2021 International Symposium on Theoretical Aspects of Software Engineering (TASE) | 978-1-6654-4163-6/21/$31.00 ©2021 IEEE | DOI: 10.1109/TASE52547.2021.00018

A Timed Automata based Automatic Framework for Verifying STL Properties of Simulink Models

Miao Tian[†], Jianqi Shi[*†], Zhe Hou[‡], Yanhong Huang[†], Shengchao Qin[§¶]
[†]National Trusted Embedded Software Engineering Technology Research Center,
East China Normal University, Shanghai, China
[‡]School of Information and Communication Technology, Griffith University, Brisbane, Australia
[§]School of Computing, Engineering, & Digital Technologies, University of Teesside, Tees Valley, UK
[¶]College of Computer Science & Software Engineering, Shenzhen University, Shenzhen, China
miao.tian@ntesec.ecnu.edu.cn, {jqshi, yhhuang}@sei.ecnu.edu.cn
z.hou@griffith.edu.au, S.Qin@tees.ac.uk

***Abstract*—Simulink has been widely used in model-based design and development. While we witness a growing demand on testing and verification for safety-critical systems, it remains a challenge to verify Simulink models, due largely to a lack of standardized formal semantics for Simulink. In this paper, we propose a comprehensive framework that allows us to automatically verify Simulink models. Our proposed framework is equipped with Signal Temporal Logic (STL) for system requirements specification and employs a formal method to translate Simulink models into UPPAAL timed automata, which can then be verified automatically by UPPAAL (against their STL specification). A novelty of our work is the integration of Simulink models with STL, allowing us to express and then verify complex time properties that may be found difficult by existing work. In our translation of Simulink models, we adopt symbolic execution to reduce the size of the translated automata that can produce accurate results. We also demonstrate the feasibility and effectiveness of the proposed framework via a case study of an autonomous driving system.**

***Index Terms*—Simulink model, Requirement, Signal Temporal Logic, Verification, UPPAAL Timed automata**

I. Introduction

MATLAB Simulink has been widely used to design and implement system models in various fields, especially in some safety-critical fields, such as avionics and autonomous driving. Beyond modeling and simulation, Simulink can also automatically generate C and HDL code from the model [1] and monitor model defects through Simulink Design Verifier [2].

Motivation: Usually, the model designed in Simulink is elaborated and expanded from the relatively straightforward requirements [3]. However, in practical work, the requirements are proposed by one group of engineers, and another group of engineers independently develops the design model. The correctness of the model largely depends on the design engineer's interpretation of the requirements. If the interpretation is wrong, the designed model may eventually cause system failures. Although Simulink Design Verifier can use Simulink verification blocks to describe requirement-related assertions and detect the violation of them, Simulink verification blocks have limited description capabilities for most complex and timing-related requirements and moreover cannot be used for verification. *Model checking*, a rigorous and correct formal technique to verify whether the properties of the model are satisfied, should be applied to Simulink model verification.

Challenge: One of the challenges is to choose an appropriate specification language and a model verification tool to realize verification automatically. Although the commonly used languages such as Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) have been supported by many model verification tools, they can not express the relationship between *real-valued signal* and time [4]. While languages such as Metric Interval Temporal Logic (MITL) and Signal Temporal Logic (STL) describe the real-time property well, but they lack the support of model verification tools [5]. Another major challenge is that Simulink has very rich blocks, which would cause the execution semantics of the model very complex and the application of formal techniques to complete verification directly very challenging. Furthermore, translating the model and requirements and finally unifying them into a suitable formal model is very important, but this process is very difficult, and few existing works complete it.

Approach: We present a fully automatic framework to verify whether the Simulink model meets the requirements. Our framework is shown in Fig. 1. STL is a popular formalism for specifying properties of dense-time real-valued signals, and it has been used in many real-life applications. Thus it is appropriate to choose STL to describe the system requirements. Timed automata [8] is chosen as the final form for unifying the Simulink model and requirement properties because it can be used to model and analyze the timing behavior of systems. Moreover, methods for verifying timed automata have been deeply studied by UPPAAL [7]. Specifically, we use MightyL [6], an open-source tool, to automatically translate a STL formula into UPPAAL timed automata. Then we design a translation scheme to automatically translate the Simulink model into UPPAAL time automata. Symbolic execution [9] is used in the translation method to make the result more accurate. In summary, contributions of this paper include:

- We design an automatic translation method with strict formal definition to translate the Simulink model into

*Corresponding author.

978-1-6654-4163-6/21/$31.00 ©2021 IEEE
DOI 10.1109/TASE52547.2021.00018

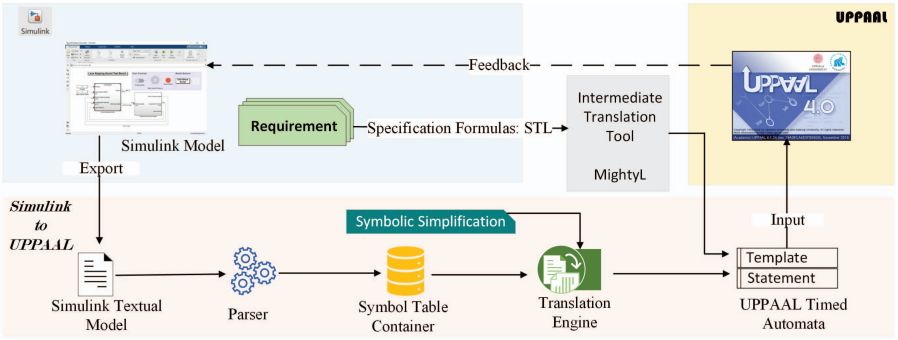

Fig. 1. The verification Framework for Simulink Models

UPPAAL, and apply symbolic execution to the generation of UPPAAL time automata.

- We propose a comprehensive and scalable framework to automatically verify whether a Simulink model meets the requirements (described as STL properties) through UPPAAL. To the best of our knowledge, we are the first to fully automatically verify the Simulink model without simulating the model at the time of verification.
- Experiments on the Lane Keeping Assist (LKA) system of the autonomous driving system demonstrate the feasibility of our method.

Organization: In Section II, we briefly introduce the basic knowledge related to the verification framework. Section III presents the method of translating the Simulink model into UPPAAL timed automata. We elaborate on the experimental details and verification results in Section IV. We discuss related work in Section V and conclude in Section VI.

II. BACKGROUND

In this section, we introduce the basis of our framework. Section II-A briefly introduces Simulink for system modeling. The STL we chose to describe the requirements will be described in Section II-B. Section II-C is the tool UPPAAL, which is a critical tool that we use to verify the systems.

A. Simulink

Simulink [1] is a graphical environment for model-based design integrated into MATLAB IDE by MathWorks. The essential elements of Simulink are *blocks*, which are connected by *lines*. Simulink provides a variety of implemented blocks, such as arithmetical blocks, logic blocks, and relational blocks. Simulink blocks fall into two categories: *nonvirtual blocks* that represent functions and play an active role in the simulation of a system, and *virtual blocks* which only build a model graphically without affecting the behavior of the model. A *subsystem* consists of interconnected blocks, which can be a set of atomic blocks or possibly other subsystem blocks.

B. Signal Temporal Logic

Signal Temporal Logic (STL) [5] is a popular temporal logic for specifying properties of real-valued signals and characterizing timed behaviors. It is widely used in analog circuits, systems biology, and hybrid physical systems. The grammar of STL is given by

$$\varphi := true \mid s_i \geq 0 \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \mathbf{U}_I \psi$$

where s_i are variables that represent real-valued signals, I is a non-singular interval over $\mathbb{R}^+$ with endpoints in $\mathbb{N} \cup \{+\infty\}$ or $[0, 0]$. φ and ψ are STL formulas. The validity of a formula φ with respect to a signal $S = s_0 s_1 s_2 ...$ at time t is defined inductively as follows:

$$\begin{aligned}
&S, t \models true \\
&S, t \models s_i \geq 0 \quad \text{iff} \quad s_i^S(t) \geq 0 \\
&S, t \models \neg\varphi \quad \text{iff} \quad S, t \not\models \varphi \\
&S, t \models \varphi \wedge \psi \quad \text{iff} \quad S, t \models \varphi \text{ and } S, t \models \psi \\
&S, t \models \varphi \mathbf{U}_I \psi \quad \text{iff} \quad \exists\, t' \in t + I \, s.t. \, S, t' \models \psi \text{ and} \\
&\qquad\qquad\qquad\quad \forall\, t'' \in [t, t'], \; S, t \models \varphi
\end{aligned}$$

We derive other usual operators as follows:

$$false := \neg true \quad \varphi \vee \psi := \neg(\neg\varphi \wedge \neg\psi) \quad \varphi \rightarrow \psi := \neg\varphi \vee \psi$$
$$\Diamond_I \varphi := true \, \mathbf{U}_I \varphi \quad \Box_I \varphi := \neg\Diamond_I \neg\varphi$$

where $\Box$ means 'globally', and $\Diamond$ means 'eventually'.

C. UPPAAL

Formally, the UPPAAL timed automata is a finite state machine extended with non-negative real-valued clocks. It can be defined as a tuple $\mathcal{A} = (L, l_0, X, A, I, \Delta)$, where L is a finite set of locations, $l_0 \in L$ is the initial location, X is a set of clocks, A is a set of actions, I is a set of invariants on the location, and $\Delta \subseteq L \times G \times A \times 2^C \times L$ denotes a set of transition edges, G is the set of guard conditions. The transition represented by an edge can be triggered when the clock value satisfies the guard labeled on the edge. The clocks may reset when a transition is taken.

III. THE PROPOSED APPROACH

Model checking is one of the most important and successful techniques to verify the correctness of a system automatically. A system *model* and a *formal property* that the system must respect should be provided to a model-checker for verification. For their implementation, many model-checkers rely on the *automata-based* approach. Therefore, to verify automatically, we provide a method to translate the Simulink model into the UPPAAL time automata. Simultaneously, the requirements

(negation) described by STL are translated into the UPPAAL time automata, and finally, complete the verification through UPPAAL. The working process is performed in following steps:

1. **Parsing the Simulink model:** The critical information of the model is extracted by parsing the textual representation (MDL file) of the Simulink model. Firstly, the system is divided into different parts according to the subsystems, and each part only extracts the relevant information of the nonvirtual blocks that have an actual influence on the model. Other graphical information related to the construction of the model, such as location, color, and font, are discarded.
2. **Formal modeling:** In this step, we give the formal definition of the Simulink model and the *Symbol Table* (described in subsection III-A).
3. **Translation:** We translate the Simulink model into UPPAAL timed automata using *translation scheme* (described in subsection III-B).
4. **Symbolic simplification:** Based on symbolic execution, we propose a simplification algorithm for calculating new locations and capturing the constraints of reaching locations. (described in subsection III-C).
5. **Verification:** The "property automata" and the "model automata" are executed synchronously in UPPAAL to complete the verification (described in subsection III-D).

A. Formal Definitions

To clarify the terms used in the framework and facilitate the construction of the translation scheme, we give the formal definition of the Simulink model and the Symbol Table in this section.

Definition 1. *A Simulink model is formally defined as a tuple* $SL = (D, B, C, TS, TC, Fun)$*, where*

- D *is a finite set of typed variables including input variables* D_I*, intermediate variables* D_X*, and output variables* D_O*,* $D = D_I \cup D_X \cup D_O$*.*
- B *is a finite set of Simulink blocks. Each block has input, output, and local variables. The input and output variables are associated with input and output ports. A Simulink block can itself be a Simulink diagram.*
- $C \subseteq B \times B$ *is an ordered relation that represents connections between the blocks. A connection* $c = (b, b') \in C$ *is a line from an output port of* b *to an input port of* b' *in the Simulink model, and represents the signal flow between the corresponding variables of* b *and* b'*.*
- TS *is the sample time of the Simulink model, which indicates when the model updates its internal states and produces outputs.*
- $TC : C \rightarrow TY$ *is a mapping from a connection to a time that represents the time taken by a block to generate output and pass it to another block,* $TY \in \{TY_{ct}, TY_{cm}, TY_{ctM}, TY_{cmM}\}$*. According to the type of signal flow between blocks, we divide the time into control time and communication time. Control time is the time taken to transmit the control signal between* b *and* b'*, such as the controller transmitting a signal to the actuator. If a communication signal is transmitted between* b *and* b'*, the time taken in this process is communication time. For example, the controller obtains the value from the sensor.* TY_{ct} *and* TY_{ctM} *are the set of control time, and* TY_{cm} *and* TY_{cmM} *are communication time.* TY_* *represents the minimum time required for the blocks to complete control or communication, and* TY_{*M} *is the maximum time.*
- Fun *is a set* $\{Fun_0, \cdots, Fun_n\}$*, where* Fun_i *captures the functionality of a Simulink block, and* i *marks the execution order of functions, consistent with the execution order of corresponding blocks in Simulink simulation.*

The Simulink model is not simulated during the translation. In order to track and record the changes of various variables after the execution of the block, we use symbols to mark the information in the Simulink model. Symbolic information of the Simulink model is recorded in the Symbol Table.

Definition 2. *A Symbol Table is formally defined as a tuple* $ST = (SI, SO, SX, SB, SFun)$*, where*

- SI*,* SO *and* SX *is a finite set of symbols of input, output and intermediate variables in the Simulink model, respectively.*
- SB *is a finite set of symbols of variables involved in each block in Simulink model.*
- $SFun : Fun_i \rightarrow SI \times SO$ *represents the symbolic values of the input and output variables of the model after the* Fun_i *is executed.* $SFun$ *records the update of the variables.*

B. Translation Scheme

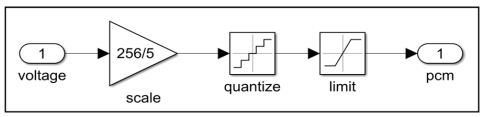

Fig. 2. Analog-to-Digital Converter (ADC) Model

Consider a Simulink model $SL = (D, B, C, T, Fun)$. SL can be systematically translated into the UPPAAL timed automata $\mathcal{A} = (L, l_0, X, A, I, \Delta)$ as below. We use the Analog-to-Digital Converter (ADC) model shown in Fig. 2 as the running example to explain the translation. This model is a subsystem of a Digital Thermometer model. The input *voltage* is converted from temperature, and the output *pcm* is the converted digital signal.

Location L: The set L of locations can be identified with the symbolic values of the input and output variables in the Simulink model. Let $v_1, \cdots, v_n$ be the input/output variables in D, and $Fun_0, \cdots, Fun_n$ be the function implementation of the corresponding block in Fun. We identify the symbolic values of the variables according to the mapping relationship in $SFun$ at the end of each block, denoted by $SFun(Fun_i) = V_1^i, \cdots, V_n^i$ for block $B_i \in B$. Then L is the set of all such tuples $(V_1^i, \cdots, V_n^i)$.

For instance, in the ADC model, we define the set of input variables $D_I = \{voltage\}$ and output variables $D_O = \{pcm\}$. These two variables are marked with the symbol VO and the symbol P, respectively, $SI = \{VO\}$, $SO = \{P\}$. Thus, $l_1 = (VO, P) \in L$. The ***scale*** in the ADC model is a Gain block that amplifies the voltage value. After the ***scale*** block is executed, $VO = 256/5 * VO$, the symbolic value of the input variable is updated. Therefore, $l_2 = (256/5 * VO, P) \in L$.

Initial Location l_0: We define the initial location as $l_0 = SFun(Fun_0) = (V_1^0, \cdots, V_n^0) \in L$, where Fun_0 is the initialization assignment function, and V_m^0 $(1 \leq m \leq n)$ is the initial symbol value of the variable. For example, in the timed automata translated from the ADC model, we define $l_0 = (VO_0, P_0) \in L$.

Clocks X: We identify the set X of real-valued variables from the *sample time* TS and the *time constraints* TC. The sample time determines the time of "a complete run" for the translated timed automata. So a *global clock* is needed to record the global time. For the time constraints TC on the connections, they are reflected as *local clocks* in the translated time automata. That is, the time taken on control and communication tasks also needs to be recorded by setting the corresponding clocks on the automata. For the sample time TS, we define a global clock g. For each connection $c_i \in C$, we set a corresponding local clock x_i according to the time $t_i = T(c_i)$ given by the mapping $T : C \to TY$. The clock is defined as $X = \{g, x_1, \cdots, x_m\}$, where m is the number of connections in C.

In our example, The value of *voltage* is obtained from the sensor. The time taken waiting for the sensor's signal is the *communication time*, which requires a clock x to record, and the remaining blocks are computational blocks without control or communication tasks. Therefore, $X = \{g, x\}$ in the timed automata translated from the ADC model.

Action A: For each Simulink block, we define and implement a *function* to realize the process of getting output from input variables and intermediate variables. Specifically, we design a *library of routines* LR to generate the *function* equivalent to basic blocks in Simulink, $LR(B_i) = Fun_i$ for block $B_i \in B$. Then the set of these functions is Fun. The function will update the variables, so in the translated time automata, we defined $Fun \in A$. Besides, in order to preserve the execution order of blocks, we define an integer variable $exeNum$ in the UPPAAL model to record the execution order of functions. Then we set an update operation $exeNumOp$ for the variable $exeNum$, and $exeNumOp \in A$. So $A = Fun \cup exeNumOp$.

For the ADC model, we respectively define the corresponding functions of ***scale***, ***quantize***, ***limit*** as $F = \{Fun_1, Fun_2, Fun_3\}$. There are three blocks in the model, so the value range of $exeNum$ is $[0, 3]$. The set of actions of the timed automata translated from ADC model is $A = \{Fun_1, Fun_2, Fun_3, exeNum = i\}$, where $i \in [0, 3]$.

Invariants I: Locations are labeled with invariants in the UPPAAL timed automata. An invariant is an *expression on clocks* that limits the time that the automata stay in a location. TY_{cmM} and TY_{ctM} in the TC are the maximum time to complete communication and control tasks, respectively, so they should be the *upper bound* of the invariant. It means that transition must occur if the local clock exceeds the maximum time. For each connection $c_i \in C$, we convert the time $t_i = T(c_i)$ given by the mapping $T : C \to TY$ into a invariant $x_i \leq t_i$ if $t_i \in TY_{cmM}$ or $t_i \in TY_{ctM}$. Therefore, the invariant on a location l is defined as

$$I(l) = \begin{cases} x_i \leq TY_{cmMi}, & \text{if } c_i \text{ is a communication type;} \\ x_i \leq TY_{ctMi}, & \text{if } c_i \text{ is a control type.} \end{cases}$$

The ***scale*** block waits for the signal from the sensor for no more than 5 time units in the ADC model. We can define the invariant on the location $I(l_1) = x \leq 5$.

Transition Δ: Transition is the set of edges. Edges are annotated with *selections*, *guards*, *synchronizations* and *updates* in UPPAAL. *Selections* non-deterministically bind a given identifier to a value in a given range. The other three labels of an edge are within the scope of this binding. A *guard* is a boolean expression, and an edge is enabled in a location if and only if the guard evaluates to be true. *Channels* are used to synchronise processes. An *update* label is a list of expressions. When executed, the update expression of the edge is evaluated. We do not set the *selections*, and we do not use the *channel synchronization mechanism* (explained further in Section III-D). So we only consider *guards* and *updates* on edge in our translation scheme.

A *guard* $G = \{B(X) \wedge BoolExp\}$ is a conjunction of simple conditions on clocks, differences between clocks, and boolean expressions not involving clocks. The set $B(X)$ of *clock constraints* g_1 over X is defined by $g_1 := true \,|\, g_1 \wedge g_1 \,|\, x \bowtie c \,|\, x - y \bowtie c$, where $\bowtie \in \{\leq, <, \geq, >\}$, $x, y \in X$. We set the c according to the time $t_i = T(c_i)$ given by the mapping $T : C \to TY$ as

$$c = \begin{cases} TY_*, & \text{if } t_i \text{ is a minimum time;} \\ TY_{*M}, & \text{if } t_i \text{ is a maximum time.} \end{cases}$$

$BoolExp$ is a set of branch conditions and conditions of judgment function execution order g_2. g_2 is defined as $g_2 := true \,|\, g_2 \wedge g_2 \,|\, branchCon \,|\, exeOrdCon$. The $branchCon$ is the constraint conditions captured from the function Fun_i through the method of symbolic execution (discussed in Section III-C). The $exeOrdCon = (exeNum == i)$ is a boolean expression that determines whether the $exeNum$ is the correct value, where i is the number of execution order.

Now we discuss the *updates* on the edge. The update expressions of the edge are evaluated when the guards are true, and the transition occurs. Therefore, the actions to be performed during the transition should be added to the update list. Besides, the update list also includes local clocks reset. For each local clock x_i, the reset operation is defined as $x_i := 0$.

Then the set of reset clock $clockRe$ is all such reset operations, where $x_i \in X \backslash g$. So $updates = A \cup clockRe$.

In summary, the *transition* $\Delta \subseteq L \times G \times A \times 2^C \times L$ is a set of edges between locations annotated with a *guard* and an *update* (including actions A and a set of clocks to be reset 2^C). We divide the *transition* into two types:

$$l_0 \xrightarrow{true,\ Fun_0,\ exeNum = 1} l'$$
$$l \xrightarrow{G,\ Fun_i,\ exeNum = i+1,\ x_i := 0} l'$$

The first transition represents the *initialization* of the model, fired at the initial location. The initialization process is unconstrained, $G = \{true\}$. The second transition is the *operation-type*. When the time and condition constraints G are evaluated to be true, the function Fun_i corresponding to the current order number i is executed. Note that location l' can be the same as l in the second transition if the symbolic values of input and output variables do not change after the execution of the Fun_i.

According to the above transition scheme, the ADC model is translated into the time automaton in UPPAAL, as shown in Fig. 3. The green label on the edge is *guard condition*, and the blue label is *update action*. Specifically, *fun1*, *fun2*, and *fun3* correspond to the execution of ***scale***, ***quantize***, ***limit*** blocks in the ADC model.

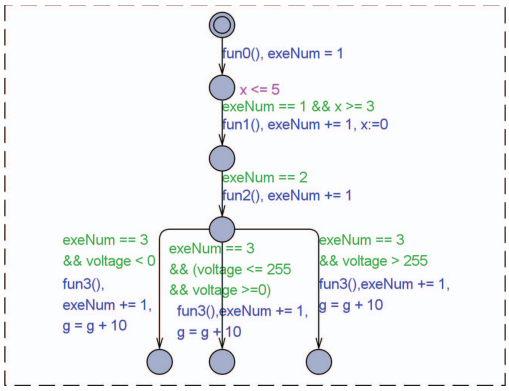

Fig. 3. The Timed Automaton of ADC Model

C. Symbolic Execution for Computing New Locations and Constraints

This section introduces the *symbolic execution* algorithm. We design this algorithm to simplify the symbolic values of variables and determine the generation of a new location according to the simplification results. Our algorithm reduces the number of states as much as possible to avoid *state explosion*. There are many condition blocks in the Simulink model, and choosing different conditions will generate different outputs. So we also use this algorithm to capture the constraints of reaching a location (*branchCon* in G).

For example, a location l is marked by a variable $y = Y$, where Y is a symbolic value. Assuming perform the action $y = y + 2$, and then l translates to l_1 that is marked by $y = Y + 2$. Then, the action $y = y - 2$ is performed on the next transition, where y is updated to $Y + 2 - 2 = Y$. However, the location marked by the symbolic value Y already exists, so we do not need to add a new location l_2, and the location should transition from l_1 to l.

We have preprocessed the function set Fun to mark the variables, the position of the statement, the statement type, and the execution flow in the function. The computation method is shown as **Algorithm 1**. The ξ is a path constraint, which collects the conditions of reaching a specific location along the execution path. ST is the Symbol Table for recording the symbol value of each variable, and α is the program counter, pointing to the position of the currently executed statement. Symbolic execution is performed using the *symbolic memory*. For the memory M_1, we write $M_1[\xi \rightarrow ST]$ for mapping the constraint to symbol values of variables for recording the state of variables on the constraint. $M_2[\alpha \rightarrow ST]$ maps the program position to symbolize variables for recording the state of variables after executing a statement. Execution starts from a Symbol Table ST_0 containing initial symbolic values for input and output variables and a path constraint ξ recorded as true, at the entry position PC_0. The operation of each statement updates the memory and the control position.

According to the operation, we divide program statements into three types: *termination statement*, *assignment statement*, and *conditional branch statement*. For an assignment, the symbolic values of the variables are updated by the function *updateSymbol*(α, ST), which evaluates the assignment expression. The $labelVariable$ is a tuple to separately record the updated symbolic values of the input and output variables that mark the current location, and it is obtained from the $updateInOut(ST)$. Then we update M_1 and M_2 according to the updated ST. For a conditional branch statement, we record all the conditions. ξ_i represents a new path constraint obtained by updating the original path constraint ξ with the ith condition $condition_i$ that is achieved by $addConstraints(condition_i, \xi)$. Then the memory M_1 is updated. The $newlocation$ is the set of symbolic values of input and output variables corresponding to all existed locations. When the termination statement is executed, we search in set $newlocation$ to check if there is an element that is the same as the tuple $labelVariable$. If $findLocaion(labelVariable, newlocation)$ returns true, there is no need to generate a new location after this function is executed. The algorithm only returns all path constraints at the time. If no equivalent location is found, then this function execution have generated a new location. We add the new location to $newlocation$ by $addLocation(labelVariable, newLocation)$.

D. Verifying with STL

Before translating the Simulink model into the time automata in UPPAAL, we use the STL formula to describe the requirement and translate the STL formula into the time automata in UPPAAL. We use MightyL, an academic tool, to translate the STL formula to UPPAAL time automata. However, MightyL can only translate a *Metric Interval Temporal*

Algorithm 1: Computation of locations and constraints

$\xi := true;\ ST := ST_0;\ \alpha := PC_0$
$M_1(true) := ST;\ M_2(\alpha) := ST$
function symbolCom(programCounter α, symbolTable ST, constraints ξ)

```
1: if typeOp(α) ≠ endProgram then
2:   switch typeOp(α) do
3:     case assignment
4:       ST = updateSymbol(α, ST);
5:       labelVariable = updateInOut(ST);
6:       M1 := M1[ξ → ST]; M2 := M2[α → ST];
7:       α := α → next;
8:       symbolCom(α, ST, ξ);
9:     end
10:    case ifElseBranch
11:      for all condition in condtionBranch do
12:        ξi := addConstraints(conditioni, ξ)
13:        M1 := M1[ξi → ST];
14:        α := α → conditionGoto;
15:        symbolCom(α, ST, ξi);
16:      end
17:    end
18:  end
19: else
20:  if findLocation(labelVariable, newLocation) then
21:    return M1;
22:  else
23:    newLocation :=addLocation(labelVariable,
       newLocation);
24:    return M1;
25:  end if
26: end if
```

Logic (MITL) formula into time automata in UPPAAL, so we first translate the STL formula to the MITL formula.

The syntax and semantics of MITL can be extended to *real-valued signals* through STL. Therefore we can define STL over MITL and thus translate STL to MITL. Firstly, we give the syntax of MITL as follows:

$$\varphi := true \mid a \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \mathbf{U}_I \psi$$

where $a \in AP$, which is the set of atomic propositions. I is a nonsingular interval over $\mathbb{R}^+$. This includes bounded intervals $[a, b]$ and unbounded intervals $[a,\ +\infty)$ for any $0 \leq a < b$. Compared with the syntax definition of STL, MITL is based on *atomic propositions*, while STL is based on *real-valued variables*. Therefore, to translate STL to MITL, it is necessary to map real-valued variables to atomic propositions.

We define a real-valued signal $S : \mathbb{T} \rightarrow \mathbb{R}^m$, and a set of predicates $P = \{p_1, ..., p_n\}$, where $p_i : \mathbb{R}^m \rightarrow \mathbb{B}$, $\mathbb{B} = \{true, false\}$. We define the STL formula Φ_{STL} over predicates P using MITL formula Φ_{MITL} over the atomic propositions AP. The semantics of STL can be defined using MITL as follows:

- Define a set of AP such that for each $p \in P$, there exists some $a_p \in AP$;
- For each signal S we define a μ such that $a_p \in \mu(t)$ iff $p(s(t)) = true$;
- $\forall t(s, t) \models \Phi_{STL}$ iff $(\mu, t) \models \Phi_{MITL}$.

According to the above definition, we project the STL *predicate expressions* into *atomic propositions* with independent truth valuations. Then we create the corresponding MITL formula from STL, and we do this automatically by coding. The subsequent translation work is done automatically by MightyL. However, the work of MightyL has certain limitations. The time automata translated into UPPAAL use global variable synchronization instead of the channel mechanism. Therefore, before translating the Simulink model to the UPPAAL automata, we extract the global variable information in the "requirement automata" obtained by MightyL. Then we set the global variables of "Simulink model automata" according to this information to realize synchronization among all automata. Finally, some properties are added to UPPAAL to describe that all automata can reach an acceptable termination state, and the verification result is obtained by verifying these properties.

IV. EXPERIMENT

In order to evaluate the translation scheme and the verification framework, we apply them to some artificial and real industrial Simulink models. We choose two cases which are the Digital Thermometer (DT) model, and the Lane Keeping Assist (LKA) system [18]. In this section, we provide a brief overview of our results.

The first artificial example is the Digital Thermometer model composed of a simple temperature sensor and an ADC. The ADC model shown in Fig. 2 is a subsystem of the DT model. We verify 1) the thermometer receives the voltage data $VolData$ from the sensor and converts it into the corresponding temperature value $TempVal$ within 10 time units, and 2) the exception flag $ExeFla$ is 1 finally when the $VolData$ exceeds the threshold $ThreVal$. In Table I we provide verification results. Since the model does not deal with exception values, the second requirement is not satisfied.

TABLE I
PROPERTY LIST AND RESULTS OF VERIFICATION FOR DT MODEL

RQ	Temporal Logics	Result
1	$\Box_{[0,10]}((VolData \geq 0 \wedge VolData \leq 5) \rightarrow\ (TempVal \geq -40 \wedge TempVal \leq 40)$	true
2	$\Diamond((VolData \geq ThreVal) \rightarrow ExeFla > 0)$	**false**

Then, we apply our framework to the LKA system in the autonomous driving system. Since this model is representative, we take this model as an example to explain our translation and verification process in detail.

*a) **The LKA system**:* A lane-keeping assist (LKA) system is a control system that aids a driver in maintaining safe travel within a marked lane on a highway. The LKA system detects when the vehicle deviates from a lane and automatically adjusts the steering to restore proper travel inside the lane without additional input from the driver.

*b) **Requirements**:* We mainly refer to ISO 11270:2014 Intelligent transport systems — Lane-keeping assistance systems (LKAS) — Performance requirements and test procedures [10] to design the requirements. We divide the requirements into two categories: *warning requirements* describe the system requirements when the LKA system automatically sends out a warning signal of vehicle deviation, and *road performance requirements* describe the requirements that the LAK system should meet when it adjusts the driving state of the vehicle on the road. The standard covers 5 *warning requirements* and 10 *road performance requirements*. Here, we present two requirements of them in natural language:

- Warning requirement (**WR1**): The warning signal must be sent out within 3 time units after detecting that the vehicle is at the warning line. And if the driver has turned on LKA, the system intervention must be activated within 2 time units after the warning signal is sent out.
- Road performance requirement (**RR1**): If the distance between the measured lane and the wheel exceeds the safe lateral distance (1m), LKA will control the controller within 100 time units to adjust the wheel direction.

*c) **Translation**:* First, we use STL formulas to describe requirements, as shown in Table II. In **WR1**, $LSensor$ represents the data detected by the sensor, $DeparDet$ represents the warning signal of detected offset, $Enable$ corresponds to whether LKA is enabled, and $Status$ represents the execution status of the LKA system. For **RR1**, $LeftLaOff$ and $RigLaOff$ are the distances between the left and right wheels and the left and right lanes, respectively. $SafeLa$ is the safe lateral distance, and $StAng$ is the wheel steering angle adjusted by LKA. Then we use the MightyL tool to translate the requirements described by STL into the time automata in UPPAAL as described in Section III-D.

Next, we translate the Simulink model of the LKA system into UPPAAL. The hierarchical Simulink model for the LKA system consists of 516 blocks, in which only 216 nonvirtual blocks such as Gain, Logical Operator, and Relational Operator are considered in the conversion. The remaining 300 virtual blocks that define the model's structure are removed during the translating, such as InPort, OutPort, BusCreator, BusSelector, and Terminator. We divide the LKA model into five subsystems and finally obtain a time automata network in UPPAAL using the translation method of Section III-B and Section III-C.

TABLE II
PROPERTY LIST AND RESULTS OF VERIFICATION FOR LKA MODEL

RQ	Temporal Logics	Result
WR1	$\Diamond((LSensor > 0 \rightarrow \Diamond_{[0,3]} DeparDet > 0) \wedge ((Enable > 0 \wedge DeparDet > 0) \rightarrow \Diamond_{[0,2]} Status > 0))$	true
RR1	$\Box((LeftLaOff < SafeLa \vee RigLaOff < SafeLa) \rightarrow \Diamond_{[0,100]}(StAng - 0.5 \leq 0))$	true

*d) **Verification**:* After the Simulink model and STL requirements are translated into timed automata, we perform the verification using UPPAAL. Generally, UPPAAL declares the properties to be verified in the '.q' file, which essentially states a timed word that leads both the model and the property timed automata to accepting locations. So after the translation, our framework also generates a '.q' file, which describes the accepting locations of "model automata" and "requirement automata". When UPPAAL verifies this .q file and returns *satisfied*, it means that the model meets the requirement. In Table II we provide verification results for requirements **WR1** and **RR1**. Based on the physical limitations of the car, the steering angle is constrained to the range $[-0.5, 0.5]\ rad/s$. So for the requirement **RR1**, in addition to verifying whether the LKA system meets the necessary standards, we also verify the safety limits of the LKA system by adjusting $SafeLa$. Table III presents an excerpt of the verification results.

TABLE III
VERIFICATION RESULTS OF SAFE LIMITS FOR LKA MODEL

$SafeLa$ **Distance (m)**	**Result**	$SafeLa$ **Distance (m)**	**Result**
0.75	true	0.67	true
0.50	true	0.37	true
0.25	**false**	**0.35**	**false**

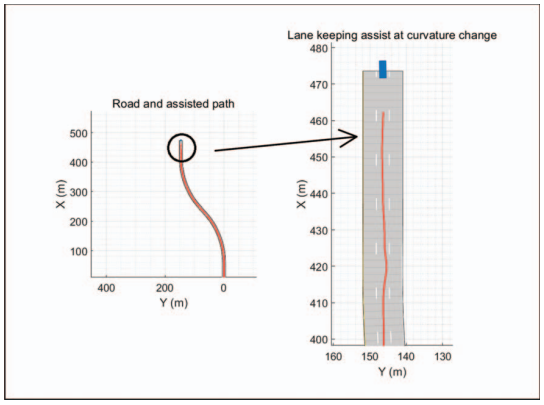

Fig. 4. Simulation driving of the vehicle within safety limits ($SafeLa = 1m$)

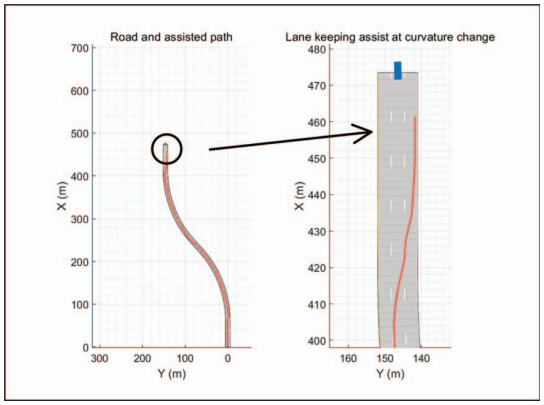

Fig. 5. Simulation driving of the vehicle exceeding safety limits ($SafeLa = 0.25m$)

In order to show the correctness of our verification results more intuitively, we also simulate the LKA model in Simulink, and the simulation results are shown in Fig. 4 and Fig. 5. The red curve represents the driving path of the vehicle.

Obviously, the LKA system can keep the vehicle traveling along the centerline of its lane taking $SafeLa = 1m$, while the vehicle deviates from the lane on the same road segment taking $SafeLa = 0.25m$. The simulation result is consistent with our verification result for **RR1** in Table II and Table III.

We compare the experimental results of the two models and show the results in Table IV. Although the number of blocks in the LKA model is much greater than the number of blocks in the DT model, the number of automata after the LKA model translation does not increase by multiples. The result shows the practical feasibility of our verification framework.

TABLE IV
COMPARISON OF EXPERIMENTAL RESULTS

The Model	Number of Blocks	Number of Automata	Time
DT	**7**	**3**	1.05s
LKA	**216**	**14**	28.30s

At present, the translation time is mainly spent on generating the functions corresponding to the blocks. Because execution semantics of Simulink are described in informal natural languages based on examples, we have not formally proven the equivalence of the translation. We currently acquire correctness by carefully compare simulation results of the translated model, including the value and state sequence step by step.

V. RELATED WORK

There are some works on the formal analysis and verification of Simulink models in related industries and academia. Commercial tools like SCADE design verifier [12], Embedded Validator [11] support formal verification of Simulink models against safety properties and deal with mainly blocks from the discrete library. Nejati et al. [15] present an industrial Simulink model benchmark, and prove the effect of applying model checking to identify requirements violations in the benchmark models. Bartocci et al. [14] propose a procedure to debug Simulink models for locating the fault, and they use STL specifications. Another research [13] apply the STL quantitative semantics, and generate a monitor to check the properties of the Simulink model at run-time. There are also some works to translate the Simulink model into the existing tools for verification. In [16], the Simulink model is translated into NuSMV for verification. And [17] provides a tool that translating the Simulink model into UPPAAL SMC.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a comprehensive framework to verify whether the Simulink model meets the requirements automatically. In order to cover some necessary real-time properties, we use Signal Temporal Logic to describe the requirements. The STL requirements are translated into timed automata by MightyL, and the Simulink model is also translated into timed automata using the formal method. **Compared to the existing work, our framework is more complete and scalable and can fully automatically verify the Simulink model without running the Simulink model. Further, our translation and verification process are all based on rigorous formal methods**. The ongoing works mainly focus on the following aspects: (1) Establish a simulation relationship between the execution semantics of the Simulink model and the execution semantics of the UPPAAL time automata to prove the correctness of the translation. (2) Extend the current translation scheme to support verification of blocks containing machine learning components in addition to basic blocks. (3) Design a new method to translate STL formulas into UPPAAL time automata directly.

ACKNOWLEDGMENT

This work is partially supported by NKRDP (No. 2019YFB2102602) and NSFC (No. 61772347).

REFERENCES

[1] J. B. Dabney and T. L Harman, "Mastering Simulink," Pearson/Prentice Hall, 2004.
[2] G. Hamon, et al., "Simulink design verifier-applying automated formal methods to simulink and stateflow," in Third Workshop on Automated Formal Methods, 2008, pp. 1–2.
[3] J. Yang, J. Bauman, and A. Beydoun, "Requirement analysis and development using MATLAB models," SAE Int. J. Passeng. Cars – Electron. Electr. Syst, vol. 2, pp. 430–437, January 2009.
[4] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in FORMATS/FTRTFT, 2004, pp. 152–166.
[5] T. Brihaye, G. Geeraerts, H.-M. Ho, and B. Monmege, "Timed-automata-based verification of MITL over signals," in 24th International Symposium on Temporal Representation and Reasoning, 2017, pp. 7:1–7:19.
[6] T. Brihaye, G. Geeraerts, H.-M. Ho, and B. Monmege, "MightyL: a compositional translation from MITL to timed automata," in International Conference on Computer Aided Verification, 2017, pp. 421–440.
[7] J. Bengtsson, K. Larsen, F. Lsrsson, P. Pettersson and W. Yi, "UPPAAL—a tool suite for automatic verification of real-time systems," in International hybrid systems workshop, 1995, pp. 232–243.
[8] R. Alur, "Timed automata," in Computer Aided Verification, 1999, pp. 8-22.
[9] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," Acm Sigplan Notices, vol. 47, no. 6, pp. 193–204, June 2012.
[10] ISO 11270:2014(en), "Intelligent transport systems — Lane keeping assistance systems (LKAS) — Performance requirements and test procedures," International Organization for Standardization, 2014.
[11] Embedded Validator web page: http://www.dspaceinc.com/ww/en/inc/home/products/sw/pcgs/automatic model validation.cfm.
[12] J.L. Camus and B. Dion, "Efficient development of airborne software with scade-suite," Esterel Technologies, vol. 62, 2003.
[13] A. Balsini, M. DiNatale, M. Celia, and V. Tsachouridis, "Generation of Simulink monitors for control applications from formal requirements," in 2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES), IEEE, 2017, pp. 1–9.
[14] E. Bartocci, T. Ferrère, N. Manjunath, and D. Ničković, "Localizing faults in Simulink/Stateflow models with STL," in Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control, CPSWEEK, 2018, pp. 197–206.
[15] S. Nejati, et al., "Evaluating model testing and model checking for finding requirements violations in Simulink models," in Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, 2019, pp. 1015–1025.
[16] B. Meenakshi, A. Bhatnagar, and S. Roy, "Tool for translating simulink models into input language of a model checker," In International Conference on Formal Engineering Methods, Springer, 2006, pp. 606–620.
[17] P. Filipovikj, et al, "Simulink to UPPAAL statistical model checker: analyzing automotive industrial systems," in International Symposium on Formal Methods, Springer, 2016, pp. 748–756.
[18] Lane Keeping Assist System web page: https://ww2.mathworks.cn/help/mpc/ug/lane-keeping-assist-with-lane-detection.html