Specification and Runtime Verification of Temporal Assessments in Simulink

Akshay Rajhans, Anastasia Mavrommati, Pieter J. Mosterman, and Roberto G. Valenti

Advanced Research & Technology Office, MathWorks, Natick, MA 01760, USA
{arajhans|amavromm|pmosterm|rvalenti}@mathworks.com

Abstract. Formalization of specifications is a key step towards rigorous system design of complex engineered systems such as cyber-physical systems. Temporal logics are a suitable expressive formalism for capturing temporal specifications. However, since engineers and practitioners are often unfamiliar with the symbols and vocabulary of temporal logic, informal natural-language specifications still are used abundantly in practice. This tool paper presents the Temporal Assessments feature in Simulink® Test™ that strives to achieve the best of both worlds. It provides graphical user interfaces and visual examples for users to interactively create temporal specifications without the need to author logical formulae by hand, yet any user-authored temporal assessment is a valid logical formula in an internal representation. Iterative folding of clauses enables the specification to be presented to read like English language sentences. Key highlights of the feature along with examples of authoring and runtime verification of temporal logic specifications are presented.[1]

Keywords: Formal specifications · Temporal logic · Runtime verification · Model-Based Design · Simulink · Simulink Test

1 Introduction

Model-Based Design of complex engineered systems involves the creation of computational models and perfecting them as much as possible before building actual physical prototypes where design iterations and finding and fixing mistakes can be costly. Requirement specifications are useful for establishing correctness of design models. However, these specifications are often captured in natural-language sentences in enumerated lists (e.g., in a spreadsheet). Such informal specifications and can be incomplete, ambiguous, and inconsistent among each other.

Temporal logics such as the Signal Temporal Logic [9] are a formal alternative naturally suited for dense-time continuous or hybrid domain behavior evolutions

[1] This is authors' preprint. The paper appears in the Proceedings of the International Conference on Runtime Verification (RV) 2021. The final authenticated version is available online at `https://doi.org/10.1007/978-3-030-88494-9_17`.

seen in engineered systems, including cyber-physical systems. The research community has seen broad adoption of such logics for specification and runtime verification: a representative list of relevant work includes [3, 4, 6, 8, 11, 12, 14]. Yet, industrial adoption by practitioners has remained a challenge. One key barrier is the lack of familiarity with logical symbols and formulae. For example, in his HSCC 2015 Keynote, Deshmukh calls out formal requirements engineering as a "grand challenge" and mentions "[h]ow do [control designers] convey their intentions without using formalisms?" as a "[k]ey challenge for Toyota, Bosch, and others" towards that grand challenge [5]. As examples of additional barriers to adoption, Kapinski posits that "[o]ne reason why formal requirements have not yet been adopted by industry is that they can be difficult to create and debug" and that "[a] remaining challenge is in creating methods to visualize, or otherwise elucidate, the envelope (or complete set) of behaviors specified by the requirements." [2].

Starting with Release R2019a, Simulink® Test™ offers a new Logical and Temporal Assessments functionality[2] that aims to address some of these challenges and make formal specifications more accessible to engineers and practitioners. It provides a mechanism for authoring logical and temporal specifications via a graphical user interface (GUI) by simply filling out pre-existing template patterns, that is, without the need to write out a logical formula by hand. This interface provides visual representations of hypothetical passing and failing behaviors at authoring time as visual feedback to the user. Evaluation of specifications involves simulation of a Simulink® model specified as the system under test (SUT). Hierarchical subexpression tree evaluation provides a visual mechanism for the user to investigate assessment, and failing examples provide graphical and textual explanation of failures. The rest of this paper presents key details about authoring and evaluation of specifications using this functionality.

2 Authoring Temporal Specifications

We begin by covering some preliminaries.

2.1 Preliminaries

Simulink® is a graphical Model-Based Design environment for modeling, simulation, and automatic code generation of engineered systems, including cyber-physical systems. Simulink models are directed graphs with *blocks* forming nodes of the graph and *signal lines* forming the edges of the graph. Formal definitions of Simulink models and blocks can be found elsewhere [13] and are dropped from this paper for brevity.

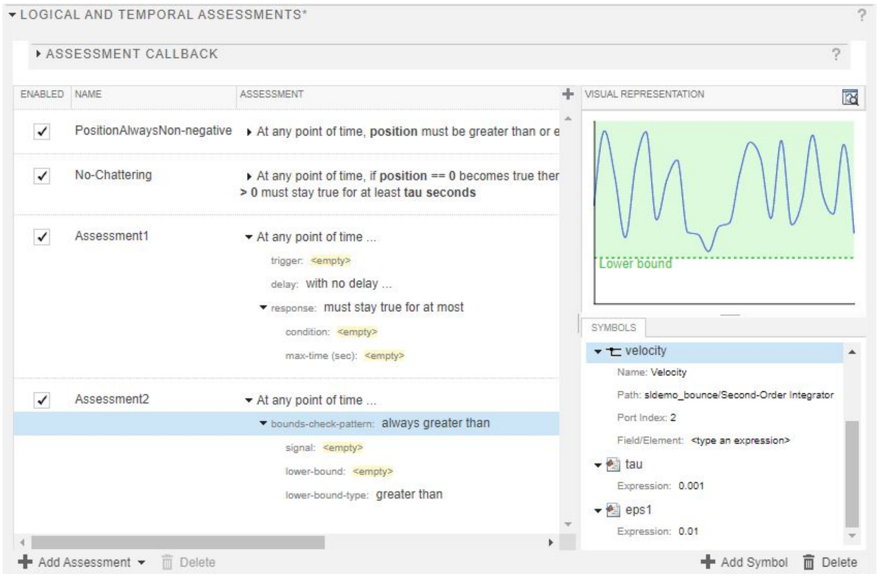
Blocks may have internal continuous-time and/or discrete-time state(s) that are updated as per the corresponding differential and/or difference equations. As a software implementation, each block must define its *output* method, and

[2] `https://www.mathworks.com/help/sltest/ug/temporal-assessments.html`

may define *update* and/or *derivative* methods to realize the corresponding equations as applicable. Simulink's execution engine calls these methods in a predetermined order in a loop, called the *simulation loop*, until the simulation stop time is reached [7].

Signal lines are buffered values computed by the output ports of the driving blocks every time the *output* method gets called. In between such consecutive *output* method calls, the last computed value is held until a new value overwrites it. Note that for discrete-time blocks, this is a zero-order hold implementation, which is actually a continuous-time signal with possible discontinuities.

2.2 Authoring Temporal Assessments via a UI element



ENABLED	NAME	ASSESSMENT
✓	PositionAlwaysNon-negative	At any point of time, **position** must be greater than or e
✓	No-Chattering	At any point of time, if **position == 0** becomes true ther > 0 must stay true for at least **tau seconds**
✓	Assessment1	At any point of time ...
		trigger: <empty>
		delay: with no delay ...
		response: must stay true for at most
		condition: <empty>
		max-time (sec): <empty>
✓	Assessment2	At any point of time ...
		bounds-check-pattern: always greater than
		signal: <empty>
		lower-bound: <empty>
		lower-bound-type: greater than

velocity
Name: Velocity
Path: sldemo_bounce/Second-Order Integrator
Port Index: 2
Field/Element: <type an expression>
tau
Expression: 0.001
eps1
Expression: 0.01

Fig. 1: UI Element for Authoring Logical and Temporal Assessments.

Logical and temporal specifications can be authored in structured English from pre-existing templates and patterns. Figure 1 shows a user interface (UI) element for authoring specifications. Clicking on 'Add Assessment' lets the user choose from pre-existing templates and patterns to construct a specification. The left half of the window in Figure 1 depicts a mechanism for users to author specifications. Shown are a couple of new assessments with highlighted fields for the user to interactively select variations and/or enter expressions, as well as a couple of filled out and folded assessments that read like English language sentences.

Table 1 depicts the three classes of template patterns available for users to author. *Custom formula* captures if a given Boolean expression holds over all

Table 1: Template Pattern Classes

Template Pattern Class	Equivalent Logical Formula
Bounds Check	$\Box$ (signal satisfies bound constraint)
Custom Formula	$\Box\ \varphi$
Trigger Response	$\Box\ (\varphi_1 \rightarrow \Diamond\ \varphi_2)$

simulation time. *Bounds check* collection provides a set of frequently-used instances of *custom formula*, where the expression is whether a given signal value or a derived expression always stays *above*, *below*, *inside*, and *outside* bounds, with Boolean combination of strict and/or non-strict inequalities (indicated using corresponding check-boxes) tabulated in Table 2.

Table 2: Bounds check patterns and variations.

Pattern	Equivalent Logical Formula	Strict Variation(s)
Always less than	$\Box\ (x < \texttt{ub})$	$\Box\ (x \leq \texttt{ub})$
Always greater than	$\Box\ (x > \texttt{lb})$	$\Box\ (x \geq \texttt{lb})$
Always inside bounds	$\Box\ ((x < \texttt{ub}) \wedge (x > \texttt{lb}))$	$\Box\ ((x \leq \texttt{ub}) \wedge (x > \texttt{lb}))$ $\Box\ ((x < \texttt{ub}) \wedge (x \geq \texttt{lb}))$ $\Box\ ((x \leq \texttt{ub}) \wedge (x \geq \texttt{lb}))$
Always outside bounds	$\Box\ ((x > \texttt{ub}) \vee (x < \texttt{lb}))$	$\Box\ ((x \geq \texttt{ub}) \vee (x < \texttt{lb}))$ $\Box\ ((x > \texttt{ub}) \vee (x \leq \texttt{lb}))$ $\Box\ ((x \geq \texttt{ub}) \vee (x \leq \texttt{lb}))$

Trigger response is a class of frequently-used temporal formulas of the type $\Box\ (\varphi_1 \rightarrow \Diamond_{[a,b]}\ \varphi_2)$. The various combinations of triggers, response delays, and responses are tabulated in Table 3. Note that a logical condition *becoming* true captures the rising edge of the evaluation of the expression from false to true. This is typically not supported out-of-the-box in as an atomic expression in logics, but is a shorthand, similar in spirit to rise and fall operators [10], provided to the user who would otherwise need to construct a compound clause themselves. Similarly, an expression *staying* true for a specified period is another shorthand that absorbs a temporal operator within it. The three flavors of response delay form the implications $\rightarrow$, $\rightarrow \Diamond_{[0,b]}$, and $\rightarrow \Diamond_{[a,b]}$ respectively.

The flavors for response conditions include when an expression evaluates to true at the point of evolution as well as those where it evaluates to true and stays true for a range of time intervals. Additionally, there is also an until operator where a condition evaluates to true and stays true until a different condition becomes true. This is a timed until, so there is a maximum timeout period that the user can specify.

Table 3: Trigger response pattern ($\varphi_1 \rightarrow \Diamond_{[a,b]}\ \varphi_2$) and its variations

Element	Variation
Trigger (φ_1)	`whenever <condition1> is true`
	`<condition1> becomes true`
	`<condition1> becomes true and stays true for at least`
	`<condition1> becomes true and stays true for at most`
	`<condition1> becomes true and stays true for between`
Response delay ($\rightarrow \Diamond_{[a,b]}$)	`with no delay`
	`with a delay of at most`
	`with a delay of in between`
Response (φ_2)	`<condition2> must be true`
	`<condition2> must stay true for at least`
	`<condition2> must stay true for at most`
	`<condition2> must stay true for between`
	`<condition2> must stay true until <condition3> becomes true`

2.3 Visual Representation

The top right corner of Figure 1 shows a visual representation of a fictitious trace that would pass the selected lower bounds check. The user can regenerate additional passing and failing visuals including ones with a dynamic lower bound in order to get a visual intuition about the kind of specification they are entering. Note that these fictitious traces are not the actual behaviors of the SUT model since at authoring time SUT simulation is not invoked. Appropriate visual examples are also available for other templates such as trigger-response.

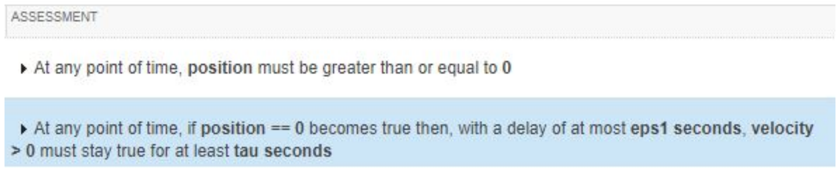
2.4 Symbol Resolution

The bottom right corner of Figure 1 shows the symbol mapping UI element, where the symbols appearing in authored assessments can be mapped to either a signal in the SUT model or to an expression. The named symbol `velocity` is shown to be mapped to a signal in the SUT model, whereas the symbols `tau` and `eps1` are mapped to simply their defined constant values. (In an alternative implementation, they could have instead been mapped to workspace variables).

2.5 Example

As a running example, let us consider a bouncing ball model in Simulink (example model `sldemo_bounce` that ships with the product) [1] as the SUT. We express a logical bounds-check condition `PositionAlwaysNon-negative`, which checks that the position of the ball always stays non-negative. This specification is intended to be a sanity check that the model has been constructed correctly. Additionally, we author a trigger-response specification `No-Chattering`, which checks whether after each bounce, within a small time period ϵ_1, whether the velocity remains positive for at least a specified finite time period of τ seconds.

Figure 2 shows the two authored specifications folded to read as English sentences.



ASSESSMENT

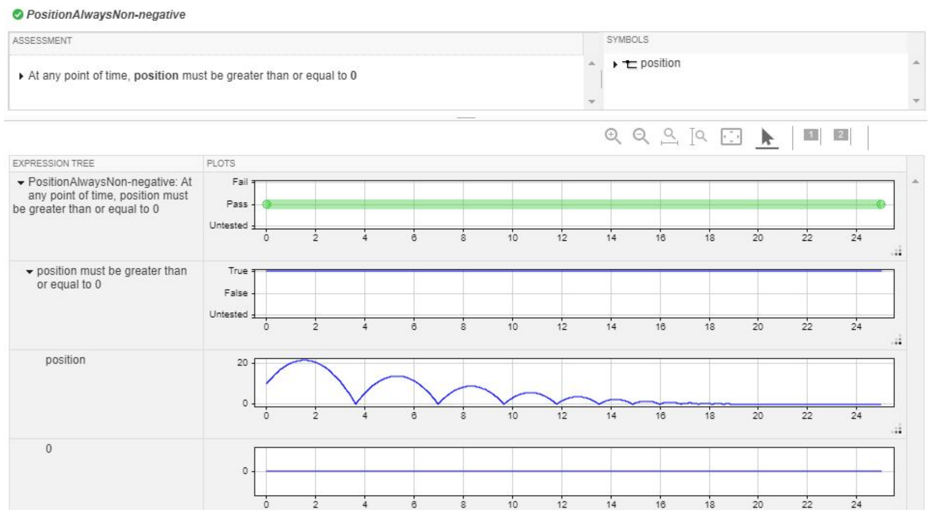
▸ At any point of time, **position** must be greater than or equal to **0**

▸ At any point of time, if **position == 0** becomes true then, with a delay of at most **eps1 seconds**, **velocity > 0** must stay true for at least **tau seconds**

Fig. 2: Example specifications for a bouncing ball model.

3 Runtime Verification of Temporal Assessments

Runtime verification of the logical and temporal assessments invokes a simulation of the SUT and checks whether the simulation trace satisfies the specifications. Figure 3 depicts a passing evaluation of the sanity-check condition `PositionAlwaysNon-negative` defined in Section 2.5. The UI shows the assessments, the symbols used in the assessment (in this case only `position`), and a foldable subexpression evaluation tree.



✓ *PositionAlwaysNon-negative*

ASSESSMENT

▸ At any point of time, **position** must be greater than or equal to **0**

SYMBOLS

▸ position

EXPRESSION TREE

▾ PositionAlwaysNon-negative: At any point of time, position must be greater than or equal to 0

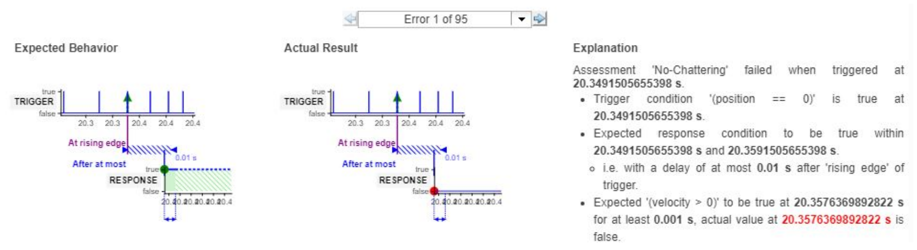
▾ position must be greater than or equal to 0

position

0

Fig. 3: Satisfaction of `PositionAlwaysNon-negative`.

The assessment `No-Chattering` from Section 2.5 turns out to be not satisfied by the SUT. Figure 4 shows a portion of the UI where we see a pictorial and textual explanation of the failure. There is at least one point in simulation time (at 20.35 s) where the response condition of (`velocity` > 0) does not stay true for at least τ s. The explanation provides the exact simulation time within less than τ s from the trigger plus a delay of ϵ_1 when it becomes false. It turns out there are 94 other simulation times when the assessment also evaluates to false, and the left and right arrows let the user navigate to other failure points in time and read the corresponding failure explanation.



Explanation

Assessment 'No-Chattering' failed when triggered at **20.3491505655398 s**.

- Trigger condition '(position == 0)' is true at **20.3491505655398 s**.
- Expected response condition to be true within **20.3491505655398 s** and **20.3591505655398 s**.
  - i.e. with a delay of at most **0.01 s** after 'rising edge' of trigger.
- Expected '(velocity > 0)' to be true at **20.3576369892822 s** for at least **0.001 s**, actual value at **20.3576369892822 s** is false.

Fig. 4: Graphical and textual explanation of failure of `No-Chattering`.

Figure 5 shows another portion of the failing assessment UI which shows the foldable and expandable subexpression evaluation tree. The top trace shows the evaluation of the overall assessment resulting in `fail`, `pass`, and `untested` values over time. In case of trigger response formulas, since they take the form of an implication ($\varphi_1 \rightarrow \varphi_2$), the formula can be vacuously true when the trigger precondition φ_1 evaluates to false. All such points in time are shown in gray (`untested` value), whereas shown in green (`pass` value) are those where the trigger condition φ_1 evaluates to true and the response condition φ_2 evaluates to true. All the failing points towards the end of the simulation are shown in red (`fail` value) where the trigger condition φ_1 evaluates to true but the response condition φ_2 evaluates to false. Such an expression tree helps the user narrow down the sources of failure in time (by zooming into the x axis as depicted) as well as, in subexpressions and thereby in corresponding SUT elements they are mapped to in order to debug the failures more quickly.

4 Discussion

This tool paper presented the new logical and temporal assessments functionality in Simulink Test which aims to make formal specifications accessible to practitioners who may not be familiar with logic symbols and vocabulary. A graphical user interface enables users to enter formal specifications without the need to write out logical formulas by hand. Iterative folding of subexpressions enables the formulas to be read like English language sentences. Yet, because these are syntactically correct formulas by construction, they are formal and unambiguous. Symbols used in the formulas can be mapped to expressions or signals from a Simulink system-under-test model.

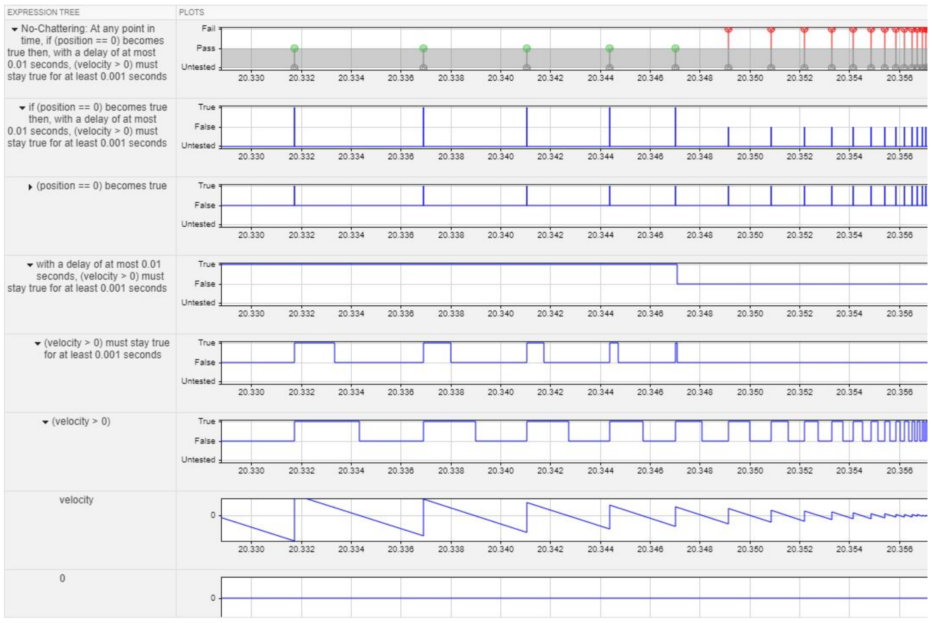


Fig. 5: Violation of non-chattering specification with a detailed expression tree of assessment evaluation.

The design choices made in developing this functionality are based on the voice of our industry practitioner customers since the early days of development. For example, the supported classes of template patterns strive to achieve the balance between expressivity (capture most commonly used specifications) and simplicity (keep it intuitive for practitioners). Visualization examples provide additional feedback to the user about whether the specification they are authoring is the one they have in mind. Lastly, the deliberate use of the `untested` value is another such design choice because showing such vacuously true instances as `pass` is non-intuitive to practitioners.

Our hope is that this new functionality facilitates broader mainstream adoption of formal specifications by industry practitioners.

Acknowledgments

Contributions by Dr. Jean-François Kempf and Dr. Khoo Yit Phang to an earlier extended abstract on this topic are gratefully acknowledged.

References

1. Simulation of a bouncing ball. `http://www.mathworks.com/help/simulink/examples/simulation-of-a-bouncing-ball.html`.

2. Frank Allgöwer, João Borges de Sousa, James Kapinski, Pieter Mosterman, Jens Oehlerking, Patrick Panciatici, Maria Prandini, Akshay Rajhans, Paulo Tabuada, and Philipp Wenzelburger. Position paper on the challenges posed by modern applications to cyber-physical systems theory. *Nonlinear Analysis: Hybrid Systems*, 34:147–165, 2019.
3. Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. S-TaLiRo: A tool for temporal logic falsification for hybrid systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 17th Int. Conf., TACAS 2011, Held as Part of the Joint European Conf. on Theory and Practice of Software, ETAPS 2011, Proceedings*, Lecture Notes in Computer Science, pages 254–257, 2011.
4. Ezio Bartocci, Niveditha Manjunath, Leonardo Mariani, Cristinel Mateis, and Dejan Ničković. CPSDebug: Automatic failure explanation in CPS models. *Int J Softw Tools Technol Transfer*, 2021.
5. Jyotirmoy Deskhmukh. "Will future cars have formally verified powertrain control software?". Keynote Talk, 18th International Conference on Hybrid Systems: Computation and Control, 2015. Slides: `https://www.cs.utexas.edu/~deshmukh/Papers/Talks/hsccKeynote.pptx`.
6. Alexandre Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 167–170, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
7. Zhi Han, Pieter Mosterman, Justina Zander, and Fu Zhang. Systematic management of simulation state for multi-branch simulations in Simulink. In *Proc. of the Symposium on Theory of Modeling and Simulation (TMS) 2013*, pages 84–89.
8. Bardh Hoxha, Nikolaos Mavridis, and Georgios Fainekos. VISPEC: A graphical tool for elicitation of MTL requirements. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3486–3492, 2015.
9. Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
10. Oded Maler and Dejan Ničković. Monitoring properties of analog and mixed-signal circuits. *Int J Softw Tools Technol Transfer*, 15:247–268, 2013.
11. Dejan Ničković, Olivier Lebeltel, Oded Maler, Thomas Ferrère, and Dogan Ulus. AMT 2.0: Qualitative and quantitative trace analysis with extended signal temporal logic. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 303–319, Cham, 2018. Springer International Publishing.
12. Dejan Ničković and Tomoya Yamaguchi. RTAMT: Online robustness monitors from STL. In Dang Van Hung and Oleg Sokolsky, editors, *Automated Technology for Verification and Analysis*, pages 564–571, Cham, 2020. Springer International Publishing.
13. Akshay Rajhans and Pieter J. Mosterman. Graphical modeling of hybrid dynamics with Simulink and Stateflow. In *Proc. of the ACM International Conference on Hybrid Systems:Computation and Control (HSCC) 2018*, pages 84–89.
14. Dogan Ulus. Online monitoring of metric temporal logic using sequential networks, 2019. `https://arxiv.org/abs/1901.00175`.