Author pre-print. Publication accepted for MODELS 2025.

DTChecker: A Real-Time Signal Monitoring and Property Specification Tool for Digital Twins

Abdelhamid Rouatbi
Université de Montréal
abdelhamid.rouatbi@umontreal.ca

Eugene Syriani
Université de Montréal
syriani@iro.umontreal.ca

Bentley Oakes
Polytechnique Montréal
bentley.oakes@polymtl.ca

***Abstract*—Specifying and monitoring temporal requirements in Digital Twin (DT) systems is challenging, as writing formal specifications in temporal logic is often complex and inaccessible to domain experts. This typically necessitates close collaboration with software engineers, introducing communication overhead and slowing development. We present DTChecker, a reusable self-contained monitoring tool for DT systems built on RabbitMQ-based architectures. The tool enables domain experts to write temporal specifications in a browser-based editor with language server support. These specifications are automatically translated into Signal Temporal Logic (STL) formulas and evaluated in real-time on data streams from sensors or services. Robustness scores are streamed to a front-end dashboard to visualize how well the system satisfies the specified requirements over time. This enables domain experts to write and verify temporal properties easily, thereby improving the real-time monitoring of the DT. We demonstrate the tool through integration with an open-source incubator DT case study.**

Video demonstration: https://youtu.be/elyhSOiGuc4?si=V-2TTsXgRcYaAU2X

***Index Terms*—Digital Twins, Model-Driven Engineering, Run-time Verification, Signal Temporal Logic.**

I. Introduction

Digital Twins (DTs) are virtual replicas of physical systems and are one of the leading technologies in the fourth industrial revolution. A pressing topic in this field is the development of reusable assets to accelerate the development of DTs [1]. In this work, we are particularly interested in *the development of reusable components to efficiently provide the monitoring capabilities of DTs*. Monitoring services allow stakeholders to observe the behavior of the Physical Twin through metrics as it evolves over time, offering support for decision-making [2].

This process can be further enhanced by using *Runtime Verification* techniques that can evaluate the correctness of the behavior of the system under study with respect to some property by analyzing execution traces [3]. Such properties are specified by using proper formalisms such as *temporal logic* (TL). *Signal Temporal Logic* (STL) [4] enables reasoning about continuous real-time properties, making it suitable for monitoring the temporal properties of real-time DTs.

Manually specifying properties with TL is known as a challenging task [5]. Autili et al. have presented a textual *Domain-Specific Language* (DSL) to abstract away the difficult syntax of TL [6]. However, DSLs such as these have not yet been integrated into reusable and open-source tools for the monitoring of DTs.

Our contribution *DTChecker* is thus a reusable, self-contained DT monitoring service that provides:

- STL run-time verification using property specification patterns.
- A browser-based editor with parsing, validation, and code completion for specifying DT behavior.
- Automatic translation of specifications into executable STL code and monitoring dashboards.

We describe our tool using a running example of beer fermentation in the next section. This is followed by background information relevant to our approach. We then detail the technical architecture and usage of DTChecker, and finally present an example of DTChecker's use on the incubator DT case study.

II. Running Example: Beer Fermentation

A. System and DT

The fermentation process in beer making is a complex biochemical transformation in which yeast consumes fermentable sugars to produce ethanol, carbon dioxide, and a variety of flavor compounds. This process unfolds over weeks and is highly sensitive to environmental factors, including temperature, pH, dissolved oxygen, and sugar concentration. Each stage of fermentation requires specific conditions to ensure efficient yeast performance and flavor development. Progress is tracked through indicators such as pH and density shifts, and parameters like conductivity and temperature influence the quality and consistency of the final product. Because of its biological complexity and sensitivity to external factors, fermentation must be carefully monitored and controlled to achieve consistent results across batches.

A DT can enhance the fermentation process by continuously collecting and synchronizing data from multiple sensors, providing brewers with insights into fermentation dynamics [7]. This reduces reliance on delayed manual sampling and enables early detection of anomalies, such as bacterial contamination or equipment failure.

B. Need for Temporal Logic

By integrating temporal logic into the monitoring framework, brewers can move beyond simple threshold-based alerts and monitor how they behave in relation to time and context.

For example, instead of just checking whether the temperature exceeds a limit, we could specify that "*if the temperature rises above 32°C, the cooling system must activate within 10 minutes and bring the temperature back down within 30 minutes*". Such specifications can be continuously evaluated on live sensor data to compute robustness scores that indicate how close the system is to violating a requirement or how severe the violation is. This provides actionable insights for managing safety and quality.

III. Background

A. Signal Temporal Logic

STL is a specification language for dense-time real-valued signals properties of continuous systems. Such specifications are used by verification algorithms that check whether the specified properties are satisfied or not.

Offline monitoring refers to the case where the signal data is assumed to be complete and available before the evaluation. While this approach is valid for simulation-based validation, it is not ideal for the monitoring of real-time systems because the evaluation of a property at a certain timestamp could depend on data observed at a future time. Meanwhile, *online monitoring* evaluates properties incrementally as data becomes available, without access to future values, by transforming the formula and postponing the evaluation to a time where all necessary data is available [8].

The evaluation of STL specifications outputs a quantitative metric called *robustness* that describes the degree of satisfaction or violation of the specified properties, providing additional feedback on the system's behavior. A positive robustness implies that the property is satisfied, while a negative value indicates it is violated.

In the context of brewing, we might consider the formula (1), which states that the temperature inside the fermentation tank should always be lower than 30°C.

$$\square(T < 30) \tag{1}$$

Fig. 1 showcases two different scenarios and their corresponding robustness evaluated against this specification.

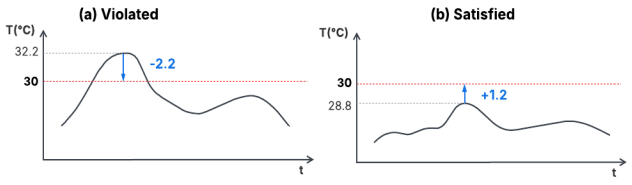

Fig. 1. Two temperature readings evaluated against the specification described by the formula (1).

B. Property Specification Patterns

Despite the effectiveness of formal specification languages, their complexity still constitutes a major challenge, requiring practitioners to be properly trained [9]. To bridge this gap, Dwyer *et al.* introduced *Property Specification Patterns*, a collection of formalism-independent abstractions of specifications [10]. The patterns were identified by analyzing multiple examples of property specification. For example, the *Response* pattern describes the case where the occurrence of one state or event must eventually be followed by the occurrence of another. In addition, patterns are combined with a *scope* that restricts the domain where the property should hold. Autili *et al.* extended this work by identifying more patterns and presenting a structured English grammar, enabling an easier specification process at a higher level of abstraction [6].

IV. DTChecker Tool

A. Architecture Overview

Fig. 2 presents the architectural overview of our DTChecker tool[1]. The domain experts write specifications in a browser-based text editor, which interacts with a language server to provide parsing, code completion, and validation.

Once finalized, the specifications are sent to a template-based code generator. This generator outputs a script where each specification is transformed into an STL formula and each signal is mapped to a corresponding RabbitMQ data stream. Signals may originate from sensor readings or other DT services, making the monitoring more flexible and enabling broader integration across components. The script is then automatically executed by the back-end server. From that point on, every signal value published by the message broker triggers the evaluation of the related specifications. Evaluation and robustness computation are handled by the *RTAMT* library [8].

The output of an evaluation is a robustness score, which is streamed back to the front-end in real time via a Server-Sent Events channel and is used to update the dashboard.

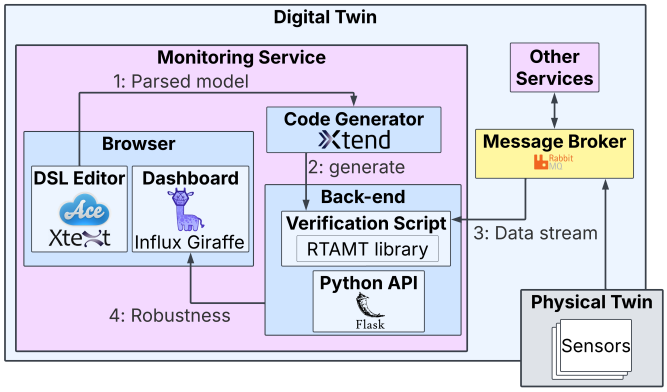

Fig. 2. DTChecker tool architecture.

B. Modeling Specifications

The specification of properties is carried out using a textual DSL based on an Xtext grammar described in listing 1. Each signal is defined by a *name* attribute, which serves as its identifier, and a *queueName* attribute, which specifies the name

[1]Available on GitHub (https://github.com/AbdelhamidRouatbi/DTChecker/)

of the RabbitMQ queue that provides its values. Specifications are defined by a *name* attribute, along with a combination of *scope* and *pattern*. In this combination, we refer to comparative boolean expressions where the operands may be signal values or constants to define the property we want to monitor.

Some patterns also require the definition of time bounds to constrain the temporal scope of the property. This is essential in the context of online monitoring, which cannot accommodate properties with unbounded or open-ended time intervals, as doing so would imply potentially waiting indefinitely for a condition to be satisfied or violated.

```
Model:
  Signals+=Signal*
  Specifications+=Specification*;

Signal:
  'signal' name=ID 'from' queueName=STRING;

Specification:
  'specification' name=ID ':'
  'scope' scope=Scope
  'pattern' pattern=Pattern;

Response:
  'if' p=Expression 'then-in-response'
      s=Expression ('eventually holds')?
      time=Time;

Expression:
  (s1=[Signal] | f1=Number) operator=('>' | '<'
       | '=') (s2=[Signal] | f2=Number);
```

Listing 1. DSL Grammar snippet

C. Code Generator

The Xtend template-based code generator handles the translation of specification patterns into temporal logic formulas. We adapted the detailed mapping table from patterns to Metric Temporal Logic provided by Autili *et al.* [6] to produce equivalent formulas in STL. Every combination of *scope* and *pattern* has a template to produce the corresponding STL formula. Listing 2 illustrates the template for the *Response* pattern with *Global* scope.

```
def static genGlobalScope(Specification spec){
   var pattern = spec.pattern as Response
   var p = expression(pattern.p)
   var s = expression(pattern.s)
   var t = time(pattern.time)
   return '''always[0,0](«p» implies(
       eventually «t» «s»)'''
}
```

Listing 2. Xtend template for generating STL formula for the *Response* pattern with *Global* scope.

D. Example Property

To demonstrate the usefulness of this DSL, let us consider the following property: *Between the time yeast is pitched and the end of active fermentation, if the temperature drops below 18°C, then it must return above 19°C within 1 hour.* The corresponding STL formula would be:

$$\square\big(\big(\neg(\text{fermentation} = 0) \wedge (\text{true}\ \mathcal{U}\ (\text{yeast\_pitched} = 1)) \\ \wedge (\text{temperature} < 18)\big) \rightarrow \Diamond_{[0,3600]}(\text{temperature} > 19)\big) \tag{2}$$

While this formula captures the intended timing and scope constraints, its low-level temporal logic syntax makes it difficult to read, understand, and maintain, especially for domain experts who are unfamiliar with formal methods. However, using our DSL, the same requirement can be expressed in a much more intuitive and readable form, as shown in listing 3.

```
1 specification TemperatureRegulation :
2   scope Between yeast_pitched=1 and
        fermentation=0
3   pattern Response:
4   if temperature<18 then-in-response
        temperature>19 within 1 h
```

Listing 3. Temperature regulation property for fermentation process.

In this specification, the *yeast_pitched* and *fermentation* signals are Boolean indicators denoting whether the yeast has been pitched and whether the fermentation process is active, respectively. The *temperature* signal is the temperature value inside the fermentation tank. We apply the *Response* pattern to express that, when the temperature drops below 18°C, it must rise above 19°C within one hour as a corrective action. The *Between* scope is used to express that this property should always hold between the time yeast is pitched and the end of active fermentation.

E. Usage

Before using the tool, it must first be configured by editing a configuration file. DT developers are required to specify the path to the Python executable, as well as the ports for the RabbitMQ server, the Python back-end, and the front-end. After launching the service using the `launch.py` script, the editor can be accessed in a web browser by navigating to `localhost:<front-end-port>` in a web browser.

From the web interface, domain experts can define signals and specify properties. The editor offers code completion features to simplify the process. Once monitoring begins by clicking on the button show in Fig 3.2, graphs automatically appear in the right-hand panel as evaluations are received from the back-end verifier. These graphs display the evolution of robustness over time for each specified property, shown in Fig. 3.4. Requirements that are violated are displayed in red, and those that are satisfied are shown in blue. Additionally, each defined signal has its own graph that visualizes its values over time, providing further insight.

If domain experts wish to focus on a particular property, they can select it from a tab menu, as shown in Fig. 3.3. This filters the view to show only the robustness graph of the

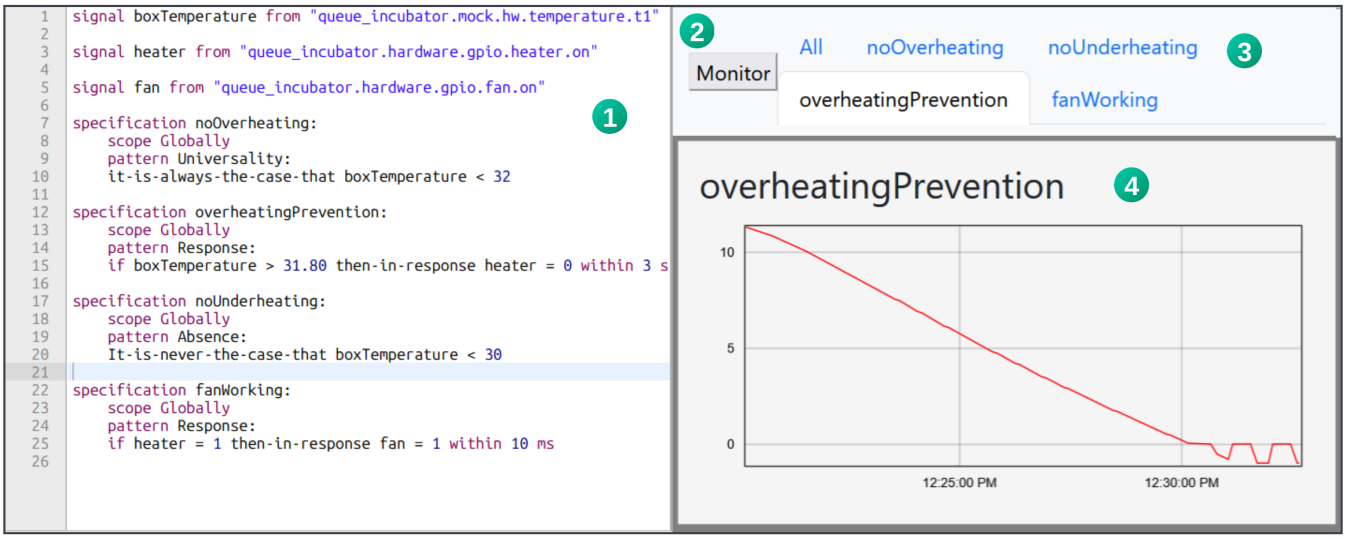

Fig. 3. Incubator monitoring.

selected property and the signal graphs that influence it. The set of signals affecting a given property can be identified by inspecting the abstract syntax tree of the model.

F. Integration into DTs

The tool is implemented as a self-contained monitoring service that can be seamlessly integrated into DT architectures using RabbitMQ as a message broker. RabbitMQ was chosen for its widespread adoption and reliability in cyber-physical system deployments. Because DTChecker interacts only with RabbitMQ, all physical system details are abstracted away, enabling integration with different platforms and configurations.

However, the tool extracts only signal values from the messages and assumes the evaluation time as the timestamp of the reading, without accounting for communication delays. This limitation makes the tool unsuitable for monitoring systems where precise timing is critical, particularly when evaluating properties defined over very short time intervals.

V. Demonstration: Incubator Digital Twin

We demonstrate our tool by integrating it into a free open-source project of an incubator DT [11].

The incubator is a compact cyber-physical system designed to regulate and maintain a target temperature within an insulated container. It comprises a styrofoam box equipped with a heatbed for thermal actuation, a fan for air circulation, and multiple temperature sensors for feedback.

A Raspberry Pi serves as the controller, executing a simple control strategy and interfacing with all components via a custom PCB. Communication between software modules and the DT is handled through a RabbitMQ server. The system supports real-time monitoring and actuation, making it suitable for evaluating DT technologies.

We use our tool to monitor four functional and/or safety-critical properties of the incubator:

- The temperature inside the box must never exceed 32°C.
- If the temperature inside the box gets close to 32°C, the heater must turn off shortly after.
- The temperature inside the box must never drop below 30°C.
- The fan must be activated whenever the heater is on.

Fig. 3.1 illustrates how these requirements are modeled using the DSL.

Fig. 4 shows the generated graphs after 12 minutes of monitoring. The *overheatingPrevention* requirement is violated because robustness is negative in the first plot. This indicates that the system fails to respond appropriately when the temperature exceeds the desired range. As a consequence, the *noOverheating* requirement is also violated. By hovering over the graph, we can read robustness values at specific timestamps to assess the severity of the violations. However, the *noUnderheating* and *fanWorking* requirements remain satisfied throughout the observation period, indicating proper functioning of the incubator with respect to these specifications.

VI. Related Work

Multiple other works have addressed similar goals in runtime verification and monitoring in general and in the context of DTs [12]. As representative examples, NuRV extends the nuXmv model checker by enabling runtime verification of Linear Temporal Logic (LTL) properties [13]. It supports online and offline monitoring modes and generates monitoring code in multiple languages. Bernaerts *et al.* introduced a contract-based design approach for formalizing requirements of complex, safety-critical automotive components [14]. It uses property specification patterns to replace ambiguous natural language requirements. These patterns are automatically translated into STL formulas, which are then verified on simulation traces. While this work is similar to ours, we bring novelty by implementing online monitoring of a system's behavior.

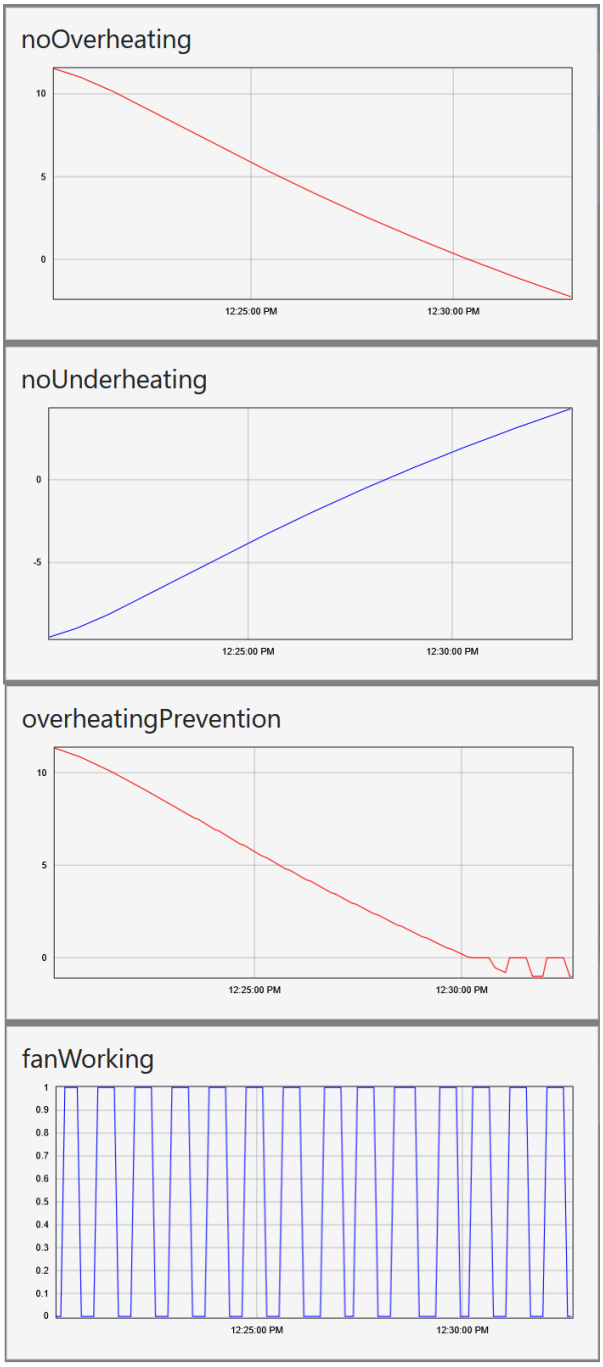

Fig. 4. Incubator monitoring dashboards

We follow a similar approach to Kristensen *et al.*, where LTL specifications of DTs are evaluated upon receiving input signals from a message broker [15].

VII. Conclusion

In this paper, we present DTChecker, a Web-based modular DT service that relies on a DSL to specify STL requirements and generates dashboards from these specifications to monitor robustness scores on live signal data from the system. DTChecker accelerates DT development by allowing domain experts to specify temporal requirements without writing low-level code. By providing a ready-to-use monitoring solution, DTChecker also reduces the need to develop custom services for temporal specification and evaluation.

Our primary focus has been on supporting DT development, but the underlying approach is general and can be applied to other systems requiring temporal requirement monitoring.

Future work will focus on improving the visualization of the graphs and containerizing the tool to facilitate its deployment. We will also address the current limitations on the STL specifications that require strict timing constraints and the assumption of noise-free sensor data, which may not hold in real-world scenarios.

References

[1] P. Talasila, C. Gomes, L. B. Vosteen, H. Iven, M. Leucker, S. Gil, P. H. Mikkelsen, E. Kamburjan, and P. G. Larsen, "Composable digital twins on digital twin as a service platform," *SIMULATION*, p. 00375497241298653, 2024.

[2] M. Frasheri, P. Katsaros, A. Iosifidis, S. T. Hansen, C. Gomes, V. Tang, and P. G. Larsen, "System monitoring through a digital twin," in *The Engineering of Digital Twins*, pp. 189–207, Springer, 2024.

[3] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, *Introduction to Runtime Verification*, pp. 1–33. Cham: Springer Publishing, 2018.

[4] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems* (Y. Lakhnech and S. Yovine, eds.), (Berlin, Heidelberg), pp. 152–166, Springer Berlin Heidelberg, 2004.

[5] P. Bellini, P. Nesi, and D. Rogai, "Expressing and organizing real-time specification patterns via temporal logics," *Journal of Systems and Software*, vol. 82, no. 2, pp. 183–196, 2009.

[6] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang, "Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar," *IEEE Transactions on Software Engineering*, vol. 41, no. 7, pp. 620–638, 2015.

[7] P.-E. Goffi, R. Tremblay, and B. Oakes, "Engineering a digital twin for the monitoring and control of beer fermentation sampling," in *Proceedings of the 28th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 2025.

[8] D. Nickovic and T. Yamaguchi, "RTAMT: online robustness monitors from STL," *CoRR*, vol. abs/2005.11827, 2020.

[9] C. Hahn, F. Schmitt, J. J. Tillman, N. Metzger, J. Siber, and B. Finkbeiner, "Formal specifications from natural language," 2022.

[10] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, (NY, USA), p. 411–420, Association for Computing Machinery, 1999.

[11] H. Feng, C. Gomes, C. Thule, K. Lausdahl, M. Sandberg, and P. G. Larsen, "The incubator case study for digital twin engineering," 2021. Code for the incubator DT is found at https://github.com/INTO-CPS-Association/example_digital-twin_incubator.

[12] Z. Hóu, Q. Li, E. Foo, J. S. Dong, and P. de Souza, "A digital twin runtime verification framework for protecting satellites systems from cyber attacks," in *2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pp. 117–122, 2022.

[13] A. Cimatti, C. Tian, and S. Tonetta, "Nurv: A nuxmv extension for runtime verification," in *Runtime Verification* (B. Finkbeiner and L. Mariani, eds.), (Cham), pp. 382–392, Springer International Publishing, 2019.

[14] M. Bernaerts, B. Oakes, K. Vanherpen, B. Aelvoet, H. Vangheluwe, and J. Denil, "Validating industrial requirements with a contract-based approach," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pp. 18–27, 2019.

[15] M. Kristensen, A. Bonizzi, C. Gomes, S. Hansen, C. Isasa, H. Iven, E. Kamburjan, P. Larsen, M. Leucker, P. Talasila, V. Tang, S. Tonetta, L. Vosteen, and T. Wright, "Runtime verification of autonomous systems utilizing digital twins as a service," pp. 121–127, 09 2024.