorial on how to run Simppaal tool on the BBW example can be accessed on the following link[2].

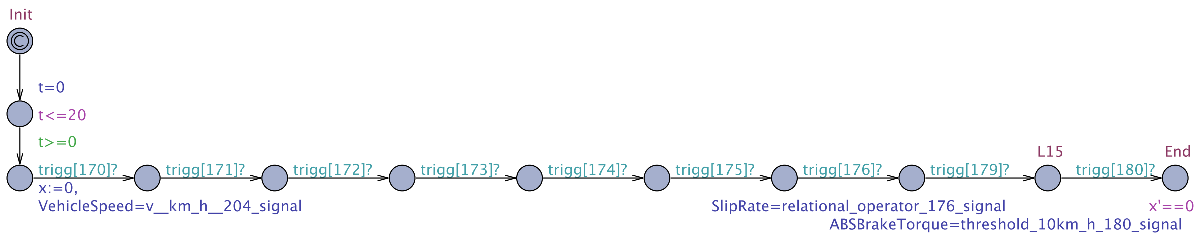


Fig. 8. The Monitor automaton for requirement $R5_{BBW}$.

Analysis and results. Once the complete model is created, one can validate the correctness of each of the STA by comparing its simulation trace in Uppaal SMC with the simulation trace of the corresponding Simulink block. Additionally, we can verify an extensive set of functional and timing requirements. In Table 1, we provide concrete verification results for the six requirements introduce above. For this, we need to implement an additional STA

[2]http://www.idt.mdh.se/ predragf/SIMPPAAL/

Table 1. Overall Results of Statistical Model Checking.

Req.	Query	Result	Runs	Time
$R1_{BBW}$	$Pr[<= 35](<> Monitor.End\ and\ Monitor.x <= 10)$	$Pr \in [0.975066, 1]$ with confidence 0.9875	201	4926.6 s
$R2_{BBW}$	$Pr[<= 75](<> Monitor.End\ and\ and\ Monitor.x <= 50)$	$Pr \in [0.990014, 1]$ with confidence 0.995	597	13872.1 s
$R3_{BBW}$	$Pr[<= 75](<> Monitor.End\ and\ Monitor.y <= 4)$	$Pr \in [0.980056, 1]$ with confidence 0.99	263	6097.2 s
$R4_{BBW}$	$Pr[<= 75](<> Monitor.End\ and\ (RequestedTorque == 0\ imply\ ABSBrakeTorque == 0))$	$Pr \in [0.998, 1]$ with confidence 0.999	3797	290362.8 s
	$Pr[<= 75](<> Monitor.End\ and\ RequestedTorque == 0)$	$Pr \in [0.851567, 0.951396]$ with confidence 0.95	145	13224 s
$R5_{BBW}$	$Pr[<= 75](<> Monitor.End\ and\ ((VehicleSpeed > 10\ and\ SlipRate > 0)\ imply\ ABSBrakeTorque == 0))$	$Pr \in [0.995005, 1]$ with confidence 0.99	1058	23892.4 s
	$Pr[<= 75](<> Monitor.End\ and\ VehicleSpeed > 10\ and\ SlipRate > 0)$	$Pr \in [0.893009, 0.992099]$ with confidence 0.95	79	2588s
$R6_{BBW}$	$Pr[<= 75](<> Monitor.End\ and\ ((VehicleSpeed <= 10\ or\ SlipRate <= 0)\ imply\ ABSBrakeTorque == RequestedTorque))$	$Pr \in [0.902602, 1]$ with confidence 0.95	36	656.8 s
	$Pr[<= 75](<> Monitor.End\ and\ (VehicleSpeed <= 10\ or\ SlipRate <= 0))$	$Pr \in [0.00790082, 0.106991]$ with confidence 0.95	79	18894.6 s

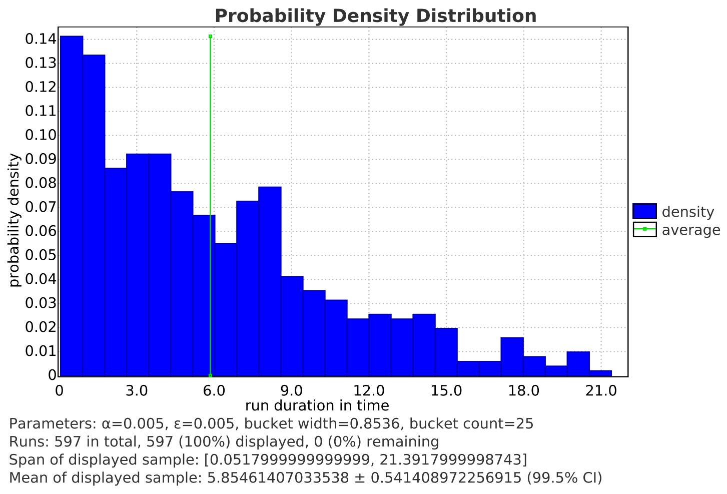
that monitors the execution of the system for a particular requirement. For instance, for requirement $R5_{BBW}$ we have implemented the monitor presented in Fig. 8 that follows the execution of each component in the ABS subsystem according to the execution order. For each simulation to start at an arbitrary moment in time, we introduce a clock t that allows for a delay between 0 and 25 time units. Similar monitors have been implemented for the other five requirements. In Fig. 8, we have added an additional clock x, that monitors the time needed for the ABS to execute. This mechanism is employed to analyze the three timing requirements.

The Uppaal SMC can achieve analysis results with different probability interval spans and different confidence levels, depending on the values of the statistical parameters. During the analysis, we have opted for different values of α, the probability of false negatives, and ϵ, the probability uncertainty, which influence the number of runs generated by the model checker. To run the verification, we have used a HP Z620 Workstation with Intel Xeon Processor 3.0 Ghz, 8 cores, and 64 GB DDR3.

Uppaal SMC can also display additional verification results, such as probability density distribution. In Fig. 9, we show the distribution of the end-to-end timing results of requirement $R2_{BBW}$).

6 DISCUSSION ON THE APPROACH

In this section we discuss the characteristics of the proposed approach for the transformation of Simulink models into NSTA and their analysis using Uppaal SMC.



Parameters: α=0.005, ε=0.005, bucket width=0.8536, bucket count=25
Runs: 597 in total, 597 (100%) displayed, 0 (0%) remaining
Span of displayed sample: [0.0517999999999999, 21.3917999998743]
Mean of displayed sample: 5.85461407033538 ± 0.541408972256915 (99.5% CI)

Fig. 9. Probability Density Distribution for requirement $R1_{BBW}$

Reusability and Automation. The core of the Simulink to NSTA transformation is the transformation patterns. Such approach provides a straightforward and highly automated transformation procedure, with the transformation result being faithful to the original model. This is achieved by instantiating patterns for both discrete- and continuous-time blocks whose functional behavior is encoded as `C` functions, implemented in our SIMPPAAL tool. The usage of the `C` routines enables us to extend the application of the patterns and faithfully represent the functional behavior of each block. This means that no matter how complex the computational routine of the given block is, it can be transformed via the given patterns. Our approach supports the transformation of the current blocks in the Simulink library, but also of custom atomic blocks built using the concepts of S-function and Mask. We verify the functional correctness of each block (its block routine) by using the `Dafny` program verifier. To be able to automatically transform all blocks of a given model, an adequate set of plug-ins has to be developed. This is by no means a limitation of the tool, but it is due to lack of time and resources needed to develop the extensive set of plug-ins. The proposed flattening procedure based on a recursive algorithm can, in principle, can be applied to flatten any model with arbitrary nested composite blocks. The limitation of the current implementation of the SIMPPAAL tool with respect to transforming Simulink models composed of several Simulink files has been solved, and the implementation will be provided in the next release of the tool. Based on our current experience, the current version of the SIMPPAAL tool, as presented in this paper, can be used to generate an NSTA model of the Brake-by-Wire industrial prototype model, which is suitable for analysis using UPPAAL SMC, after minor changes.

Despite the mentioned limitations, our approach and the SIMPPAAL tool has been successfully applied on the Simulink model of the Brake-by-Wire industrial prototype. The transformation results from the case study show that SIMPPAAL is fast and efficient in transforming Simulink models into NSTA. The positive experience of the application described in this paper, combined with the solid code base and the modular tool architecture form a solid basis towards extending it to a more complete platform, which can be further extended with new features, including the formal specification of properties to be verified, and ultimately completely automated to a *"push-button"* formal analysis of Simulink models via SMC.

Verification Lessons. The state space explosion is a real problem when attempting to exhaustively verify complex automotive systems. We overcome this by employing statistical model-checking techniques that generate stochastic simulations and employ statistical methods to estimate probabilities and probability distributions over time with given confidence levels. However, to achieve a high confidence level, Uppaal SMC might need to generate a large number of simulation, which is time consuming process. For example, to verify $R5_{BBW}$ (see Table 1), Uppaal SMC needs around 80 hours to generate 3797 runs. The work proposed in this paper shows promising results and enables verification of functional and timing requirements of automotive embedded systems modeled in Simulink, but further improvements might be required to ensure the efficiency and scalability required for industrial adoption.

7 RELATED WORK

Verification of *control* algorithms implemented in Simulink has been formulated as a hybrid-automata *reachability problem*, like in CheckMate [7], where Simulink models are transformed into polyhedral-invariant hybrid automata (PIHA). This method is limited to a restricted class of models, as reachability is known to be undecidable for hybrid automata, in the general case, and it does not scale well to the complexity of real industrial cases, containing large numbers of very diverse modules: continuous, discrete, StateFlow, etc. Verification of more complex Simulink models has been addressed following three strategies:

(a) Generation and abstraction of simulation traces. The simulation capabilities of Simulink are used for generating and collecting simulation traces that are later transformed, by abstraction, into a state machine representing the system's behavior, which can be model checked without difficulty [5]. PlasmaLab follows this approach, relying on SMC for model checking [17]. This strategy is limited by the feasibility to generate an exhaustive simulation of the system and raises concerns about completeness. Moreover, it is not adequate for analyzing extra-functional properties, at least not without further changes on the initial model. On the positive side, the approach is generic, applicable to any kind of Simulink diagram, and does not require adding more computation if new blocks are considered.

(b) Abstraction of blocks into contracts/theories and formal analysis. First, the system designer "lifts" the specification of each block using some logics. Second, the whole specification is composed and fed into an analysis engine. Ferrante et al. [12] use contract-based theory in order to lift the block specification, and rely on a combination of SAT solvers and the NuSMV model checker for analysis. Hocking et al. [15] use the PVS specification language for writing the specification, and rely on the PVS theorem prover for analysis. A limitation of this strategy is that both steps still require much user interaction, so it is error-prone and requires certain understanding of the formal analysis engines, which is not common among embedded systems engineers.

(c) Model to model transformation followed by model checking. This strategy tries to minimize user intervention. It applies some kind of automated model-to-model (M2M) transformation from Simulink into an automata language that can be verified with model checking. This strategy has received much attention in the literature. The approach proposed by Barnat et al. [4] focuses on transforming the Simulink models into the language of an LTL explicit model checker called DiViNE. The authors show how this can be integrated with the Honeywell formal verification environment. They only provide support to discrete blocks, yet they show it suitable for the aeronautics industry. Similarly, the approach by Meenakshi et al. [22] proposes a transformation of discrete blocks into NuSMV. In contrast, Agrawal et al. [2] propose a transformation approach of Simulink models into networks of automata, without providing concrete means for formal verification. The work by Miller [23] proposes a translation from Simulink to Lustre, and enables formal verification with a constellation of model checkers and provers. The transformation of StateFlow design elements has been addressed in research endeavors by Manamcheri [19] and Jiang et al. [16], in which the authors propose transformation frameworks from StateFlow/Simulink into timed and hybrid automata, respectively, without considering other types of Simulink blocks.

In general, the solutions available for automated M2M transformation of Simulink (i) are quite *restrictive* with respect to the number of supported block types (only discrete blocks or only StateFlow diagrams), and (ii) have been applied only to academic or middle-size Simulink models, such as the engine control system appearing in the Simulink distribution, thus raising concerns about scalability. The only exception is Zuliani et al. [25], which uses Bayesian statistical model checking for analyzing the specification and can scale better to larger-size models. Despite that, the approach has been applied to a medium-size Simulink model only, and it seems to have practical limitations such as not accepting multi-file Simulink models.

Simppaal follows the third strategy, but it goes beyond the current state of the art, by reducing the modeling effort (M2M transformation is based on templates and fully automated), and by supporting a larger number of Simulink blocks (some of them is still under development). To our best knowledge, it is also the only approach that formally verifies the encodings of the Simulink blocks functionalities, by using `Dafny`.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we have extended and improved our already existing pattern-based approach for transforming Simulink models into NSTA semantics [14]. For that purpose, we have proposed the following extensions: i) a formal definition of Simulink blocks, to facilitate the soundness proof between the formalized Simulink model and the SPTA, ii) a definition of a Simulink model as a serial composition of interconnected Simulink blocks, iii) a soundness proof for the mapping of the formalized Simulink blocks into the respective STA, for discrete-time models, and iv) a tool, called Simppaal, which embodies our approach and is intended for automating the complete process of transforming Simulink models into NSTA.

The main purpose of the tool is to enable formal analysis of large Simulink industrial models, and keep the formal modeling effort to a minimum, by adding automation to the transformation. A secondary goal is to make the approach applicable for practitioners, who are not expert in formal methods. Both the scalability and the suitability for engineers of out tool Simppaal await validation.

The approach described in this paper is suitable for transforming Simulink models that contain both continuous-time and discrete-time blocks, which has been identified as a major limitation of the existing academic approaches. Another strong point of our approach is the fact that it can be applied on both Simulink-provided and user-defined blocks. Additionally, the Simppaal tool provides a high degree of automation, thus minimizing the interaction with the user during the formal model generation phase. This is achieved through the complete automation of the M2M transformation from Simulink to NSTA. This feature of the approach makes it a promising candidate for adoption in industrial settings, where analysis and verification approaches are evaluated and approved based on how fast, accurate and user-demanding they are. Another benefit of the proposed approach is the fact that all the functional behavior of the model is verified. For that purpose, we use `Dafny`, a program and language verifier by which we prove the correctness of each computational routine that encodes the functional behavior of a Simulink block. Similar to the generation of the formal model, the generation of the `Dafny` verification routines is in principle completely automated and handled by the Simppaal tool, thus almost no additional modeling effort is required from the user.

The ability of the tool to automatically generate the NSTA model of any type of industrial system modeled in Simulink depends on the coverage of the Simulink block types by the plug-in library. The current version of the Simppaal plug-in library consists of ten plug-ins that are enough to cover most of the block types found in the Brake-by-Wire model. In order for the tool to be applicable on a large and diverse set of industrial Simulink models, the plug-in library has to be extended accordingly.

Our future work can proceed in several directions. First, we aim at improving the efficiency and scalability of our approach, by proposing a new transformation procedure for the triggered subsystem blocks. Second, we intend to implement the missing features of the Simppaal tool, such that it can be applied on larger industrial systems. By doing that, we seek for more industrial penetration. This is tightly connected with the next direction of our work, which includes more extensive validation of the approach. The goal is to consider at least two examples of industrial Simulink models of operational systems, and i) test the scalability of the Simppaal tool to generate formal models of such Simulink models, and ii) perform statistical model checking of the obtained models using Uppaal SMC. Finally, we plan to explore the possibilities of generating formal models required by other verification tools, such as for instance the STORM probabilistic model checker [11], in an attempt to enhance the class of systems that can be tackled.

REFERENCES

[1] 2009. *ISO/DIS 26262-1 - Road vehicles âĂŤ Functional safety âĂŤ Part 1 Glossary.* Technical Report. Geneva, Switzerland.

[2] A. Agrawal, G. Simon, and G. Karsai. 2004. Semantic Translation of Simulink/Stateflow Models to Hybrid Automata using Graph Transformations. *ENTCS Journal* 109 (2004), 43–56.

[3] Rajeev Alur and P. Madhusudan. 2004. Decision problems for timed automata: A survey. In *In Proceedings of SFMâĂŹ04, Lect. Notes Comput. Sci. 3185, 1âĂŞ24.* Springer, 1–24.

[4] J. Barnat, J. Beran, L. Brim, T. Kratochvíla, and P. Ročkai. 2012. Tool chain to Support Automated Formal Verification of Avionics Simulink Designs. In *FMICS*. Springer, 78–92.

[5] B. Boyer, K. Corre, A. Legay, and S. Sedwards. 2013. PLASMA-lab: A flexible, distributable statistical model checking library. In *QEST*. Springer, 160–164.

[6] P. Bulychev, A. David, K.G. Larsen, A. Legay, G. Li, and D.B. Poulsen. 2012. Rewrite-based Statistical Model Checking of WMTL. In *RV Conference*. Springer, 260–275.

[7] Alongkrit Chutinan and Bruce H Krogh. 2003. Computational techniques for hybrid system verification. *IEEE transactions on automatic control* 48, 1 (2003), 64–75.

[8] J. B. Dabney and T. L Harman. 2004. *Mastering Simulink.* Pearson/Prentice Hall.

[9] A. David, D. Du, K.G. Larsen, A. Legay, M. Mikučionis, D.B. Poulsen, and S. Sedwards. 2012. Statistical Model Checking for Stochastic Hybrid Systems. *arXiv preprint arXiv:1208.3856* (2012).

[10] Alexandre David, K.G. Larsen, A. Legay, M. Mikučionis, and D.B. Poulsen. 2015. Uppaal SMC Tutorial. *STTT Journal* 17, 4 (2015), 397–415.

[11] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. 2017. A storm is Coming: A Modern Probabilistic Model Checker. *arXiv preprint arXiv:1702.04311* (2017).

[12] Orlando Ferrante, Luca Benvenuti, Leonardo Mangeruca, Christos Sofronis, and Alberto Ferrari. 2012. Parallel NuSMV: A NuSMV extension for the verification of complex embedded systems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 7613 LNCS. 409–416. https://doi.org/10.1007/978-3-642-33675-1_38

[13] Predrag Filipovikj, Nesredin Mahmud, Raluca Marinescu, Guillermo Rodriguez-Navas, Cristina Seceleanu, Oscar Ljungkrantz, and Henrik Lönn. 2017. *Analyzing Industrial Simulink Models by Statistical Model Checking*. Technical Report. http://www.es.mdh.se/publications/4714-

[14] Predrag Filipovikj, Nesredin Mahmud, Raluca Marinescu, Cristina Seceleanu, Oscar Ljungkrantz, and Henrik Lönn. 2016. *Simulink to UPPAAL Statistical Model Checker: Analyzing Automotive Industrial Systems.* Springer International Publishing, Cham, 748–756. https://doi.org/10.1007/978-3-319-48989-6_46

[15] Ashlie B. Hocking, M. Anthony Aiello, John C. Knight, and Nikos Aréchiga. 2016. Proving Critical Properties of Simulink Models. In *Proceedings of IEEE International Symposium on High Assurance Systems Engineering*, Vol. 2016-March. 189–196. https://doi.org/10.1109/HASE.2016.38

[16] Y. Jiang, Y. Yang, H. Liu, H. Kong, M. Gu, J. Sun, and L. Sha. 2016. From Stateflow Simulation to Verified Implementation: A Verification Approach and A Real-Time Train Controller Design. In *RTAS'16*. 1–11. https://doi.org/10.1109/RTAS.2016.7461337

[17] A. Legay and L.M. Traonouez. 2015. Statistical Model Checking of Simulink Models with Plasma Lab. In *FTSCS'15*. Springer, 259–264.

[18] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR'10*. Springer, 348–370.

[19] K. Manamcheri Sukumar. 2011. Translation of Simulink-Stateflow Models to Hybrid Automata. (2011).

[20] Raluca Marinescu, Henrik Kaijser, Marius Mikučionis, Cristina Seceleanu, Henrik Lönn, and Alexandre David. 2014. Analyzing Industrial Architectural Models by Simulation and Model-Checking. In *Third International Workshop on Formal Techniques for Safety-Critical Systems*. http://www.es.mdh.se/publications/3762-

[21] Raluca Marinescu, Saad Mubeen, and Cristina Seceleanu. 2016. Pruning Architectural Models of Automotive Embedded Systems via Dependency Analysis. In *42nd Euromicro Conference series on Software Engineering and Advanced Applications*. http://www.es.mdh.se/publications/4363-

[22] B Meenakshi, A. Bhatnagar, and S. Roy. 2006. Tool for Translating Simulink Models into Input Language of a Model Checker. In *ICFEM*. Springer, 606–620.

[23] Steven P. Miller. 2009. Bridging the Gap Between Model-Based Development and Model Checking. In *TACAS*. Springer, 443–453.

[24] Inc. The MathWorks. 2017. *Simulink Reference, Matlab&Simulink* (R2017a ed.). The MathWorks, Inc., 3 Apple Hill Drive Natick, MA 01760-2098.

[25] Paolo Zuliani, André Platzer, and Edmund M Clarke. 2013. Bayesian statistical model checking with application to Stateflow/Simulink verification. *Formal Methods in System Design* 43, 2 (2013), 338–367.