11. SIMPPAAL - A Framework For Statistical Model Checking of Industrial Simulink Models

Predrag Filipovikj, Nesredin Mahmud, Raluca Marinescu, Cristina Seceleanu, Oscar Ljungkrantz and Henrik Lönn. *Submitted to ACM Transactions on Software Engineering and Methodology (TOSEM).*

Abstract: 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 system complexity, by supporting functional design from a set of block libraries that can be simulated and analyzed for hidden errors, but also used for code generation. Since such implementations are at most as correct as the models that they are generated from, providing assurance that Simulink models fulfill given functional and timing requirements is desirable. Exhaustive verification methods like model checking might easily encounter the state-space explosion problem for large Simulink descriptions. To tackle such problem, 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 be formally analyzed with UPPAAL Statistical Model Checker. The latter employs scalable yet statistical methods for reasoning, rather than exhaustive ones. To enable the formal analysis, we first define the formal syntax and semantics of Simulink blocks and their composition, and show that the proposed transformation is provably correct for a certain class of Simulink models. Our method is supported by the SIMPPAAL tool, which we introduce and apply on two industrial Simulink models, of a prototype called the Brake-by-Wire system, and of an operational Adjustable Speed Limiter system. This work enables the formal analysis of industrial Simulink models, by automatically generating their stochastic timed automata counterparts.

11.1 Introduction and Motivation

The current trend in automotive systems is to replace mechanical components with electronic ones (e.g. in the drive-by-wire technology). While beneficial, this step also results in highly complex, difficult to analyze software-based systems that control *safety-critical* functions. Such systems need to be developed by following safety standards like ISO26262 [1], which require provable assurance of safe operation at all design levels.

To achieve some form of assurance with respect to safety-critical requirements, as well as gain valuable design insight, *model-based design* enables industry to create executable specifications, for instance in the form of Simulink [35][9] models that can be simulated and formally analyzed [30][24] to detect hidden design errors and requirements violations. The formal analysis of Simulink models has been a research target for a while now. Existing work [24][4][20] provides solutions based on (stochastic) hybrid automata, extended finite automata etc., yet no integrated framework exists, which could serve as a basis for automated tool support applicable to complex industrial Simulink models. To address this gap, in this paper, we introduce a pattern-based approach (Section 11.3) that captures formally the behaviors of Simulink blocks, as networks of stochastic timed automata, and report our experience from analyzing two industrial systems, with UPPAAL SMC (Statistical Model Checker) [11] (Section 11.5). Since we target the formal analysis of industrial-size Simulink models, we chose statistical instead of exahustive model checking due to the scalability of the former. Both of our use cases, that is, the *Brake-by-Wire* (BBW) prototype and the operational *Adjustable Speed Limiter* (ASL) system come 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 whose functionality is realized through a set of atomic 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 for Simulink blocks. The definition captures the functionality of an atomic Simulink block as a `blockRoutine()`, described in C, which 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. To ensure that our C encodings are correct, we verify them with the program verifier `Dafny` [25], yet we omit presenting the details in this paper. Next, we define the semantics of Simulink blocks in terms of timed transition systems, and 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, for 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 our transformation of Simulink into stochastic timed automata is the preservation of the correct execution order of 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. To be able to apply our approach on the selected industrial use cases, 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.

The crux of our method is threefold: (i) the automated transformation relies on transformation patterns, which eases the modeling process while preserving the execution semantics of Simulink blocks, (ii) for models containing discrete-time blocks only, which form a large class of industrial Simulink artifacts, we prove that their stochastic timed automata encoding is correct with respect to our proposed Simulink block semantics, and (iii) the proposed tool support SIMPPAAL is validated on industrial use cases.

Our endeavor is justified by the industrial needs of ensuring correctness with respect to both functional and timing behaviors of automotive embedded systems. Moreover, an initial investigation of verifying large Simulink models with the existing commercial tool called Simulink Design Verifier [30] shows limitations in terms of scalability and coverage of all types of requirements. By applying our SIMPPAAL tool on the selected industrial systems, BBW and ASL, respectively, we obtain the following results. In the BBW case, we show that SIMPPAAL can automatically generate the network of stochastic timed automata corresponding to the Simulink blocks of the BBW model and their sorted order of execution, and analyze it with UPPAAL SMC 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 that cannot be verified exhaustively unless reduction techniques are applied [27][28]. It is then foreseeable that applying exhaustive model checking on more complex industrial models would most likely expose a similar 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. In the ASL case on the other hand, which is a much larger system than BBW, we are able to automatically transform a large part of the Simulink model into the formal counterpart, yet the resulting model is not analyzable as such. Despite this fact, our findings from applying SIMPPAAL to ASL help us to expose some implementation-based limitations of the tool and its parser, which are to be addressed in the subsequent releases of SIMPPAAL.

The remainder of the paper is organized as follows. In Section 11.2 we overview Simulink, stochastic timed automata, UPPAAL SMC, and `Dafny` language and program verifier, after which we present our Simulink to stochastic timed automata transformation approach in Section 11.3. The architecture of the SIMPPAAL tool is described in Section 11.4, and its validation by applying it on the BBW prototype is shown in Section 11.5. In Section 11.7 we compare to related work, before concluding the paper and outlining future lines of research in Section 11.8.

11.2 Preliminaries

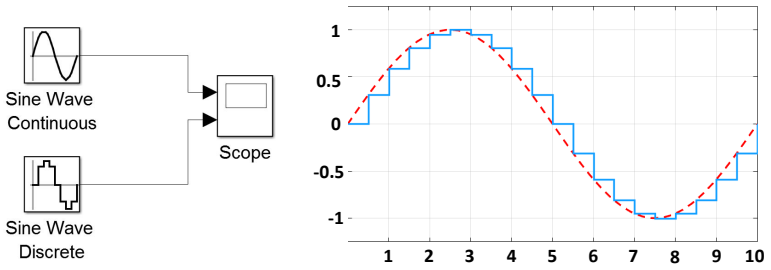
In this section, we present an overview of Simulink, stochastic (priced) timed automata, the UPPAAL SMC tool, and `Dafny` language and program verifier.

11.2.1 Simulink

Simulink [35][9] 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 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 an 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. 11.1 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 interfaces, encapsulated logic, and hidden data.

A hierarchical model is achieved by the implementation of at least one *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* blocks from the *Ports and Subsystems* library, and *Goto* and *From* blocks from the *Signal Routing* library.

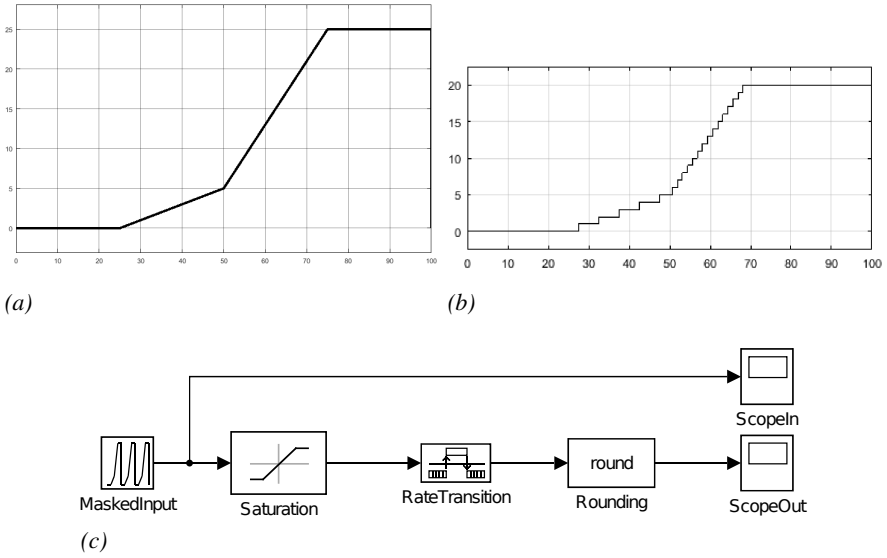
As an example, in Fig. 11.2c we show a small Simulink model, which is an excerpt from the BBW model. The first block is a masked block called



(a) *(b)*

Figure 11.1: Example (Sine Wave Block): (a) Simulink Diagram and (b) Simulation Result

MaskedInput that produces the signal presented in Fig. 11.2a. 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, being displayed in Fig. 11.2b. Between the *Saturation* and the *Rounding* blocks, 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)

Figure 11.2: 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 refer to as *nested level*. This means that, when a non-virtual subsystem is to be executed during simulation, all blocks inside the subsystem will be executed first, after which 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 *slist*.

11.2.2 UPPAAL SMC

UPPAAL SMC [10] is a statistical model checker (implemented as an extension of UPPAAL toolset) for system models represented as networks of stochastic priced timed automata. In UPPAAL SMC the automata have a stochastic interpretation based on: (i) the probabilistic choices between multiple enabled transitions, and (ii) the non-deterministic time delays that can be refined based on probability distributions, either uniform distributions for time-bounded delays or user-defined exponential distributions for unbounded delays.

A *stochastic priced timed automaton* (SPTA) is defined as the following tuple:

$$SPTA = \langle L, l_0, X, \Sigma, E, R, I, \mu, \gamma \rangle, \tag{11.1}$$

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 $\mathbb{R}^X$, $R : L \to \mathbb{N}^X$ 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 s is either a uniform or an exponential distribution, depending on the invariant of location l. Let E_l denote the disjunction of guards such that $(l,g,o,-,-) \in E$ for some output o. Then, $D(l,v) = \sup\{d \in \mathbb{R}_{\geq 0} : v + d \models I(l)\}$ 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), via the parallel composition operator ($A1 \parallel A2 \parallel ... \parallel An$). The states of the NSPTA are defined as a tuple $s = \langle s_1, ..., s_n \rangle$, 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 as 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 a network of stop-watch timed automata, which is a subset of NSPTA.

UPPAAL SMC uses a probabilistic extension of *weighted metric temporal logic* (WMTL) [7] 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 ($Pr[<= bound](\star_{x \leq C} \phi) \geq p$),
- *Probability evaluation*: calculate the probability $Pr[<= bound](\star_{x \leq C} \phi)$ for some NSPTA,
- *Probability comparison*: is $Pr[<= bound](\star_{x \leq C} \phi_1) > Pr[<= bound](\star_{y \leq D} \phi_2)$?,

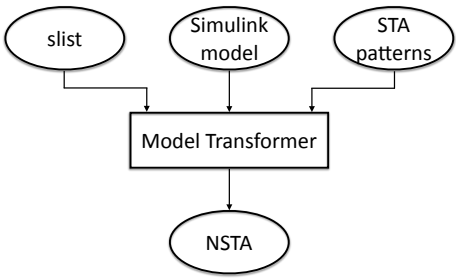


Figure 11.3: Overall Approach of the Simulink to UPPAAL SMC Transformation

where $\star$ stands for either *future* ($\Diamond$) or *globally* ($\Box$) temporal operator, and *[<= bound]* denotes the time bound of the executions.

11.2.3 Dafny

`Dafny` [25] is an imperative, sequential programming language with generic classes and dynamic allocation that allows for built-in specification constructs, such as standard pre- and postconditions, framing constructs, and termination metrics. The language also offers updatable ghost variables, recursive functions, and types like sets and sequences. The specification style is based on dynamic frames [21] and the language also includes user-defined mathematical functions. These features permit programs to be specified for modular verification, so that the separate verification of each part of the program implies the correctness of the whole program.

`Dafny` includes a static program verifier that can be used to verify the functional correctness of the programs. The tool translates a given (`Dafny`) program into the intermediate verification language `Boogie` [5], which ensures that the correctness of the `Boogie` program implies the correctness of the `Dafny` program (the semantics of `Dafny` is defined in terms of `Boogie`). The `Boogie` tool is then used to generate first-order verification conditions that are passed to a theorem prover, in particular to the `Z3` SMT solver [12].

For example, pre- and postconditions have a standard declaration based on the keywords *requires* and *ensures*, respectively. The caller has the responsibility to fulfill the precondition and the implementation has the responsibility to establish the postcondition. If either fails, an error is reported by the verifier.

11.3 Simulink to UPPAAL SMC: Approach

In order to be able to analyze a Simulink model against functional and timing requirements, we propose a general approach of transforming Simulink

models into NTSA, which we present in the following. The transformation handles a wide range of block types, including both atomic and composite blocks that belong to various categories such as: non-virtual, continuous, discrete, s-functions, etc. To ensure the fact that we cover the commonly used Simulink blocks in industry, we base our selection on a study of the types of blocks used in various industrial Simulink models collected from our industrial partners. Fig. 11.3 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 of the Simulink model and the atomic Simulink blocks (Section 11.3.1);

(ii) We propose a flattening algorithm that transforms hierarchical Simulink models into flat models that do not contain composite Simulink blocks (Section 11.3.3);

(iii) We transform the continuous-time and discrete-time atomic blocks into their respective stochastic timed automata (STA) counterparts, using *Transformation Patterns* (Section 11.3.2);

(iv) We give a proof of transformation soundness for Simulink models consisting of discrete-time Simulink blocks only (Section 11.3.4).

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 an automatic translation into corresponding `C` functions.

11.3.1 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 [35]. In the following, we introduce the defintions of the 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 1 (Simulink block) *An atomic Simulink block, denoted by B, is defined as the following tuple:*

$$B = \langle s_n, V_{in}, V_{out}, V_D, \Delta, Init, blockRoutine \rangle, \tag{11.2}$$

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) $\Delta = \{\Delta_0, \Delta_1, \cdots, \Delta_k\}$ - represents the totally ordered set of time points at which an output is produced. For discrete-time Simulink blocks, the value of a time point Δ_j is calculated as $\Delta_j = offset + j * t_s$, where t_s, $offset \in \mathbb{R}_{\geq 0}$ are the sample time and the offset of the atomic Simulink block, respectively, and $0 \leq j \leq k \in \mathbb{N}$ is the index of the element. For continuous blocks $\Delta_j = j * t_s$, where t_s is infinitely small (tends towards zero).

(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, as follows.

Definition 2 (Simulink model) *A Simulink model is formally defined as a sequential composition of n Simulink blocks that communicate via shared variables, as follows:*

$$S = B_1 \otimes B_2 \otimes B_3 \cdots \otimes B_n, \tag{11.3}$$

with: $s^S = \{s_1, s_2, \ldots, s_n\}$ is an ordered list of execution of blocks $B_1, B_2, \ldots, B_n$, such that $\forall (i, j), i, j \in [1, ..n].i < j \Rightarrow s_i < s_j$, $V_{in}^S = \bigcup_{i=1}^{n} V_{in}^i$ is the set of input variables of S, $V_{out}^S = \bigcup_{i=1}^{n} V_{out}^i$ is the set of output variables of S, $V_D^S = \bigcup_{i=1}^{n} V_D^i$ 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, denoted by $\|_{=\Delta_i}$, $1 \leq i \leq n$.

Semantics of Simulink blocks.

Let us rewrite $\Delta_j = j * t_s + offset$ of Definition 1, as an integral multiple of Simulink's simulation step $\delta \in \mathbb{Q}_{\geq 0}$, that is, $\Delta_j = n*(m*\delta)+(r*\delta)$, $n,m,r \in \mathbb{N}$. 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 (Semantics of a Simulink block) *Assume B is a Simulink block as given in Definition 1. The semantics of B is a timed transition system, as follows:*

$$T_B = \langle q_0, Q, \mathscr{L}, \longrightarrow \rangle, \tag{11.4}$$

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, $t_0 \in \mathbb{R}_{\geq 0}$, $\mathscr{L} = \mathscr{L}_a \cup \mathscr{L}_t$ is the set of labels, with $\mathscr{L}_a$ the set of action labels: $\mathscr{L}_a = \{Init, blockRoutine\}$, $\mathscr{L}_t$ the set of time labels: $\mathscr{L}_t = \{r*\delta, m*\delta\}$, and $\longrightarrow$ is the transition relation: $\longrightarrow \subseteq Q \times \mathscr{L}_a \times \mathscr{L}_t \times Q$ with two types of transitions:

$$q_0 \xrightarrow{Init, r*\delta} q' \iff \begin{cases} \text{if } V_D \neq \emptyset \text{ then } = t_0 + r*\delta, \text{ and } \exists y_0|_{t'} \text{ such that } y|_{t'} = y_0|_{t'} \\ \text{else } r = 0, t' = t_0 \end{cases}$$

$$q \xrightarrow{blockRoutine, m*\delta} q' \iff t' = t + m*\delta, \text{ and } \exists u|_t, x|_{t'} \text{ such that } u|_{t'} = f(x|_{t'}, u|_t), \text{ and } y|_{t'} = f(x|_{t'}, u|_{t'}).$$

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 transition, 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 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:

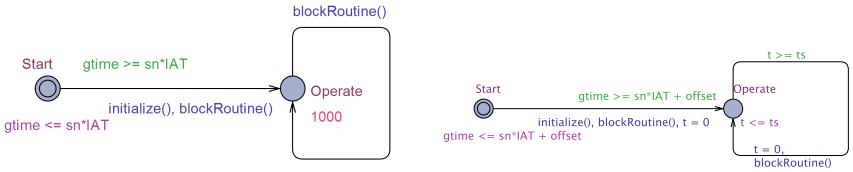
$$q_0 \xrightarrow{Init, r*\delta} q_1 \xrightarrow{blockRoutine, m*\delta} \ldots \xrightarrow{blockRoutine, m*\delta} q_n$$

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 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.

11.3.2 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 they conform to the semantics of the tuple introduced in Equation 11.3. Fig. 11.4 shows our transformation patterns encoded in the input language of UPPAAL SMC.



(a) *(b)*

Figure 11.4: 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 Fig. 11.4a proceeds according to an exponential distribution for unbounded delays, whereas the execution of the discrete-time pattern presented in Fig. 11.4b 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 that determines the probability of the automaton to remain in this location at each simulation step, according to an exponential distribution, in other words $Pr(\text{leaving } \texttt{Operate} \text{ after t}) = 1 - e^{-\lambda t}$.

(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 (Fig. 11.4a) is given as `gtime` $\geq$ `sn`$\star$`IAT`, where: `gtime` is the global clock, `sn` is the block's execution number, and `IAT` is the constant inter-arrival time between two consecutive releases of automata. For the discrete blocks (Fig. 11.4b) the release time depends on the `offset` parameter that 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()` that updates the output.

(iv) 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 check the correctness of the `blockRoutine` functions, which we implement in the subset of the C language used by UPPAAL SMC, we use pre-/post-condition verification using the Dafny [25] 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 the verification procedure for the `blockRoutine` using `Dafny` has been omitted for brevity. For full details, we refer the reader to our technical report [17].

11.3.3 Flattening Algorithm for Preserving the Block Execution Order

In this section, we describe our proposed *flattening* procedure used to assign a unique execution number to each block inside a hierarchical Simulink model, such that the execution order enforced by Simulink is preserved.

The flattening of a Simulink model is performed automatically, in two steps, as follows: (i) remove the non-virtual composite Simulink blocks from the model and replace them with a set of atomic Simulink blocks, and (ii) assign a correct execution order number for the atomic blocks, respectively, 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 ← 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 ← getChildren(currentBlockId)
7:          concatenatedParentId ← 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 11.2.1, the sorted order list *slist* models 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 referred 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 pseudocode of the algorithm, used for assigning a global execution order number to the atomic Simulink blocks nested arbitrary deep inside a given Simulink model, is given as Algorithm 1. The algorithm produces a 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 gives the overall behavior of the original Simulink model, when executed as such. In the newly generated *slist* all the non-virtual subsystems (local contexts) are replaced with their respective set of atomic blocks.

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

11.3.4 Proof of Transformation Soundness

Assume a Simulink model, as described by Definition 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 show that the set of runs of the resulting NSTA, obtained by using the semantic pattern given in Fig. 11.4b, is *refined* by the set of runs of the Simulink model, under the assumption of the latter containing discrete-time blocks only.

We say 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 prove 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 Fig. 11.4b, 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 Fig. 11.4b 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 [3]. This means that constructing a sampled version of the STA automaton of Fig. 11.4b 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 Fig. 11.4b, as follows:

Definition 4 (Sampled semantics of STA of Fig. 11.4b) *Given a timed automaton A as in Fig. 11.4b, 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 $a \in \Sigma \cup \{\varepsilon\}$, 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 $\zeta = (t_0, a_0)(t_1, a_1)(t_2, a_2)\ldots(t_n, a_n)$ is a sequence of transitions:

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

Theorem 1 *Let us assume a discrete-time Simulink block B defined by Definition 1, and a discrete transformation pattern described by a timed automaton with sampled semantics A_δ, as in Definition 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 1 and Definition 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, that is, the STA pattern automaton of Fig. 11.4b, 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 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; hence, 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, and the fact that sequential composition of timed automata is closed under refinement, it follows that a discrete-time Simulink model described as in Definition 2 refines the network of STA that it is transformed into.

The above proof shows that for Simulink models that contain only discrete-time blocks, which are often encountered in industry, proving invariance properties on our STA encoding is sufficient to infer that the original Simulink model satisfies the same property. However, in the following, when we apply our method on the Brake-by-Wire industrial use

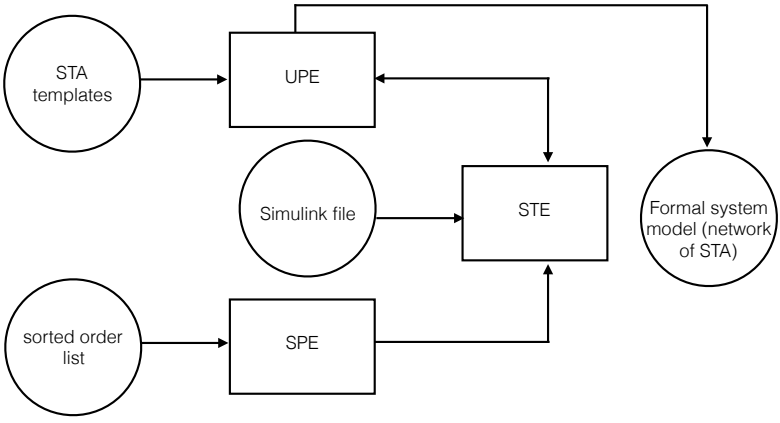


Figure 11.5: SIMPPAAL tool architecture.

case, we analyze the generated formal model by statistical model checking against probabilistic qualitative and quantitative properties. The latter are weaker properties than any type of "for all paths" property, so the result holds in these cases too.

If the Simulink model contains one or more continuous-time blocks that are being transformed by instantiating the transformation pattern of Fig. 11.4a, 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.

11.4 SIMPPAAL Tool

To automate the process of transforming Simulink models into NSTA suitable for analysis using the UPPAAL SMC tool, as described in Section 11.3, we propose a tool called SIMPPAAL (SIMulink to uPPAAL), which does the job. In the following, we describe the architecture and the functionality of SIMPPAAL.

11.4.1 SIMPPAAL Architecture

Our tool is based on a modular design, where the overall functionality is achieved by a set of communicating (software) modules, via well-defined

Application Programming Interfaces (APIs). The architectural design of SIMPPAAL is given in Fig. 11.5, and is based on the following concepts: *artifacts*, which can be input or output represented as circles, and *modules* represented as squares. In this paper, 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.

As shown in Fig. 11.5, the transformation process is based on three different input artifacts: a Simulink model file, an *slist* given as a text 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 this, the STE module is responsible for parsing the Simulink files, the UPE module handles the reading from and writing to UPPAAL-specific .xml files, while the sorted order list is handled by the SPE module.

The STE module

This is the core module of the SIMPPAAL tool. Its main responsibility is to transform a Simulink model file into an 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 a submodule 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

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

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 (note that UPPAAL SMC is part of the UPPAAL toolset) that contains the patterns for the continuous- and discrete-time blocks, as well as for writing the resulting UPPAAL model into a new file. The module provides an API that allows the manipulation of UPPAAL files, including operations such as: deserializing a UPPAAL file into a UPPAAL memory model, adding and retrieving elements from the UPPAAL model (automaton, location, edge) and serializing the UPPAAL memory model back into a file that is used as an input to the UPPAAL toolset. The UPE can be used as a stand-alone library or as part of any other tool for manipulating UPPAAL models.

The SPE module
This is the module that implements the flattening algorithm discussed in Section 11.3.3. 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 to STE. Unlike UPE, the SPE module is bound to a specific purpose and cannot be reused outside the initially-intended context.

11.4.2 SIMPPAAL work flow

The transformation of Simulink models into networks of STA, as implemented in SIMPPAAL, is performed according to 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 block predecessors and successors, respectively
3. Transforming atomic Simulink blocks into STA
   (a) Mapping Simulink block information into STA constructs
   (b) Determining the output signal type
   (c) Instantiating the STA inside the UPPAAL model
4. Saving the generated UPPAAL model in a new file.

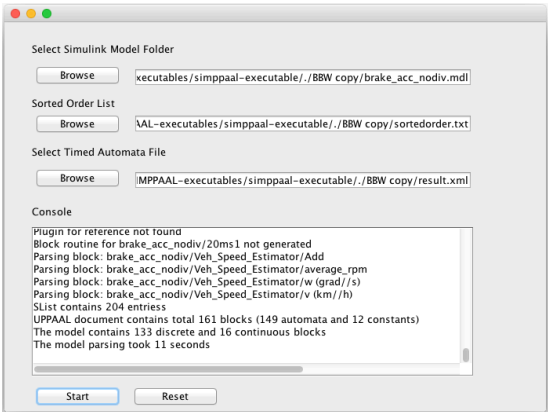


Figure 11.6: SIMPPAAL GUI.

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

The flattened *slist* is then used as a primary input of STE, for transforming the Simulink model into an 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, STE tries to identify all of its *predecessors* that are collected in a list of non-virtual atomic Simulink blocks (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 module 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 a block is gathered, the STE module transforms the Simulink block into an STA in Step 3. The transformation is done in several steps: first, in Step 3.a, STE calls UPE to provide the list of patterns. Once the patterns are loaded, 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, 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 instantiated pattern becomes an STA.

Once the automaton is obtained, in Step 3.b, 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, in general, the type of the output 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 output, but if the input is vector it cannot produce a scalar output).

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 to 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 a simple interface, as presented in Fig. 11.6. To create a UPPAAL model file, the user has to select the root Simulink model, the corresponding slist, and the destination file where the resulting 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.

11.4.3 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.

As for now, SIMPPAAL 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.

In order to validate our method and tool, we apply the latter to analyze the Simulink models corresponding to two industrial use cases, as described in the following.

11.5 Application on Industrial Use Cases

In this section, we introduce the Brake-by-Wire (BBW) and the Adjustable Speed Limiter (ASL) use cases, whose Simulink descriptions we transform into corresponding NSTA, with the SIMPPAAL tool. Further, we verify the BBW Simulink model with UPPAAL SMC.

11.5.1 The Brake-By-Wire Use Case

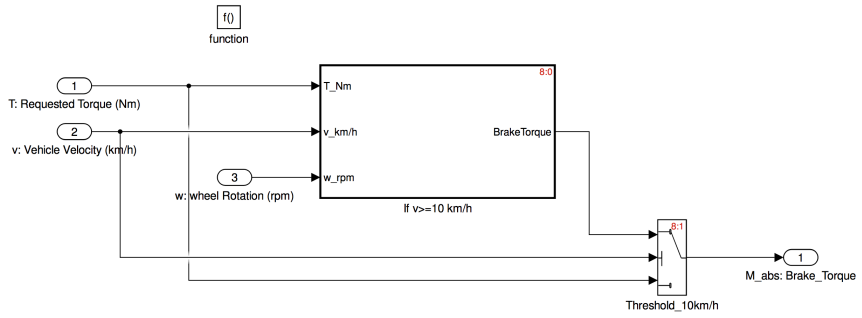
BBW is a prototype implementation of a braking system equipped with an anti-lock braking (ABS) function, and without any mechanical connection between the brake pedal and the four brake actuators. The dedicated pedal sensor reads its current position, which is used to compute the desired brake torque 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 Fig. 11.7.

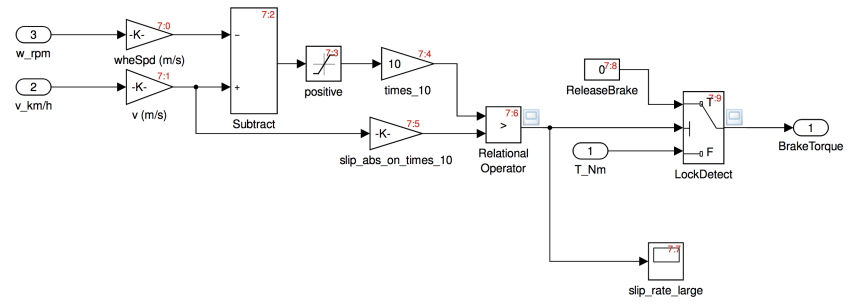
At the system level, the BBW system has a set of 13 functional and 4 timing requirements that need to be verified. In this parer, we present six of them, as follows:

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

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



(a)



(b)

Figure 11.7: The ABS function: (a) The ABS subsystem (b) The "If v>=10km/h" subsystem

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

R4$_{BBW}$ If the brake request is 0, then the ABS shall 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 $\leq$ 10 km/h or the slip rate $\leq$ threshold, then the ABS shall apply the requested torque.

Transformation

As depicted in Fig. 11.6, SIMPPAAL produces a network of 149 automata corresponding to the computational blocks in the Simulink model (e.g., gain, sum, rounding), while the 171 non-computational blocks have been removed during the flattening and transformation phase. For example, the 12 blocks that generate a constant value in the Simulink model are transformed in 12 constants in the UPPAAL SMC model. In the generated formal model, 133

STA are created using the discrete-time pattern, whereas 16 STA are created using the continuous-time pattern. SIMPPAAL also generates 133 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 and the absence of the monitor automata, 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 at the given link[2].

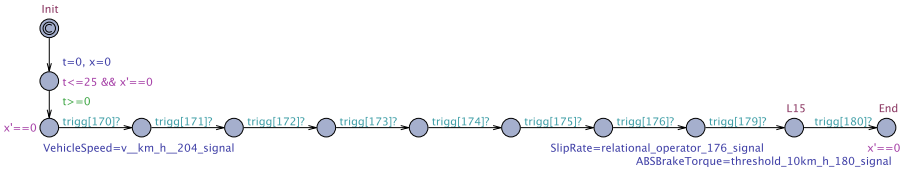


Figure 11.8: The Monitor automaton for requirement $R5_{BBW}$.

Analysis and results

Once the complete formal model is created, we can use the capability of the UPPAAL SMC model checker to display simulation traces that can be compared to the corresponding Simulink simulations for validation purposes. With UPPAAL SMC, we can verify an extensive set of functional and timing requirements. In Table 11.1, we provide the concrete verification results for the six requirements introduced above, which include both hypothesis testing and probability estimation. We perform hypothesis testing, which represents a qualitative property, only for one of the properties, namely $R1_{BBW}$ (first row in Table 11.1). For the given property, we assume a probability threshold of at least 0.999. To make the analysis more efficient, for hypothesis testing we use a pruned model of the BBW system, in which a part of the model relevant for the given property is isolated and analyzed, thus optimizing the analysis time. The pruned model is created based on the control and data dependency between the blocks [29], and consists of five blocks plus the monitor. For the probability estimation (the rest of the properties), we use the original BBW model.

To carry out the analysis, we need to extend the network of STA with an additional automata that monitors the execution of the system for a particular requirement. For instance, for requirement $R5_{BBW}$, we implement

[2]https://github.com/predragf/simpppaal-executables

the monitor presented in Fig. 11.8, which follows the execution of the ABS subsystem. For this, we use a set of synchronization channels named $trigg[sn]$, where sn represents the block's execution number. 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. 11.8, we use an additional clock x, that monitors the ABS subsystem's execution in order to compute timing results. This mechanism is employed to analyze the three timing requirements namely $R1_{BBW}$, $R2_{BBW}$ and $R3_{BBW}$ in Table 11.1, where the monitors follow the execution of the entire system, from the pedal sensor to the wheel actuator.

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 opt 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 an HP Z620 Workstation with Intel Xeon Processor 3.0 Ghz, 8 cores, and 64 GB DDR3.

11.5.2 The Adjustable Speed Limiter Use Case

The Adjustable Speed Limiter (ASL) is an operational servomechanism integrated into modern Volvo trucks produced by Volvo Group Trucks Technology (VGTT). The intended functionality of the system is to limit vehicle speed such that it does not exceed a predefined upper-bound value specified by the driver. It is classified as an ASIL-A safety-critical system according to the ISO 26262 standard [1]. The system realizes more than 300 system-level requirements covering the following aspects: i) the vehicle variability, mode of operations, the vehicle and engine speed, etc., ii) the road condition, including the flat and downhill motions of the vehicle intended to maintain a smooth driving experience, iii) the driver requests, coming from the HMI system (see Fig. 11.9) as well as from the accelerator pedal, which override the system functionalities, and iv) various road speed limits, including legal speed limit, temporary road speed limit, tracking road speed limit, etc. Fig. 11.9 shows the HMI system, used by the driver to operate the ASL system. Given its size and complexity, and especially the fact that it represents an operational system, we use ASL to gain further insights about the scalability of our approach and the SIMPPAAL tool.

The ASL model is constructed from 10 Simulink referenced models. Such design provides numerous benefits, including modular development, accelerated simulation, incremental code generation and so forth. Due to the intricacies in the referencing mechanism, the referenced models have been loaded as subsystems prior to the transformation. Executing the MATLAB command *sldiagnostics* on the model shows that 4817 communicating

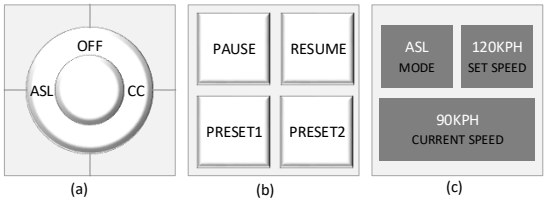


Figure 11.9: The HMI System of ASL - (a) a free-wheel device for vehicle speed limit mode selection; (b) a rocker device for ASL control options; (c) a display device for showing vehicle speed limit information to the driver)

Block Type	Count	Block Type	Count	Block Type	Count	Block Type	Count
Constant	1056	Terminator	38	SaturationDynamic	16	Rounding	4
Inport	870	DataStoreMemory	33	ModelReference	12	TruthTable	4
Relational Operator	707	DataStoreWrite	33	MinMax	9	RateTransition	3
Outport	461	Product	32	Goto	9	IntervalTest	3
Switch	450	Sum	27	Integrator	7	BusCreator	3
Logic	381	DataTypeDuplicate	24	Stateflow	6	Saturate	2
SubSystem	244	From	24	UnaryMinus	6	Derivative	1
UnitDelay	79	S-Function	18	Mux	6	Clock	1
InportShadow	71	Demux	17	SampleTimeMath	5	EnablePort	1
Gain	55	CompareToConstant	17	RateLimiterDynamic	5	Trigonometry	1
DataStoreRead	55	DataTypePropagatio	16	CompareToZero	5		

Figure 11.10: ASL Simulink Block Types

Simulink blocks realize the ASL model, and Fig. 11.10 displays the blocks' types and their distribution in the model. Furthermore, the model is constructed from 244 subsystems, 20 masked blocks, 6 Stateflows and 18 S-functions. All the remaining blocks are atomic with either discrete or continuous behavior. Due to confidentiality reasons, further details about the design of the ASL model are omitted. Overall, compared to BBW, ASL is more than 10 times larger in terms of number of blocks and encompasses a much more complex model-referencing mechanism.

Transformation

We use our SIMPPAAL tool to generate the NSTA model that corresponds to the ASL Simulink model. The formal model that consists of 1548 blocks, modeled as 1024 automata and 524 constants, is produced within 720 seconds. The difference between the number of transformed blocks included in the NSTA model and the number of blocks in the Simulink model accounts for the existence of composite blocks eliminated during the flattening procedure, the non-computational ones eliminated by the transformation engine and the inner contents of the masked blocks, which are not accessible to the parsing engine. In addition to this, we observe that a number of atomic blocks could not be located in the model by the SIMPPAAL model parsing engine, and are reported as non-existing.

The generated NSTA UPPAAL model is not analyzable as such, due to the following reasons: i) missing automata in the model corresponding to the non-existing blocks that cannot be located by the parsing engine, ii) syntax errors in the block routines caused by non-existing signals corresponding to the non-existing blocks, and iii) the fact that the current set of SIMPPAAL plug-ins is primarily developed to cover the block types in the BBW model, meaning that for the block types specific for the ASL model (ex: DataStoreRead, DataStoreWrite, TruthTabel, etc.) block routines could not be automatically generated. While the third issue is of a known technical nature (inherent incompleteness of the plug-in library) and can be addressed by extending the library, the first two reveal a bug within the model parsing engine of SIMPPAAL, which comes from the ConQAT library. To be able to address this issue, we need further investigations to identify the exact cause of such behavior and develop a strategy for improving the tool. Despite the fact that the produced NSTA model is not analyzable, the results of applying SIMPPAAL on the ASL use case are encouraging. The important observation from this case study is the fact that the tool is able to cope with the size and complexity of production-size Simulink models and can generate the corresponding formal model within reasonable time.

11.6 Discussion on the Approach

To summarize the experience of applying our approach, we discuss its characteristics and lessons learned with respect to the transformation of Simulink models into NSTA and their analysis using UPPAAL SMC.

Reusability and Automation

The pattern-based Simulink to NSTA transformation approach described in this paper provides a straightforward and highly automated transformation procedure, with the transformation result being behaviorally 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. Using the `C` routines enables us to extend the application of the patterns and faithfully represent the functional behavior of each block. This means that, regardless of how complex the computational routine of the given block is, it can be transformed using the given patterns. Our approach supports the transformation of the blocks from various Simulink libraries provided by the vendor, 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. The generation of the block routine is performed by a SIMPPAAL plug-in, which is dynamically loaded into the tool. This means that in order to be able to automatically transform all blocks in a

given model, an adequate set of plug-ins has to be developed. The proposed flattening procedure based on a recursive algorithm can, in principle, be applied to flatten any model with arbitrary nested composite blocks. Based on our current experience, the current version of SIMPPAAL, 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.

Our approach and the SIMPPAAL tool have been successfully applied to generate the complete formal model of the Simulink model of the Brake-by-Wire industrial prototype, and to partially transform the Simulink description of the operational Adjustable Speed Limiter system. The transformation results from the case studies show that SIMPPAAL has the potential for application on industrial Simulink models of considerable sizes. The positive experience of applying SIMPPAAL, 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 exhaustive verification techniques are applied on complex industrial 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 simulations, which is a time-consuming process. For instance, to verify $R5_{BBW}$ (see Table 11.1), 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, yet further improvements might be required to ensure the efficiency and scalability needed for industrial adoption.

11.7 Related Work

Simulink includes its own tool for design verification, the so-called *Simulink Design Verifier* (SDV) [30], which uses a combination of static analysis and explicit model checking in order to identify design errors and requirements violations. The numerous limitations of this tool have been discussed by Nellen et al. [34]. The authors mention issues such as inconclusiveness, lack of reliability, inconsistency and irreproducibility of the results, and

difficulties with respect to verifying temporal properties. Due to these, complementary formal verification means are necessary.

The verification of *control* algorithms implemented in Simulink has been formulated as a hybrid-automata *reachability problem*, like in CheckMate [8], 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 the transformation does not scale well to the complexity of real industrial cases, which contain large numbers of very diverse modules: continuous, discrete, StateFlow, etc. The (formal) verification of complex Simulink models is addressed by the following three categories of approaches:

(a) Abstraction of blocks into contracts/theories and their formal analysis. In such approaches, 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. [15] 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. [19] 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. The latest endeavor from this class is the refinement calculus for reactive systems (RCRS) toolset [14], which is a fully automated compositional framework for modeling and reasoning about Simulink models. The toolset consists of two main engines, namely *Translator* that transforms a given Simulink model provided as an .slx file into an RCRS model of the diagram, which is then analyzed via the *Analyzer*. The toolset can be used for static analysis and behavioral type checking and inference for Simulink models. The authors, on the other hand, do not show how this approach can be used for the verification of Simulink models with respect to properties that describe the functional or behavioral characteristics of the system, which SIMPPAAL focuses on.

(b) 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 or by reachability analysis. This strategy has received much attention in the literature and is the one that we have also followed in SIMPPAAL. The approach proposed by Barnat et al. [4] focuses on transforming the Simulink models into the language of the LTL explicit model checker DiViNE. The authors show how the latter can be integrated with the Honeywell formal verification environment. The work provides support only for discrete Simulink blocks, yet they show it suitable for the aeronautics industry. Similarly, the approach by Meenakshi et al.

[31] 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 [32] 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 [26] and Jiang et al. [20], in which the authors propose transformation frameworks from StateFlow/Simulink into timed and hybrid automata, respectively, without considering other types of Simulink blocks. The formal verification of C-code automatically generated from Simulink modules, as presented by Berger et al. [6], is another example of M2M transformation. The approach has been applied in two case studies of the automotive industry with moderate success because of complexity issues. For the requirements for which finding unbounded correctness proofs is not feasible, the authors advocate the use of subsystem verification. Overall, the paper shows that much human intervention is still required for verifying a Simulink model, with a significant percentage of the time dedicated to requirement formalization, as other studies indicate [16].

In general, the solutions available for automated M2M transformation of Simulink are restrictive with respect to the number of supported block types (only discrete blocks or only StateFlow diagrams), and 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. Two notable exceptions are Zuliani et al. [36] and the SL2SX Translator [33]. The former uses Bayesian statistical model checking for analyzing the specification and can scale better to larger-size models. However, it 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. The SL2SX Translator performs a semi-automated transformation from Simulink diagrams into a hybrid automata formalism, *SpaceEx*, preserving the model architecture. It is based on a rather simple definition of a Simulink model, which does not include some of the complex features addressed by SIMPPAAL. It is constrained by the limitations of SpaceEx (piecewise constant and affine dynamics only) but it has been recently complemented with a technique called *syntactic hybridization* [22], which allows analysis of non-linear dynamics.

(c) Generation and abstraction of simulation traces. A third approach, suggested in the Plasma Lab platform [24], consists of generating and collecting simulation traces directly from Simulink, and transforming them, by abstraction, into a state machine representing the system's behavior, which is model checked. Plasma Lab is, in fact, a statistical model checking architecture that can be connected to different simulation engines [23], Simulink being one of them. It accepts properties specified in bounded linear

temporal logic (BLTL) and offers the three basic modes of statistical model checking: simple Monte Carlo, Monte Carlo using Chernoff confidence bound, and sequential hypothesis thesis. The approach has been enhanced with runtime mechanisms for generation of optimal schedules, rare event simulation and change detection, in order to increase the trust on the properties obtained by SMC [23].

Even though the approach is similar to ours, it differs in several aspects: first, the system properties are expressed in linear temporal logic, while in SIMPPAAL we use weighted metric temporal logic. Based on the fact that the two specification formalisms are orthogonal, the approaches complement each other in that regard. Second, due to the absence of an input model, the PLASMa-lab platform relies on external simulation engines to generate execution traces, and as such is strongly coupled to such tools. In SIMPPAAL the model generation and analysis is completely independent from external tools, as only the system model and the sorted order execution are required once, that is, at the time the formal model is generated. Third, Plasma Lab does not require a system model and thus is not hampered by the complexity of M2M transformation. However, for completely discrete-time Simulink models of moderate size and complexity, by using SIMPPAAL one can perform exhaustive model checking thus obtaining either a full guarantee that the property is satisfied, or a counter-example in the opposite case.

With this plethora of approaches for Simulink verification, it seems that the next step will be to investigate how they can be integrated with current industrial practices and safety standards' recommendations. Many of the suggested methods are complementary to each other and exhibit different strengths.

11.8 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 [18]. 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 STA, ii) a definition of a Simulink model as a sequential 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, iv) a tool, called SIMPPAAL, which embodies our approach and automates the complete process of transforming Simulink models into NSTA, and v) an extended validation of SIMPPAAL on two industrial use cases.

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 our tool SIMPPAAL need to be further exercised.

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 lacking in 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 offers 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 [13], in an attempt to enhance the class of systems that can be tackled.

Acknowledgement

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

11.9 References

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

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

[3] Rajeev Alur and P. Madhusudan. Decision problems for timed automata: A survey. In *In Proceedings of SFM'04, Lecture Notes Computer Science 3185, 1-24*, pages 1–24. Springer, 2004.

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

[5] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.

[6] Philipp Berger, Joost-Pieter Katoen, Erika Ábrahám, Md Tawhid Bin Waez, and Thomas Rambow. Verifying auto-generated c code from simulink. In Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik de Vink, editors, *Formal Methods*, pages 312–328, Cham, 2018. Springer International Publishing.

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

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

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

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

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

[12] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[13] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A storm is coming: A modern probabilistic model checker. *arXiv preprint arXiv:1702.04311*, 2017.

[14] Iulia Dragomir, Viorel Preoteasa, and Stavros Tripakis. The refinement calculus of reactive systems toolset. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 201–208. Springer, 2018.

[15] Orlando Ferrante, Luca Benvenuti, Leonardo Mangeruca, Christos Sofronis, and Alberto Ferrari. 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)*, volume 7613 LNCS, pages 409–416, 2012.

[16] P. Filipovikj, M. Nyberg, and G. Rodriguez-Navas. Reassessing the pattern-based approach for formalizing requirements in the automotive domain. In *2014 IEEE 22nd International Requirements Engineering Conference (RE)*, pages 444–450, Aug 2014.

[17] Predrag Filipovikj, Nesredin Mahmud, Raluca Marinescu, Guillermo Rodriguez-Navas, Cristina Seceleanu, Oscar Ljungkrantz, and Henrik Lönn. Analyzing industrial simulink models by statistical model checking. Technical report, March 2017.

[18] Predrag Filipovikj, Nesredin Mahmud, Raluca Marinescu, Cristina Seceleanu, Oscar Ljungkrantz, and Henrik Lönn. *Simulink to UPPAAL Statistical Model Checker: Analyzing Automotive Industrial Systems*, pages 748–756. Springer International Publishing, Cham, 2016.

[19] Ashlie B. Hocking, M. Anthony Aiello, John C. Knight, and Nikos Aréchiga. Proving Critical Properties of Simulink Models. In *Proceedings of IEEE International Symposium on High Assurance Systems Engineering*, volume 2016-March, pages 189–196, 2016.

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

[21] Ioannis T Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *International Symposium on Formal Methods*, pages 268–283. Springer, 2006.

[22] Nikolaos Kekatos, Marcelo Forets, and Goran Frehse. Constructing verification models of nonlinear simulink systems via syntactic hybridization. In *Proceedings of Applied Verification for Continuous and Hybrid Systems*, 2017.

[23] Axel Legay, Sean Sedwards, and Louis-Marie Traonouez. Plasma lab: A modular statistical model checking platform. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*, pages 77–93, Cham, 2016. Springer International Publishing.

[24] Axel Legay and Louis-Marie Traonouez. Statistical model checking of simulink models with plasma lab. In Cyrille Artho and Peter Csaba Ölveczky, editors,

Formal Techniques for Safety-Critical Systems, pages 259–264, Cham, 2016. Springer International Publishing.

[25] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR'10*, pages 348–370. Springer, 2010.

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

[27] Raluca Marinescu, Henrik Kaijser, Marius Mikučionis, Cristina Seceleanu, Henrik Lönn, and Alexandre David. Analyzing industrial architectural models by simulation and model-checking. In *Third International Workshop on Formal Techniques for Safety-Critical Systems*, November 2014.

[28] Raluca Marinescu, Saad Mubeen, and Cristina Seceleanu. Pruning architectural models of automotive embedded systems via dependency analysis. In *42nd Euromicro Conference series on Software Engineering and Advanced Applications*, September 2016.

[29] Raluca Marinescu, Saad Mubeen, and Cristina Seceleanu. Pruning architectural models of automotive embedded systems via dependency analysis. In *Software Engineering and Advanced Applications (SEAA), 2016 42th Euromicro Conference on*, pages 293–302. IEEE, 2016.

[30] Mathworks. Mathworks: Simulink design verifier - user's guide, 2018. [Online; accessed 24-October-2018].

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

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

[33] Stefano Minopoli and Goran Frehse. Sl2sx translator: From simulink to spaceex models. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*, HSCC '16, pages 93–98, New York, NY, USA, 2016. ACM.

[34] Johanna Nellen, Thomas Rambow, Md Tawhid Bin Waez, Erika Ábrahám, and Joost-Pieter Katoen. Formal verification of automotive simulink controller models: Empirical technical challenges, evaluation and recommendations. In Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik de Vink, editors, *Formal Methods*, pages 382–398, Cham, 2018. Springer International Publishing.

[35] Inc. The MathWorks. *Simulink Reference, Matlab&Simulink*. The MathWorks, Inc., 3 Apple Hill Drive Natick, MA 01760-2098, R2017a edition, March 2017.

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

Req.	Query	Result	Runs	Time
$R1_{BBW}$	$Pr[<= 35]([]\ Monitor.End\ imply\ Monitor.x <= 10) >= 0.999$	Pr([] ...) >= 0.9991 with confidence 0.9995	37965	17991.783s
$R1_{BBW}$	$Pr[<= 35](<>\ Monitor.End)$	Pr $\in [0.990015, 1]$ with confidence 0.99	528	12391.89s
	$Pr[<= 35]([]\ Monitor.x <= 10)$	Pr $\in [0.990015, 1]$ with confidence 0.99	528	62320.3s
$R2_{BBW}$	$Pr[<= 75](<>\ Monitor.End)$	Pr $\in [0.9900061, 1]$ with confidence 0.991	538	13751.28s
	$Pr[<= 75]([]\ Monitor.x <= 50)$	Pr $\in [0.9900061, 1]$ with confidence 0.991	538	125579.68s
$R3_{BBW}$	$Pr[<= 75](<>\ Monitor.End)$	Pr $\in [0.980056, 1]$ with confidence 0.99	263	6362.99s
	$Pr[<= 75]([]\ Monitor.y <= 4)$	Pr $\in [0.980056, 1]$ with confidence 0.99	263	62372.65s
$R4_{BBW}$	$Pr[<= 75](<>\ Monitor.End\ and\ RequestedTorque == 0)$	Pr $\in [0.851567, 0.951396]$ with confidence 0.95	79	3214s
	$Pr[<= 75](<>\ (Monitor.End\ and\ RequestedTorque == 0)\ imply\ ABSBrakeTorque == 0)$	Pr $\in [0.998, 1]$ with confidence 0.999	3797	290362.8s
$R5_{BBW}$	$Pr[<= 75](<>\ Monitor.End\ and\ VehicleSpeed > 10\ and\ SlipRate > 0)$	Pr $\in [0.893009, 0.992099]$ with confidence 0.95	79	2588s
	$Pr[<= 75](<>\ (Monitor.End\ and\ VehicleSpeed > 10\ and\ SlipRate > 0)\ imply\ ABSBrakeTorque == 0)$	Pr $\in [0.995005, 1]$ with confidence 0.99	1058	23892.4s
$R6_{BBW}$	$Pr[<= 75](<>\ Monitor.End\ and\ (VehicleSpeed <= 10\ or\ SlipRate <= 0))$	Pr $\in [0.00790082, 0.106991]$ with confidence 0.95	79	18894.6s
	$Pr[<= 75](<>\ (Monitor.End\ and\ (VehicleSpeed <= 10\ or\ SlipRate <= 0))\ imply\ ABSBrakeTorque == ReqTorque)$	Pr $\in [0.980029, 1]$ with confidence 0.995	297	7494.04s

Table 11.1: Overall Results of Statistical Model Checking.