Generation of Simulink Monitors for Control Applications from Formal Requirements

Alessio Balsini Marco Di Natale
Scuola Superiore Sant'Anna
Pisa, Italy
Email: alessio.balsini marco.dinatale @santannapisa.it

Marco Celia Vassilios Tsachouridis
UTRC Research
Cork, Ireland
Email: CeliaME TsachoV @utrc.utc.com

***Abstract*—The increasing complexity of embedded systems requires an improved capability of detecting and fixing errors. The availability of a modeling environment like Simulink allows the verification by simulation or model checking of system properties and of the correct behavior of the design. This verification is possible upon condition that the requirements are expressed in a formal way.**

Test and verification in Simulink is often a time-consuming process that requires the systems developers to translate requirements in model blocks for the verification. The capability of performing such translation is seldom available and prone to translation and interpretation errors.

We present in this paper a monitor generation tool and a Simulink library that enable a methodology to translate requirements in structured natural language into formal Signal Time Language (STL) constraints, leading to the automatic generation of Simulink monitors that check at run-time the desired properties. The tool automatically creates and connects the monitor blocks to a target Simulink model.

I. Introduction

Model-based development of embedded systems is today an established industrial practice. The use of models allows a precise and formal definition of the behavior with respect to time and also allows to raise the level of abstraction of the controller logic allowing verification by model checking and by simulation.

Simulink by the Mathworks [1] is among the most popular modeling environments. The Simulink models used for the representation of cyber-physical systems are based on a synchronous reactive semantics. The model of the controlled (physical) system is defined by a system of differential equations, integrated in continuous time, while the model of the controller is typically discrete-time.

Simulink allows for model verification of discrete-time models using formal proofs through the Simulink Design Verifier add-on (which is internally based on the Prover engine [2]) and supports checking asssertions at simulation time using a simple library with basic assertion blocks.

This leaves the system developers with the task of bridging the gap between the requirements (often expressed in natural language) and the definition of monitors that check the requirements constraints at simulation time and possibly also at run-time. This process can be divided in steps. First, the requirements need to be translated from the natural language into a formal language. To ease this translation, the natural language can be constrained to be semantically as close as possible to a suitably selected formal language. Once the requirements are expressed formally, the language can be used to verify the correctness of the system model offline by model checking or theorem proving, or the constraint formulas can be parsed to automatically generate monitors that check them on-line (while the simulation is running) or off-line on the execution traces.

A temporal logic is a language in which formal specifications can be written for computer systems. In the late 70s, Amir Pnueli [3] introduced temporal logic to reason formally about the temporal behaviors of reactive systems. In the Linear Temporal Logic LTL [3] and the Computation Tree Logic or CTL [4], time is implicitly represented as an enumerated sequence of reaction steps occurring in a discrete time space. These temporal logics were developed to check properties in (typically hardware) systems with boolean, discrete-time signals and focused on the verification, specification, and synthesis of concurrent systems.

Other models and languages were later developed [5], [6] to improve the expressive power of LTL and CTL and to define and verify properties in real (continuous) time as applied to hybrid systems.

Today, there are several examples of temporal logic, differing in the model of time, the semantics of reactions and the language that can be used to define properties and constraints. The Property Specification Language PSL [7] is an extension of LTL in which constraints are composed of boolean expressions written in the host language (often VHDL or Verilog) together with temporal operators and sequences to describe the relationship between states over time. The Metric Temporal Logic (MTL) [5] allows reasoning over Boolean signals over dense-time domains and the Signal Temporal Logic (STL) [6] was proposed in the context of analog and mixed-signal circuits as a specification language for constraints on real-valued signals defined in continuous time.

The verification of timed properties using these languages has been studied in depth and so the possibility of using techniques for monitoring the properties off-line on system traces or on-line using monitors at simulation time. The general verification problem is discussed in several surveys and books such as [8]. Other works discuss the application of formal verification (by model checking) to systems with STL

constraints. A recent work on this subject is [9].

A formal language like STL also offers the option of generating monitors for checking the properties of a simulated system. Offline techniques for monitoring STL properties on execution traces are discussed in [10]. This is an example of timed pattern matching, which consists in finding all segments of a continuous-time boolean signal that match a pattern defined by a timed regular expression. This problem has been formulated and solved in [11] via an offline algorithm that takes the signal and expression as inputs and produces the set of all matches.

Another possibility is the automatic generation of monitors that can be used to check properties at run-time, that is, while the simulation is running. In the context of timed regular expressions [12] an online matching algorithm has been presented in [13], but an on-line monitor generation technique is still not explicitly available for STL.

Finally, while the use of a formal temporal logic allows in principle the use of automatic verification techniques, bridging the gap between informal requirements and formal statements is not an easy task. Libraries and automatic implementation techniques can be used to ease the use of STL formulas in designs. In [14] Kapinsky et al propose the use of STL to verify typical control constraints in automotive applications modeled in Simulink, However, despite the title of the work, a library implementing the sample STL constraint presented in the paper is not described, nor it is available.

As for the problem of translating informal requirements into formal (possibly STL) constraints, several approaches are possible. It is possible to parse the natural language to extract formal predicates (as in [15] or [16], with a more recent discussion of the possible approaches in [17]), or to restrict the natural language (using editors or forms) in such a way that only a readable form of formal statements (typically constructed by replacing the formal language operators with natural language tokens) is allowed. A comparison of the two approaches is presented in [18].

An example of controlled composition of natural language tokens is described also in [19], in which the requirements formulation approach is coupled with the proposal of a contract language for the expression of requirements.

Paper contributions.

The purpose of this work is not to provide a formal language or a formal extension of existing methods, but rather to provide a usable tool and library to improve the applicability of existing langages, methods and techniques.

This paper presents an open source tool that generates Simulink monitor blocks for the validation at simulation time of a given models against constraints expressed according to a restriction of the STL language. The blocks are generated according to rules expressed as STL formulas and the monitor generation makes use of a set of library blocks that provide an implementation of the STL operators and an implementation of the typical control constraints described in [14].

Thanks to the availability of the source code and the modular structure of the project, the user can customize the tool and the library by directly accessing the software classes. It is thus possible to modify or improve the tool to extend the supported formal languages, or support other environments in addition to Simulink.

Paper organization. The remainder of this paper is organized as follows. Section II presents the proposed methodology from the requirements editor to the generation of the monitors. Section III introduces the STL language and the restriction of the language currently supported by the generation tool. Section IV provides an overview of the tool, from a high level user perspective to some of the internals and implementation details. This section also presents the typical user interaction with respect to a given model in order to (i) define the STL constraints, (ii) create Matlab code for the generation of the monitor blocks, (iii) add the monitor blocks to the model and connect them to the model signals. Section VI provides a description of the Simulink library developed as a support for the generation of monitors and finally, Section VII provides a discussion of a simple example to show the applicability of the proposed tool. Section VIII concludes the paper and highlights some future work.

II. From Requirements to Monitors

The work described in this paper is part of a general framework that is meant to improve the quality of the requirements and automate their translation into runtime Simulink monitors. A graphical description of the methodology is shown in Figure 1 The framework is centered on the availability of STL specifications (actually using a restriction of the STL language) that express the constraints to be verified on the system. From the STL specifications, a tool automatically generates monitors that are automatically connected to the model signals to check the correct behavior of the system at simulation time. The monitors are generated using elements from a purposely developed Simulink library (freely available from [20]), that provides, among other things, a practical implementation of the library proposed in [14].

The following sections describe the methodology and the tools to generate the monitors from STL statements. However, in this section we provide a short description of the other stages of the process to provide some context to our work. These stages and tools (currently under development) are a first step to address the problem of bridging the gap from natural language requirements to the generation of monitors in Simulink.

To restrict the scope of the work to a manageable size, we are initiall targeting typical control requirements, of the type presented in [14]. A typical requirement expressed in natural language (for a control application) is the following.

The Driver Subsystem (DRV) shall accelerate the motor from zero to x1 rpm in less than t1 sec, with an overshoot of less than x2 rpm.

We developed a customized Eclipse editor that supports the user in writing structured requirement by separating assumptions from assertions. The editor provides syntax highlighting, context help and direct access to a library of symbols (of

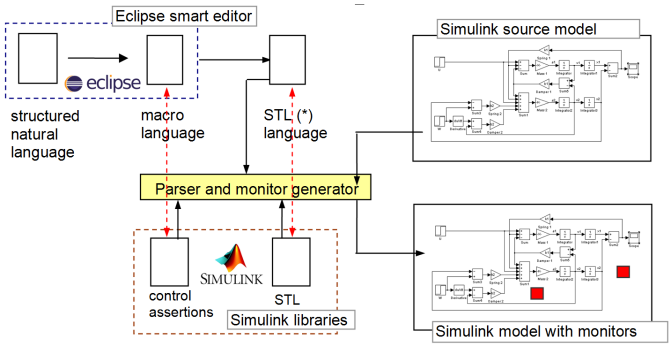

Figure 1. The framework for monitor generation from requirements.

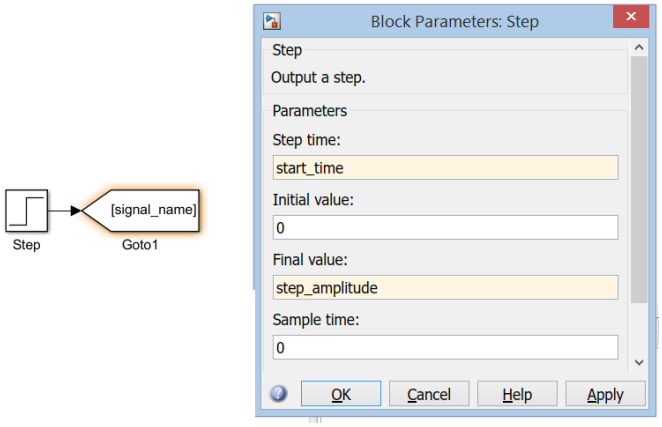

Figure 2. Step signal generator.

signals, subsystem and parameter identifiers) in an attempt to enforce the definition of requirements in a structured language with predefined natural language sentences or tokens, and the use of names from a data dictionary.

The informal requirement shown as an example then becomes.

R1. Acceleration bound and overshoot

Assumption:

The system inertia `sys_inertia` ***is less than or equal to*** `i1` ***and*** `reference` ***is a*** **`step with amplitude`** `A`.

Assertion:

Inside the Driver Subsystem (`DRV`), the speed (`spd`) signal **`shall rise from 0 to x1 in less than t1`** ***and*** **`the overshoot shall be less than x2`**.

In the new requirement formulation, the fixed size font indicates names of signals or parameters, the fixed size font in bold indicates macros; the italics bold indicates operators (logic and comparison) and the bold sans serif indicates a scoping operator. Finally, with limited additional reasoning or processing, the requirement can be rewritten using macros as in the following (DRV/spd indicates the signal with name spd defined inside the subsystem DRV).

R1. Acceleration bound and overshoot

Assumption:

```
UPPERBOUND(sys_inertia, i1);
STEP(reference, 0, A).
```

Assertion:

```
RISETIME(DRV/spd, 0, t1, 0, x1) and
OVERSHOOT(DRV/spd, 0, x2).
```

At this point the macros expressing the specification can either be translated into STL or, for simplicitly, be directly transformed into signal generator or assertion checker blocks. For example, the signal generator macro

```
STEP(signal_name, start_time, step_amplitude)
```

could be implemented with the library source subsystem of Figure 2.

Similarly, the macro

```
OVERSHOOT(sig_name, start_time, oversh_bound)
```

can be easily translated in STL or implemented using the library blocks described in section VI.

III. The STL Language

This section introduces the restriction of the STL language that is used as a backbone of our generation tool.

A. The STL Language

In STL, a formula ϕ is evaluated on a sequence of inputs $\mathcal{X} = (x_1, x_2, \ldots, x_n)$ at a (continuous) time instant t, resulting in the evaluation of $(\mathcal{X}, t)$ pairs.

An STL formula ϕ can be:

- p: a proposition that evaluates a state variable.

$$(\mathcal{X}, t) \models p \Leftrightarrow p\,[t] = \text{TRUE}.$$

- $\neg\phi$ (Negation): the logical negation of ϕ.

$$(\mathcal{X}, t) \models \neg\phi \Leftrightarrow \neg\left((\mathcal{X}, t) \models \phi\right).$$

- $\phi_1 \wedge \phi_2$ (And): the logical *and* between ϕ_1 and ϕ_2.

$$(\mathcal{X}, t) \models \phi_1 \wedge \phi_2 \Leftrightarrow (\mathcal{X}, t) \models \phi_1 \wedge (\mathcal{X}, t) \models \phi_2.$$

- $\bigcirc\phi$ (Next): a temporal operator that evaluates ϕ at the subsequent input value.

$$(\mathcal{X}, t) \models \bigcirc\phi \Leftrightarrow (\mathcal{X}, t+1) \models \phi.$$

- $\phi_1 \mathcal{U} \phi_2$ (Until): a temporal operator that is satisfied if ϕ_1 holds until ϕ_2 becomes true.

$$(\mathcal{X}, t) \models \phi_1 \mathcal{U} \phi_2 \Leftrightarrow$$
$$\exists t' \geq t : (\mathcal{X}, t') \models \phi_2 \wedge \forall t'' \in [t, t'), (\mathcal{X}, t'') \models \phi_1.$$

From the previous primitive operators, it is possible to derive other temporal operators:

- $\Diamond\phi = \text{TRUE}\ \mathcal{U}\phi$ (Eventually): the condition is verified at least once.

$$(\mathcal{X}, t) \models \Diamond\phi \Leftrightarrow \exists t' \geq t : (\mathcal{X}, t') \models \phi.$$

- $\Box\phi = \neg(\Diamond\neg\phi)$ (Globally): the condition is always verified.

$$(\mathcal{X}, t) \models \Box\phi \Leftrightarrow \forall t' \geq t : (\mathcal{X}, t') \models \phi.$$

In STL, temporal operators may be bounded inside an implicit $[0, +\infty)$ or explicitly specified time interval. The Until operator with an interval bound has the meaning

$$(\mathcal{X}, t) \models \phi_1 \mathcal{U}_{[a,b]} \phi_2 \Leftrightarrow$$
$$\exists t' \in [t+a, t+b] : (\mathcal{X}, t') \models \phi_2 \wedge \forall t'' \in [t, t'], (\mathcal{X}, t'') \models \phi_1,$$

from which is possible to obtain the following relations.

$$\Diamond_{[a,b]} \phi = \text{TRUE } \mathcal{U}_{[a,b]} \phi.$$
$$\Box_{[a,b]} \phi = \neg \left(\Diamond_{[a,b]} \neg \phi \right).$$

B. Language Implementation

Each STL formula or *STLFormula*, can be one of the following:

- *BoolExpr*: an expression resulting in a boolean value.
- *!STLFormula*: the logical negation of an *STLFormula*.
- { *STLFormula* } `AND` { *STLFormula* }: a logical *AND* operation between two *STLFormula*s.
- *STLUntil*: the *Until* temporal operator.
- *STLGlobally*: the *Globally* temporal operator.
- *STLEventually*: the *Eventually* temporal operator.

Temporal Operators. The STL temporal operators can be written in a parsable text syntax.

The *Until* operator is expressed in the following ways:

- { *STLFormula* } `U_`*TimeExpr* { *STLFormula* }: timed *Until*.
- { *STLFormula* } `U` { *STLFormula* }: untimed *Until*.

On the other hand, the *Globally* and *Eventually* temporal operators, is expressed as follows:

- `[]` { *STLFormula* }: untimed *Globally*.
- `[]_`*TimeExpr* { *STLFormula* }: timed *Globally*.
- `<>` { *STLFormula* }: untimed *Eventually*.
- `<>_`*TimeExpr* { *STLFormula* }: timed *Eventually*.

Expressions. The previously mentioned *TimeExpr* defines the time interval in which the temporal operator is evaluated. It can be any kind of interval: closed [*Expr*, *Expr*], left open (. . .], right open [. . .), or open (. . .).

The *Expr* keyword identifies an expression with integer or floating point value:

BoolExpr is an expression with true or false evaluation, and can be one of the following,

- *Expr CmpOp Expr*: a comparison expression.
- *BoolExpr BoolOp BoolExpr*: a logical expression.
- *BoolFunction*: a function that returns a logical value.
- *BoolVal*: a constant logical value.

Operators. The operators recognized by the tool can be the basic mathematical, comparison, or boolean operators:

Values. *Val* or *BoolVal* represent values that can be either a variable defined by the user, the name of a signal or parameter belonging to the Simulink model or a constant value:

Functions. The language also allows using predefined functions such as:

- `abs(` *Expr* `)`: the absolute value of *Expr*.
- `diff(` *Expr* `)`: the left-derivative of *Expr*.
- `step(` *Expr* `,` *Expr* `)`: returns true when the first expression is recognized to be a step function with a height of at least the value defined by the second expression.

Timed Behaviors.

In STL, timed formulas can be nested such as, for example,

```
<>_[0, T] { q AND []_[a, b] { p } }.
```

The proposition *p* is nested one level deeper than proposition *q*. The meaning is that there has to be one time instant t in $[0, T]$ (the outer Eventually condition) such that **q** is satisfied in t and for all the system evolutions starting from time t, the condition *p* is verified at some time between $t + a$ and $t + b$.

In a runtime monitor implementation, the evaluation of the global condition with *p* depends not only on the time range of its temporal operator, but also on the time t in which *q* is satisfied. If t_q is the time at which *q* is satisfied, the time range in which *p* is evaluated becomes $[a + t_q, b + t_q]$. The nested time interval $[a, b]$ is therefore not an absolute time, but is relative to the time instant identified by the outer clause.

C. Language Restriction

In order to generate online monitors, we introduce the following restrictions to the STL language.

- The maximum level of nesting for temporal operators is two.
- If there is a nested temporal operator, the condition on which the outer operator is evaluated must be a conjunction and at least one of the terms of the conjunction must be a proposition (not a temporal operator).
- If T_b is the maximum value for all the endpoints of the intervals defined in the inner (nested) temporal operators, then the terms of the conjunction that are not temporal operators can only be true at time instants that are separated by a time interval always greater than T_b.

For example, in

```
<>_[0, T] { q AND []_[a, b] { p } }.
```

The outer temporal operator is defined on the conjunction **q** AND []_[a, b] { **p** }. In order to correctly generate a monitor from this formula, the proposition **q** can only be true at time instants that are separated by more than b time units.

The purpose of the restrictions is to simplify (or make altogether possible) the online monitor definition and generation. However, despite these restrictions the language is still powerful enough to handle the typical control requirements defined for the library in [14].

D. Examples

The following examples show how it is possible to express some simple system constraints using the (restricted) STL language. The language is used to express the condition resulting in the violation of the constraint (and the corresponding activation of the assertion block).

```
/* Doors must never be open while the
 * elevator is moving */
<> {doorOpen == TRUE AND elevatorSpeed != 0} ;

/* The elevator must never exceed given
 * speed and acceleration limits */
<> {abs(elevatorSpeed) > maxSpeed OR
     abs(diff(elevatorSpeed)) > maxAccel} ;

/* If the elevator is called at the fourth
 * floor, it must reach the destination in
 * less than 100 time units */
<> {floorRequest == 4 AND
     []_[0,100] elevatorFloor != 4 } ;
```

Other examples of typical control specifications in STL can be taken from [14] and are used for the synthesis of the control monitor blocks described in Section VI.

IV. The Monitor Generation Tool

The tool presented in this paper consists of a MATLAB/Simulink front-end, implemented as scripts in the Matlab language, and an STL parser and Monitor code generator, implemented in C++.

As shown in Figure 3, the parser and generator tool takes as input two files, one containing the list of requirements and a Data Dictonary description containing (among others) information about all the subsystems, signals, and parameters defined in the requirements and having a corresponding definition in the Simulink model. The Data Dictionary file (currently a .csv Excel file) can be automatically synchronized with the definitions in the Simulink model by one of the Matlab scripts in the framework.

The tool parses the two files and outputs a new file containing the Matlab code that is used to generate the Simulink Monitor blocks for the runtime validation of the STL rules in the requirements.

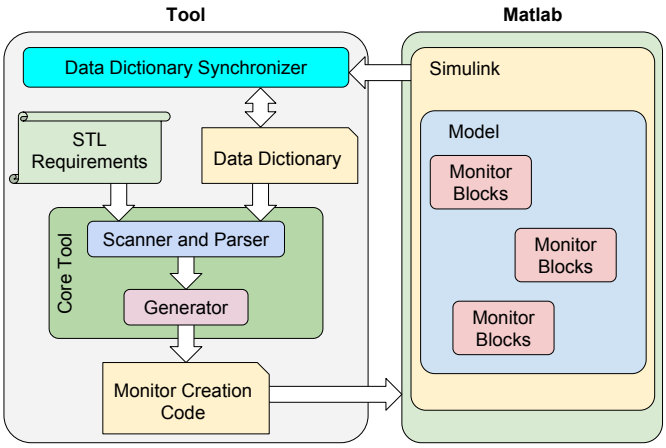

Figure 3. Block diagram representing the elements involved in the project.

This section provides a high-level description of the main tool subsystems and the input/output files by following the logical flow that the user follows to generate the monitors.

A. Data Dictionary File and Requirements File

The tool provides a Matlab function called `syncDD()`, that takes as input parameters the name of the Simulink model to be synchronized and the name of the Data Dictionary file. The function ensures that all the Simulink names of signals, subsystems and parameters are in the DD file and, if not, it updates the DD. The DD also contains the definition of all the constants (with values possibly computed as expressions of other constant values).

The requirements file (from the editor or written manually by the user) is composed by a sequence of STL formulas to be monitored.

A label can be associated with each STL formula to ease the identification of the constraint that is checked by each monitor, as in:

```
maxExceeded : <> { x > maxValue };
```

Moreover, the tool accepts single-line and multi-line code comments expressed using the C language syntax.

B. Monitor Block Code Generator

When the STL requirements are parsed by the tool all the identifiers encountered in the requirements are checked to be valid constant values or signals as defined in the DD file.

When the the parsing of the requirements is completed, the tool generates in an output file the Matlab code containing the instructions to generate the Simulink monitor blocks.

C. Monitor Block Creation

The Matlab code generated by the parser is finally used to create the Simulink monitor blocks. The tool provides a Matlab function called `addMonitorBlock()`, which takes as parameters the name of the Simulink model in which the block will be added and the position where the monitor block is located.

The function creates the monitor block in the model and connects its inputs to the signals in the model using pairs of From/Goto blocks.

D. Model Validation

After the creation of the monitor blocks, the model can be validated by launching a Simulink simulation. In the default monitor creation process, each output port of a monitor is connected to an assertion block. Whenever a requirement is violated, the simulation is aborted and an error showing the violated condition is prompted to the user.

V. Parsing and Generation Tool

This section describes the implementation details of the subsystems described in Section IV.

A. STL Requirements Parser

The requirements file is first processed by the Flex (The Fast Lexical Analyzer) [21] Flex passes every STL language token detected in the source file to the syntax parser, implemented with the GNU Bison tool [21].

Constant values are computed and replaced, and the variable names and their values are stored in a (standard library) `map` data structure.

Each token recognizable by the parser has a corresponding C++ class, derived from a pure virtual `TreeNode` class that provides the following members and data:

- `left`, `right`: pointers to `TreeNode` classes.
- `generate()`: pure virtual function that creates the associated Matlab code.

When the Bison parser identifies a token, it creates an object from the C++ class representing the associated operator or expression and, if needed, sets the `left`, `right` (or both) data fields in order to create a binary tree of parsed objects.

Each class derived from `TreeNode` must provide an implementation of `generate()` (defined as pure virtual in the generic parent class). This function generates a Simulink block container that implements the clause expressed by the associated language token and then recursively calls `generate()` on its children nodes, creating the associated sub-blocks. The set of recursive calls at all the tree nodes, results in the generation of the Matlab code for the creation of the hierarchy of nested Simulink blocks inside the monitor.

The monitor block generated by the tool has a sub-block for each formula, as shown in the example of Figure 4. Those sub-blocks output a signal with boolean value representing the validity of the associated formula ($true/1$ when verified, $false/0$ otherwise). The output of the block is meant to be connected to the *Assertion* block provided by the Simulink standard library after being complemented by a NOT. The Assertion block takes as input a signal and, as default behavior, stops the simulation and prompt an error message when it receives a truth value. The block can be configured also to continue the simulation but signal the assertion violation with a prompt.

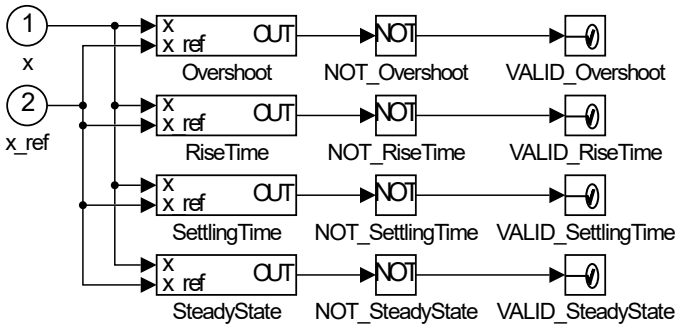

Figure 4. Validation bock content.

In order to make the timed temporal operators valid only in the time interval that is defined for them, they are provided with an additional boolean input port. The timed temporal operator is enabled only when this boolean value is true. This input port is connected with a time comparison block that outputs a true value only when the simulation time is in the given range.

The implementation of timed relationships between STL formulas is performed by extending the time range structure. Considering the Timed Behaviors description provided in Section III-B, the time instant when the untimed terms in the inner conjunction (q in the example) are all satisfied is stored in a memory block and added to the blocks containing the interval edges. See for example, the implementation of the timed Until monitor of Figure 6.

B. Simulink Functions

In Matlab, the signals and parameters defined inside the model can be extracted with the `getSignalsList()` and `getParameterList()` functions.

These function open the Simulink model passed as a parameter and scan it All the signals and parameters in the model are searched in the .csv DD file and, if missing, they are added to it.

Another important Matlab function provided by the tool is the one responsible for the creation and insertion of the monitor block in the Simulink model. The *AUTOGEN_testBlock.m* file is created by the parser tool, and used for the generation of the monitor blocks in the Simulink model by calling the `addValidationBlock()` function in Matlab.

The function takes as input parameter:

- The name of the Simulink model in which the validation block must be added.
- The name to be assigned to the validation block.
- The position of the validation block, expressed as the coordinates of the edges: left, top, right, bottom.

The function creates an empty block, with the requested position and name, as a monitor block container and runs the *AUTOGEN_testBlock.m* script to creates its content and the connections with the input model signals.

VI. The Simulink Libraries for Monitoring STL and Control Constraints

To simplify the code generator, some of the standard functions that can be internally used by the validation block are developed as a Simulink library called STL Library and implemented in the file *STLLib.slx*, and a Control Monitor library in the file *CtrlMonitorLib.slx*.

A. STL Library

This library provides Simulink blocks implementing the STL temporal operators and the *AND* operator: *Eventually*, *Always*, *Until*, and *ANDSTL* (shown in Figure 5).

Consider, for example, the timed until block (labelled as UNTIL in the Figure, the other blocks follow similar conventions). The block contains an implementation of the clause $\phi U_{[SOI,EOI]}\psi$. The block inputs are: IN_INTERVAL that needs to be set to true if the current time is inside the interval $[SOI, EOI]$, false otherwise; the EOI value, the

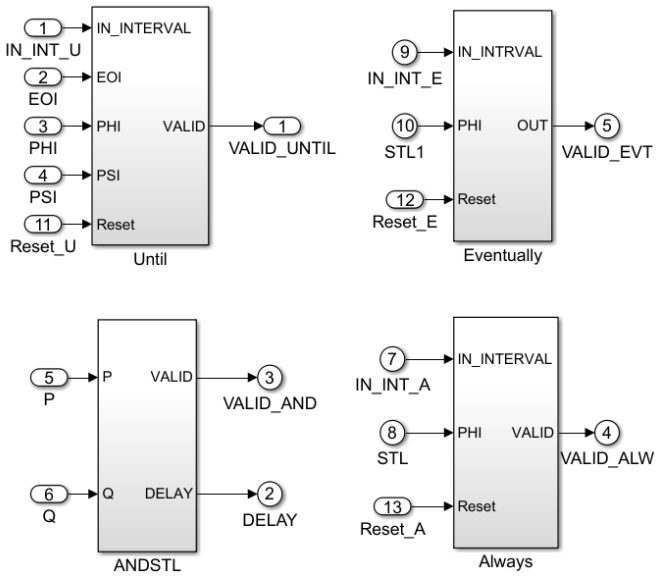

Figure 5. Library for the generation of STL monitors.

current evaluations for the ϕ and ψ formulas, and a RESET input.

Figure 6 shows the internals of the timed Until block. All the monitor blocks keep their output constant after a violation of the rule is detected. However, to facilitate their use in simulations concatenating several test cases, a reset input is also provided. This is implemented by the set-reset block at the end of the chain, on the far right.

The definition of the timed until $\phi U_{[SOI,EOI]}\psi$ is (from section III)

$$(\mathcal{X},t) \models \phi \mathcal{U}_{[SOI,EOI]}\psi \Leftrightarrow$$
$$\exists t' \in [t+SOI, t+EOI] : (\mathcal{X},t') \models \psi \wedge$$
$$\forall t'' \in [t,t'], (\mathcal{X},t'') \models \phi,$$

with the availability of the signal IN_INTERVAL, the condition becomes

$$\exists t' \text{ such that: IN\_INTERVAL} \wedge (\mathcal{X},t') \models \psi \wedge$$
$$\forall t'' \in [t,t'], (\mathcal{X},t'') \models \phi,$$

In the model implementation of Figure 6 the top left part is in charge of the implementation of the first conjunction (highlighted in red); whereas the bottom part (in blue) implements the final clause of the conjunction.

B. Control Monitor Library

To simplify the creation of monitors for typical control systems, we also defined a library of control monitors (shown in Figure 7.) The library has blocks for checking overshoot (undershoot) and rise time (or fall time) constraints on triggers derived from generic inputs signals (steps, but also ramps). These conditions are verified on a selected input signal (typically a system variable or an input/output of the controller) with respect to another reference or trigger input. The library is constructed in layers. A set of blocks checks the conditions upon reception of a generic trigger signal. Other blocks are built on this set including the logic that detects the trigger from conditions on a generic signal.

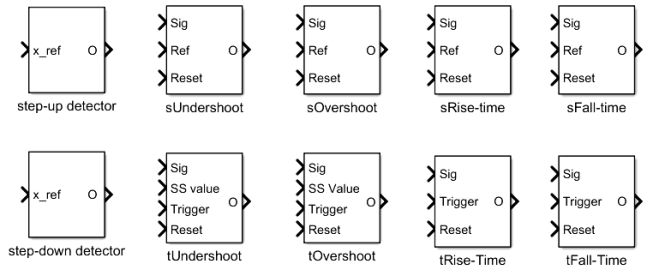

Figure 7. Library for the generation of Control monitors.

Each block of the control monitor library (such as the overshoot block, shown in Figure 8) is built on top of (using) the STL library blocks.

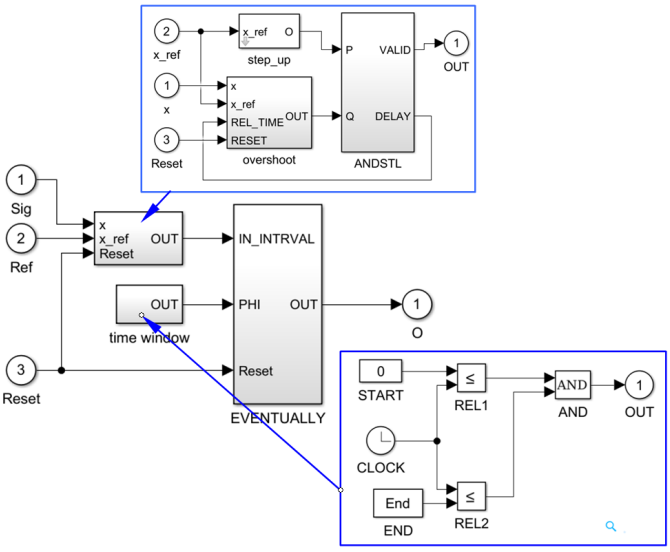

Figure 8. Overshoot monitor block internals as defined in the Control Monitor library.

For the implementation of the Control Monitor library, we simply used the STL formulations provided in [14]. For example, the STL encoding of the overshhot condition (with the corresponding block implementation of Figure 8) is

$$\Diamond_{[0,T]}(step(x_{ref}, r) \wedge \Diamond(x - x_{ref} > c))$$

VII. Usage Example

This section presents a simple example model showing how the tool can be used to generate monitor blocks and what is the final result. The example system is a Simulink model of

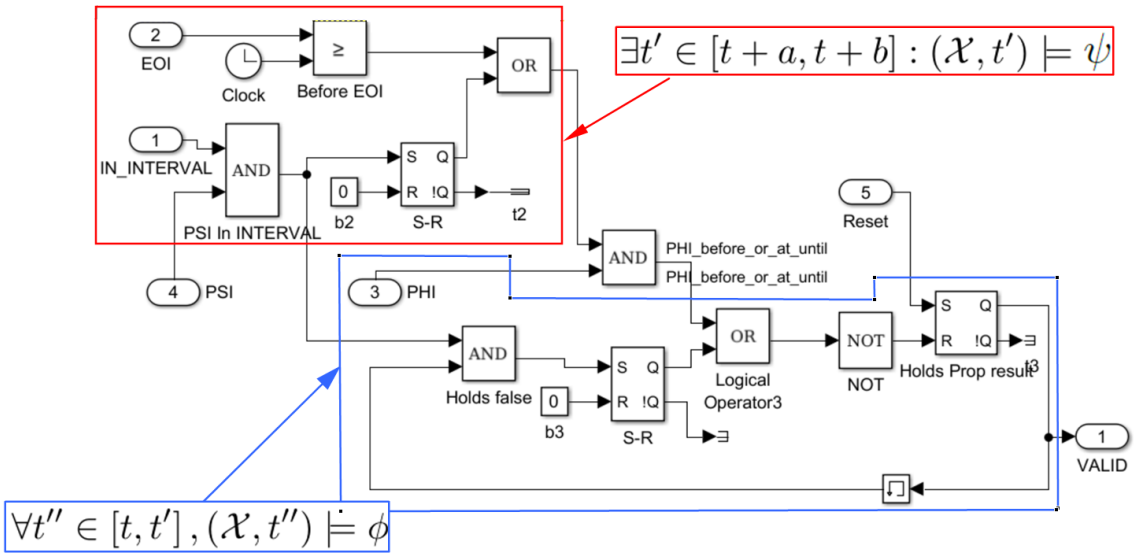

Figure 6. The STL library block for checking the timed Until condition.

a dual pole system controlled in a closed-loop, as shown in Figure 9.

The model is originally as shown in the bottom side, without the highlighted monitor blocks that are automatically added by the framework tool, as described in the next sections.

The function

```
>>> getSignalsList('SimulinkModelExample');
```

is executed from the Matlab prompt to verify that the signals in the model are contained in the DD file. The DD file also contains the constants are parameters used in the STL constraint formulas. The set of relevant variables and symbols in the DD file is shown in Table I.

name	value
T	10
r	5
c	3
zeta	0.5
mu	0.95
steadyStateValue	10
s	3
beta	0.02
a	0.01

Table I
VALUES IN THE DD FILE FOR THE SIMULINK EXAMPLE WITH THE STL CONSTRAINTS.

Listing 1 shows the example requirements file with the list of STL formulas representing the system requirements:

- *Overshoot*: after the detection of a step in the input signal, the output value of the system exceeds the reference value for more than a given quantity.
- *RiseTime*: after the detection of a step in the input signal, the system is not able to reach a specified value in a given time.
- *SettlingTime*: after the detection of a step in the input signal, the system is not able to keep the output bounded in a given range after a given time.
- *SteadyState*: when the system reaches its steady state condition, the value it outputs differs from the reference signal for more than a given value.

```
Overshoot : <>_[0, T] { step(x_ref, r) AND
  <> { x - x_ref > c } };

RiseTime : <>_[0, T] { step(x_ref, r) AND
  []_[0,zeta] { x < mu * steadyStateValue } };

SettlingTime : <>_[0, T] { step(x_ref, r) AND
  <>_[s, T] {abs(x - x_ref) > beta * x_ref} };

SteadyState : <>_[T, T] { abs(x - x_ref) > a};
```

Listing 1. Example of requirements file

The tool executes by passing as a first argument the requirements file and as a second argument the path of the folder containing the DD file.

After the execution of the tool, the *AUTOGEN_testBlock.m* file is created in the same path of the signals file.

To insert the validation block in the given Simulink model, the following function can be executed from the Matlab prompt:

```
>>> addValidationBlock('SimulinkModelExample',
    'STL_TEST', [60,240,90,280]);
```

The result of the the `addValidationBlock()` function is the creation of the monitor blocks highlighted in Figure 3

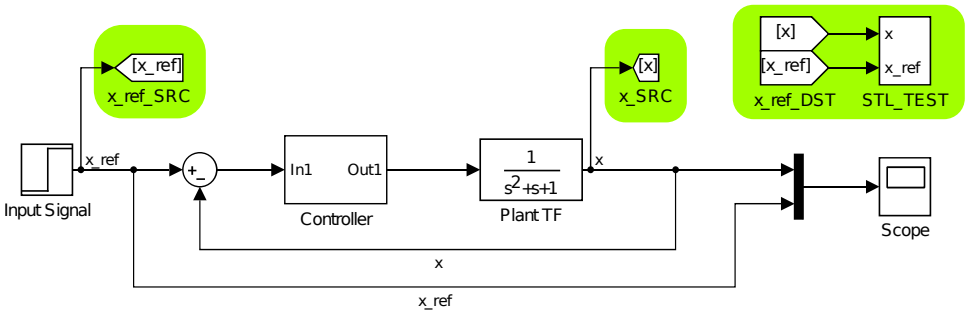

Figure 9. Example of Simulink block diagram of a model with a closed loop controller.

and their connection to the specified input and output signals by means of From/Goto blocks..

VIII. Conclusion

This paper presented a framework for the generation of monitor Simulink blocks for model validation at simulation time. The STL formal language is used as reference for the definition of the model requirements. The paper shows an overview of the tool and the supporting libraries, including implementation details and a practical example of its usage on a real model.

As a future work, we plan to extend the tool by integrating it in a complete environment that supports the user to describe the model requirements in a formal language with a syntax closer to the natural languages.

References

[1] T. Mathworks., "Simulink user manual," in *Product web page*, 2017.

[2] Prover., "Company web page: www.prover.com," 2017.

[3] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS)*, 1977, pp. 46–57.

[4] E. C. E. Emerson, "Design and synthesis of synchronisation skeletons using branching time temporal logic," in *Logic of Programs, Proceedings of Workshop, Lecture Notes in Computer Science*, vol. 131. Springer, Berlin, 1981, pp. 52–71.

[5] R. Koymans, "Specifying real-time properties with metric temporal logic," in *Real-Time Systems*, vol. 2(4), 1990, p. 255299.

[6] O. Maler and D. Nickovic., "Monitoring temporal properties of continuous signals," in *Proc. of Formal Modeling and Analysis of Timed Systems/ Formal Techniques in Real-Time and Fault Tolerant Systems*, 2004, pp. 152–166.

[7] C. Eisner and D. Fisman, "A practical introduction to psl," 2006.

[8] O. Maler, D. Nickovic, and A. Pnueli, "Checking temporal properties of discrete, timed and continuous behaviors," in *Pillars of Computer Science: Lecture Notes in Computer Science*, vol. 4800. Springer, 2003, pp. 475–505.

[9] P. S. Duggirala, C. Fan, S. Mitra, and M. Viswanathan, "Meeting a powertrain verification challenge," 2015.

[10] Donze, T. Ferrere, and O. Maler, "Efficient robust monitoring of stl formula," in *Proceedings of the CAV 13 Conference*, 2013.

[11] D. Ulus, T. Ferrre, E. Asarin, and O. Maler, "Legay, a., bozga, m. (eds.) timed pattern matching," in *FORMATS 2014. LNCS, vol. 8711*, vol. 8711. Springer, Heidelberg, 2014, pp. 222–236.

[12] P. C. E. Asarin and O. Maler, "Timed regular expressions," in *The Journal of the ACM*, vol. 49, 2002, pp. 172–206.

[13] D. Ulus, T. Ferrre, E. Asarin, and O. Maler, "Online timed pattern matching using derivatives," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS 2016: Tools and Algorithms for the Construction and Analysis of Systems*, 2016, pp. 736–751.

[14] J. Kapinski, X. Jin, J. Deshmukh, A. Donze, T. Yamaguchi, H. Ito, T. Kaga, S. Kobuna, and S. Seshia, "St-lib: A library for specifying and classifying model behaviors," in *SAE Technical Paper*. SAE International, 04 2016. [Online]. Available: http://dx.doi.org/10.4271/2016-01-0621

[15] J. Flores, "Semantic filtering of textual requirements descriptions," in *Natural Language Processing and Information Systems*, 2004, pp. 474–483.

[16] S. Gnesi, G. Lami, and G. Trentanni, "An automatic tool for the analysis of natural language requirements," in *CSSE Journal*, vol. 20(1), 2005, pp. 53–62.

[17] S. L. Obispo, "Parsing of natural language requirements," in *Thesis presented to the Faculty of California Polytechnic State University*, Oct. 2013.

[18] K. Deemter, V. E. Krahmer, and M. Theune, "Real versus template-based natural language generation: A false opposition?" in *Computer Linguist*, vol. 31(1), 2005, pp. 15–24.

[19] L. Mangeruca and O. A. F. Ferrante, "Formalization and completeness of evolving requirements using contracts," in *8th IEEE International Symposium on Industrial Embedded Systems (SIES 2013)*, 2013.

[20] A. Balsini, "Repository," https://github.com/balsini/SignalTemplateLibraryAutogen/.

[21] J. Levine, "Flex and bison," in *O'Reilly Media*, August 2009.