Bounded Verification of Simulink Models

Predrag Filipovikj
Mälardalen University
Västerås, Sweden
predrag.filipovikj@mdh.se

Guillermo Rodriguez-Navas
Mälardalen University
Västerås, Sweden
guillermo.rodriguez-navas@mdh.se

Cristina Seceleanu
Mälardalen University
Västerås, Sweden
cristina.seceleanu@mdh.se

ABSTRACT

Simulink is the de-facto standard for model-based software development in many domains. With the ever-increasing size and complexity of the models, it is very beneficial to apply highly automated and rigorous verification techniques based on formal methods, which go beyond simulation, in order to check whether a model satisfies its specification. While tools employing such methods do exist for Simulink, one can mainly check errors such as integer overflow, array access violations, etc., or statistical properties.

In this paper, we show how Simulink models can be formally analyzed against invariance properties using bounded model checking reduced to satisfiability modulo theories solving. In its basic form, the technique provides means for verification of an underlying model over bounded traces rigorously, however, in general the procedure is incomplete. We identify common Simulink block types and compositions by analyzing selected industrial models, and we show that for some of them the set of non-repeating states (reachability diameter) can be visited with a finite set of paths of finite length, yielding the verification complete. We complement our approach with a tool, called SyMC that automates the following: i) calculation of the reachability diameter size for some of the designs, ii) generation of finite (bounded) paths of the underlying Simulink model and their encoding into SMT-LIB format and iii) checking invariance properties using the Z3 SMT solver. To show the applicability of our approach, we apply it on a prototype implementation of an industrial Simulink model, namely Brake by Wire from Volvo Group Trucks Technology, Sweden.

KEYWORDS

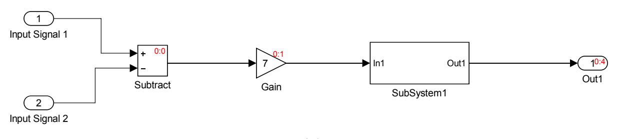
Simulink, Formal verification, Bounded invariance checking

1 INTRODUCTION

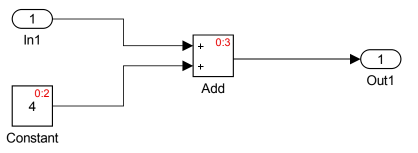
Simulink is a graphical development environment that supports system-level design, simulation, continuous verification and automatic code generation for embedded systems. Due to all these features, it is used as the de-facto development standard in many domains such as the automotive and avionics. For quality assurance, industry has become interested in adopting formal verification techniques, which bear the advantage of being systematic and sometimes exhaustive. One of the commonly-used tools is the Simulink Design Verifier, a formal verification suite provided by Mathworks based on the Prover plug-in [21]. The tool is suitable for finding design errors such as buffer overflows, division by zero, array access violations, etc., but not for complex invariance properties or properties that involve timing constraints [19]. Model checking [8] is a suitable candidate for verification of industrial designs with respect to the aforementioned properties, however, despite the drastic improvements in the memory efficiency of the approach, it still falls short of verifying complex models mainly due to their large state space. To cope with this challenge, there are solutions that resort to statistical model checking (SMC) [15, 18] for analyzing properties of industrial-size examples of Simulink models. Despite the fact that the method is not exhaustive, the SMC approach is rigorous and scalable. On the down side, the verification result comes as a probability estimation accompanied by a confidence interval that a given property is satisfied, which does not provide usable feedback to the designers for the purpose of refining the model.

One way forward is to resort to a different type of model checking called bounded model checking (BMC) [4]. BMC is a specialized model checking technique, tailored for checking system properties over bounded executions of the model, up to some predefined bound *k*. This characteristic makes it similar to SMC and suitable for checking industrial designs with large state spaces, while still being able to generate counter-example execution traces in case an invariance property is violated. Additionally, the set of reachable states of an underlying model in BMC is encoded as a set of propositional formulas, which means that the model checking is then reduced to a satisfiability problem. This way of exploring the state-space has been shown to scale well when applied on industrial embedded software [25].

In this paper, we propose an approach for bounded invariance checking of Simulink models based on the principles of BMC. The main idea behind our approach is that instead of using state-of-the art model-checking tools, we automatically generate a bounded reachable state space of a given Simulink model and directly encode it as a set of SMT constraints. Additionally, we investigate whether there are commonly used designs of Simulink models for which the complete set of reachable non-repeating states also know as the *reachability diameter* [4] can be visited using paths of finite length. We base our method on the execution semantics of Simulink blocks and models proposed previously in the literature. Assuming the former defined, we automatically generate the set of bounded execution paths for a given Simulink model and encode them as a set of satisfiability modulo theory (SMT) assertions. For the analysis of such model with respect to invariance properties we use Z3 SMT solver [9]. In order to determine the completeness of the bounded invariance checking procedure we perform the following: we isolate commonly used Simulink designs (compositions of blocks) from two industrial Simulink models, namely Brake by Wire (BBW) and Adjustable Speed Limiter (ASL) both from Volvo Group Trucks Technology (VGTT) and we create different configurations by instantiating different block types into the commonly used Simulink designs. Our results show that for some Simulink designs the set of reachable non-repeating states are reachable via paths of finite length, thus yielding the invariance checking complete. For automating our approach, we propose a tool called *SyMC* that automates all of the above-mentioned steps. Last but not least, we validate our approach on the BBW system provided by VGTT.



(a)



(b)

Figure 1: Simple Simulink model: (a) Root model and (b) Inner contents of SubSystem1 block

The paper continues as follows: in Section 2 we present the preliminaries including Simulink, SMT, BMC and the formal syntax and semantics definitions of Simulink blocks and model. Next, in Section 3 we present the industrial use cases that we use in our work, followed by Section 4 where we list the relevant identified Simulink blocks and types of compositions. In Section 5 we describe our method for bounded invariance checking for the considered Simulink models and the SyMC tool. Next, in Section 6 we show the applicability of our methodology on the BBW system. Finally, we compare to related work in Section 7 followed by the concluding remarks and directions for future research in Section 8.

2 PRELIMINARIES

In this section, we present the preliminary concepts that are used in the rest of the paper.

2.1 Simulink: Structure and Simulation Semantics

Simulink is an integrated environment for model-based development of multi-domain dynamic systems. It provides a graphical user interface with modeling functionalities, extensions for simulation and verification of the underlying models and automatic code-generation, making it appealing to industrial practitioners.

A Simulink model represents a hierarchical diagram composed out of various types of blocks interconnected with signals that model the control and data flow inside the model. In this paper, we use the terms signal and variable interchangeably. An illustrative example is given in Figure 1a, where the blocks are denoted using squares and triangles, signals via directed lines and input and output ports as ovals.

The fundamental building units of a Simulink model are the atomic blocks that model either a basic input-output relation (ex: *Gain*, *Subtract* in Figure 1a) or generate signals' values (ex: *Constant* in Figure 1b). A hierarchical Simulink model of an arbitrary depth is created using composite blocks (*SubSystems*) (ex: SubSystem1 in Figure 1a). Unlike atomic blocks that have a predefined input-output mapping function, the input-output function of the composite blocks is realized through a set of atomic blocks. Each signal in a Simulink model has at most one source block and one or more destination blocks. The source block has write access over the signal and controls its value, whereas the destination blocks have read access only.

A Simulink simulation represents a sequential valuation of the signals in the model for a finite amount of time, denoted as T_{sim}. The sequential order at which the atomic blocks are executed during simulation is called *sorted order* or *slist*. Each block has a unique execution order number (s_n). The notion of time of a Simulink simulation is modeled via two *simulation steps*: *major* (T_{maj}) and *minor* (T_{min}) simulation step. T_{min} represents an integer fraction of the T_{maj} and is used to improve the accuracy of the numerical computation for the signals' values. Continuous-time blocks perform computation at each T_{min}, whereas the discrete-time blocks compute their outputs at specific time points. The distance between two executions of discrete-time block is an integer number of T_{maj} and is called *sample time*. Additionally, Simulink allows delaying the first execution of some discrete-time blocks through a feature called *offset*, expressed as an integer number of T_{maj}. Any block in a given Simulink model can operate in discrete-time manner if a sample time is specified.

2.2 Formal Semantics of Simulink

In this section, we present the formal definitions for the atomic Simulink block and Simulink model, respectively, introduced in our previous work on formal analysis of Simulink models using statistical model checking [14].

The syntax of an atomic Simulink block (B) is given as the following tuple:

$$B = \langle s_n, V_{in}, V_D, V_{out}, \Delta, init, blockRoutine \rangle \quad (1)$$

where:

(i) $s_n \in \mathbb{Z}^+$ is the execution order number;
(ii) V_{in} represents the finite (possibly empty) set of real-valued input variables;
(iii) V_D represents the finite (possibly empty) set of real-valued data variables;
(iv) V_{out} represents the finite non-empty set of real-valued output variables;
(v) $\Delta = \{\Delta_0, \Delta_1, \cdots, \Delta_k\}$ represents the totally ordered set of time points at which an output value is produced. For discrete-time Simulink blocks, the value of each time point Δ_j is calculated as $\Delta_j = \mathit{offset} + j * t_s$, where: t_s, offset $\in \mathbb{R}_{\geq 0}$ are the sample time and the offset of the atomic Simulink block, respectively, and $0 \leq j \leq k \in \mathbb{N}$ is the index of time point. For continuous blocks, $\Delta_j = j * t_{min}$.
(vi) $init()$ is the initialization procedure that initializes the internal state variables of the block to the configuration parameters.
(vii) $blockRoutine() : V_{in} \times V_D \mapsto V_{out}; V_{in} \times V_D \mapsto V_D$ is sequential update first of the outputs followed by the update of the internal state variables.

We assume that all the time points in Δ from Formula (1) can be expressed as an integer multiple of Simulink simulation steps, which represents the basic quanta of time in the system as follows: $\Delta_j = j * (m * \delta) + (r * \delta)$, n, m and $r \in \mathbb{N}$ (here δ corresponds to

T_{min} simulation step). Also we assume that $x \in V_{in}$, $u \in V_D$ and $y \in V_{out}$ represent input, internal state (data) and output variables, respectively.

The operational semantics of a Simulink block can be interpreted over a timed transition system T, defined as follows:

$$T = \langle \Sigma, \Sigma_0, L, \rightarrow \rangle \tag{2}$$

where: Σ is the set of states, where each state $\sigma = (x|_t, u|_t, y|_t)$ is given by the values $y|_t$ 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, $u|_t$, $\Sigma_0 \subseteq \Sigma$ is the set of initial states, $L = L_a \cup L_t$ represents the set of labels, where $L_a = \{init, blockRoutine\}$ is the set of action labels and $L_t = \{m * \delta, r * \delta\}$ is a set of time labels, and $\rightarrow \subset \Sigma \times L_a \times L_t \times \Sigma$ is the transition relation that consists of the following types of transitions:

$$\sigma_0 \xrightarrow{init, r*\delta} \sigma \iff \begin{cases} \text{if } V_D \neq \emptyset \text{ then } t = t_0 + (r * \delta), \textit{ and} \\ \qquad init() : u = u_0, y_0 = u \\ \text{else } t = t_0 \textit{ and } init() : void \end{cases} \tag{3}$$

$$\sigma \xrightarrow{blockRoutine, m*\delta} \sigma' \iff t' = t + m * \delta, \textit{ and} \\ u' = f(x', u), y' = f(x', u') \tag{4}$$

The first transition type called *init* is executed once at the beginning of the blocks' execution only for those which have an internal state, whereas the second type of transitions called *blockRoutine* perform update of the internal state (if any) and the output variables for given inputs at particular time points denoted as $t + m * \delta$. If in the definition of Δ we instantiate $r = m = 1$ we obtain the continuous-time behavior of a blocks that execute the *blockRoutine* infinitely often, that is at every simulation step δ.

From the above definition, an infinite run ρ of a Simulink block can be defined as the following sequences of states:

$$\sigma_0 \xrightarrow{init, r*\delta} \sigma_1 \xrightarrow{blockRoutine, m*\delta} \dots \xrightarrow{blockRoutine, m*\delta} \sigma_n \tag{5}$$

A Simulink model (S) is defined as a sequential composition of atomic Simulink blocks, as follows:

$$S = B_1 \otimes B_2 \otimes B_3 \cdots \otimes B_n \tag{6}$$

where: $s_{n_S} = \bigcup_{i=1}^{n} s_{n_i}$ is an ordered list of execution with $\forall (i, j) \cdot i < j \Rightarrow s_i < s_j$, $V_{in_S} = \bigcup_{i=1}^{n} V_{in_i}$, $V_{out_S} = \bigcup_{i=1}^{n} V_{out_i}$, $V_{D_S} = \bigcup_{i=1}^{n} V_{D_i}$ being the sets of input, output and internal state variables, $\Delta_S = \bigcup_{i=1}^{n} \Delta_i$ represents the set of time points at which the variables are updated, $(init)_S \triangleq (init_1)|_{=r_1*\delta}; (init_2)|_{=r_2*\delta}; \dots; (init_n)|_{=r_n*\delta}$ is an ordered list of block initialization functions, and $(blockRoutine)_S \triangleq (blockRoutine_1)|_{=\Delta_1}; (blockRoutine_2)|_{=\Delta_2}; \dots; (blockRoutine_n)|_{=\Delta_n}$ is an ordered list of block input-output relations executed atomically at given times Δ_i. The blocks in the sequential composition communicate via shared variables.

2.3 Satisfiability Modulo Theories and Z3

The problem of determining whether a Boolean formula can be made true by assigning true/false values to the constituent Boolean variables is called the Boolean satisfiability problem (SAT). If a given Boolean formula is satisfiable, the boolean decision procedure generates a valuation of the variables such that the formula is true. In the opposite case, there exists no valuation for the constituent variables that will make the formula true. Satisfiability Modulo Theories (SMT) represents an extension of SAT, where some of the symbols are interpreted by a background theory [10]. For instance, the proposition might contain a variable of type integer over which arithmetic operations are applied.

In our work we use the Z3 tool [9] from Microsoft Research, which is a state-of-the-art SMT solver and theorem prover. The input to the tool is a script specified in the SMT-LIB language [2]. The satisfiability of the formulas present on the Z3 stack is checked via the `check-sat` command. If the set of formulas is satisfiable Z3 returns *SAT*. In the opposite case Z3 returns *UNSAT*. When the Z3 tool cannot determine whether the set of formulas is satisfiable or not it returns *UNKNOWN*. There are additional commands that can provide more information for *SAT* and *UNSAT* cases, respectively. The `get-model` command returns an interpretation that makes all formulas on the Z3 internal stack true. On the other hand, the `unsat-core` command returns the minimal inconsistent set of formulas.

2.4 Bounded Model Checking

Bounded model checking (BMC) [4] is a specialized model checking technique for verification of system properties over executions of finite length. The length of the paths is called *bound* and is usually denoted as *k*. Initially proposed as an efficient refutation technique, it has been shown that BMC can also be used for full verification of underlying designs [11].

Let M be the system model with set of states S, $I \subseteq S$ be the set of initial states and a state transition relation T, which is a binary relation on S. We write $T(s, s')$ to indicate that the state s is related to state s' via the transition relation T. Based on this, we define a path in M as follows:

$$path(s[0, .., n]) \triangleq \wedge T(s_i, s_{i+1}), \forall i \,.\, 0 \leq i < n \tag{7}$$

Consequently, any sequence of states that satisfies Formula (7) is a valid path in M. The size of a path is determined by the number of transitions taken. An invariance property *(P)* is a property that holds in every reachable state in model M, or formally expressed:

$$\forall s_0, .., s_n,\ \forall i \cdot 0 \leq i \leq n \cdot (I(s_0) \wedge path(s_{[0,..,i]})) \implies P(s_i) \tag{8}$$

where P(s_i) is a predicate denoting that a given state s_i satisfies property P. Using this definition, one starts from the set of initial states $I(s_0)$ and repeatedly applies the transition relation T, and each new state satisfies P. This is called a *forward reachability*. The same property can be checked using a *backwards reachability*, where the idea is to start from a non-initial state in which P does not hold and then show that it is not possible to reach an initial state by applying the inverse transition relation. The backwards reachability is formulated as follows:

$$\forall s_0, .., s_n,\ \forall i \cdot 0 \leq i \leq n \cdot \neg(\neg P(s_i) \wedge path(s_{[0,..,i]}) \implies I(s_0)) \tag{9}$$

In any case, one can prove that the model M satisfies an invariance property P by proving the following conjunction:

$$\forall s_0, .., s_n,\ \forall i \cdot 0 \leq i \leq n \cdot \neg(I(s_0) \wedge path(s_{[0,..,i]}) \wedge \neg P(s_i)) \quad (10)$$

where n denotes the length of the longest path of non-repeating states, defined as follows:

$$max\{i | \exists s_0, \cdots s_i \cdot I(s_0) \bigwedge_{j=0}^{i-1} T(s_j, s_{j+1}) \wedge \bigwedge_{j=0}^{i-1} \bigwedge_{m=j+1}^{i} \cdot s_j \neq s_m\} \quad (11)$$

Provided that the transition relation $T(s, s')$ can be expressed as a predicate, it is clear how the reachability problem can be reduced to a satisfiability problem. In cases when the transition relation T is constrained by a background theory an SMT is applied. This representation of the transition relation and the reachability procedure gives advantage to BMC over the BDD-based symbolic model checking on two fronts: i) it alleviates the infamous state-space explosion of model checking, and ii) it is very efficient for fast detection of errors in bounded traces up to 100 transitions [4].

The BMC procedure terminates when one of the following two conditions is satisfied: i) Formula (10) cannot be satisfied or ii) a predefined number of transitions (denoted as k) of the transition relation has been reached. The termination in the first case is due to the fact that a state that does not satisfy property P is detected, in which case a counter-example is generated. In the second case, all of the states along the generated path satisfy the property P. When the procedure terminates according to the second case, the property is proven to hold only until the bound but not beyond that. This yields the procedure incomplete, as there is no information whether the states reachable beyond k satisfy the property or not. However, there are some designs in which the paths might be infinite, yet the set of non-repeating states is finite and all of them are reachable within k. Such designs usually contain a *back-loop* in the transition relation, meaning that $T(s_i, s_j)$ represents a transition from some current state s_i to a state s_j that has been previously visited. The minimal path that contains the complete set of non-repeating states is called the *reachability diameter*, and the size of the same is called *completeness threshold (CT)*. The existence of the reachability diameter of finite size allows one to perform complete verification of invariance properties in general and a restricted class of liveness properties [17] over models with infinite state-space.

In our work, we use the Formula (10) for checking invariance properties over bounded executions of Simulink models.

3 INDUSTRIAL USE CASES

In this section we present the industrial systems, namely ASL and BBW both from VGTT, whose Simulink models we analyze here.

3.1 Adjustable Speed Limiter

ASL is an operational software function integrated into the modern heavy load vehicles produced by Volvo GTT. The intended functionality of the system is to control the servomechanism of the vehicle in order to limit the vehicle speed such that it does not exceed the predefined threshold defined by the driver. ASL is an ASIL-A safety critical system [1]. Its specification consists of more than 300 requirements at the system level, which cover the correct functioning of the system with respect to: i) variability, modes of operation, vehicle and engine speed; ii) road conditions, maintaining smooth driving experience in various road conditions such as flat or up/downhill roads, and iii) driver's requests that are passed to the function via the provided human machine interface, etc.

The complete ASL system consists of 22 modules, each of them realized in a separate Simulink model file. The total number of Simulink blocks inside the ASL system exceeds 4000. For this paper, we have used only two of the modules, called the Engine Manager and the Road Speed Limit Manager.

3.2 Brake by Wire

BBW is a prototype implementation of a software function that controls the breaking system of a vehicle equipped with an anti-lock breaking system. It is realized completely in electronics, thus eliminating all mechanical connections between the braking pedal (the sensor) and the four brake actuators on the wheels. The breaking pedal sensor installed in the car cabin reads the position of the brake and sends it to the main computational module, which then computes the brake torque to be applied on the wheels. The rotational speed of the wheels is monitored by sensors installed on all wheels. The ABS functionality is applied if the velocity of the vehicle exceeds some predefined threshold, which prevents the wheels from locking and skidding.

The BBW function is implemented as a Simulink model composed of 320 blocks. The system specification of the BBW that describes the intended functionality consists of 13 functional, and 4 timing requirements.

4 COMMON BLOCKS AND COMPOSITIONS

In this section, we first describe the commonly-occurring Simulink block types and compositions that we identify by studying the Simulink models of ASL and BBW, and give formal definitions to the latter. Next, we investigate whether there exists a finite bound k which represents the CT for the given compositions.

4.1 Identified Block Types

The atomic Simulink blocks used in the industrial use cases (see Section 3) belong to the following types: *feedthrough*, *delay*, and *SFunction*. In the following, we discuss in more details the specific block types and their characteristics relevant for this work. Additionally, we assume that all signals in the model are scalars.

Feedthrough (FT) blocks. FT represents a category of atomic Simulink blocks characterized by the absence of an internal state and by the immediate execution of the block routine when the input(s) change(s). Some examples of feedthrough blocks include: *relational*, *logic* and *arithmetic operators*, etc. The type of the block is determined by the type of the transfer function ($f()$ in Formula (4)). In our work, for the sake of simplicity we assume that the number of arguments in the transfer function can be arbitrary, but it always has only one output that is scalar. Having more then one output of the transfer function does not influence neither the formalization of the implementation as long as they are scalars.

Based on the possible transitions for atomic Simulink blocks given by Formulas (3) and (4), the execution trace of a feedthrough block B_{FT}, with $V_{in_{B_{FT}}}$ the set of input variables and $V_{out_{B_{FT}}}$ the set of output variables, the value of any variable $y_{B_{FT}} \in V_{out_{B_{FT}}}$ in any state $\sigma_{i_{B_{FT}}}$ from Formula (5) (denoted as $y_{i_{B_{FT}}}$) is always defined as follows:

$$y_{i_{B_{FT}}} = f_{B_{FT}}(x_{i_{B_{FT}}}), \tag{12}$$

where $x_{i_{B_{FT}}}$ and $y_{i_{B_{FT}}}$ are the valuations of the $x_{B_{FT}} \in V_{in_{B_{FT}}}$ $y_{B_{FT}} \in V_{out_{B_{FT}}}$ in a given state $\sigma_{i_{B_{FT}}}$, respectively.

Delay blocks. The blocks from this category delay the input value for a certain period of time. There are various block types in this category. In the industrial use cases considered for this paper only the *UnitDelay* and *RateTransition* types of delay blocks are used. *RateTransition* blocks represent a generalization of *UnitDelay* with many modes of operation, for instance: ZOH (Zero-order-hold), 1/Z (UnitDelay), Buf (semaphore controlled copying of the input to the output), NoOp (does nothing), etc. However, in our industrial models, all of the *RateTransition* blocks operate as *UnitDelay* in a discrete-time manner ($ts > 0$). The *offset* of the delay blocks is always 0.

The execution semantics of a UnitDelay block B_D is given as follows: before the first computation, the *init*() transition is executed, setting $u_{B_D} \in V_{D_{B_D}}$ to the initial value (*initV*) as specified in the block's configuration. After the block starts, the block executes its *blockRoutine*(). If we assume that t denotes the time that has elapsed along the execution of the block (Equation (5)), the *blockRoutine*() is executed whenever $t \ mod \ ts = 0$. Consequently, the values of the output variable y_{B_D} of a UnitDelay block is defined as follows:

$$\begin{cases} u_{i_{B_D}} = initV, & \text{executed before the execution starts} \\ y_{i_{B_D}} = u_{i_{B_D}},\ u_{i_{B_D}} = x_{i_{B_D}}, & \text{if } t \ mod \ ts = 0 \end{cases} \tag{13}$$

SFunctions. In Simulink one can extend the set of predefined block types via the concept of *SFunction*. In order to define custom a block, one first needs to implement the block routine in either Matlab, C or Fortran. Once the transfer function is implemented, the block is then wrapped with a mask that makes it look like any other block. The purpose of defining a mask is to encapsulate the block routine and to provide I/O interface which represents the input(s) into the computation and the output(s) of the same. In our use cases, the *SFunctions* are either stateless or the internal state is constant, that is, it does not change during execution. Due to this, all the *SFunction* blocks are treated as feedthrough.

4.2 Identified Compositions

The structural analysis of the industrial use cases shows that in principle there are two major designs of blocks compositions: *linear* and *feedback-loop*. The term *composition* is used to denote a subset of atomic blocks of a Simulink model, which in isolation can be treated as a Simulink model. Consequently, the syntax and semantics definitions for Simulink models apply to compositions as well. In this section, we provide definitions for the identified compositions.

Before we introduce and define the different types of compositions, we define the notion of *predecessor* for Simulink blocks inside a given Simulink model.

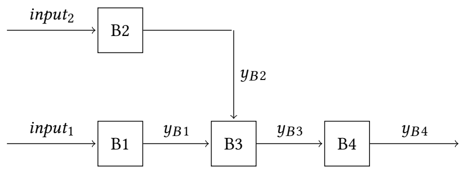


Figure 2: Linear composition of Simulink blocks.

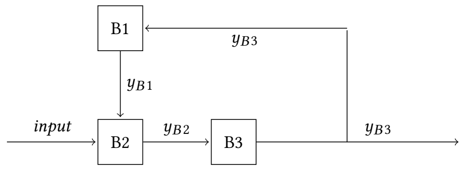


Figure 3: Feedback-loop composition of Simulink blocks.

Definition 1. *Let $S = B_1, B_2, \dots B_n$ be a set of $n \in \mathbb{N}$ atomic Simulink blocks with their sets of input and output variables $V_{in_{B_i}}, V_{out_{B_i}}, \forall B_i \cdot i \in [0,n]$. A* ***Predecessor*** *(B_i is a predecessor of B_j) is a binary relation over S defined as follows: $\forall B_i, B_j \in S \cdot predecessor(B_i, B_j) \triangleq \exists y \cdot y \in V_{out_{B_i}} \wedge y \in V_{in_{B_j}}$.*

The set of $Predecessors(B_i) \subset S$ is: $predecessors(B_i) \triangleq \forall B_j \in S \cdot (predecessor(B_j, B_i) \vee (\exists B_r \in predecessors(B_i) \wedge B_j \in predecessors(B_r)))$.

Linear composition. This is the most common composition of Simulink blocks. An illustrative example of a linear composition is given in Figure 2, and it consists of three blocks (B_1, B_2 and B_3) and four signals (*input*, y_{B1}, y_{B2}, y_{B3}). The main characteristic of the linear compositions is that the signals propagate in one direction, meaning that the current value of any output signal depends on the current value of the input signal(s) and internal state variables only. For instance, let us consider the signal y_{B_3} from Figure 2. The value of the signal can be computed using a function obtained as a sequential (forward) composition of transfer functions of the predecessor blocks $y_{B_3} = f_3(f_2(input_2), f_1(input_1))$, where f_1, f_2 and f_3 are the transfer functions of B_1, B_2 and B_3, respectively and $input_1$ and $input_2$ are the inputs. Based on this, we define a linear composition as follows:

Definition 2. *Let $B_1, B_2, \dots B_n$ be a set of $n \in \mathbb{N}$ atomic Simulink blocks with their sets of input and output variables $V_{in_{B_m}}, V_{out_{B_m}}$, for any $m \in [0,n]$ and execution traces: $\rho_{B_m} = \sigma_{0_{B_m}}, \sigma_{1_{B_m}}, \cdots, \sigma_{k_{B_m}}$, $m \le n$, respectively. Let $y_{B_m} \in V_{out_{B_m}}$ be the output signal variable of B_m, and $y_{i_{B_m}} \cdot i \le k$ be its valuation in a given state $\sigma_{i_{B_m}}$. We say that composition C is linear if $\forall m \cdot 0 \le m < n \cdot \forall y_{B_m}, \ \forall y_{j_{B_m}} \cdot 0 \le j < k_{B_m}, \ \forall y_{r_{B_m}} \cdot j < r \le k_{B_m} \cdot \forall z \ne m, \ 0 < z < n \cdot (y_{r_{B_m}} \ne f_{B_m}(y_{j_{B_m}}) \wedge y_{r_{B_m}} \ne f_{B_m}(f_{B_z}(\dots(f_{B_1}(y_{j_{B_m}})))) \wedge (\forall input_j \in V_{in_{B_1}} \cdot y_{j_{B_m}} = f_{B_m}(f_{B_{m-1}}(\dots f_{B_1}(input_j)))))$.*

Feedback-Loop composition. The second design pattern identified in the industrial use cases is the feedback-loop composition. Opposite to the linear composition, in the feedback-loop composition there exists at least one output signal for which the current value is computed based on some of its previous values. Formally, we define a feedback-loop composition as follows:

Definition 3. *Let $C = B_1 \otimes B_2 \otimes \cdots \otimes B_n$ be a composition of atomic Simulink blocks $B_1, B_2, \ldots B_n$ defined as in* ***Definition 2****. The composition C is feedback-loop if $\exists y_{B_i} \cdot \forall y_{j_{B_i}} \cdot 0 < j \leq k_{B_i} \cdot \exists r \cdot j < r \leq k_{B_i} \cdot \exists m \neq i, 0 < m < n \cdot (y_{r_{B_i}} = f_{B_i}(y_{j_{B_i}}) \vee y_{r_{B_i}} = f_{B_i}(f_{B_m}(\ldots(f_{B_1}(y_{j_{B_i}})))))$.*

This broad definition for the feedback-loop composition incorporates both direct and indirect feedback loops. A direct feedback-loop assumes that the feedback signal is input to the current block B ($y_B = f(y_B)$), whereas an indirect one assumes that the feedback signal is used as an input signal to some of the predecessors of B, and it eventually becomes an input of the current block ($y_{B_n} = f_{B_n}(\ldots f_{B_1}(y_{B_n})))$. An illustrative example of an indirect feedback-loop composition is given in Figure 3, which is composed of three blocks (B_1, B_2 and B_3) and four signals (*input*, y_{B1}, y_{B2}, y_{B3}), with y_{B3} being the feedback signal. In our work we assume that the feedback-loops cannot be algebraic[1] since the Simulink models that contain them cannot be used for code generation. This means that in the feedback-loop compositions, the feedback signal must be delayed using some type of a delay block. Such requirement introduces a new signal (y_{B1} in Figure 3) into the composition. Consequently, the signal y_{B1} holds either the initial value of the delay block or some previous value of y_{B3}.

Another important observation is that in the general case the type of a given composition cannot be implied from its graphical representation. An example of graphically linear composition that in fact is a feedback-loop is a composition where at least one of the blocks is a continuous-time block (ex: Integrator, Derivative, etc.).

4.3 Completeness of Bounded Invariance Checking for Identified Compositions

In this section, we investigate whether there exists a CT, for the linear and feedback-loop compositions, such that the bounded invariance checking procedure is complete, under the following assumptions: i) each block in the composition has an execution order id which is monotonically increasing ($sn_1, sn_2, \cdots sn_n$), and ii) the invariance properties are always specified over the output signal of the last block in the composition. Recalling Formula (10) that is used for checking invariance properties, as well as the definition of CT (Section 2.4), a given composition of Simulink blocks has a CT iff all the possible values of an output signal y constrained by the invariance property are reachable within paths of a finite length.

Linear Composition. According to the formal definition for linear composition (Formula (2)), the value of each output signal in the model is computed based on the current values of the input and the internal state variables of the constituent blocks. In the following, we look at two possible configurations: i) linear compositions of feedthrough blocks only, and ii) linear compositions of feedthrough and delay blocks.

In the first configuration (Figure 2), there is no internal state in the composition. Assuming that $input_1$ and $input_2$ are the inputs of B_1 and B_2, respectively, the value of the signal y_{B4} is always calculated by sequential application of the transfer functions $f_{B_1}, f_{B_2}, f_{B_3}$ and f_{B_4} corresponding to the B_1, B_2, B_3, B_4 blocks, respectively, over the input signal, in that specific order. Consequently, the value of the output signal y_{B4} in all states of execution $\rho_{B_4} = \sigma_{0_{B_4}}, \sigma_{1_{B_4}}, \cdots, \sigma_{k_{B_4}}$ is always calculated by the following expression:

$$\forall i \cdot 0 \leq i < k,\ y_{i_{B_4}} = f_{B_4}(f_{B_3}(f_{B_2}(input_{2_i}), f_{B_1}(input_{1_i}))) \quad (14)$$

Theorem 1. *Let $C = B_1 \otimes B_2 \otimes \cdots \otimes B_n$, $n \in \mathbb{N}$ be a linear composition of feedthrough blocks with $f_{B_1}, f_{B_2}, \cdots, f_{B_n}$ transfer functions, respectively, $input \in V_{in_C}$ is the input of block B_1 and $output \in V_{out_C}$ is the output signal of block B_n in the composition. For such compositions, there exists a reachability diameter of size 0 (called the CT of C), over which the satisfaction of an invariance property P can be checked. We say that P is an invariance property of C iff the following conjunction does never evaluate to true: $f_{B_n}(f_{B_{n-1}} \cdots (f_{B_1}(input))) \wedge \neg P(y_{B_n})$.*

Proof. Let us assume that $y_{i_{B_n}}$ and $y_{j_{B_n}}$ represent the valuations of an output signal y_{B_n} in states $\sigma_{i_{B_n}}$ and $\sigma_{j_{B_n}}$, respectively, such that $y_{i_{B_n}} \neq y_{j_{B_n}}$ for $input_i = input_j$. Since the value of $y_{i_{B_n}}$ is defined by Formulas (12) and (14), $\forall i \cdot 0 \leq i < k$, the assumption is clearly a contradiction as $f_{B_n}(\cdots f_{B_2}(f_{B_1}(input_i))) = f_{B_n}(\cdots f_{B_2}(f_{B_1}(input_j)))$ for $input_i = input_j$. Consequently, all the reachable values of the variable y_{B_n} can be computed in the initial state, thus the claim that $CT = 0$ holds. □

In the second configuration, at least one of the constituent blocks (for instance B_2) in the linear composition is a delay block. In order to determine whether the existence of a CT is preserved, let us recall the definition of *blockRoutine*() for delay blocks given in Formula (13). It follows that the value of a delayed signal y_{B_2} is either the initial value of the block's internal state ($initV_{B2}$) given in the block configuration or exactly the same value of the input signal $input_2$ delayed by one sample time of B_2. For compositions composed of delay and feedthrough blocks, the execution sequence can be divided into two segments: i) when y_{B_4} is calculated based on the initial value of B_2 ($y_{B_4} = f_{B_4}(f_{B_3}(initV_{B2}, f_{B_1}(input_1)))$), and ii) when block B_2 starts to execute, and the value of y_{B_3} is calculated as: $y_{B_4} = f_{B_4}(f_{B_3}(input_2, f_{B_1}(input_1)))$. We refer to these two segments as *initialization* and *feedthrough*, respectively. Once the execution of the composition enters the *feedthrough* segment, based on Theorem 1 the composition reaches a CT. Therefore, the index of the state at which the execution of the last signal in the composition enters the *feedthrough* segment is the CT of the composition.

Theorem 2. *Let $C = B_1 \otimes B_2 \otimes \cdots \otimes B_n$, $n \in \mathbb{N}$ be a linear composition of feedthrough and delay Simulink blocks with $V_{out_{B1}}, V_{out_{B2}}, \cdots, V_{out_{Bn}}$ set of output variables, and $ts_1, ts_2, \cdots ts_n$ sample times, respectively. Then for all such compositions there exists k_i for each signal $y \in V_{out_{Bi}} \cdot 0 < i \leq n$ which is the completeness threshold (CT_i) for the signal y. The value of k_i for*

[1]an algebraic loop occurs when a signal loop exists with only direct feedthrough blocks within the loop. Direct feedthrough means that the block output depends on the value of its input; the value of the input directly controls the value of the output.

each $y \in V_{out_{B_i}} \cdot 0 < i \leq n$ *is computed as follows:*

$$\begin{cases} k_i = max(k_j), B_j \in predecessors(B_i), \text{ if } B_i \text{ is a feedthrough block,} \\ \qquad \text{and there exists a delay block in } predecessors(B_i) \\ k_i = m * ts_i, m = max(ceil(\frac{CT_j}{ts_i})), B_j \in predecessors(B_i), \text{ if } B_i \text{ is a} \\ \qquad \text{sampled feedthrough block} \\ k_i = (m+1) * ts_i, \text{ if } B_i \text{ is a delay block} \end{cases}$$

Proof. ***Part I***: *Correctness of computation of* k.

Recalling Theorem 1, the CT for feedthrough blocks is 0. Consequently, if such a block is placed inside a composition where there are sampled blocks it will not have any effect on computing the k of the entire composition.

In the second case, that is, if a feedthrough block $B_i \in S$ is sampled (ts > 0), there are two possibilities: i) if $ts_i = max(k_j) \cdot B_j \in predecessors(B_i)$, then $\frac{max(k_j)}{ts_i} = 1$ and k_i equals to the largest CT of its predecessor blocks; ii) when $ts_i \neq max(k_j) \cdot B_j \in predecessors(B_i)$, the k of the sampled feedthrough block is the first execution of the same after the maximal k of its predecessors, which is assured by the ceiling ($ceil()$) function when calculating m.
Finally, for the delay block, the calculation of CT is similar to the one of sampled feedthrough block, except that it needs one additional execution. This is because of its execution semantics (see Formula (13)), according to which the input read at a given sample time will become an output during the next execution, hence $m + 1$.
Part II: k_i *is indeed the CT for the specified signal. Proof by contradiction.*

Let us assume that B_i is a sampled feedthrough block with $y \in V_{out_{B_i}}$ output variable, and k as in Theorem 2 is not the CT of y. This means that there is a state along the execution path at position $r \in \mathbb{N}$ such that $k < r$ at which B_i produced new value. From the calculation of k, we know that $\exists m \in \mathbb{N}$ such that $m \leq r$ at which all the inputs of B_i have reached their respective CTs. Starting from position k the value of y in all subsequent states $r \geq k$ is calculated using the following expression $y_r = f_i(x_{1_{r B_i}}, \cdots x_{z_{r B_i}})$, where f_i is the transfer function of B_i and $x_{1B_i}, \cdots x_{zB_i} \in V_{in_{B_i}}$ are the set of input signals. Consequently, it is not possible to reach a new value for y after step k, thus the assumption that $k \neq CT$ is false.

Finally, let us assume that B_i is a delay block with $y_{B_i} \in V_{out_{B_i}}$ output variable and $\exists r$ such that $k < r$ is the CT of y_{B_i}. Given the way we compute k, we know that $\exists m \in \mathbb{N}$ such that $m \leq k$ at which all the inputs of the B_i have reached their respective CTs. Starting from position k the value of y in all subsequent states $r \geq k$ y_{B_i} is always assigned as follows: $y_{rB_i} = x_{(r-ts)B_i}$, where $x_{B_i} \in V_{out_{B_i}}$ and ts is the sample time of B_i expressed in number of transitions. From the calculation of k it is clear that $r - ts > k_j$, where k_j is the CT of the input signal, thus the initial assumption does not hold. □

Feedback-Loop Composition. The main characteristic of the feedback loop compositions is that the value of some signal depends on one or more previous values of the same signal. The previous value of the feedback signal must be delayed using a delay block before returned into the main sequence of computation (see Section 4.2). Due to the dependency of previous values, in order to compute the value of the output signal in the current state, one must compute all the previous values of the same. To be able to show the completeness of BMC for the composition, we must demonstrate that there exists a state after which no new values of the output signal can be produced.

Theorem 3. *Let* $C = B_1 \otimes B_2 \otimes \cdots \otimes B_n$ *be a feedback loop composition of blocks. For such compositions, there exists* CT *if the value range of the inputs of the composition and feedback signal are constrained with closed and enumerable intervals and the output signal* $y \in V_{out_{B_n}}$ *over which the invariance property is specified is strictly monotonic.*

Proof. Let us assume that in the above composition C $y_{B_n} \in V_{out_{B_n}}$ is the output signal of the last block of the composition over which the invariance property is specified and that $y_{B_n} = f_n(f_{n-1} \cdots f_{n-m}(x1_{B_n}, x2_{B_n}, x3_{B_n}, \cdots xh_{B_n})), m \in \mathbb{N} \cdot 1 \leq m < n, x1_{B_n}, x2_{B_n}, \cdots xh_{B_n} \in V_{in_{B_n}}$ is the function that assigns values of the output variable y_{B_n}. We assume that one of the inputs $(x1_{B_n}, x2_{B_n}, \cdots xh_{B_n})$ is the feedback signal and that the value range of each of the signals is constrained by enumerable closed interval $[x1_{B_{n_l}}, x1_{B_{n_u}}], [x2_{B_{n_l}}, x2_{B_{n_u}}], \cdots [xh_{B_{n_l}}, xh_{B_{n_u}}]$.

1) Let us assume that $f_n(f_{n-1} \cdots f_{n-m}(x1_{B_n}, x2_{B_n}, x3_{B_n}, \cdots, xh_{B_n}))$ is strictly monotonically increasing function. Given that one of the inputs is the feedback signal, with each new computation y_{B_n} takes new value that is bigger than the previous one. Consequently, there exists minimal difference between the current and previous value of the y_{B_n} signal. Using the minimal increment of the y_{B_n} and the enumerable value ranges of the inputs it is possible to calculate the maximal number of steps of the transition relation k at which y_{B_n} reaches its maximal value, thus $k = CT$.

2) Let us now assume that $f_n(f_{n-1} \cdots f_{n-m}(x1_{B_n}, x2_{B_n}, x3_{B_n}, \cdots xh_{B_n}))$ is a strictly monotonically decreasing function under the same setup as in 1). Analogously, each new value for y_{B_n} is smaller than the previous one with some minimal difference. We can compute the maximal number of steps k at which y_{B_n} reaches its minimal value, hence $k = CT$. □

5 SMT-BASED BOUNDED INVARIANCE CHECKING

In this section, we present our SMT-based bounded invariance checking method (Section 5.1) and the SyMC tool (Section 5.2) that automates the later.

5.1 Method

The proposed bounded invariance checking approach for Simulink models is illustrated in Figure 4. It consists of the following steps: i) automatic generation of finite execution paths of the Simulink model according to the formal semantics presented in Section 2.2, ii) automatic encoding of the generated executions as an SMT-LIB script suitable for analysis using Z3, and iii) the analysis of the generated SMT-LIB script. The boxes denote the processing steps, whereas circles denote the artifacts. In the rest of this section we describe each of the steps in detail.

The proposed methodology starts with Step 1, during which finite paths (as defined by Formula (7)) with a predefined bound k are generated. As inputs, the path generation procedure assumes a

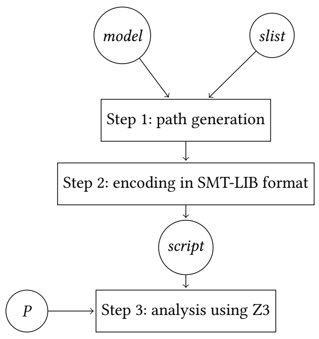


Figure 4: Bounded model checking approach for the analysis of Simulink models.

Simulink model (*model*) and a sorted order execution list (*slist*) that contains the execution number of each atomic block in the model. The length of the generated paths (bound k) is always expressed as a number of *simulation steps* (see Section 2.1). For generating the bounded paths, a number of configuration parameters must be provided: the duration of the paths in absolute time, the relation between one unit of absolute time and the major simulation step (T_{maj}), and the ratio between T_{maj} and minor (T_{min}) simulation steps.

Depending on the type of the model, a different strategy is employed for generating the state-space of the model. For instance, if the model is purely discrete time, T_{min} is not used, since all of the sample times of the model can be expressed as an integer number of T_{maj}. Additional optimization includes the introduction of the *fundamental sample time* (T_f) that represents the least common divisor of all different sample times used in the model. In the worst case, $T_f = T_{maj}$, but usually it is larger, thus resulting in paths of less transitions that describe the same set of reachable states. As an additional optimization, the procedure assigns sample time of the non-sampled feedthrough blocks based on the rate of change of their input(s).

As the paths are already encoded symbolically during Step 1, during the second step the generated paths are transformed into SMT assertions that can be analyzed using any tool that accepts input specified in SMT-LIB. For illustration, let us recall the simple model from Figure 1b. We assume the following names for the signals: the output signal of block *In1* is denoted by y_{in1}, the output signal of *Constant* is denoted by $y_{constant}$, and the output signal of the block *Add* is y_{add}. In Step 1, each state in the path is represented by the following tuple: (y_{in1}, $y_{constant}$, y_{add}). Following this, a single one-transition execution path for this composition can be encoded as follows: *((x, 4, y), (5, 4, 9))*. In the first state $(x, 4, y)$, the values of the signals y_{in1} and y_{add} are not known, thus they are represented with symbols x and y, respectively. A possible successor state of the previous is *(5, 4, 9)*, meaning that setting the value of the signal y_{in1} to 5 results in signal y_{add} becoming 9. It is clear that there might be other one-transition paths. To be able to encode all the possible successors of $(x, 4, y)$, we encode the successor state analytically, as follows: *T((x, 4, y), (y_{in1}, 4, (+ 4 y_{in1})))*. By encoding the value of the signal y_{add} analytically, we have implicitly encoded all the reachable values of that signal. The value of signal y_{in1} does not represent an input to the system (see Figure 1a), which means that it can be further expressed as a function of the *Gain* block. If we continue repeating the procedure, eventually an analytical expression for each signal is created, where the only variables are the inputs. If the model is encoded as such, then we can say that the analytical expression represents all the reachable values (states) for all the signals in the model, which according to Theorem 1 is the CT of the model.

Finally, the SMT script that can be analyzed in order to determine whether the system model satisfies an invariance property for all bounded executions is generated. The presented methodology does not provide means for the correct encoding of the invariance properties such that they can be checked over the generated path. In the current version, their existence is assumed as an artifact (P in Figure 4), expressed to correspond to Formula (10).

5.2 SyMC tool

We use an initial prototype of our SyMC tool to automatically execute Steps 1-3 of the proposed methodology in Figure 4. The tool is implemented in Python, and the source code is available on github [13].

The implementation of the tool is divided into different modules. The module that automates Step 1 of the proposed methodology provides complete automation for the following features: i) determining the fundamental sample time for the discrete-time and hybrid Simulink models, ii) the generation of the path that describes the evolution of the signals until the predefined bound k, which is specified in the configuration file, and iii) calculation the size of reachability diameter (CT) for linear compositions. In the current version of the SyMC tool we assume a simplified Simulink model created based on real one. The simplified model file contains the same information as the original one, however, the information is structured in JavaScript Object Notation (JSON) format such that the model can be parsed with minimal effort. It is important to note that this is due to purely technical reasons (lack of developers) and does not affect the proposed methodology in any way. During Step 2, the tool automatically determines the block type and instantiates the corresponding *blockRoutine()*. It is important to notice that the implementation of this module is inherently incomplete, as it is not feasible with the available resources to encode all the Simulink supported block types *blockRoutines()*. Nevertheless, for the purpose of analyzing the BBW model, all of the relevant *blockRoutines()* have been encoded and the transformation of the atomic blocks is fully automated. Lastly, during this step, the encoding of the invariance properties is added to the generated reachable state space.

During the last step, that is Step 3, the generated SMT-LIB script is analyzed using Z3 via the provided Python API (Z3Py), which enables seamless integration of the latter into the SyMC tool.

6 APPLICATION

In this section, we present in detail the application of the SyMC tool on the BBW industrial prototype. A snapshot of the original BBW model is given in Figure 5. The intended purpose is to give a glimpse of the size and complexity of the model at the root level. In order to inspect the model in details, we refer the readers to the SyMC code repository.

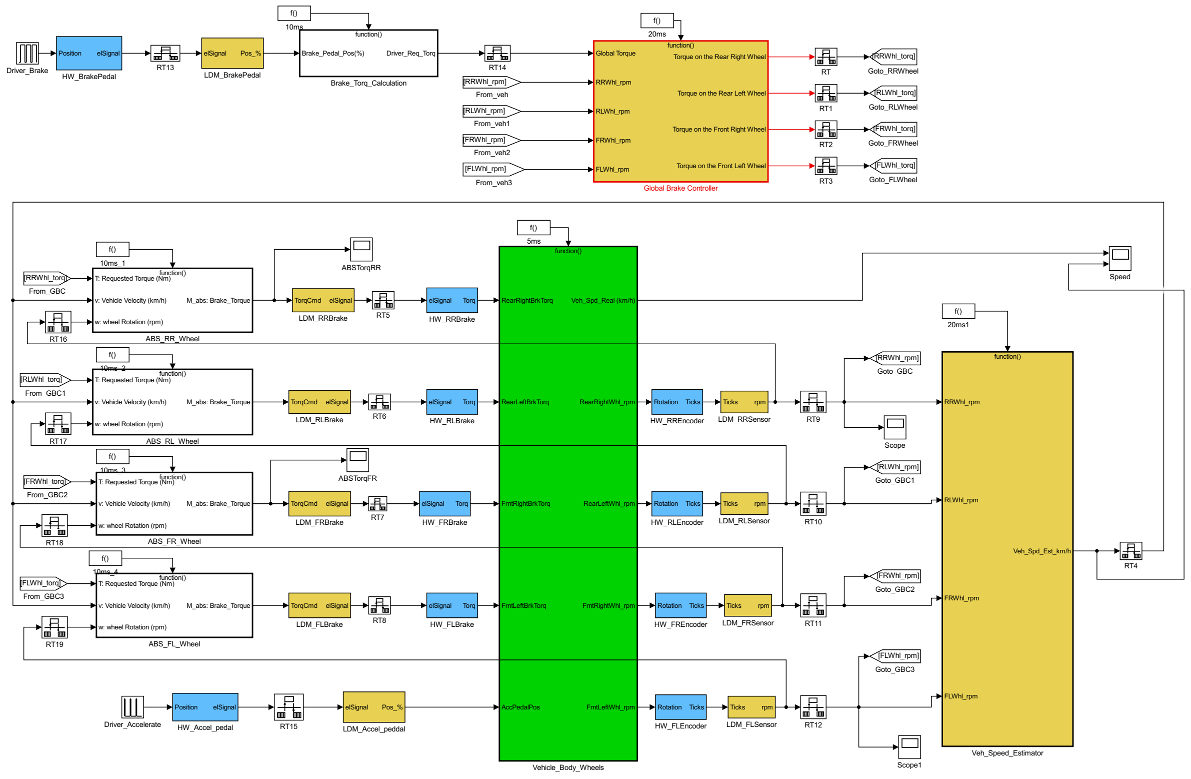


Figure 5: Brake by Wire Simulink model.

6.1 Transformation of BBW

We use our SyMC tool to analyze the BBW model based on the simplified Simulink model.All of the blocks in the model are either discrete-time (delay or feedthrough) or feedthrough blocks with no sample time. Eventually, these blocks are also transformed into discrete-time blocks during the model optimization (see Section 5.1). The set of sample times in the model includes the following values: 5, 10 and 20, each given in milliseconds (ms). Consequently, the fundamental sample time for the model is 5 ms, meaning that blocks are sampled at each 1st, 2nd or 4th fundamental time steps.

The atomic Simulink blocks are predominately of the following categories: arithmetic operators (*Add*, *Subtract*, *Divide*, *Multiply*, *Gain*), logical operators ($<$, $\leq$, $=$, $>$, $\geq$, *if-then-else*), *Constant*, *Delay*, *Saturation*, etc. These types of blocks belong to the library of standard Simulink blocks and as such are supported by our tool. Additionally, the model contains blocks for which the automatic transformation is not supported in at the moment, hence manual intervention is required. Of these types, we have the following blocks: LookUp Table and State-flow diagrams.

The internal structure of the BBW contains both linear and feedback-loop compositions, as it can be seen in Figure 5. The figure does not depict the actual complexity of the feedback loops, as once one goes into the subsystem blocks (*Vehicle_Body_Wheels*, colored in green, or the *ABS_XX_Wheel* denoted with white), the loops become much more complicated. The complex structure, combined with the number of missing features of the tool, such as the one for the automatic computation of the CT for feedback loop compositions, prevent us to precisely determine whether BBW's bounded invariance checking is complete or not for all requirements. For properties which are specified over the portion of the model that contains a feedback loop, we perform invariance checking over bounded paths of length 40 execution steps, that is, a duration of 200 ms expressed in absolute time.

6.2 Application results

All possible bounded execution paths of 40 steps have been generated in approximately 2 seconds on a standard Macbook Pro laptop machine running OSX 10.10.5, with the following hardware characteristics: 2.6 GHz Intel Core i5 CPU and 8 GB of RAM. The bound of 40 steps was used because it was the maximum size for which the SMT analysis terminates on the specified machine. The analysis model, that is, the SMT-LIB script is composed of 15319 assertions, out of which exactly 8000 represent variable declarations and 7278 represent constraints over the set of variables.

The generated model is verified against the following two safety requirements, encoded as invariance properties:

P1 *The value of the brake pedal position shall not exceed its maximal value of 100.*

P2 *If the slip rate exceeds 0.2, then the applied braking torque shall be set to 0.*

The compositions of Simulink blocks that contribute to property *P1* are of type *linear composition* of *feedthrough* blocks only. From earlier, we know that CT of such compositions is $k = 0$, meaning that the analysis of the generated path is complete, and the result is a full guarantee that $P1$ holds. The composition of blocks that contributes to property *P2* on the other hand, contains a feedback loop, consequently for analyzing the model against $P2$, we apply the incomplete bounded invariance checking procedure.

The verification of both of the above properties over the generated path took approximately 8.5 and 9.7 seconds, respectively. The results deem that *P1* is indeed an invariance property of the BBW Simulink model. Similarly, *P2* is proven correct over the bounded path of $k = 40$ execution steps, with no information about its correctness beyond this limit (due to our theoretical results).

To demonstrate a case of finding a violation, we modified P1 to create P1' as follows: *"The value of the brake pedal position shall not exceed its maximal value of 10".* When the property is checked against a model, a violation is detected and the input is synthesized within 25 seconds.

7 RELATED WORK

There is an increasing body of work on the formal verification of Simulink models. We compare our work to other approaches based on BMC, as well as to other related types of formal analysis techniques. We focus primarily on formal verification of Simulink models at *design time.*

5Schrammel et al. [25] show that BMC has a great potential for verification on industrial systems. Their approach, which is based on incremental BMC, yields encouraging results when validated on industrial use-cases. The difference compared to our work is that they focused on the verification of the actual code implementation, while our approach targets the verification of design-time models. Chaves et al. [6] propose the DSVerifier tool for verification of digital systems with respect to overflow, limit cycle, stability, etc. As an input, the tool accepts a digital system design specified in MATLAB from which an ANSI-C code is generated and passed into a highly efficient BMC tool. Herde et al. [16] propose an analysis method for dynamic behavior of safety-critical system based on HySAT. Compared to our work, they translate the behavioral system into a hybrid model, which is analyzed using bounded reachability. The approach guarantees absence of errors until some predefined bound k (which is not necessarily a CT), but not beyond that. In comparison to this, we also discuss different compositions of Simulink blocks and completeness of bounded invariance checking for the same. Similarly, Minopoli and Frehse [22] translate Simulink models into an intermediate SL2SX model using the SpaceEx translator. The SL2SX model can then be translated into number of formal notations based on hybrid automata, such as for example HyDI model that can be checked using the HYCOMP tool [7]. The Simulink to SL2SX translation is limited to the block of discrete-time type and is missing support for commonly used blocks industrial models such as Mux, DeMux, etc. Additionally, the approach has been applied on a Simulink models of relatively moderate size, thus the scope of the applicability remains an open question. In comparison, in our approach we: i) avoid the intermediate encoding as we generate the paths directly from the Simulink model, ii) successfully deal with structural elements (Mux, DeMux, etc.) and iii) we retain the possibility to use various analysis tools as the paths are encoded as a set of assertions encoded in SMT-LIB, a format suitable as input to most of the modern SMT solvers. Bauch et al. [3] propose a hybrid approach based on a combination of explicit model checking with SAT-based representation of the variables, based on a set of possible evaluations. To model the valuations of the variables in the system, they rely on the theory of bit vectors. In contrast to this, in our approach we deem the system safe by showing that refutation of the invariance properties is not possible for all possible inputs and executions, according to the principle idea of BMC. Approaches based on statistical model checking are being proposed as a compromise between the exhaustive verification using symbolic model checking and simulation [18] [14]. Such approaches are useful for establishing probabilistic estimation for the correctness of the underlying model, but any information beyond that is not available. Compared to these approaches, our approach for Simulink models shows the following advantages: i) the ability to verify certain invariance properties (such as P2 from Section 6) over a complete reachable state-space or over bounded one and ii) ability to generate a counter-example in cases the property is violated. On the other hand, our approach is applicable to invariance properties, which represents a subset of all the possible properties that can be verified using any of the above model-checking-based verification techniques.

As other related approaches, we list the following. Reicherdt and Glesner [24] propose a theorem-proving approach based on the Boogie tool in order to check whether the model invariants are preserved in all possible executions. Despite having provided full automation for the transformation and analysis phases, the approach is suitable only for Simulink models composed of discrete-time feedthrough blocks only, which substantially limits its range of applicability. Liu et al.[20] combine statistical debugging and model slicing in order to improve fault localization in Simulink models, opening an interesting direction for searching the state space more efficiently, which might be compatible with our approach.

8 CONCLUSIONS

In this paper, we have presented an approach for invariance checking of Simulink models based on the principles of BMC using satisfiability modulo theories. Our main contributions are on two main fronts: first, we show that we can automatically generate finite execution paths based directly on the Simulink model, and second, we show that there are certain Simulink designs for which the bounded invariance procedure is complete. In order to determine whether the invariance checking is complete, we have identified the commonly used Simulink designs and block types based on two industrial models. We complement our approach with a tool, called SyMC which automatically generates the set of finite bounded paths of a Simulink models, calculates the CT for linear compositions and integrates Z3 such that the analysis can be automatically executed.

For determining the completeness of the bounded invariance checking for certain classes of Simulink models, we have first analyzed the structure of two industrial systems, namely ASL and BBW from Volvo GTT, Sweden, from which we have identified the commonly used block types and compositions of blocks. As a result, we have identified three broad categories of Simulink blocks: *feedthrough*, *delay* and *SFunction* and two common compositions: *linear* and *feedback-loop* compositions. Based on the identified compositions and blocks, we first prove the completeness of the bounded invariance checking procedure for the linear compositions, by proving the existence of a CT. For the feedback-loop compositions, a CT does not exist in the general case, however, there exists a special subclass of feedback-loop compositions for which a CT does exist. To automate the proposed approach, in this paper we have performed the following: i) we have introduced an automated procedure (implemented in the SyMC tool) for generating execution paths of finite length based on the Simulink model and an execution order of the blocks, ii) we propose a template-based encoding of the execution paths into the SMT-LIB format suitable for analysis using the Z3 SMT solver. For validation, we have applied our SyMC tool on the Simulink model of BBW, which we prove to be correct with respect to two safety properties.

There are several directions for future research around the approach proposed in this paper. First, we need to improve the SyMC tool on multiple fronts in order to: i) use the original Simulink model as an input into the tool, and ii) optimize the generated paths such that the same information is encoded using less states (less SMT variables and constraints). For the latter, we can consider symbolic execution [5] for removing unfeasible paths, and model slicing [23] for removing non-relevant parts with respect to certain properties. The proposed improvements could potentially enable us to consider more industrial systems for validation, in order to further test the boundaries of the applicability of our approach. Additionally, we also aim to implement the automation of the CT calculation for the feedback loop compositions. We also aim to tackle the problem of incompleteness of the invariance checking procedure for the feedback-loop designs by complementing the proposed approach with static analysis of the model [12]. Finally, we aim to investigate whether we can extend the approach for the verification of invariance properties with timing information.

ACKNOWLEDGEMENTS

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

REFERENCES

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

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2015. *The SMT-LIB Standard: Version 2.5.* Technical Report. Department of Computer Science, The University of Iowa. Available at www.SMT-LIB.org.

[3] Petr Bauch, Vojtěch Havel, and Jiří Barnat. 2016. Accelerating Temporal Verification of Simulink Diagrams Using Satisfiability Modulo Theories. *Software Quality Journal* 24, 1 (March 2016), 37–63. https://doi.org/10.1007/s11219-014-9259-x

[4] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, Yunshan Zhu, et al. 2003. Bounded model checking. *Advances in computers* 58, 11 (2003), 117–148.

[5] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering.* ACM, 1066–1071.

[6] Lennon Chaves, Iury Bessa, Lucas Cordeiro, Daniel Kroening, and Eddie Lima. 2017. Verifying digital systems with MATLAB. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis.* ACM, 388–391.

[7] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. 2015. HyComp: An SMT-based model checker for hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 52–67.

[8] Edmund M Clarke, E Allen Emerson, and Joseph Sifakis. 2009. Model checking: algorithmic verification and debugging. *Commun. ACM* 52, 11 (2009), 74–84.

[9] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proc. of the Theory and Practice of Software, 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08).* Springer-Verlag, Berlin, Heidelberg, 337–340.

[10] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability Modulo Theories: Introduction and Applications. *Commun. ACM* 54, 9 (Sept. 2011), 69–77.

[11] Leonardo De Moura, Harald Rueß, and Maria Sorea. 2003. Bounded model checking and induction: From refutation to verification. In *International Conference on Computer Aided Verification.* Springer, 14–26.

[12] Christian Dernehl, Norman Hansen, Thomas Gerlitz, and Stefan Kowalewski. 2015. Static Value Range Analysis for Matlab/Simulink-Models. In *Informatik 2015 : Tagung vom 28. Sep. - 02. Okt. 2015 Cottbus / Douglas W. Cunningham ... (Hrsg.)* (2015-09-28) *(GI-Edition : lecture notes in informatics)*, Vol. 246. Ges. fur Informatik, [Bonn], 1649–1660. https://publications.rwth-aachen.de/record/573834

[13] Predrag Filipovikj. 2018. SyMC. https://github.com/predragf/symc.

[14] Predrag Filipovikj, Nesredin Mahmud, Raluca Marinescu, Guillermo Rodriguez-Navas, Cristina Seceleanu, Oscar Ljungkrantz, and Henrik Lönn. 2017. *Analyzing Industrial Simulink Models by Statistical Model Checking.* Technical Report.

[15] Predrag Filipovikj, Nesredin Mahmud, Raluca Marinescu, Cristina Seceleanu, Oscar Ljungkrantz, and Henrik Lönn. 2016. Simulink to UPPAAL Statistical Model Checker: Analyzing Automotive Industrial Systems. In *FM 2016: Formal Methods*, John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.). Springer International Publishing, Cham, 748–756.

[16] Christian Herde, Andreas Eggers, Martin Fränzle, and Tino Teige. 2008. Analysis of hybrid systems using HySAT. In *IN ICONS '08: Proceedings of the Third International Conference On Systems.* 196–201.

[17] Daniel Kroening and Ofer Strichman. 2003. Efficient computation of recurrence diameters. In *International Workshop on Verification, Model Checking, and Abstract Interpretation.* Springer, 298–309.

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

[19] Florian Leitner-Fischer and Stefan Leue. 2008. Simulink Design Verifier vs. SPIN: a comparative case study. (2008).

[20] Bing Liu, L. Lucia, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. [n. d.]. Simulink fault localization: an iterative statistical debugging approach. *Software Testing, Verification and Reliability* 26, 6 ([n. d.]), 431–459. https://doi.org/10.1002/stvr.1605 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1605

[21] Matlab. 2018. Simulink Design Verifier Features. https://se.mathworks.com/products/sldesignverifier/features.html

[22] Stefano Minopoli and Goran Frehse. 2016. SL2SX Translator: From Simulink to SpaceEx Models. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control (HSCC '16).* 93–98.

[23] Robert Reicherdt and Sabine Glesner. 2012. Slicing MATLAB simulink models. In *Software Engineering (ICSE), 2012 34th International Conference on.* IEEE, 551–561.

[24] Robert Reicherdt and Sabine Glesner. 2014. Formal verification of discrete-time MATLAB/Simulink models using Boogie. In *International Conference on Software Engineering and Formal Methods.* Springer, 190–204.

[25] Peter Schrammel, Daniel Kroening, Martin Brain, Ruben Martins, Tino Teige, and Tom Bienmüller. 2015. Successful use of incremental BMC in the automotive industry. In *International Workshop on Formal Methods for Industrial Critical Systems.* Springer, 62–77.