Analyzing Industrial Simulink Models by Statistical Model Checking

Predrag Filipovikj[1], Nesredin Mahmud[1], Raluca Marinescu[1], Guillermo Rodriguez-Navas[1], Cristina Seceleanu[1], Oscar Ljungkrantz[2], and Henrik Lönn[2]

[1] Mälardalen University, Västerås, Sweden,
{first.last}@mdh.se
[2] Volvo Group Trucks Technology, Gothenburg, Sweden,
{oscar.ljungkrantz, henrik.lonn}@volvo.com

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

1 Introduction

Designing modern automotive systems is as rewarding as it is challenging. Trends like the *drive-by-wire* technology, in which standard vehicle operations such as braking are carried out by electronic components rather than mechanical ones, make the assurance of a modern vehicle's correct operation extremely difficult.

The cost of drive-by-wire systems is often greater than for conventional systems. The extra costs stem from higher complexity, development costs and the redundant elements needed to make the systems safe. Such systems fall under the category of *safety-critical* systems that are recommended to be developed under standards like ISO26262 [1], which require extensive changes in the traditional development process due to the need of providing assurance of the system's safe operation at all development stages.

To achieve some form of assurance with respect to safety-critical requirements, as well as valuable design insight, *model-based design* enables industry to create executable specifications in the form of Simulink [2] models that can be

simulated and formally analyzed [3] to detect hidden design errors and requirement violations.

Analyzing Simulink models formally has been a research target for a while now. Existing work [4–6] provides solutions based on (stochastic) hybrid automata, extended finite automata etc., yet no integrated framework exists, which could serve as an immediate tool applicable to complex industrial Simulink models. To address this gap, in this paper, we introduce a pattern-based approach (Section 3) that captures formally the behaviors of Simulink blocks, as networks of stochastic timed automata, and report our experience with analyzing an industrial system, with `UPPAAL SMC` (Statistical Model Checker) [7] (Section 5). Our use case, that is, the *Brake-by-Wire* (BBW) prototype comes from Volvo Group Trucks Technology (VGTT), Sweden, a well-known truck manufacturer.

We classify the Simulink blocks into *atomic* (do not contain sub-blocks) and *composite* (hierarchical) blocks, and we separate further 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 the atomic block. The definition captures the functionality of a Simulink block as a `blockRoutine()` that updates data (state) variables, after which it produces an output observable at particular time instances defined as a multiple of the block's sample time (in case of discrete blocks), or continually observable in case of a continuous block. Next, we define the semantics of the Simulink blocks in terms of timed transition systems, and we sketch the proof of the soundness of the transformation into particular stochastic timed automata, by showing that the Simulink atomic block refines our proposed stochastic patterns, in the discrete-time case. If the model contains continuous-time blocks then the soundness resorts to comparing simulation traces generated by simulating the model in Simulink and `UPPAAL SMC`, respectively. A crucial aspect of the Simulink to stochastic timed automata transformation is ensuring and preserving the correct execution order of the Simulink blocks, both at the system and subsystem level. We do this by introducing a *flattening algorithm* of the hierarchical Simulink models that removes the hierarchy by capturing only the execution order of the blocks as computed by Simulink at the beginning of the simulation in the so-called sorted list.

The crux of our method is twofold: (i) using patterns in the transformation, which eases the modeling process while preserving the execution semantics of Simulink blocks, and (ii) verifying the encodings of the Simulink blocks behaviors as C routines in `UPPAAL`, with the program verifier `Dafny` [8]. To be able to apply our approach on the selected Brake-by-Wire industrial use case, we also provide the tool called `SIMPPAAL` that takes as input the Simulink model together with the sorted list and generates automatically the flattened formal model to be statistically model checked.

Our endeavor is justified by the industrial needs of ensuring correctness with respect to both functional and timing behaviors of automotive embedded systems. Moreover, an initial investigation of verifying large Simulink models with the existing Simulink Design Verifier tool shows limitations in terms of scalabil-

ity and coverage of all types of requirements. The application of our approach to the Brake-by-Wire Simulink model does not yet confirm the scalability of our approach, but it shows its feasibility. We show that we can automatically generate the network of stochastic timed automata corresponding to the Brake-by-Wire Simulink blocks and their sorted order of execution, and analyze via statistical model checking the complete transformed model, against probabilistic functional and timing requirements, with high accuracy. Applying exhaustive model checking on the 4-wheels architectural model of the Brake-by-Wire, integrated with formal semantics in terms of timed automata generates a very large state-space, unless reduction techniques are applied [9, 10]. It is then foreseeable that applying exact model checking on more complex industrial models would most likely run into the state-space explosion problem. This motivates our choice of SMC as the analysis solution for Simulink models, despite the fact that the method is not exact.

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

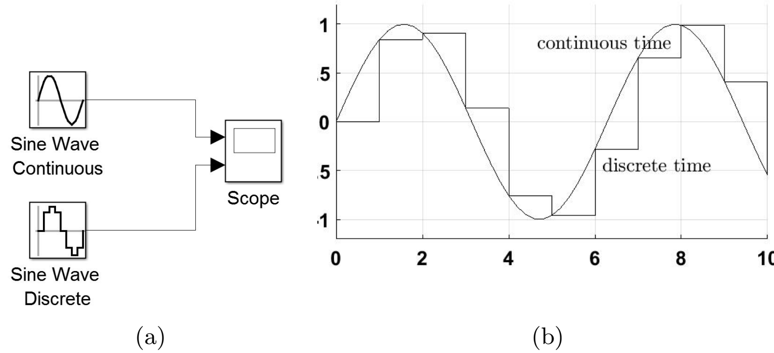
2 Preliminaries

In this section, we recall the basics of the notations, languages and tools needed to comprehend the rest of the paper. We overview Simulink, stochastic (priced) timed automata, `UPPAAL SMC`, and `Dafny`.

2.1 Simulink

Simulink [2] is a graphical programming environment for model-base design, simulation, verification, and code generation of multi-domain dynamic systems. The model-based design is achieved based on predefined libraries of atomic blocks, e.g., *Sum*, *Product*, *Gain*, *Sine* in the *Math Operations* library, *Logical Operator*, *Relational Operator* in the *Logic and Bit Operations*. Such blocks represent computational modules that produce an output based on a equation or another modeling concept either continuously (i.e., continuous-time blocks) or at specific points in time (i.e., discrete-time blocks). Fig. 1 shows a visual representation of the continuous-time and discrete-time behaviors of the *Sine* wave Simulink block. The tool also enables the definition of custom blocks modeled as Stateflow diagrams or user-defined functions via the concept of *S-function* written in Matlab, C, C++, or Fortran, and *Block Masks* modeled in Simulink with a userdefined interface, encapsulated logic, and hidden data from the user.

A hierarchical diagram is achieved through the implementation of a *Subsystem*, a block that encapsulates a set of atomic blocks and possibly other



(a) (b)

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

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

For example, in Fig. 2 we show a small Simulink model. This is an excerpt from the BBW model that is used as a use case in this paper. The first block in the model is a masked block called *MaskedInput* that produces the signal presented in Fig. 2a. Next, this signal is limited to a maximum value of 100, that is, if the input signal to *Saturation* is grater than 100, then the output will be set to 100, otherwise will remain unchanged. This signal is then rounded to the closed integer by the *Rounding* block and represents the output signal of the system witch is displayed by Fig. 2b. Between the *Saturation* and the *Rounding* block, there is the *RateTransition* block, which does not alter the value of the signal, but handles the data transfer between the two blocks.

In Simulink, the dynamic models can be simulated and the results can be displayed as simulation runs. The order in which the blocks are invoked is determined during simulation by the model compiler. The system's block invocation ordering is called the *sorted order*. Some blocks, such as *Mux*, *Demux*, *Goto*, *From*, are related to the signal flow and have no computational value, thus they are not part of the sorted order. Similarly, virtual subsystems do not execute as a unit and are also not part of the sorted order, but the blocks inside the virtual subsystem are part of the root-level system's sorted order and share the system's index.

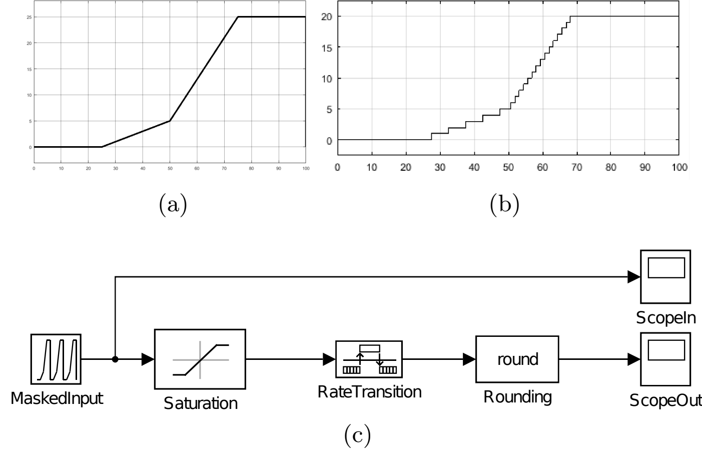


Fig. 2: Example (Simulink model): (a) Simulink input signal (Scope1), (b) Simulink output signal (Scope2), and (c) The Simulink model.

2.2 UPPAAL SMC

`UPPAAL SMC` [11] is a statistical model checker for system models represented as networks of stochastic priced timed automata. A stochastic priced timed automaton (SPTA) is defined as the following tuple:

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

where L is a finite set of locations, $l_0 \in L$ is the initial location, X is a finite set of continuous variables, $\Sigma = \Sigma_i \uplus \Sigma_o$ is a finite set of actions partitioned into inputs (Σ_i) and outputs (Σ_0), E is a finite set of edges of the form (l, g, a, φ, l'), where l and l' are locations, g is a predicate on $\mathbb{R}^X$, action label $a \in \Sigma$, and φ is a binary relation on $\mathbb{R}^X$, $R : L \to \mathbb{N}^X$ 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 delays and discrete transitions from (l, v) to (l', v'). The delay density function μ_s over delays in $\mathbb{R}_{\geq 0}$ for each state a is either uniform or exponential distribution depending on the invariant of l of the state s. Let E_l

denote the disjunction of guards such that $(l, g, o, -, -) \in E$ for some output o. With $D(l, v) = \sup\{d \in \mathbb{R}_{\geq 0} : v + d \models I(l)\}$ we denote the supremum delay, whereas with $d(l, v) = \inf\{d \in \mathbb{R}_{\geq 0} : v + d \models E_l\}$ we denote 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 the uniform distribution over the set $\{o : (l, g, o, -, -) \in E \wedge v \models g\}$ whenever the set is non empty.

Under the assumption of input-enabledness, disjointedness of clock sets and output actions, a collection of composable SPTA can be defined as a network of SPTA (NSPTA) $(A1 \parallel A2 \parallel ... \parallel An)$. 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 (that is, based on a given delay density function and output probability function) how much to delay before outputting and what output to broadcast at that moment.

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

`UPPAAL SMC` uses a probabilistic extension of WMTL [12] 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 $(Pr(\Diamond_{x \leq C} \phi) \geq p)$,
- *Probability evaluation*: calculate the probability $Pr(\Diamond_{x \leq C} \phi)$ for some NSPTA,
- *Probability comparison*: is $P(\Diamond_{x \leq C} \phi_1) > P(\Diamond_{y \leq D} \phi_2)$?

2.3 Dafny

`Dafny` [8] 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 [13] 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` [14], which ensures that the correctness of the `Boogie` program implies the correctness of the `Dafny` program

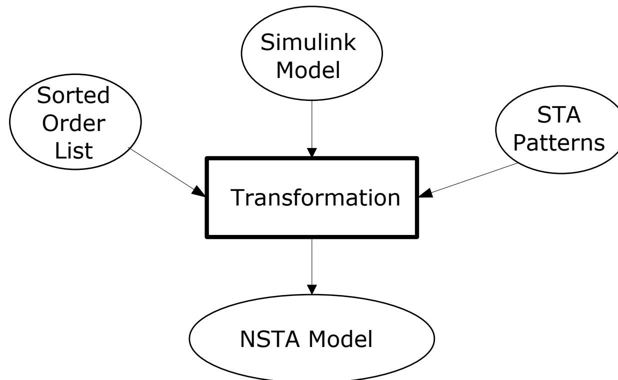


Fig. 3: Overall Approach of the Simulink to UPPAAL SMC Transformation

(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 [15].

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.

3 Simulink to UPPAAL SMC: Approach

In our approach, the transformation from a Simulink model to a Network of Stochastic Automata (NSTA) handles a wide range of Simulink block types, and these blocks include virtual and non-virtual blocks, atomic and composite blocks, continuous and discrete blocks and special blocks such as S-function, Custom and Masked blocks. The wide consideration of the Simulink blocks in our transformation is motivated by a study conducted on the block of types used in modeling various industrial Simulink models.

During the transformation, we also preserve the Simulink model's execution semantics on the `UPPAAL SMC` model checker, that includes the behavior of the blocks and their execution order. Figure 3 illustrates the high level process of the transformation (the oval shapes represent artifacts and the rectangular shape represents functionality). The *Transformation* block implements functionalities for parsing, transformation and generation of artifacts. The *Simulink model* represents the model to be transformed, the *Sorted Order List* contains the sorted order of the blocks inside the Simulink model, while the Stochastic Timed Automata (STA) patterns are the patterns used for transforming the atomic Simulink blocks into NSTA. In order to guarantee correct functionality of the Simulink blocks, we also verify the functional correctness of the blocks

using `Dafny`. The transformation of Simulink models into a NSTA is performed in following steps:

(i) first, we categorize the Simulink blocks into atomic and composite. The atomic blocks are primitive types e.g., Gain, Integrator, etc., whereas the composite blocks such as the Subsystem and Reference Model blocks consist of communicating atomic blocks, hence allowing hierarchical modeling. Further, we classify the atomic blocks by the blocks' execution behavior into continuous-time and discrete-time. The continuous-time blocks execute at infinitely small time intervals that approximate continuous behavior, e.g., Integrator; whereas the discrete-time blocks execute with sample time, t_s;
(ii) second, we give a formal definition for the atomic Simulink blocks as a tuple. The semantics of the composite blocks is interpreted according to its atomic blocks constituents and the connections among the atomic blocks (described in Subsection 3.1);
(iii) third, we propose a flattening algorithm that transforms the composite blocks into their equivalent flat NSTA. Our implementation preserves the atomic execution behaviors of the Subsystem blocks (described in Subsection 3.3);
(iv) fourth, we transform the continuous-time and discrete-time atomic blocks into their respective STA using *Transformation Patterns* (described in Subsection 3.2);
(v) finally, we model the connections among blocks using globally shared memory.

The user defined blocks via the extension mechanisms of Simulink such as S-function, Masked and Custom blocks are also handled in our transformation approach with a special treatment. Since S-function blocks are implemented on the basis of user defined functions, and the Custom and the Masked blocks conceal the content of their blocks, we only transform the blocks' execution behavior into their corresponding STA. Their respective routines that map input and state variables into output are manually encoded on, or migrated to the *blockRoutine* functions, hence treating the special blocks as atomic blocks.

In the following subsections, we give formal definitions of atomic Simulink block and Simulink model, followed by detail descriptions of the transformation patterns and the flattening algorithm.

3.1 Formal definitions

The Simulink documentation [16] provides informal descriptions of Simulink blocks, e.g., short descriptions and usages of blocks, detailed blocks' parameters, datatypes support and runtime characteristics of blocks, e.g., sample time, zero-crossing, etc. For the purpose of clarity in the usage of terminologies in our context, we provide formal definitions of a *Simulink block* and a *Simulink model* respectively. The formal definitions are later employed to reason about the soundness of our transformation, by establishing connections between formal syntactic definitions and their corresponding semantics.

Simulink blocks are the main elements used to build models in MATLAB Simulink. Since non-computational blocks, such as MUX, virtual SubSystem blocks, etc., are not involved in the execution, the formal definition only addresses computational blocks, e.g., Gain, Integrator, etc.

Definition 1 (Simulink block). *An atomic Simulink block, denoted by B, is defined as the following tuple:*

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

where:

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

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

Definition 2 (Simulink model). *A Simulink model is formally defined as a sequential composition of n Simulink blocks, as follows:*

$$S = B_1 \otimes B_2 \otimes B_3 \cdots \otimes B_n \quad (3)$$

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

Semantics of Simulink blocks. Let us rewrite $\Delta = n * t_s + \mathit{offset}$ of Definition 1, as an integral multiple of Simulink's simulation step $\delta \in \mathbb{Q}_{\geq 0}$, that is, $\Delta = n*(m*\delta)+(r*\delta)$, $n, m, r \in \mathbb{N}$. Let us also assume that $x \in V_{in}$, $u \in V_D$, and $y \in V_{out}$ are input, data, and output variables, respectively. Then, we define the semantics of a Simulink block in terms of the following discrete-time transition system.

Definition 3 (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:*

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

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

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

The first transition is the *Init*-type transition, fired at the beginning of the block's execution, at t_0, and the second is the *Operation*-type corresponding to generating outputs for given inputs, at particular time points, $t' = t + m*\delta$. Note that state q' can be the same as q if the input does not change between two sample times. If the Simulink block is continuous, then $m = 1$, $r = 1$, meaning that transitions are fired "infinitely" often, that is, every δ. Note that Definition 3 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.

3.2 STA Patterns

In order to facilitate the transformation of atomic Simulink blocks into their equivalent STA, we propose transformation patterns for the continuous-time

and discrete-time blocks. The transformation patterns are reusable and conform to the semantics of the tuple discussed in Equation 3. Figure 4 shows our transformation patterns encoded in the input language of `UPPAAL SMC`.

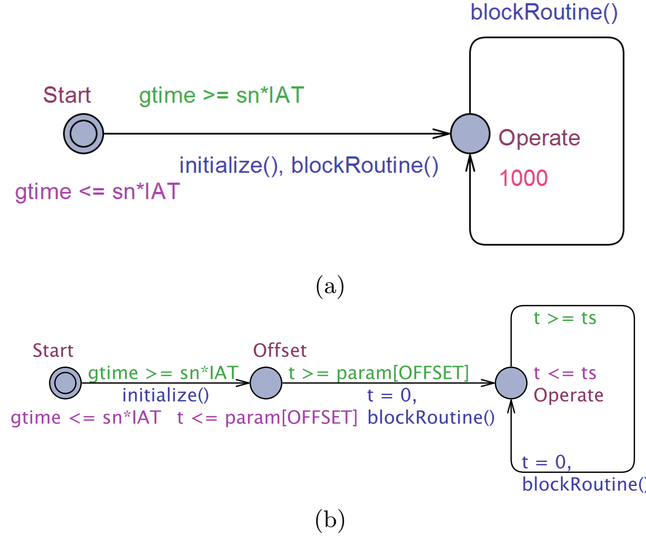


Fig. 4: STA Transformation Patterns: (a) Continuous-time and (b) Discrete-time Blocks

The patterns presented in Figures 4a and 4b are similar to a great extent. The major difference is the absence of the `Offset` location from the contentious-time pattern, as the continuous-time blocks are not allowed to delay their execution. Additionally, the patterns differ in the way they implement the execution mechanism. The execution of the discrete-time pattern proceeds according to the uniform distribution for time-bounded delays modeled via the invariant, whereas the continuous-time patterns execute according to an exponential distribution for unbounded delays.

The elements of the STA patterns are explained as follows:

(i) Locations {`Start, Offset, Operate`} - In the `Start` location, the automaton waits for the release according to the order of execution given in the sorted order list. In the `Offset` location of Figure 4b, the automaton waits until the offset time expires, if there is any. Then, it moves to the `Operate` location, where the automaton executes for fixed simulation time units. The invariants, `I(Start), I(Offset), I(Operate)` are boolean conditions that are defined over the clock variables $\mathbb{R}^{\{gtime,t\}}$ used to limit the time that the automaton is allowed to stay in each location, respectively. In case of continuous-time STA, the exponential rate λ at location `Operate` determines

the probability of the automaton to remain in that location at each simulation step, according to an exponential distribution.

(ii) Edges {`(Start, Offset)`, `(Start, Operate)` } - The edge from the `Start` to `Offset` (in case of the discrete-time STA) is enabled when the guard condition for the release of the block, `gtime` $\geq$ `sn*IAT`, is satisfied, where: `gtime` is the the global clock, `sn*IAT` is the global time when the block is scheduled to be released, defined using the block's sorted order number `sn` and the constant inter-arrival time (IAT) between two consecutive releases of automata `IAT`. On the same edge, when the edge is enabled, the *initialization* function is executed, hence initializing the data variables, including the parameters of the corresponding block. During the initialization function, all the internal state variables are initialized to the corresponding configuration values, and the local clocks are reset.

(iii) The edge `(Offset, Operate)` - The edge from `Offset` to `Operate` is traversed when the offset time expires `t`$\geq$ `para[OFFSET]`, where: `t` is a local clock, and para[OFFSET] is the offset value stored at the index of `OFFSET` in the list of parameters `para`.

(iv) The edge `(Operate, Operate)` - In the case of continuous-time STA, the self-loop in the location `Operate` fires with probability $Pr(\text{leaving after } t) = 1 - e^{-\lambda t}$ according to the exponential distribution. The exponential rate λ is a user-configurable variable that determines how early the automaton leaves the location; the higher the value of λ, the earlier the automaton leaves the location. In case of discrete-time STA, the automaton leaves the location when the guard condition `t`$\leq$`ts` is true. In both cases, the block's routines `blockRoutine()` are executed. The blocks' routines are implemented in the C language and verified using the `Dafny` programmer verifier.

3.3 Flattening Algorithm for Preserving the Execution Order

In this section, we introduce the *flattening* procedure that is used for transforming a hierarchical Simulink model into a flat (non-hierarchical) model composed of atomic blocks only.

The flattening of a Simulink model is performed in two steps, as follows: (i) removing the non-virtual composite Simulink blocks from the model and replacing them with a set of atomic Simulink blocks, and (ii) assigning a correct execution order number for the atomic blocks such that the original behavior of the model is preserved. The proposed flattening procedure is implemented using a recursive algorithm, meaning that it can be applied on Simulink models with arbitrary levels of nesting and terminates once all the virtual hierarchical structures are eliminated from the result model.

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 model itself does not contain information about the execution order of the contained blocks. Instead, the execution order of the blocks, called the sorted order list is always computed at the beginning of the simulation. To obtain the execution order of the blocks inside a given model, the `MATLAB` environment has to be set in debug mode by running the **`sldebug`** command with the Simulink model name as input parameter from the `MATLAB` console. Once in the debug mode, the sorted order list is generated by executing the **`slist`** command. The command generates a list of tuples, where each tuple corresponds to a Simulink block, be it composite or atomic. A tuple in the list contains information for a single block, including: execution order number, block unique identifier and block type. There is also additional information not relevant for the flattening procedure. The sorted order list also contains information about the hierarchical structure of the Simulink model for the non-virtual blocks, as the virtual ones are automatically flattened. Given the structure and the information contained inside the sorted order list, it is better to apply the flattening procedure on the list directly. In the rest of the section, we give an overview of the structure of the sorted order list and the algorithm that is applied for its flattening.

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
```

The sorted order list represents the hierarchical structure of the Simulink model as a collection of *contexts*. There are two types of contexts in a Simulink model: the root context, which is the model itself, and a number of local contexts representing the inner contents of the non-virtual subsystem blocks inside the model. For the virtual and atomic subsystem blocks, no contexts are created, as such blocks are flattened by Simulink automatically, at the time when the "slist" command is executed. Each local context creates a *"nested level"*. Simulink supports infinite levels of hierarchy of subsystem blocks (of any type). Given such structure, the procedure for "flattening" a model is reduced to a procedure of assigning global execution numbers to all atomic Simulink blocks within the atomic subsystems. A global execution order number is an execution order number relative to the root context, that is the root Simulink model file itself. By doing that, we perform an implicit flattening of the Simulink model,

as the correct order of execution of the atomic Simulink blocks in the model for achieving the model's behavior is known. After the flattening procedure is complete, all the local contexts can be safely replaced with the respective set of atomic blocks. The pseudo code of the algorithm which is used to assign global execution order for atomic Simulink blocks nested arbitrary deep inside a given Simulink model is given in Algorithm 1. The algorithm produces new sorted order list, 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.

The sorted order list is available as output in the MATLAB console. In order to automatically flatten the list, the output from the MATLAB console is saved as a textual file which is then given as input to the SIMPPAAL tool (see Section 4). The flattening procedure is fully automated, meaning no user interaction is required.

3.4 Block Routine Verification using Dafny

As already presented in Section 3, the function that maps inputs and state variables into outputs for Simulink blocks is encoded as a C function called *blockRoutine*. To verify the correctness of these functions, we use Dafny [8], a language and program verifier.

To prove the correctness of the input-output mapping functions we use pre-/post-condition verification. We use a set of pre-conditions to describe the state of 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. In the following, we give an overview of blockRoutine verification by applying Dafny on an illustrative example.

Listing 1.1: Example of Dafny verified Simulink block routine

```
/* saturate block routine */
method saturate(input: real, lowerBound:real, upperBound:
    real) returns (output: real)
  requires (lowerBound <= upperBound)
  ensures (lowerBound <= output && output <= upperBound)
{
  output := input;
  if(output <= lowerBound){
    output := lowerBound;
  }
  if(output >= upperBound){
    output := upperBound;
  }
}
```

Listing 1.1 shows the `Dafny` implementation of the basic functionality of the `Saturation` Simulink block. Saturation is an atomic Simulink block that is used to constrain the range of an input signal between a lower and an upper limit [3]. The illustrated saturation function is given as a ternary function that takes as input the incoming signal, and the lower and the upper limits for the output signal. The saturation is executed only if the lower limit is lower or equal to the upper allowed limit for the signal, which is captured by the precondition: `requires (lowerBound <= upperBound)`. Under this precondition, the implementation of the saturation function ensures that the postcondition `ensures (lowerBound <= output && output <= upperBound)` is maintained for all possible executions. In other words, the given implementation of the saturation guarantees that the output of the saturation function will always be within the allowed range for all input values. By adjusting the parameters of the Saturation block, one can significantly influence the behavior of the saturation routine. For instance, if the designer specifies that the output signal of the saturation function should be an integer value, then the saturation function is combined with one of the rounding functions (ceiling, floor, rounding, etc.). In such cases, the postcondition for the saturation function has to be modified accordingly.

In our previous work [17], we have used `Dafny` to verify the block routines at a pattern level. This means that for each block type it is required to generate the most general block routine that captures all the possible behaviors. Then a set of parameters is used to control the flow and decide which of the functionalities of the given routines should be executed for each instance of that routine. Such functions are big and introduce unnecessary complexity for blocks that have simple computational routines, as discussed above. With the automation of the transformation procedure, it is possible to generate custom block routines for each block instance individually, with the pre- and post-conditions specific to that actual implementation. For illustration, for the Saturation blocks that do not have to perform rounding of the output signal, we generate a simple saturation procedure as the one given in Listing 1.1. Such implementations of the routines are easier to read, understand and debug. Our new approach enables verification of each individual block routine, compared to template-level verification, as presented in previous work [17].

In the next subsection, we sketch the transformation's proof of soundness, assuming our particular case of STA patterns.

3.5 Proof Sketch 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 can show that the set of runs of the resulting NSTA, obtained by using the semantic pattern given in Figure 4b, is *refined*

[3] https://se.mathworks.com/help/simulink/slref/saturation.html

by the set of runs of the Simulink model, under the discrete-time blocks only assumption. We have that a Simulink model A' is a refinement of the NSTA model A if and only if $Runs'_A \subseteq Runs_A$, meaning that if the model A satisfies a safety or inevitability property p, and A' refines A, it then follows that A' also satisfies p.

We show the refinement at the discrete-time block level first and then we explain how the result extends to Simulink models with discrete-time blocks. For this, we use the results on decision problems for timed automata, overviewed by Alur and Madhusudan [18].

The authors show that reachability is decidable for the discrete-time or sampled semantics of timed automata [18], assuming an unknown non-negative rational sample time. If we consider the STA pattern of Figure 4b, we notice that the output probabilities over edges outgoing from locations `Start` and `Offset`, according to the uniform distribution γ, are 1, since there is only one outgoing edge from each location, respectively. Similarly, the delay density function μ gives probability 1 of delaying in location `Operate` for t_s time units, due to the disjointness of the invariant $t \leq t_s$ and the guard $t \geq t_s$. Basically, the automaton of Figure 4b is a deterministic closed timed automaton (since clock constraints are of the form $x \bowtie c$, with $\bowtie \in \{\leq, \geq\}$).

Refinement also equates to the problem of language inclusion between timed automata, which is an undecidable problem in general. An important class of timed automata for which the inclusion problem is decidable involves the notion of *digitization* [18]. A timed language L is said to be *closed under digitization* if discretizing a timed word (a string of symbols tagged with occurrence times) in the language, by approximating the events of the timed word to the closest tick of a discrete clock results in a word that is also in L. It is a proven fact that closed timed automata are closed under digitization. This means that constructing a sampled version of the STA automaton of Figure 4b yields an automaton that is a refinement of the original pattern, since $L_{A_d} \subseteq L_A$, where A is an automaton conforming to our STA pattern, and A_d is its discretized version.

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

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

Note that in any reachable state of A_δ, the values of clocks are integral multiple of δ. A run of A_δ with initial state s_0, over a finite timed trace $\zeta = (t_0, a_0)(t_1, a_1)(t_2, a_2) \ldots (t_n, a_n)$ 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, the STA pattern automaton of Figure 4b, it follows by transitivity of the refinement relation that the Simulink block B refines the STA pattern automaton A, assuming the discrete-time behavior.

This result extends to a Simulink model defined by Definition 2, if the former contains only discrete-time blocks. Given the execution order of each block, only one block at a time can be enabled and executed in the Simulink model, therefore the probability distributions of the network of STA that represents the Simulink model's transformation render transitions with probability one, so the parallel composition of STA is in fact a parallel composition of deterministic timed automata, which due to the enforced execution order is in fact a sequential composition of deterministic timed automata that can be further sampled. Given the result of Theorem 1, it follows that a discrete-time Simulink model described as in Definition 2 refines the network of STA in which it is transformed.

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

4 SIMPPAAL Tool

In this section, we present our tool called `SIMPPAAL` (SIMulink to uPPAAL), which automates the process of transforming Simulink models into networks of STA suitable for analysis using the `UPPAAL SMC` tool. The tool represents an implementation of the transformation approach described in Section 3. In the following subsections, we describe the architecture and the functionalities of the `SIMPPAAL` tool.

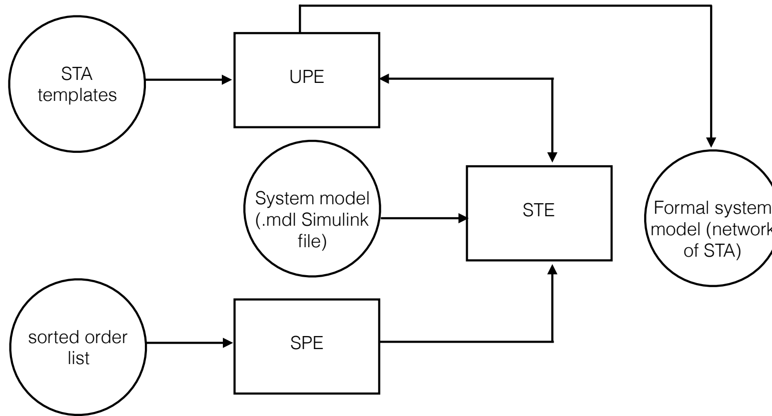


Fig. 5: SIMPPAAL tool architecture.

SIMPPAAL Architecture

The architecture of the `SIMPPAAL` tool is based on a modular design, where the overall functionality is achieved through a set of communicating software modules (called modules for simplicity), via well-defined Application Programming Interfaces (APIs). Each module is designed to perform a specific role in the overall design. The advantages of such design are numerous, including: decreasing implementation complexity by separation of concerns, increasing system maintainability and re-usability of the modules. The architectural design is given in Figure 5, and is based on the following concepts: *artifacts*, which can be input or output represented as circles, and the *modules* represented as squares. In the following, we use the term module and *software engine* interchangeably. The tool is implemented in `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 one can see in Figure 5, the transformation process is based on three different input artifacts: the Simulink model given as an .mdl file, the sorted order list for the execution order given as a textual file, and a `UPPAAL` .xml file that contains the templates for continuous-time and discrete-time blocks. Each of these inputs is handled by a specific module. Each module that is handling a given artifact is responsible for: i) reading and parsing that input in a format such that it can be consumed by other modules, and ii) writing back to that file if required. Given that, the coupling between the inputs and the modules is the following: the STE module is responsible for parsing the Simulink .mdl files, the *UPPAAL Parser Engine (UPE)* handles the reading from and writing to `UPPAAL` specific .xml files, while the sorted order list are handled by the *SList Parser Engine (SPE)*.

The STE module is the core module of the SIMPPAAL tool. Its main responsibility is to transform a Simulink model given as an .mdl file into a network of STA suitable for analysis using UPPAAL SMC. This is by no means a trivial procedure, and for that purpose, the module itself is further decomposed into submodules as follows: a submodule for reading and manipulating Simulink models, and another submodule for transforming Simulink blocks into STA. The role of the first submodule is to read an .mdl 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 [4], 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 .mdl file. To mitigate this problem, we have developed a *"context-switching"* procedure that enables the tool to switch contexts, that is, go from one .mdl file and back, without information loss.

The second STE submodule is responsible for transforming an atomic Simulink block into a corresponding STA, by mapping Simulink parameters into STA specific constructs, such as: sample time, execution order number and the block routine. The submodule also generates Dafny verification expressions for each block routine. The main challenge during this phase is to generate an adequate block routine for the given atomic block. The reason for this is that each Simulink block performs a specific computation routine based on different sets of inputs and different configurations, respectively. Consequently, the transformation submodule must know how to parse each block type in the Simulink library. Moreover, the number of atomic Simulink blocks is not finite as it is possible to introduce new atomic block types via the concepts of S-function and block masking. To cope with this challenge, we have resorted to the *"plug-and-play"* concept, meaning that the generation of the block routine for a specific block type is performed by a *plug-in*, which is dynamically loaded into STE. The given approach facilitates the extension of the STE functionalities to scale with the modeling capabilities of Simulink, with additional advantages such as: i) the effort and complexity of developing a new plug-in is relatively low, and ii) plug-ins are developed independently from the STE, which means that the source code of the STE module does not have to be modified for extending its functionality.

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

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

tool. The UPE can be used as a stand alone library or as part in any other tool for manipulating `UPPAAL` models. The implementation of the module is not bound to the given context, which means it can be reused either as a standalone library for managing `UPPAAL` files or as a part of another system.

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

SIMPPAAL work-flow

The transformation of the Simulink models into a network of STA implemented by the `SIMPPAAL` tool is performed in the following steps:

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

The transformation process of the Simulink models into a network of STA starts by loading and flattening the sorted order list such that the global execution number to each atomic block in the model is assigned (Step 1). As presented in the architecture of the tool, this set of actions is performed by the SPE module. The newly generated sorted order list, which has been flattened and sorted according to the execution order number is composed of atomic Simulink blocks only, and contains all the blocks that participate in producing the overall behavior of the Simulink model.

The flattened sorted order list is then used as a primary input in the STE for transforming the Simulink model into a network of STA. 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 .mdl. This is enabled by our context-switching technique. Once the block has been identified, the

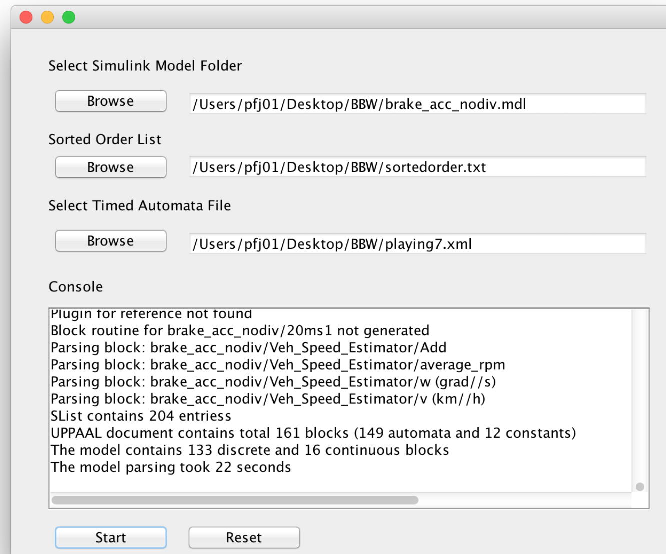


Fig. 6: SIMPPAAL graphical user interface.

STE tries to identify all of its *predecessors*, which is a list of non-virtual atomic Simulink blocks, performed in Step 2.b. In our implementation, the following blocks are considered as virtual: Mux, Demux, Inport and Outport. Additionally, some non-virtual blocks that do not perform computational routines, such as Scope and RateTransition are added to the list of virtual blocks. In other words, a predecessor is an atomic block whose output is consumed by the block that is currently being transformed. In a similar way, the STE identifies the list of *successors*, which is a list of non-virtual atomic Simulink blocks, which uses the output of the given Simulink block as an input for producing an output (this is given as Step 2.c).

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

routine, all the template constructs have been instantiated with block-specific ones. With this, the block transformation is complete and the pattern becomes an instantiation of an STA.

Once the automaton is obtained, in Step 3.b the STE determines the type of the output produced by the block, which can be either scalar or vector of type `Boolean` or `Double`. Even though the type of the output in general is defined for each block type, sometimes it can be determined by other factors, such as: the type and format of the input (ex: a Gain block that has input scalar can produce either scalar, vector or matrix, but if the input is vector it cannot produce scalar). In the current version of the tool, this procedure is implemented in the STE, but to be able to scale with the number of blocks, this responsibility will be transferred to the plug-in for that specific block type.

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

In the current version, `SIMPPAAL` is a standalone tool that can be used via the simple interface, as presented in Figure 6. To create a `UPPAAL` model file, the user has to select the root Simulink model, the sorted order list, and the destination .xml file where the result UPPPAAL 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.

Applying SIMPPAAL on a Toy Example

We apply `SIMPPAAL` on the small Simulink example introduced in Fig. 2. The result of this transformation is a network of 3 STA, all generated based on the continuous-time template introduced in Fig. 4a. For the given blocks, only two block routines could be automatically generated by the tool, that is, the Saturation and Rounding block routines. The MaskedInput is an S-function, for which we need to implement the block routine manually. In Listing 1.2 we present the three block routines.

Listing 1.2: Applying SIMPPAAL: the block routines

```
/* MaskedInput STA blockROutine()*/
void blockRoutine(){
if(25.0>t) {MaskedInput_1_signal=0.0;}
if(t>=25.0 && 50.0>t){MaskedInput_1_signal=t*0.2-5.0;}
if(t>=50.0 && 75.0>t) {MaskedInput_1_signal=t*0.8-35.0;}
if(t>=75.0) {MaskedInput_1_signal=25.0;}
```

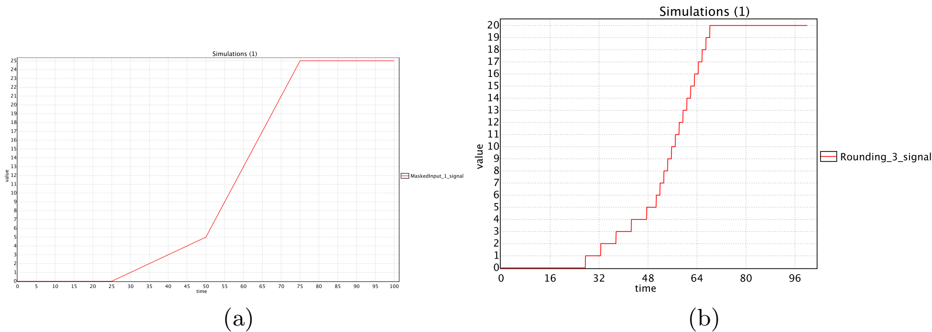
```
}

/* Saturation STA blockROutine()*/
void blockRoutine(){
if(MaskedInput_1_signal >= 20.0) {Saturation_2_signal =
    20.0;}
if(MaskedInput_1_signal <= 0.0) {Saturation_2_signal=0.0;}
if(MaskedInput_1_signal <20.0 && MaskedInput_1_signal >0.0)
    {Saturation_2_signal=MaskedInput_1_signal;}
}

/* Rounding STA blockROutine()*/
void blockRoutine()
{ rounding=true; i=0.0;
  while(rounding)
    { i = i + 1.0;
      if (i>Saturation_2_signal)
      { if(2 == 0) Rounding_3_signal = i-1.0;//floor
        else if(2 == 1) Rounding_3_signal = i;//ceiling
             else if(2 == 2)//round
                  { if(i - Saturation_2_signal >0.5)
                      Rounding_3_signal = i-1.0;//floor
                    else Saturation_2_signal = i;//ceiling
                    rounding=false;
                    }
          }
      }
}
```

We use the UPPAAL SMC model checker to simulate the system and display the values of the outputs for the MaskedInput and Rounding automata. The simulations are presented in Fig.7, and they match the Simulink simulations depicted in Fig.2.

Scope of Application

The modular architecture of the SIMPPAAL tool represents a solid foundation for future improvements, in order to reach a version that can be used to automatically transform industrial Simulink models into a network of STA suitable for analysis using UPPAAL SMC. However, 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 Brake-By-Wire model.

The SIMPPAAL tool cannot properly handle model referencing in cases when a parent model references directly a model instead of a library. This is due to the fact that the structures of the referenced models and the referenced libraries are different. The referenced libraries always start with a subsystem block that has



(a) (b)

Fig. 7: Applying SIMPPAAL: (a) simulating the MaskedInput STA, and (b) simulating the Saturation STA.

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 plan to address this limitation in the subsequent releases of the `SIMPPAAL` tool.

5 Application on Brake-by-Wire Use Case

5.1 The Brake-by-Wire System

System Description. Brake-by-Wire (BBW) is a prototype implementation of a braking system equipped with an ABS function, and without any mechanical connection between the brake pedal and the four brake actuators. Despite being a prototype implementation, the given BBW Simulink model is a faithful representation of a realistic industrial system. The functionality of the BBW is given as follows. The sensor attached to the brake pedal reads its position, which is used to compute the desired global brake torque. At each wheel, the dedicated sensor measures the speed of the wheel, which is used by the ABS algorithm together with the brake torque to compute the actual brake torque that will be applied. The wheel's slip rate s is computed as follows:

$$s = (v - w \times R)/v, \tag{5}$$

where v is the speed of the vehicle, w is the speed of the wheel, and R is the radius of the wheel. The friction coefficient has a nonlinear relationship with the slip rate: when s starts increasing, the friction coefficient also increases, and its value reaches the peak when s is around 0.2. After that, a further increase in s reduces the friction coefficient of the wheel. For this reason, if s is greater than 0.2 the brake actuator is released and no brake is applied, otherwise the requested brake torque is used by the actuator.

At system level, the BBW system has a set of 13 functional and 4 timing requirements that need to be verified. Below, we give a subset of the requirements, described in natural language:

R1$_{BBW}$ The time needed for a brake request to propagate from the brake pedal sensor to the wheel actuator should not exceed 200 ms.
R2$_{BBW}$ The difference between the time needed for a brake request to propagate from the brake pedal sensor to two different wheel actuators should not exceed 20 ms.
R3$_{BBW}$ The value of the brake pedal position shall not exceed its maximal value of 100.
R4$_{BBW}$ If the slip rate exceeds 0.2, then the applied brake torque shall be set to 0.

Transformation. In Simulink, the BBW is a hierarchical Simulink model with four levels of nesting, which contains 320 connected blocks. The result of the transformation is as follows:

- A network of 143 automata corresponding to the computational blocks in the Simulink model (e.g., gain, sum, rounding), since the 177 non-computational blocks (e.g., subsystem, inport, outport, from, goto, rate transition) have been removed during the flattening and transformation phases,
- 133 STA were created using the discrete-time pattern and 10 STA were created using the continuous-time pattern,
- 137 block routines have been generated automatically, with only 6 blocks routines left to be implemented manually (that is, two S-functions and four Sateflow blocks).

The original BBW model contains four (identical) Sateflow blocks, modeled as simple flow charts. We transform the StateFlow blocks by a 1-to-1 mapping to the automata model (i.e., flow chart conditions are mapped to guards, and flow chart actions are mapped to updates).

Analysis results. Once the complete model has been constructed, we validate the correctness of each of the STA by comparing its simulation trace in `UPPAAL SMC` with the simulation trace of the corresponding Simulink block. For instance, Fig. 8 shows the value of the brake pedal position (see requirement $R3_{BBW}$, which can be obtained with the following command:

$$simulate 1[<= 100]\{pedal_map_161_signal\} \tag{6}$$

For the BBW system, we have verified an extensive set of functional and timing requirements. In Table 1 we provide concrete verification results for requirements $R1_{BBW}$, $R2_{BBW}$, $R3_{BBW}$, and $R4_{BBW}$. For some properties, we need to implement another STA that monitors the execution of the system. For instance, for requirement $R1_{BBW}$ we have implemented the monitor presented in Fig. 9. The vertical part of the automaton models the time until the monitor

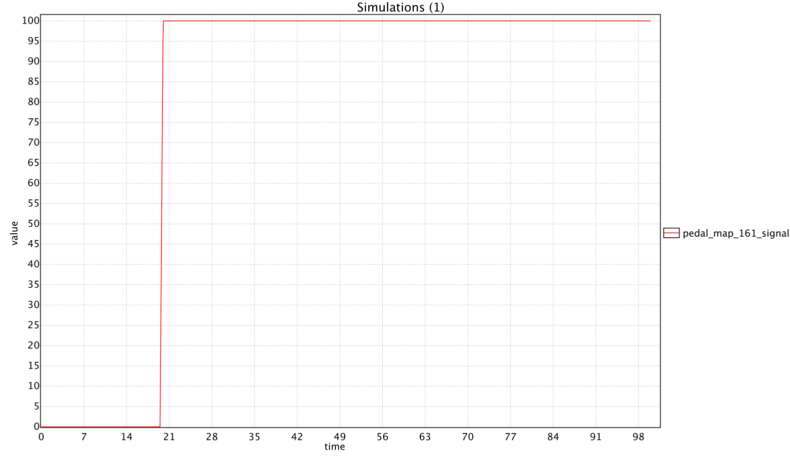


Fig. 8: Simulation of the brake pedal position value.

starts following the execution of the system. To start different simulations at arbitrary moments in time, we introduce a clock that allows for a delay between 0 and n. The vertical part describes the sequence of relevant actions of the monitor (i.e., trigg[160] is a channel that provides broadcast synchronization between the monitor and the block with the sorted order 160, which is the braking pedal modeled as an S-function). In this case, between the pedal and actuator, there are only 8 blocks that propagate the requested torque through the system. Similar monitors have been implemented for analyzing requirements $R2_{BBW}$ and $R4_{BBW}$. No monitor is needed for requirement $R3_{BBW}$.

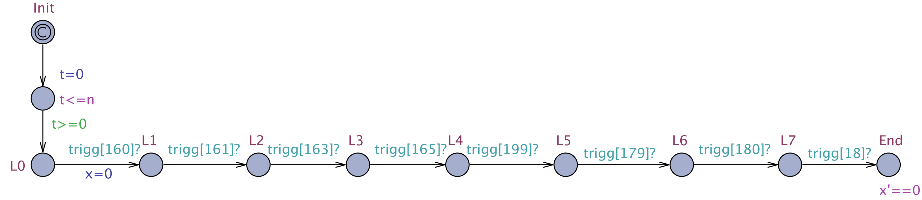


Fig. 9: The Monitor automaton for requirement $R1_{BBW}$.

The UPPAAL SMC statistical model checker can achieve analysis results with different probability interval spans and different confidence levels, depending on the values of the statistical parameters. During the analysis, we have opted for different values of α, the probability of false negatives, and ϵ, the probability uncertainty, to show how these influence the results and the number of runs generated by the model checker. In Fig. 10, we also show the probability density distribution for requirement $R1_{BBW}$.

Table 1: Overall Results of Statistical Model Checking.

Req.	Query	Result	Runs
$R1_{BBW}$	$Pr[Monitor.x <= 200](<> Monitor.End)$	Pr $\in [0.998, 1]$ with confidence 0.999	3797
$R2_{BBW}$	$Pr[Monitor1.x <= 200](<> Monitor1.End \ and \ (Monitor1.x - Monitor2.x <= 20 \ and \ Monitor1.x - Monitor2.x >= -20))$	Pr $\in [0.990014, 1]$ with confidence 0.995	597
$R3_{BBW}$	$Pr[<= 200](<> pedal_map_161_signal <= 100)$	Pr $\in [0.995002, 1]$ with confidence 0.9975	1334
$R4_{BBW}$	$Pr[<= 200](<> Monitor.End \ and \ Monitor.s > 20 \ and \ Monitor.torque == 0)$	Pr $\in [0.902606, 1]$ with confidence 0.95	36

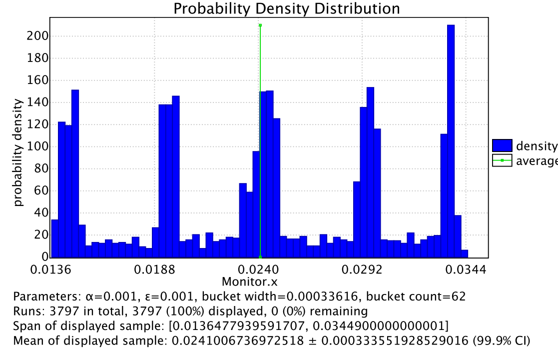


Fig. 10: Probability Density Distribution for requirement $R1_{BBW}$

6 Discussion on the Approach

In this section we discuss the characteristics of the proposed approach for the transformation of Simulink models into networks of STA and their analysis using `UPPAAL SMC`.

The core of the Simulink to NSTA transformation is the pattern-based approach. Such approach provides a straightforward transformation procedure, with the transformation result being faithful to the original model. This is achieved by instantiating patterns for both discrete-time and continuous-time blocks whose functional behavior is encoded as `C` functions. The usage of the `C` routines enables us to extend the application of the patterns and faithfully represent the functional behavior of each block. This means that no matter how complex the computational routine of the given block is, it can be transformed via the given patterns. Our approach supports the transformation of the current blocks in the Simulink library, but also of custom atomic blocks built using the

concepts of S-function and Mask. We verify the functional correctness of each block (its block routine) by using the `Dafny` program verifier.

Our general flattening procedure based on a recursive algorithm can, in principle, be applied to flatten any model with arbitrary nested composite blocks. The procedure however is limited to Simulink models that use libraries as a way of extending the model. For models that utilize model referencing, the procedure does not work, as the atomic blocks from the referenced model are not included in the sorted order list for the root model. Additionally, we have detected an inconsistency between the block identifiers in the sorted order list and the block identifiers in the Simulink model. The inconsistency comes from the fact that Simulink allows new line characters into block identifiers, whereas the new line characters, when exported as sorted order list, are replaced with characters for blank space.

Based on our current experience, the current version of the `SIMPPAAL` tool, as presented in this paper, can be used to generate an NSTA model of the BBW industrial prototype model, which is suitable for analysis using `UPPAAL SMC`, after minor changes. The tool implements functionalities for model flattening, Simulink block retrieval, predecessor and successor identification and transformation of atomic Simulink blocks into STA. As a drawback, the tool is capable of parsing complex Simulink models that are extended via libraries only. This means that the tool is not suitable for a more extensive application, as it does not support manipulation of Simulink models extended via referenced models. The reason for the limitation is the different structure of the referenced models as compared to the referenced libraries, thus requiring different ways of parsing. Additionally, to be able to automatically transform all blocks of a given model, an adequate set of plug-ins has to be developed. This is by no means a limitation of the tool, but it is due to lack of time and resources needed to develop the extensive set of plug-ins.

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

7 Related Work

Simulink is considered as the de-facto standard for designing embedded systems, particularly in the automotive, robotics and automation domains. Formal verification of Simulink models has been studied in a number of publications, yet with diverse objectives. For instance, the verification of control algorithms im-

plemented in Simulink has been formulated as a hybrid-automata *reachability problem*, where the verification objective is proving either that erroneous states are unreachable, or that certain desirable properties hold in every reachable state. An example of this type of verification is the tool CheckMate [19], which transforms Simulink models into a class of hybrid automata known as polyhedral-invariant hybrid automata (PIHA). The main limitation of this method is that it can be applied on a restricted class of models, as reachability is known to be undecidable, in the general case, for hybrid automata. Furthermore, the method does not scale well to the complexity of real industrial cases, which contain a large number of very diverse modules: continuous, discrete, StateFlow, etc.

The verification of more complex Simulink models that has been proposed in the literature follows three different strategies:

1. Generation and abstraction of simulation traces.
2. Abstraction of blocks into contracts/theories and formal analysis.
3. Model to model transformation followed by model checking.

The first strategy exploits the simulation capabilities of Simulink in order to generate and collect simulation traces that are later transformed, by abstraction, into a state machine representing the system's behavior, which can be model checked without difficulty [3]. For instance, PlasmaLab follows this approach and uses SMC for model checking [4]. However, this strategy is limited by the feasibility to generate an exhaustive (although up to a given time) simulation of the system, and thus raises concerns about the completeness of the obtained abstraction. Moreover, since it is based on system traces, it is not adequate for analyzing extra-functional properties, at least not without further changes on the initial model. On the positive side, the approach is generic, applicable to any kind of Simulink diagram, and does not require adding more computation if new blocks are considered.

The second strategy is implemented in two steps. First, the system designer "lifts" the specification of each block using some type of logical language. Second, the whole specification is composed and fed into an analysis engine. Ferrante et al. [20] 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. [21] 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 third strategy tries to reduce user intervention to the minimum. It is based on applying some kind of automated model-to-model (M2M) transformation from Simulink into an automata language that can be verified with model checking. This strategy has received much more attention in the literature. The approach proposed by Barnat et al. [5] focuses on transforming the Simulink models into the language of an LTL explicit model checker called DiViNE. The authors show how this can be integrated with the Honeywell formal verification

environment. They only provide support to discrete blocks, yet they show it suitable for the aeronautics industry. Similarly, the approach by Meenakshi et al. [22] propose a transformation of discrete blocks into NuSMV. In contrast, Agrawal et al. [23] propose a transformation approach of Simulink models into networks of automata, without providing concrete means for formal verification. The work by Miller [24] 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 [25] and Jiang et al. [6], in which the authors propose transformation frameworks from StateFlow/Simulink into timed and hybrid automata, respectively, without considering other types of Simulink blocks.

In general, the solutions available for the automated M2M transformation of Simulink are quite restrictive with respect to the number of block types supported (typically only discrete blocks or only StateFlow diagrams). Also, they have been applied only to academic or middle-size Simulink models, such as the engine control system appearing in the Simulink distribution, which raises concerns about the scalability of the approach. The only exception is the approach by Zuliani et al. [26], which uses Bayesian statistical model checking for analyzing the specification and can scale better to models of larger sizes. Despite that, the approach has been applied to a medium-size Simulink model only, and it seems to have practical limitations such as not accepting multi-file Simulink models.

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

8 Conclusions and Future Work

In this paper, we have extended and improved our already existing pattern-based approach for transforming Simulink models into NSTA semantics [17]. 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 stochastic priced timed automata, ii) a definition of a Simulink model as a serial composition of interconnected Simulink blocks, iii) a soundness proof-sketch for the mapping of the formalized Simulink blocks into the respective stochastic timed automata patterns, for discrete-time models, and iv) a tool, called `SIMPPAAL`, which embodies our approach and is intended for automating the complete process of transforming Simulink models into networks of STA.

The main purpose of the tool is to enable formal analysis of large Simulink industrial models, and keep the formal modeling effort to a minimum, by adding

automation to the transformation. A secondary goal is to make the approach applicable for practitioners, who are not expert in formal methods. Both the scalability and the suitability for engineers of out tool `SIMPPAAL` await validation.

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

Despite the approach being extended and improved as shown above, it is still not ready to be applied on Simulink models of operational systems, as used in industry. The main reason for this is the fact that although stable, the `SIMPPAAL` tool does exhibit technical limitations that are beyond the scope of the approach per se. For example, the current major limitation of the tool is its inability to transform Simulink models that rely on model referencing for loading external libraries. This limitation is of a purely technical nature, and will be addressed in the subsequent updates and releases of the tool. Another technical limitation is the lack of plug-ins for generating all the block routines. For now, 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 Simulink 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 is the 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 [27], in an attempt to enhance the class of systems that can be tackled.

References

1. ISO/DIS 26262-1 - Road vehicles Functional safety Part 1 Glossary. Technical report, Geneva, Switzerland, July 2009.
2. J. B. Dabney and T. L Harman. *Mastering Simulink*. Pearson/Prentice Hall, 2004.
3. B. Boyer, K. Corre, A. Legay, and S. Sedwards. Plasma-lab: A flexible, distributable statistical model checking library. In *QEST*, pages 160–164. Springer, 2013.
4. A. Legay and L.M. Traonouez. Statistical Model Checking of Simulink Models with Plasma Lab. In *FTSCS'15*, pages 259–264. Springer, 2015.
5. 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.
6. 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.
7. Alexandre David, K.G. Larsen, A. Legay, M. Mikučionis, and D.B. Poulsen. Uppaal smc tutorial. *STTT Journal*, 17(4):397–415, 2015.
8. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR'10*, pages 348–370. Springer, 2010.
9. 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.
10. 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.
11. 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.
12. 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.
13. Ioannis T Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *International Symposium on Formal Methods*, pages 268–283. Springer, 2006.
14. 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.
15. 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.
16. Inc. The MathWorks. *Simulink Reference, Matlab&Simulink*. The MathWorks, Inc., 3 Apple Hill Drive Natick, MA 01760-2098, R2017a edition, March 2017.

17. 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.
18. Rajeev Alur and P. Madhusudan. Decision problems for timed automata: A survey. In *In Proceedings of SFM04, Lect. Notes Comput. Sci. 3185, 124*, pages 1–24. Springer, 2004.
19. Alongkrit Chutinan and Bruce H Krogh. Computational techniques for hybrid system verification. *IEEE transactions on automatic control*, 48(1):64–75, 2003.
20. 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.
21. 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.
22. 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.
23. A. Agrawal, G. Simon, and G. Karsai. Semantic Translation of Simulink/Stateflow Models to Hybrid Automata using Graph Transformations. *ENTCS Journal*, 109:43–56, 2004.
24. Steven P. Miller. Bridging the Gap Between Model-Based Development and Model Checking. In *TACAS*, pages 443–453. Springer, 2009.
25. K. Manamcheri Sukumar. Translation of Simulink-Stateflow Models to Hybrid Automata. 2011.
26. 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.
27. 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.