Simulink to UPPAAL Statistical Model Checker: Analyzing Automotive Industrial Systems

Predrag Filipovikj[1], Nesredin Mahmud[1], Raluca Marinescu[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 advanced technology used for developing modern automotive systems increases their complexity, making their correctness assurance very tedious. To enable analysis, but also enhance understanding and communication, by simulation, engineers use MATLAB/Simulink modeling during system development. In this paper, we provide further analysis means to industrial Simulink models by proposing a pattern-based, execution-order preserving transformation of Simulink blocks into the input language of UPPAAL Statistical Model checker, that is, timed (hybrid) automata with stochastic semantics. The approach leads to being able to analyze complex Simulink models of automotive systems, and we report our experience with two vehicular systems, the Brake-by-Wire and the Adjustable Speed Limiter.

1 Introduction

Features for automating driving tasks, such as the Adjustable Speed Limiter (ASL) that enables drivers to set a maximum speed in order to reduce the risk of over speeding, as well as 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 challenging.

Model-based design enables industry to create executable specifications in the form of MATLAB/Simulink [1] models that can be simulated and formally analyzed [2] to detect hidden design errors and requirement violations.

In this paper, we introduce a pattern-based approach (Section 3) that captures formally the behaviors of a large set of Simulink blocks, as networks of stochastic timed/hybrid automata, and report our experience with analyzing two industrial systems from Volvo Group Trucks Technology, the *Brake-by-Wire* (BBW) prototype and the operational *Adjustable Speed Limiter* (ASL), with UPPAAL SMC (Statistical Model Checker) [3] (Section 4). 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 [4].

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 ASL's Simulink models with the Simulink Design Verifier (SDV) shows limitations in terms of verifying large models and that a substantial part of the requirements cannot be directly concluded due to, for instance, translation problems and boundaries not being defined. The application of our approach to BBW and ASL (specifically ASL's Engine Manager) shows improved scalability in the sense of being able to functionally analyze via statistical model checking the complete transformed Simulink models, but it also reveals limitations in tackling timing requirements, due to using only information from Simulink models.

Related work. Several works have already tackled the formal analysis of Simulink models. Barnat et al. [5] and Meenakshi et al. [6] propose transformations that target only Simulink blocks with discrete-time behavior. The work of Agrawal et al. [7] focuses on the transformation of Simulink into networks of automata, without providing concrete means for formal verification. Miller [8] investigates how translating Simulink to Lustre enables formal verification with a constellation of model checkers and provers. Transformation frameworks for Stateflow into timed and hybrid automata are presented in [9] and [10], respectively, with the former one applicable on a restricted class of Stateflow diagrams. Compared to these frameworks, our approach covers both continuous- and discrete-time blocks, and we show how our transformation leads to the formal verification of industrial automotive systems models, against a wide set of requirements. This is an endeavor not really carried out before. One other solution is the use of PLASMA-Lab [11], a tool that is able to take as input different Simulink simulations and provide statistical model checking results. Compared to this approach, we generate a formal model that can be extended further (e.g., with extra-functional information) to provide additional verification results.

2 Preliminaries

In this section, we present the two tools used in our framework: (i) Simulink, which is used to model the automotive systems, and (ii) UPPAAL SMC, which is used to analyze the systems.

Simulink. Simulink [1] is a graphical programming environment for modeling, simulation and code generation targeting multi-domain dynamic systems. The tool provides a set of libraries with predefined *blocks* that can be combined to create a hierarchical diagram of the system. A block represents an *atomic* dynamic system that computes an equation or another modeling concept to produce an output, either continuously (*continuous-time* block), or at specific points in time (*discrete-time* block). Besides these atomic blocks, Simulink supports definition of custom blocks via Stateflow diagrams or user-defined functions called *S-Functions* written in MATLAB, C, C++ or Fortran. The hierarchical diagram is achieved through the implementation of *subsystem*, a block that contains a set of atomic blocks and possibly other subsystem blocks. Such subsystems can be *virtual* (blocks are evaluated according to the overall model), or *non-virtual* (blocks executed as a single unit). A non-virtual subsystem can also be conditionally executed based on a predefined triggering function. During simulation, Simulink determines the order in which to invoke the blocks. This block invocation order is done based on a predefined *sorted order*. In Simulink, the dynamic models can be simulated and the results can be displayed as simulation runs.

UPPAAL SMC. The UPPAAL SMC [12] tool provides statistical model checking for stochastic hybrid systems. A hybrid automata (HA) is defined as a tuple:

$$HA = \langle L, l_0, X, \Sigma, E, F, I \rangle \tag{1}$$

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 $\mathbb{R}^X$, $F(l)$ a delay function for the location $l \in L$, and I assigns an invariant predicate $I(l)$ to any location l. With this definition, UPPAAL SMC extends the timed automata (TA) tuple used by UPPAAL [13] with the delay function F that allows the continuous variables to evolve according to ordinary differential equations. 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 model in UPPAAL SMC consists of a network of interacting stochastic HA that communicate through broadcast channels and shared variables. In the network, the automata repeatedly race against each other, that is, they independently and stochastically decide how much to delay before delivering the output, and what output to broadcast at that moment, with the "winner" being the component that chooses the minimum delay.

UPPAAL SMC uses an extension of WMTL [14] to provide probability evaluation ($Pr(*_{x \leq C} \phi)$), where $*$ stands for $\Diamond(eventually)$ or $\Box(always)$, which calculates the probability that ϕ is satisfied within cost $x \leq C$, but also hypothesis testing and probability comparison.

3 Simulink to UPPAAL SMC: Transformation Approach

There are two major aspects of transforming Simulink models into a network of stochastic timed/hybrid automata: (1) transforming the individual blocks, and (2) synchronizing their execution to preserve the behavior of the model. In this section we present how we transform Simulink models into a network of TA with stochastic semantics, suitable for statistical model checking with UPPAAL SMC.

Discrete-time blocks execute their computational routine on a predefined observable time interval called *sample time*, whereas continuous-time ones execute the routine over infinitely small time intervals. The same classification applies for the *S-Functions* that are masked, preserving only the specification of their input-output relation. For the subsystem blocks, the transformation is reduced to a flattening procedure which eliminates a subsystem block from the model and replaces it with its inner content with preserved atomicity of execution. The details and algorithm for flattening are given later in the section. The flattening procedure, however, does not apply for *Referenced models* given as as executables only, as no Simulink model is available for them. Such blocks are treated as atomic.

In the following, we provide a formal definition of a Simulink block as a tuple and *patterns* for transforming both discrete- and continuous-time blocks into TA with stochastic semantics.

Each atomic Simulink block can be formally defined as a tuple:

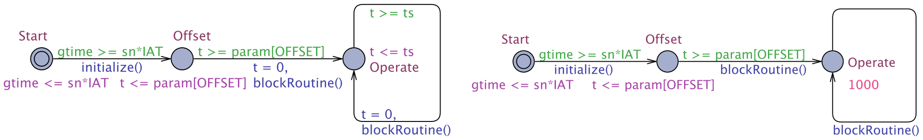
$$B = \langle V_{in}, V_{out}, V_D, t_s, Init, blockRoutine \rangle \tag{2}$$

where: V_{in}, V_{out} and V_D denote the set of input, output and data variables, respectively; t_s denotes the sample time, $Init$ is the initialization function, whereas the *blockRoutine* is a function that maps inputs and state variables onto output values. Our transformation is basically a semantic anchoring of tuple B of equation (2) onto the HA tuple given by equation (1).

The automata patterns corresponding to discrete and continuous categories is given in Figures 1a and 1b, respectively. Each of them is composed out of three locations, namely *Start*, *Offset* and *Operate*, with *Start* being the initial one. The *Offset* location is used to model the delay of the block execution. The last location is *Operate*, in which the automaton produces output either at predefined time intervals, or continuously. A local clock t is used to model the delay of the execution in both cases, and also to trigger the periodic behavior of the discrete blocks, whereas the continuous behavior is modeled via assigning *exponential rates* on the *Operate* location. The exponential rate is a mechanism used to specify the probability of the automaton to leave a location, according to an exponential distribution [3]. Simulation time is represented via the global clock *gtime*, which is used as part of the synchronization mechanism. The input parameters relevant for the pattern and its instantiation for a particular Simulink block are passed as array called *param*. The start time of the atomaton is calculated as a combination of the block execution order (*sn*) and the inter-arrival time of the block (IAT).

Preserving Block Execution Order. The execution order (sorted order) of the Simulink model blocks is generated by calling the *"slist"* function while Simulink is in debug mode. Simulink uses the assigned execution order to invoke blocks during simulation, with smaller execution order number denoting higher priority. We perform the flattening of the sorted order automatically, using Algorithm 1, which parses the *"slist"* output and assigns execution order number to atomic blocks nested arbitrary deep inside a subsystem.

We use this execution order to release the discrete and continuous time blocks during initialization in the UPPAAL model, and to arbiter their execution at



(a) Pattern for discrete blocks

(b) Pattern for continuous blocks

Fig. 1: Our used TA patterns

Algorithm 1 Flattening algorithm for slist.

```
function flatten(String currentBlockId, String currentBlockOrderNo, String parentBlockOrderNo)
    $orderedList \leftarrow emptyList$                                  ▷ Ordered list containing blocks IDs.
    if $isAtomicBlock(currentBlockId)$ then                            ▷ The current block is atomic.
        $orderedList.append(parentBlockOrderNo.concat(currentBlockOrderNo))$
    else                                                               ▷ The current block is a subsystem.
        $currentChildren \leftarrow getChildren(currentBlockId)$
        $concatenatedParentId \leftarrow parentBlockOrderNo.concat(currentBlockOrderNo)$
        for all $child$ in $currentChildren$ do
            $orderedList.append(flatten(child.id, child.orderNo, concatenatedParentId))$
    return $orderedList$
```

times when two or more blocks are ready to execute. Also, to assure the data integrity and predictability in the model we provide transformation for the *Rate-Transition* blocks.

Verifying UPPAAL Simulink Block Routines With Dafny. We use Dafny [4], a language and program verifier, to prove the functional correctness of the block routines encoded as C functions in UPPAAL. Below we present an example that shows verification of simple block routine using Dafny.

Rounding is one of the fundamental operations in Simulink, with several variants including rounding to floor, ceiling, fix, etc. In this example, we consider the floor variation of the function for non-negative real numbers. Due to space limitation, we omit the encoding of the function and present only the assertions that are used for proving the correctness. By using the Dafny, we establish the correctness of the function by checking the following pre- and postconditions, denoted as *requires* and *ensures*, respectively: "**requires** input $\geq$ 0.0", "**ensures** 0.0 $\leq$ (input - output) $<$ 1.0", where output $\in \mathbb{Z}_{\geq 0}$. We use the same approach to verify the correctness of all Simulink block behaviors that we encode as C functions in UPPAAL.

4 Application on Industrial Use Cases: Results

The proposed transformation has been validated on two industrial use cases, namely the Brake-by-Wire (BBW) industrial prototype, and the Adjustable Speed Limiter operational system. In this section we provide a brief overview of our results.

The BBW Use Case. The BBW system is a braking system equipped with an ABS function, and without any mechanical connection between the brake pedal and the brake actuators. A sensor reads the pedal's position, which is used to compute the desired brake torque. At each wheel, the ABS algorithm decides whether to apply the brake torque based on the slip rate. When the slip rate increases above 0.2, the friction coefficient of the wheel starts decreasing. For this reason, if the slip rate is greater than 0.2 the brake actuator is released and no brake is applied, otherwise the requested brake torque is used. The BBW system has a set of 13 functional and 4 timing requirements that need to be verified. Here, we present two such requirements, in natural language:

R1$_{BBW}$ **(End-to-end deadline):** The time needed for a brake request to propagate from the brake pedal sensor to the wheel actuator should not exceed 200 ms.

R2$_{BBW}$ **(Functional requirement):** If the slip rate exceeds 0.2, then the applied brake torque shall be set to 0.

Transformation. The hierarchical Simulink model for the BBW system consists of 320 blocks, out of which only 174 are computational blocks. The remaining 146 blocks define the structure of the model (e.g., Subsystem, Inport, Outport, From, Goto, Reference) and they are removed during the flattening. Consequently, the transformation provides a network of 174 TA. In this network, only 10 automata have continuous-time behavior, while the rest compute output only at sample times.

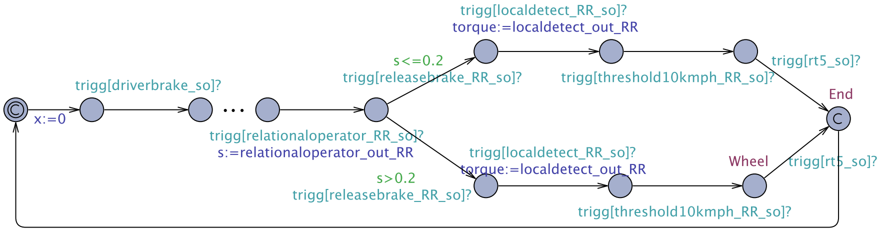

trigg[localdetect_RR_so]?
torque:=localdetect_out_RR
s<=0.2
trigg[releasebrake_RR_so]?
trigg[threshold10kmph_RR_so]?
trigg[rt5_so]?
trigg[driverbrake_so]?
End
x:=0
…
trigg[relationaloperator_RR_so]?
s:=relationaloperator_out_RR
trigg[localdetect_RR_so]?
torque:=localdetect_out_RR
Wheel
trigg[rt5_so]?
s>0.2
trigg[releasebrake_RR_so]?
trigg[threshold10kmph_RR_so]?

Fig. 2: BBW's Monitor Automaton.

Verification. In order to verify the system properties mentioned above, we have implemented a *Monitor* automaton that follows the propagation of data throughout the system, from sensors to actuators. It relies on the definition of an array of broadcast channels trigg[N], with N ∈ [1, 174]. Each TA in the network broadcasts the message trigg[own_id]! when it performs a new computation blockRutine(), and the Monitor receives these messages in a predefined order. For own_id we have used the predefined sorted number, since it is unique for each TA. Figure 2 presents an excerpt of the *Monitor* implemented for requirements **R1**$_{BBW}$ and **R2**$_{BBW}$.

For the BBW system we have verified all functional and timing requirements. In Table 1 we provide concrete verification results for requirements **R1**$_{BBW}$ and **R2**$_{BBW}$.

Table 1: Overall Results of Statistical Model Checking.

Req.	Query	Result	Runs
$\mathbf{R1}_{BBW}$	$Pr[Monitor.x <= 200](<> Monitor.End)$	Probability $\in$ [0.902606, 1] with confidence 0.95	36
$\mathbf{R2}_{BBW}$	$Pr[Monitor.x <= 200]([]\ Monitor.Wheel\ and\ Monitor.slipRate > 0.2\ and\ Monitor.torque == 0)$	Probability $\in$ [0.900924, 1] with confidence 0.975	42

The ASL Use Case. ASL is used to limit the truck speed to not exceed a maximum speed set by the driver. The driver normally enables and disables the function using control buttons located on the dashboard, and the freewheel. However, ASL can also be disabled when the accelerator pedal is pressed beyond a hard point, or the truck is subjected to overspeed, for example, in downhill, or becomes faulty during operation. ASL implements around 300 requirements, and is modeled using 4845 Simulink blocks, of which 2835 are non virtual blocks. We limit our verification to ASL Engine Manager (*ASL-EM*), which is a logical component, and an interface to the power-train of the truck's engine. It enables several functions of the truck, e.g., engine start and stop, climate control, fuel economy strategy, and road speed limitation. In our case study, we have transformed 94 non-virtual Simulink blocks, and verified all their functional and timing requirements. Examples of the ASL requirements are: (i) $\mathbf{R1}_{ASL}$(Min. speed limit): The ASL-EM controller shall be able to handle road speed limit requests down to 5 km/h, (ii) $\mathbf{R2}_{ASL}$(Lowest speed limit): When several road speed limit sources are active at the same time, ASL-EM shall use the lowest speed limit value, (iii) $\mathbf{R3}_{ASL}$(Max. latency): The maximum latency of the ASL-EM block shall be 20 ms.

5 Discussion and Conclusions

In this paper, we have introduced a pattern-based transformation of discrete- and continuous-time Simulink blocks into networks of stochastic timed automata. The approach is motivated by the industry's need of increasing the assurance of vehicular systems developed using Simulink, and the limited coverage obtained by employing the SDV for verification. Applying our approach on the BBW and ASL-EM systems has provided improved scalability for verification, that is, we have analyzed statistically the complete Simulink models, but we have also encountered concrete challenges and limitations, as follows:

1. The formal model needs to obey the same execution order as the Simulink one. For this, we have enforced the *sorted order* as generated by Simulink, which is usually respected during execution, except for block methods (blocks operating at the same rate and in the same task). These exceptions need to also be taken into account during the transformation.
2. Simulink allows for the integration of code in the model by using *S-function*. In our transformation, we do not provide direct means to verify this code. We view such components as "black boxes", modeled based on their defined mask and not the code itself.

3. Simulink lacks the possibility for modeling the timing behavior of the system (beyond the sample time), thus limiting the formal verification of extra-functional requirements. By pairing the Simulink model with an architectural model that allows for the representation of a wide set of extra-functional properties (such as timing behavior and possibly resource consumption), the transformation and the verification could provide a deeper insight to the engineers. Moreover, in the current version of our transformation, we have not exploited the full power of UPPAAL SMC. We have used TA with stochastic behavior, rather than stochastic HA. This is due to the fact that for more complex blocks (e.g., Derivative, Integrator) we have chosen to use the numerical approximation performed by Simulink, instead of implementing the function directly in UPPAAL SMC. This modeling decision will be further investigated.

Acknowledgement

This work was funded by the Swedish Governmental Agency for Innovation Systems (VINNOVA) under the VeriSpec project 2013-01299.

References

1. J. B. Dabney and T. L Harman. *Mastering Simulink*. Pearson/Prentice Hall, 2004.
2. 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.
3. Alexandre David, K.G. Larsen, A. Legay, M. Mikučionis, and D.B. Poulsen. Uppaal smc tutorial. *STTT Journal*, 17(4):397–415, 2015.
4. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR'10*, pages 348–370. Springer, 2010.
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. 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.
7. 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.
8. Steven P. Miller. Bridging the Gap Between Model-Based Development and Model Checking. In *TACAS*, pages 443–453. Springer, 2009.
9. K. Manamcheri, S. Mitra, S. Bak, and M Caccamo. A Step Towards Verification and Synthesis From Simulink/Stateflow Models. In *HSCC'11*, pages 317–318. ACM, 2011.
10. 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.
11. A. Legay and L.M. Traonouez. Statistical Model Checking of Simulink Models with Plasma Lab. In *FTSCS'15*, pages 259–264. Springer, 2015.
12. 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.
13. K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *STTT Journal*, 1(1):134–152, 1997.

14. 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.