Trace-Checking Signal-based Temporal Properties: A Model-Driven Approach

Chaima Boufaied
University of Luxembourg
chaima.boufaied@uni.lu

Claudio Menghi
University of Luxembourg
claudio.menghi@uni.lu

Domenico Bianculli
University of Luxembourg
domenico.bianculli@uni.lu

Lionel Briand
University of Luxembourg
University of Ottawa
lionel.briand@uni.lu

Yago Isasi Parache
LuxSpace Sàrl
Isasi@luxspace.lu

ABSTRACT

Signal-based temporal properties (SBTPs) characterize the behavior of a system when its inputs and outputs are signals over time; they are very common for the requirements specification of cyber-physical systems. Although there exist several specification languages for expressing SBTPs, such languages either do not easily allow the specification of important types of properties (such as spike or oscillatory behaviors), or are not supported by (efficient) trace-checking procedures.

In this paper, we propose SB-TemPsy, a novel model-driven trace-checking approach for SBTPs. SB-TemPsy provides (i) SB-TemPsy-DSL, a domain-specific language that allows the specification of SBTPs covering the most frequent requirement types in cyber-physical systems, and (ii) SB-TemPsy-Check, an efficient, model-driven trace-checking procedure. This procedure reduces the problem of checking an SB-TemPsy-DSL property over an execution trace to the problem of evaluating an Object Constraint Language constraint on a model of the execution trace.

We evaluated our contributions by assessing the expressiveness of SB-TemPsy-DSL and the applicability of SB-TemPsy-Check using a representative industrial case study in the satellite domain. SB-TemPsy-DSL could express 97% of the requirements of our case study and SB-TemPsy-Check yielded a trace-checking verdict in 87% of the cases, with an average checking time of 48.7 s. From a practical standpoint and compared to state-of-the-art alternatives, our approach strikes a better trade-off between expressiveness and performance as it supports a large set of property types that can be checked, in most cases, within practical time limits.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; **Specification languages**.

KEYWORDS

trace checking, run-time verification, temporal properties, specification patterns, model-driven, cyber-physical systems, signals

ACM Reference Format:
Chaima Boufaied, Claudio Menghi, Domenico Bianculli, Lionel Briand, and Yago Isasi Parache. 2020. Trace-Checking Signal-based Temporal Properties: A Model-Driven Approach. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416631

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416631

1 INTRODUCTION

Trace checking is a *run-time verification* [6] technique that uses an automated decision procedure to check properties (based on a system's requirements) on traces representing system executions. Trace-checking tools are used for software verification & validation (V&V) activities as test oracles [38] and run-time monitors [46]. In this paper we consider a specific subset of trace-checking tools, which can be defined using the concepts recently introduced in a taxonomy for run-time verification tools [26]:

- supporting *explicit, declarative, temporal specifications*, i.e., tools that require users to formally express the requirements to be checked, using a declarative specification formalism that allows for expressing constraints over time;
- deployed at the *offline* stage, i.e., tools that run after the system has finished its execution, which has been recorded in traces;
- yielding a *verdict* as *output*, i.e., an indication (e.g., a Boolean value) of whether the input trace satisfies the property being checked.

In such tools, the requirements to check are expressed using a declarative specification formalism such as (temporal) logic-based or domain-specific languages. Temporal logic-based languages (e.g., LTL, MTL, MLTL [34], QTL [30]) provide mathematical-based constructs to express *arbitrarily complex* requirements using a basic set of temporal operators. Domain-specific languages (DSLs) for temporal specifications (e.g., PROPEL - DNL [47], Structured English Grammar for real-time specifications [33], Temporal OCL [32], OCLR [18], VISPEC - graphical formalism [31], TemPsy [19], TemPsy-AG [10], ProMoboBox - property language [39], FRETISH [29]) provide a set of predefined constructs that concisely capture certain types of requirements that are *specific to* a certain domain, possibly relying on property specification patterns [2, 13, 21]. In terms of applicability for requirement specifications, logic-based languages typically require a strong mathematical background, limiting their

adoption among practitioners. On the other hand, DSLs are more accessible for domain experts since they make available, as first-class constructs in the language, concepts (or patterns) that are specific to the domain. For example, two recent empirical studies [14, 15] provide evidence that high-level languages based on property specification patterns result into a higher level of understandability than logic-based languages like LTL.

Typically, for both types of specification languages, the expressiveness of the language is inversely related to the efficiency of the corresponding trace-checking procedure. Therefore, the main challenge faced when defining a trace-checking approach suitable for industrial contexts, is finding a reasonable trade-off between these two conflicting aspects.

In this work, we consider the problem of trace-checking requirements expressed in terms of signal-based temporal properties (SBTPs). SBTPs characterize the behavior of the system when its inputs and outputs are signals over time; they are used in a variety of cyber-physical system (CPS) domains, including aerospace, automotive, and defense [41]. An example of an SBTP, coming from our case study in the satellite domain, is: "*The velocity of the satellite along the X-axis shall oscillate with a maximum amplitude of* 8000 km/h *and a maximum period of* 180 min", where "the velocity of the satellite along the X-axis" is a signal. This property requires the system to ensure that the signal exhibits an oscillatory behavior with certain parameters (in this case, the "amplitude" and the "period"). SBTPs are usually evaluated on traces that are collected by recording the values of signals over time. Trace entries can be recorded at fixed or variable sampling rates, meaning that trace entries are recorded at time instants that are separated by fixed or variable-time intervals, respectively. System and software engineers have to assess, either manually or by means of tools, whether the recorded traces satisfy or violate the system requirements. Although there exist several logic-based (e.g., STL [36], STL* [12], SFO [4], RFOL [38]) and domain-specific [7] languages that have been proposed in the literature to express SBTPs, such languages either do not support the specification of important types of properties [11], or are not supported by (efficient) trace-checking procedures [4, 12].

In this paper, we propose *SB-TemPsy*, a trace-checking approach for SBTPs that strikes a good balance in industrial contexts as it can be efficiently trace-checked and covers the most important types of properties in practice across CPS domains. SB-TemPsy provides:

- *SB-TemPsy-DSL*, a domain-specific language that allows the specification of SBTPs covering the most frequent requirement types in CPS domains;
- an efficient trace-checking procedure, implemented in a prototype tool called *SB-TemPsy-Check*.

SB-TemPsy-DSL is a pattern-based specification language. It has been defined in collaboration with system engineers in the satellite domain and supports the specification of the most common types of SBTPs in CPS, recently identified in a taxonomy [11]. SB-TemPsy-DSL differs from the pattern-based specification languages for temporal properties mentioned above, since it is tailored to express SBTPs. Its syntax provides constructs that allow engineers to specify in a simple and precise way complex signal-based behaviors such as spikes and oscillations, without requiring a strong theoretical background in logic-based languages for SBTPs.

Our trace-checking procedure is based on the idea of *model-driven trace checking*, originally proposed by Dou et al. [19] for the verification of temporal properties based on Dwyer et al. [21]'s specification patterns (and thus not supporting SBTPs). Using a model-driven trace checking approach, we *reduce* the problem of checking an SB-TemPsy-DSL property over an execution trace to the problem of evaluating an Object Constraint Language (OCL) constraint, that is semantically equivalent to the SB-TemPsy-DSL property, on a model of the execution trace. We made this choice since OCL is a standardized constraint specification language defined by OMG [43] and, as a result, is supported by a mature constraint checking technology, such as the constraint checker included in Eclipse OCL [22]. Based on these observations and the encouraging efficiency results reported in the literature for model-driven trace checking approaches [10, 19], we surmised that this choice would allow the development of a trace-checking tool able to analyze complex requirements on real-world execution traces within practical time limits. The evaluation of this conjecture was part of our empirical investigation.

We evaluated our solution by assessing the expressiveness of our specification language SB-TemPsy-DSL and the applicability of our trace-checking tool SB-TemPsy-Check to a representative industrial case study in the satellite domain; we also compared them to state-of-the-art approaches. Using SB-TemPsy-DSL, we could express 98 out of 101 requirements of our case study. SB-TemPsy-DSL was considerably more expressive than STL, which is supported by publicly available trace-checking tools. Furthermore, in most cases ($\approx$ 87%), SB-TemPsy-Check completed the verification of these requirements on industrial execution traces within a set time-out of two hours, which we deemed practical based on the development context of our case study. Overall, the results of our empirical investigation show that SB-TemPsy represents a viable trade-off between an expressive specification language for SBTPs (SB-TemPsy-DSL) and an efficient trace-checking procedure (SB-TemPsy-Check). Furthermore, the results suggest that SB-TemPsy could be combined with existing approaches efficiently supporting STL. In this way, we show we can make optimal use of a given verification budget while avoiding most time-outs by relying on the best tool option depending on the type of the checked property.

The rest of this paper is organized as follows. Section 2 presents our case study in the satellite domain and discusses the motivations for this work. Section 3 explains the notation and the background concepts used in the rest of the paper. Section 4 provides an overview of SB-TemPsy, further detailed in section 5 (which presents SB-TemPsy-DSL) and in section 6 (which presents our model-driven trace checking procedure implemented in SB-TemPsy-Check). Section 7 reports on the empirical investigation of our contributions. Section 8 discusses related work. Section 9 concludes the paper and provides directions for future work.

2 CASE STUDY AND MOTIVATIONS

LuxSpace [35], our industrial partner, has developed, in collaboration with ESA [24] and ExactEarth [25], a maritime micro-satellite to collect AIS (automatic identification system) tracking information from vessels operating on Earth and to relay those data to the ground.

Throughout the satellite development, our partner follows different development phases. This work is set in the context of the *design phase* (i.e., phases B–C in the satellite domain [23])[1], which includes several activities, such as the definition of the system requirements and interfaces, the definition of the spacecraft, payload, launcher and ground segment, and the design and development of the on-board satellite software (OBSW).

The OBSW is a complex mission-critical software component, which includes several modules that control and monitor the operations and the physical behavior of the satellite. Its main modules are the *on-board data handling* (controlling the satellite platform and payload, and collecting and storing the data), the *electric power system* (regulating the power of the satellite), the *telemetry and telecommand* (receiving tele-commands from and sending data to the ground), the *thermal control* (ensuring that each component of the satellite remains within its operational temperature ranges), and the *ADCS - attitude determination and control system* (estimating and regulating the satellite attitude).

V&V play a crucial role during the development of the OBSW, given the complexity of the physical behavior being controlled by the OBSW, and the various hardware components (e.g., sensors and actuators) and software modules involved. For example, a typical V&V step is an in-depth testing of the ADCS module of the OBSW [37, 38].

At the very last stage of the development, testing is performed on the actual hardware-based facilities and involves all the software modules of the OBSW. As in many other cyber-physical domains, our industrial partner relies on a multi-purpose simulator [44] that supports test execution both on the actual hardware components of the satellite and on software stubs replacing such components. This feature allows developers to anticipate testing activities, when hardware components are not yet available, and to reduce the risk of hardware damages, by running high-risk test cases relying on simulation and thus ensuring beforehand there is no unexpected and harmful behavior. During OBSW testing, a specific configuration of the satellite (software/hardware components) is loaded on the simulator for each test case; then, the satellite behavior over a given time is simulated and data are collected in traces[2]. Due to the complexity of the physical models involved during simulation, itself resulting from the complex physical behavior being controlled by OBSW, the time to complete a simulation and generate the corresponding trace is on the order of several days.

In the current practice, LuxSpace engineers are using ad hoc solutions to inspect whether these traces satisfy the system requirements. In the case of the OBSW, requirements are complex properties constraining the behavior of an input or output signal (e.g., in terms of the type of oscillations allowed); we will present examples of these properties in section 3.2.

In this context, the main challenge is to automate the checking of system requirements (expressed as complex SBTPs) on simulation traces, by means of an efficient offline run-time verification (i.e., trace-checking) procedure.

[1]We remark that our solution can also be applied in phases D–E.

[2]Note that simulation is used in its broad sense, generally indicating a simulation scenario. As such, a simulation also considers the case in which all the hardware components of the satellite are in place.

Although this case study is set in the satellite domain, it is in many ways representative of other complex cyber-physical domains, where the behavior being controlled involves convoluted physical dynamics (such as the satellite attitude) and the system requirements are therefore expressed as complex properties on the shape of the input and output signals (e.g., spikes and oscillations).

3 BACKGROUND

3.1 Traces

A simulation trace, yielded at the end of the simulation, contains entries with the value of a subset of the signals and the simulation time at which they were recorded. More precisely, let $S = \{s_1, s_2 \dots s_n\}$ be a set of signals. We use the symbol T to indicate the sequence $T = (t_1, t_2, \dots, t_m)$ containing the progressive simulation times associated with the entries of a simulation trace. We also use $s_i = v$ to denote a record assigning the value $v \in \mathbb{R}$ to signal s_i. Let $\mathcal{A}$ be the universe of all possible assignments for the signals in S. A *trace* π is a tuple $\langle T, f \rangle$, with T defined as above and $f\colon T \to 2^{\mathcal{A}}$. An entry is a tuple $\langle t, f(t) \rangle$ that contains the simulation time $t \in T$ at which the entry was sampled, and a set of records $f(t)$ that specify the values of the signals at time t. We say that a signal s is assigned a value v at time t if there exists a record $(s = v) \in f(t)$, for $t \in T$.

We now introduce some useful notations used in the rest of the paper. Let $\pi = \langle T, f \rangle$ be a trace, s be an arbitrary element of S (i.e., a signal) and $t \in T$ be a simulation time. The *initial value* of s, denoted by $init(\pi, s)$, is v if v is the value assigned to s through the first assignment performed on s in π; we assume that such an initial assignment always exists, as part of the initialization of the simulation. The *last-seen value* of s at time t, denoted by $last(\pi, s, t)$, is the value v if (1) s is assigned v at time t or (2) s was assigned v in the most recent assignment to s in π at a simulation time preceding t or (3) v is equal to $init(\pi, s)$ if s still has not been assigned a value since the initialization. The *next-seen* value of s at time t, denoted by $next(\pi, s, t)$, is the value v if (1) s is assigned v at time t or (2) s is assigned v in the first assignment to s in π at a simulation time ensuing t.

3.2 Signal-based temporal properties (SBTPs)

CPS requirements are specified using SBTPs. In this section, we present—with examples from our case study—the main types of SBTP proposed in a recent taxonomy [11].

Data Assertion. It specifies a constraint on the value of a signal. Example (pDA): *The beta angle shall vary between* $+90°$ *and* $-90°$.

Spike. It is a large increase or decrease of the value of a signal. When it represents an increase of the signal value, a spike is characterized by three extrema, shown in Figure 1a: a local maximum p_2 surrounded by two local minima p_1 and p_3. A spike behavior can be characterized in terms of two features: the *width*, defined as $|t_3 - t_1|$; the *amplitude*, defined as $\max(a_1, a_2)$, where a_1 is the amplitude of the first-half of the spike shape $a_1 = |v_2 - v_1|$ and a_2 is the amplitude of the second-half of the spike shape $a_2 = |v_3 - v_2|$. Example (pSK): *The beta angle shall show a spike with an amplitude less than* $90°$.

Oscillation. It is a repeated variation over time of the value of a signal. An oscillation occurs when the signal value swings from one extremum to the adjacent extremum of the same type, by traversing

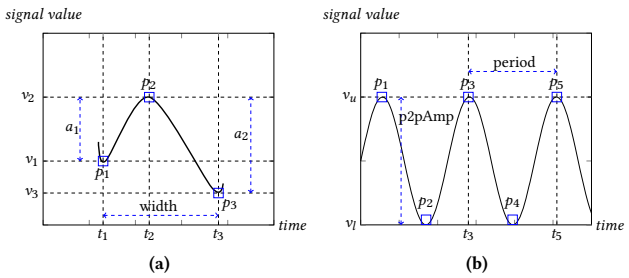

Figure 1: Main features (adapted from [11]) used to characterize (a) a spike and (b) an oscillatory behavior

an extremum of the other type. For instance, in figure 1b one can see an oscillation when the signal goes from p_1 to p_3 (two peak points) through p_2 (a valley point), and another one when the signal goes from p_3 to p_5 through p_4. An oscillation can be characterized in terms of the *period*, which is the time required to perform a complete oscillation (e.g., $t_5 - t_3$), and the *peak-to-peak amplitude (p2pAmp)*, which is the difference between two adjacent extrema (e.g., $v_u - v_l$). Example (pOS): *The velocity of the satellite along the X-axis shall oscillate with a maximum amplitude of* 8000 km/hour *and a maximum period of* 180 min.

Rise Time. It is a constraint on the (transient) behavior of a signal, while it reaches—possibly monotonically—a target value. Its dual is called "fall time". Example (pRT): *The X-current of the sun sensor shall rise monotonically reaching the value of* 3650 µA.

Overshoot. It specifies a maximum value, above the target value, that a signal can reach when overshooting (i.e., when the signal exceeds its target value); its dual is called "undershoot".
Example (pOV): *The X current of the sun sensor shall overshoot* 3650 µA *by at most* 50 µA.

Order Relationship (Response). It captures the *response* pattern proposed by Dwyer et al. [21]. It expresses a constraint over the sequence of a pair of *cause* and *effect* events, possibly with a temporal distance between the two. Example (pOR): *If the Safe Mode Convergence Status is equal to "B-dot Converged mode", then it should become equal to "Sun Acquired mode" within at most 20 minutes.*

4 THE SB-TEMPSY APPROACH

Model-driven trace checking [19] is an approach that reduces the problem of checking a temporal property ϕ over a trace π to the problem of evaluating an OCL constraint, that is semantically equivalent to ϕ, on a model of the trace (equivalent to π).

In this paper, we present our model-driven approach, *SB-TemPsy*, for trace checking of SBTPs. We have decided to follow a model-driven paradigm for trace checking because OCL is a standardized constraint specification language defined by OMG [43] and, as a result, is supported by a mature constraint checking technology, such as the constraint checker included in Eclipse OCL [22]. Furthermore, existing approaches for model-driven trace checking (for checking linear temporal logic properties [19] and properties with temporal aggregations [10]) have been shown to yield encouraging efficiency results. Hence, we surmise that choosing a model-driven approach allows the development of a trace-checking tool able to analyze SBTPs on real-world execution traces within practical time limits; we report on the empirical investigation of this conjecture in section 7.

Our SB-TemPsy approach is illustrated in figure 2; it takes as input a trace π and a property to check ϕ; it returns a Boolean verdict, indicating whether π satisfies or violates property ϕ. The trace π records the values of signals sampled at different simulation times, as discussed in section 3.1. The property ϕ represents a requirement of a CPS, expressed using one of the types of SBTPs illustrated in section 3.2.

We have defined an expressive, pattern-based domain-specific language to ease the specifications of such requirements as SBTPs. The language, called *SB-TemPsy-DSL* (Signal-Based Temporal Properties made easy), has been defined in collaboration with LuxSpace system engineers. It draws inspiration from TemPsy [19]—an existing pattern-based language for the specification of temporal properties—and supports the specification of the most common types of SBTPs in CPS (e.g., spike, oscillation), recently identified in a taxonomy [11] and presented in section 3.2.

SB-TemPsy-DSL differs from existing pattern-based specification languages for temporal properties (such as PROPEL - DNL [47], Structured English Grammar for real-time specifications [33], Temporal OCL [32], OCLR [18], VISPEC - graphical formalism [31], TemPsy [19], TemPsy-AG [10], ProMoboBox - property language [39]), since it is tailored to express SBTPs. Furthermore, differently from existing logic-based specification languages for SBTPs (e.g., STL [36], STL* [12], RFOL [38], SFO [4]), SB-TemPsy-DSL enables practitioners to specify in a precise way key requirements of CPS (which in many cases are not supported by the aforementioned languages, see [11]), without requiring a strong theoretical background.

Our approach for trace checking, called SB-TemPsy-Check and implemented in a prototype tool, includes two main steps: *pre-processing*, which prepares the trace for the verification, and *model-driven trace checking*, which computes the verification verdict.

Pre-processing. As discussed in section 3.1, the trace π is obtained by recording the values of a set of signals sampled at different simulation times; therefore, at a given simulation time, a signal may be unassigned. The pre-processing step analyzes the trace $\pi = \langle T, f \rangle$ and generates, using an interpolation function η, a new trace $\bar{\pi} = \langle T, \bar{f} \rangle$ that includes assignments to signals for the simulation times at which such signals were unassigned in π. We discuss function $\bar{f}$, the interpolation function η, and the pre-processing step in more detail in section 6.1.

Model-driven trace checking. This step (detailed in section 6.3) checks whether property ϕ (expressed in SB-TemPsy-DSL) holds over the trace π by (a) converting the pre-processed trace $\bar{\pi}$ into an instance of a trace meta-model; (b) evaluating an OCL constraint semantically equivalent to ϕ over the model of $\bar{\pi}$.

5 THE SB-TEMPSY-DSL LANGUAGE

5.1 Syntax

The syntax of SB-TemPsy-DSL is shown in figure 3; optional items are enclosed in square brackets; the symbol | separates alternatives. A *property* (non-terminal ϕ) is defined using a *scope* (non-terminal sc) or as a Boolean expression over other properties.

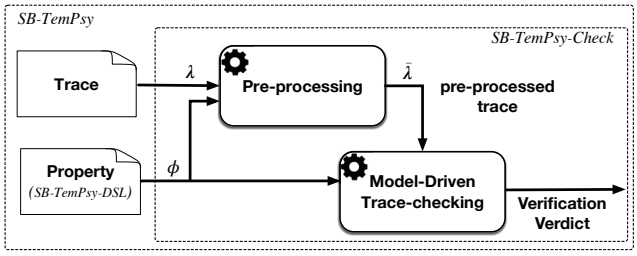

Figure 2: Overview of SB-TemPsy

Property	ϕ ::=	ϕ_1 `and` ϕ_2 | ϕ_2 `or` ϕ_1 | `not` ϕ | sc
Scope	sc ::=	`globally` p | `before` t p | `after` t p | `at` t p | `before` p_1 p `after` p_1 p | `between` t_1 `and` t_2 p | `between` p_1 `and` p_2 p
Pattern	p ::=	`assert` c | s `becomes` ~ v | `if` p_1 `then` [`within` ⋈ t] p_2 | `exists spike in` s [`with` [`width` ~ v_1][`amplitude` ~ v_2]] | `exist oscillations in` s [`with` [`p2pAmp` ~ v_1][`period` ~ v_2]] | s `rises` [`monotonically`] `reaching` v | s `falls` [`monotonically`] `reaching` v | s `overshoots` [`monotonically`] v_1 `by` v_2 | s `undershoots` [`monotonically`] v_1 `by` v_2
	⋈ ::=	`exactly` | `at most` | `at least`
Condition	c ::=	c_1 `and` c_2 | c_1 `or` c_2 | `not` c | s ~ v

t, t_1, t_2 ∈ ℝ; v, v_1, v_2 ∈ ℝ; ~∈ {<, >, =, ≠, ≤, ≥};
s is a signal in S or a mathematical expression over the signals in S

Figure 3: SB-TemPsy-DSL syntax

A scope operator constrains a *pattern* (non-terminal p) to hold within a given time interval delimited either by absolute time instants (denoted by, t, t_1, and t_2) or by events, i.e., occurrences of a pattern, (denoted by p, p_1, and p_2). For example, the "`between` t_1 `and` t_2 p" scope operator specifies that pattern p holds between the time instants t_1 and t_2. The scope operators supported by SB-TemPsy-DSL are inspired by those proposed by Dwyer et al. [21] (`globally`, `before`, `after`, `between and`): they support not only events (i.e., in the form of occurrences of a pattern) but also absolute time instants as scope boundaries. Furthermore, SB-TemPsy-DSL includes a punctual scope (`at`) to reference a specific time instant.

A pattern specifies a constraint on the behavior of one or more signals. For example, pattern "`exist oscillations in` s `with p2pAmp` v_1 `period` v_2" specifies that the value of signal s shows an oscillatory behavior with a peak-to-peak amplitude (denoted by the keyword `p2pAmp`) equal to value v_1 and a period (denoted by the keyword `period`) that is equal to value v_2; it corresponds to the property type "oscillation" discussed in section 3.2.

A *condition* (non-terminal c) is a logical predicate on a signal s (of the form "s ~ v", with ~∈ {<, >, =, ≠, ≤, ≥}) or a logical expression over predicates.

Below we show how the example properties from our case study, presented in English in section 3.2, can be expressed in SB-TemPsy-DSL (for simplicity, we omitted the corresponding scopes):

```
pDA  assert (s1 >= -90 and s1 <= 90)
pSK  exists spike in s2 with amplitude < 90
pOS  exist oscillations in s3 with p2pAmp <= 8000
     with period <= 10800
pRT  s4 rises monotonically reaching 3650
pOV  s4 overshoots 3650 by 50
pOR  if assert (s5 == 1) then
     assert (s5 == 2) within at most 120
```

5.2 Formal Semantics

Figure 4 presents the formal semantics of SB-TemPsy-DSL. The semantics is defined over a pre-processed trace $\bar{\pi} = \langle T, \bar{f} \rangle$, with $T = (t_1, t_2, \ldots, t_m)$; optional items are enclosed in square brackets with a Greek letter subscript. We use the notation $f_p(t, s)$ to denote the value x assigned to signal s at time t in the pre-processed trace $\bar{\pi} = \langle T, \bar{f} \rangle$, if $(s = x) \in \bar{f}(t)$ for $t \in T$. We omit the semantic definition of language elements for which the dual is available (i.e., fall time vs rise time, Overshoot vs Undershoot) and whose semantics can be easily derived.

Conditions are constraints on signals that should hold punctually, i.e., in a specific time instant $t \in T$. We use the notation $\bar{\pi}, t \models$ c to indicate that condition c holds in the pre-processed trace $\bar{\pi}$ at time t. For example, the predicate s ~ v holds at time t if the value $f_p(t, s)$ assigned to signal s at time t satisfies the predicate $f_p(t, s)$ ~ v.

Patterns are evaluated over a time interval $[t_l, t_u]$, where $t_l, t_u \in T$ and $t_l < t_u$. We use the notation $\bar{\pi}, [t_l, t_u] \models$ p to indicate that pattern p holds in the pre-processed trace $\bar{\pi}$ within the time interval $[t_l, t_u]$. The semantics of the spike and oscillation patterns rely on the auxiliary predicates *uni_m_min*, *uni_sm_min*, *uni_m_max*, and *uni_sm_max* defined in Table 1. These predicates evaluate to true if, within a given time interval, the signal has an extremum (minimum or maximum) and its value changes in a certain way (see column "Description" in Table 1 for the complete description). For example, in the case of the oscillation pattern, the semantics requires the signal to exhibit a maximum followed by a minimum followed by a maximum, or viceversa (see also section 3.2). In addition, in the first case, the value of the signal shall increase strictly monotonically (1) before the first maximum and (2) between the minimum and the second maximum, and shall decrease strictly monotonically (1) between the first maximum and the minimum and (2) after the second maximum.

Also *scopes* are evaluated over a time interval $[t_l, t_u]$. For example, the semantics of the "`between` t_1 `and` t_2 p" scope operator evaluates pattern p in the time interval $[t_1, t_2]$.

The semantics of a *property* is defined by evaluating the satisfaction of a scope sc within the time interval $[t_1, t_m]$, where t_1 and t_m are, respectively, the first and last simulation time in the time sequence T of the pre-processed trace $\bar{\pi}$. The semantics of properties obtained by composing other properties through Boolean operators follows the standard semantics of such operators.

Finally, we define the semantics for checking an SB-TemPsy-DSL property over an input trace. Let π be a trace, η be an interpolation function, and ϕ be an SB-TemPsy-DSL property. We say that the trace π satisfies the property ϕ (when using the interpolation function η in the pre-processing), denoted by $\pi \models_\eta \phi$, if the pre-processed trace $\bar{\pi}$ obtained from π using the interpolation function η is such that $\bar{\pi} \models \phi$.

Table 1: Definition of predicates *uni_m_max*, *uni_sm_max* (predicates *uni_m_min*, *uni_sm_min* are the dual)

Predicate	Mathematical Formulation	Description
$uni\_m\_max(\bar{\pi}, s, t, [t_l, t_u])$	$f_p(t, s) = x$ and $\forall t_1 \in [t_l, t_u], f_p(t_1, s) < x$ and $\forall t_1, t_2 \in [t_l, t]$, if $t_1 < t_2$ then $f_p(t_1, s) \leq f_p(t_2, s)$ and $\forall t_1, t_2 \in [t_l, t]$, if $t_1 < t_2$ then $f_p(t_1, s) \geq f_p(t_2, s)$	The value x of signal s at time instant t is the minimum value assigned to s within the interval $[t_l, t_u]$. Furthermore, the value of x changes according to a unimodal function, i.e., for every time instant in $[t_l, t]$ the value of s is monotonically increasing and for every time instant in $[t, t_u]$ the value of s is monotonically decreasing.
$uni\_sm\_max(\bar{\pi}, s, t, [t_l, t_u])$	$f_p(t, s) = x$ and $\forall t_1 \in [t_l, t_u], f_p(t_1, s) < x$ and $\forall t_1, t_2 \in [t_l, t]$, if $t_1 < t_2$ then $f_p(t_1, s) < f_p(t_2, s)$ and $\forall t_1, t_2 \in [t_l, t]$, if $t_1 < t_2$ then $f_p(t_1, s) > f_p(t_2, s)$	As above, except that for every time instant in $[t_l, t]$ the value of s is *strictly* monotonically increasing and for every time instant in $[t, t_u]$ the value of s is *strictly* monotonically decreasing.

Property

$\bar{\pi} \models \text{sc} \Leftrightarrow \bar{\pi}, [t_1, t_m] \models \text{sc}$
$\bar{\pi} \models \phi_1$ **and** $\phi_2 \Leftrightarrow (\bar{\pi} \models \phi_1)$ and $(\bar{\pi} \models \phi_2)$
$\bar{\pi} \models \phi_1$ **or** $\phi_2 \Leftrightarrow (\bar{\pi} \models \phi_1)$ or $(\bar{\pi} \models \phi_2)$
$\bar{\pi} \models$ **not** $\phi \Leftrightarrow (\bar{\pi} \not\models \phi)$

Condition

$\bar{\pi}, t \models \text{s} \sim \text{v} \Leftrightarrow f_p(t, s) \sim \text{v}$
$\bar{\pi}, t \models \text{c}_1$ **and** $\text{c}_2 \Leftrightarrow (\bar{\pi}, t \models \text{c}_1)$ and $(\bar{\pi}, t \models \text{c}_2)$
$\bar{\pi}, t \models \text{c}_1$ **or** $\text{c}_2 \Leftrightarrow (\bar{\pi}, t \models \text{c}_1)$ or $(\bar{\pi}, t \models \text{c}_2)$
$\bar{\pi}, t \models$ **not** $\text{c} \Leftrightarrow (\bar{\pi}, t \not\models \text{c})$

Absolute Scope

$\bar{\pi}, [t_l, t_u] \models$ **globally** $\text{p} \Leftrightarrow \bar{\pi}, [t_l, t_u] \models \text{p}$
$\bar{\pi}, [t_l, t_u] \models$ **before** t $\text{p} \Leftrightarrow t_l \leq \text{t} \leq t_u$ and $\bar{\pi}, [t_l, \text{t}] \models \text{p}$
$\bar{\pi}, [t_l, t_u] \models$ **after** t $\text{p} \Leftrightarrow t_l \leq \text{t} \leq t_u$ and $\bar{\pi}, [\text{t}, t_u] \models \text{p}$
$\bar{\pi}, [t_l, t_u] \models$ **between** n **and** m $\text{p} \Leftrightarrow t_l \leq \text{n} < \text{m} \leq t_u$ and $\bar{\pi}, [\text{n}, \text{m}] \models \text{p}$
$\bar{\pi}, [t_l, t_u] \models$ **at** t $\text{p} \Leftrightarrow \exists t : t_l \leq \text{t} \leq t_u$ and $\bar{\pi}, [\text{t}, \text{t}] \models \text{p}$

Event Scope

$\bar{\pi}, [t_l, t_u] \models$ **before** p_1 $\text{p} \Leftrightarrow \forall t_1, t_2, t_l < t_1 < t_2 \leq t_u, \bar{\pi}, [t_1, t_2] \models \text{p}_1$ and $\exists t_3, t_4, t_l < t_3 < t_4 < t_1, \bar{\pi}, [t_3, t_4] \models \text{p}$
$\bar{\pi}, [t_l, t_u] \models$ **after** p_1 $\text{p} \Leftrightarrow \forall t_1, t_2, t_l < t_1 < t_2 \leq t_u, \bar{\pi}, [t_1, t_2] \models \text{p}_1$ and $\exists t_3, t_4, t_2 < t_3 < t_4 < t_u, \bar{\pi}, [t_3, t_4] \models \text{p}$
$\bar{\pi}, [t_l, t_u] \models$ **between** p_1 **and** p_2 $\text{p} \Leftrightarrow \forall t_1, t_2, t_3, t_4, t_l \leq t_1 < t_2 < t_3 < t_4 \leq t_u,$ $(\bar{\pi}, [t_1, t_2] \models \text{p}_1$ and $[t_3, t_4] \models \text{p}_2) \Rightarrow \bar{\pi}, [t_2, t_3] \models \text{p}$

Data Assertion Pattern

$\bar{\pi}, [t_l, t_u] \models$ **assert** $\text{c} \Leftrightarrow \forall t \in [t_l, t_u], (\bar{\pi}, t \models \text{c})$
$\bar{\pi}, [t_l, t_u] \models \text{s}$ **becomes** $\sim \text{v} \Leftrightarrow \exists t \in (t_l, t_u], (f_p(t, \text{s}) \sim \text{v}$ and $\forall t_1 \in (t_l, t), (f_p(t_1, \text{s}) \nsim \text{v}))$

Spike Pattern

$\bar{\pi}, [t_l, t_u] \models$ **exists spike in** s $\left[\text{with } [\text{width} \sim \text{v}_1]_\beta [\text{amplitude} \sim \text{v}_2]_\gamma\right]_\alpha \Leftrightarrow \exists t_1, t_2, t_3 \in [t_l, t_u], t_1 < t_2 < t_3,$
$\Big((uni\_m\_min(\bar{\pi}, s, t_1, [t_l, t_2])$ and $uni\_sm\_max(\bar{\pi}, s, t_2, [t_1, t_3])$ and $uni\_m\_min(\bar{\pi}, s, t_3, [t_2, t_u]))$ or
$(uni\_m\_max(\bar{\pi}, s, t_1, [t_l, t_2])$ and $uni\_sm\_min(\bar{\pi}, s, t_2, [t_1, t_3])$ and $uni\_m\_max(\bar{\pi}, s, t_3, [t_2, t_u]))$
$\Big[[\text{and } |t_3 - t_1| \sim \text{v}_1]_\beta \left[\text{and } \max\left(\left|f_p(t_1, s) - f_p(t_2, s)\right|, \left|f_p(t_2, s) - f_p(t_3, s)\right|\right) \sim \text{v}_2\right]_\gamma\Big]_\alpha \Big)$

Oscillation Pattern

$\bar{\pi}, [t_l, t_u] \models$ **exist oscillations in** s $\left[\text{with } [\text{p2pAmp} \sim \text{v}_1]_\beta [\text{period} \sim \text{v}_2]_\gamma\right]_\alpha \Leftrightarrow \exists t_1, t_2, t_3, t_4, t_5 \in [t_l, t_u], t_1 < t_2 < t_2 < t_3 < t_4 < t_5,$
$\Big((uni\_sm\_min(\bar{\pi}, s, t_2, [t_1, t_3])$ and $uni\_sm\_max(\bar{\pi}, s, t_3, [t_2, t_4])$ and $uni\_sm\_min(\bar{\pi}, s, t_4, [t_3, t_5]))$ or
$(uni\_sm\_max(\bar{\pi}, s, t_2, [t_1, t_3])$ and $uni\_sm\_min(\bar{\pi}, s, t_3, [t_2, t_4])$ and $uni\_sm\_max(\bar{\pi}, s, t_4, [t_3, t_5]))$
$\Big[\left[\text{and } \left|f_p(t_2, s) - f_p(t_3, s)\right| \sim \text{v}_1 \text{ and } \left|f_p(t_3, s) - f_p(t_4, s)\right| \sim \text{v}_1\right]_\beta [\text{and } (t_4 - t_2) \sim \text{v}_2]_\gamma\Big]_\alpha \Big)$

Rise Time Pattern

$\bar{\pi}, [t_l, t_u] \models \text{s}$ **rises** $[\text{monotonically}]_\alpha$ **reaching** $\text{v} \Leftrightarrow \exists t \in (t_l, t_u], \Big(f_p(t, \text{s}) \geq \text{v}$ and $\forall t_1 \in (t_l, t), (f_p(t_1, \text{s}) < \text{v})$
$\left[\text{and } \forall t_2 \in (t_l, t), \forall t_3 \in (t_2, t], (f_p(t_2, \text{s}) < f_p(t_3, \text{s}))\right]_\alpha\Big)$

Overshoot Pattern

$\bar{\pi}, [t_l, t_u] \models \text{s}$ **overshoots** $[\text{monotonically}]_\alpha$ v_1 **by** $\text{v}_2 \Leftrightarrow \exists t \in (t_l, t_u], \Big(f_p(t, \text{s}) \geq \text{v}_1$ and $\forall t_1 \in (t, t_u], (f_p(t_1, \text{s}) \leq \text{v}_1 + \text{v}_2)$
$\left[\text{and } \forall t_2 \in (t_l, t), \forall t_3 \in (t_2, t], (f_p(t_2, \text{s}) < f_p(t_3, \text{s}))\right]_\alpha\Big)$

Order Relationship Pattern

$\bar{\pi}, [t_l, t_u] \models$ **if** p_1 **then** $[\text{within } (\text{exactly}|\text{at most}|\text{at least})\ \text{t}]_\alpha\ \text{p}_2 \Leftrightarrow \forall t_1, t_2 \in [t_l, t_u], t_1 < t_2, \Big(\bar{\pi}, [t_1, t_2] \models \text{p}_1 \Rightarrow$
$\exists t_3, t_4 \in [t_2, t_u], t_3 < t_4, (\bar{\pi}, [t_3, t_4] \models \text{p}_2 \left[\text{and } (t_3 - t_2) [\![\bowtie]\!] \text{t}\right]_\alpha)\Big)$

where $\bowtie \in \{\text{exactly}, \text{at most}, \text{at least}\}$ and $[\![\bowtie]\!]$ is defined such that $[\![\text{exactly}]\!] \equiv$ '=', $[\![\text{at most}]\!] \equiv$ '<=', $[\![\text{at least}]\!] \equiv$ '>='
$\text{t}, \text{t}_1, \text{t}_2 \in \mathbb{R}$; $\text{v}, \text{v}_1, \text{v}_2 \in \mathbb{R}$; $\sim \in \{<, >, =, \neq, \leq, \geq\}$; s is a signal in S or a mathematical expression over the signals in S; sc is a scope; p is a pattern.

Figure 4: SB-TemPsy-DSL formal semantics

6 SB-TEMPSY-CHECK

In this section, we present the main steps of and the artifacts used within SB-TemPsy-Check. Section 6.1 describes the pre-processing step, Section 6.2 illustrates the meta-model of the pre-processed trace, and Section 6.3 explains how the OCL constraint solver is used to perform trace checking.

6.1 Pre-processing

The *pre-processing* step converts the original simulation trace, in which signal values are recorded at different simulation times, to a new trace in which signal values are recorded for *every entry*.

Our conversion relies on an interpolation function to generate the missing signal values. More precisely, let $\eta : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ be an interpolation function over real values, and S and π be, respectively, the set of signals and the simulation trace as defined in section 3.1; the *pre-processed trace* obtained from $\pi = \langle T, f \rangle$ and denoted by $\bar{\pi}$ is defined as $\bar{\pi} = \langle T, \bar{f} \rangle$, where $\bar{f}$ is a function such that for all $t \in T$ and for all $s \in S$, the assignment $(s = \eta(last(\pi, s, t), next(\pi, s, t))) \in \bar{f}(t)$. Intuitively, in the pre-processed trace, at every time instant t_i the value of signal s is the interpolation between the last-seen value and the next-seen value of s at t_i. Note that this definition is compatible with the case in which signal s is actually assigned a value v in the record sampled at time t_i. Indeed, in such a case both function *last* and function *next* yield v, and we have $\eta(v, v) = v$.

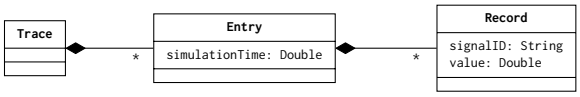

Figure 5: UML Class Diagram of the Trace Meta-model

Users can choose different interpolation functions (e.g., piecewise constant, linear, cubic) depending on the domain of the values of the signals, and their expected variation over time. We discuss the choice of the interpolation for our case study in section 7.

Before doing this conversion, our pre-processing step also performs some filtering on the trace. A trace contains records for many signals; however, a property to be checked on the trace may refer to only a subset of these signals. Hence, we remove from the trace all the entries that do not contain any record with a signal referred to by the property to be checked.

6.2 Trace Meta-model

Our trace meta-model of the pre-processed trace is an extension of the one proposed by Dou et al. [19], tailored to the CPS domain to support (i) SBTPs and (ii) trace entries recording the values of several signals at a certain time instant.

The trace meta-model includes the basic entities that are used to represent a trace when a new instance is created from a trace file. These entities are accessed by the OCL functions during the model-driven trace checking step.

The model, depicted in Fig. 5 with a UML class diagram, contains a `Trace`, which is composed of a sequence of `Entry`s. A `Entry` has an attribute representing the simulation time at which the record has been sampled, and contains one or more `Record`s (one for each signal). A `Record` has two attributes, representing the signal identifier and its value.

6.3 Model-driven Trace Checking

Our model-driven trace checking procedure checks whether an SB-TemPsy-DSL property ϕ holds over a trace π by evaluating an OCL constraint semantically equivalent to ϕ over a model (i.e., an instance of the trace meta-model described in section 6.2) of the pre-processed trace $\bar{\pi}$ (derived from π). This check is done using a standard OCL checker, such as Eclipse OCL.

Input preparation phase. First, our approach (1) builds an instance $\bar{\pi}_{obj}$ of the trace meta-model from the pre-processed trace $\bar{\pi}$ and (2) translates the property ϕ into an OCL constraint ϕ_{OCL}. The translation from SB-TemPsy-DSL properties to OCL constraints is syntax-directed and covers all the constructs of SB-TemPsy-DSL defined in figure 3. As an example, below we illustrate the OCL function (shown in figure 6) corresponding to the *oscillation* pattern.

This function takes as input a trace, an object representing the parameters of the oscillation pattern, and the bounds of the trace interval on which the pattern should be evaluated. Notice that the OCL variables `tl` and `tu` correspond to the variables t_l and t_u used in the definition of the semantics of the oscillation pattern in figure 4. The function first saves the value of the pattern parameter `s` (signal name) in the corresponding variable `s` (line 2), which is used inside the expression at lines 6–7. This expression encodes the semantics of the pattern presented in figure 4. For instance, lines 3–5 constrain the existence of five trace entries `el1`, `el2`, `el3`, `el4` and `el5` such that they have consecutive simulation times. In addition, lines 6–7 express the presence of consecutive maxima and minima according to the semantics presented in figure 4. Functions `isLMin` and `isLMax` implement functions *uni_sm_min* and *uni_sm_max* described in section 5.2. In addition, line 8 checks the optional constraints (on the peak-to-peak amplitude and/or the period) associated with the pattern.

Constraint evaluation phase. The second and final phase uses the OCL checker to evaluate the constraint ϕ_{OCL} on the object $\bar{\pi}_{obj}$, denoted by $\textsc{eval}(\bar{\pi}_{obj}, \phi_{OCL})$. The result of this evaluation is a Boolean value that corresponds to the verdict of checking property ϕ over trace π. More formally, we have that $\pi \models_\eta \phi$ if and only if $\textsc{eval}(\bar{\pi}_{obj}, \phi_{OCL})$ yields *true*.

Correctness. The correctness of our procedure (intuitively) follows from these observations. The semantics of SB-TemPsy-DSL presented in section 5.2 depends on the interpolation function selected by the user. More precisely, given a user-selected interpolation function η, the semantics specifies that property ϕ is satisfied on trace π, i.e., $\pi \models_\eta \phi$, if the pre-processed $\bar{\pi}$ trace obtained using the interpolation function η satisfies the property ϕ, i.e., $\bar{\pi} \models \phi$. Since (i) the $\bar{\pi}_{obj}$ object is built from $\bar{\pi}$, which is obtained from the pre-processing step described in section 6.1 using the η interpolation function, and (ii) the OCL constraint ϕ_{OCL} is obtained from ϕ with a one-to-one mapping from the formal semantics defined in figure 4, our procedure is correct, i.e., the verdict returned by our trace-checking procedure is consistent with the semantics presented in Section 5.2.

Time complexity. The time complexity of our procedure depends on the size of the trace and on the OCL definitions for the different constructs of SB-TemPsy-DSL. The evaluation of constructs like "condition", "property" and "absolute scope" does not depend on the size of the trace since "condition" is evaluated at a specific time instant, "property" is evaluated at the first time instant, and "absolute scope" is evaluated by setting the values of the time bound where the pattern is evaluated. The evaluation of the **`assert`** variant of the "data assertion" construct is linear in the size of the trace. The evaluation of all the other constructs is polynomial in the size of the trace. For example, the encoding of the oscillation pattern presented in Listing 6 contains five nested existential operators, leading to a procedure with a time complexity of $O(|\bar{\pi}|^5)$.

In light of these complexity results, we defined an alternative, semantically equivalent OCL definition, which relies on an optimized usage of OCL collections, specialized for each construct. More specifically, the optimization replaces the use of first-order quantifiers with collection operations, in particular `iterate` expressions. Thanks to these optimizations, the complexity of evaluating the "data assertion", "spike", "oscillation", "rise/fall time", and "overshoot/undershoot" patterns is linear in the size of the trace; the complexity of evaluating the "event" scope and "order relationship" pattern is still polynomial in the size of the trace. We used this alternative OCL definition for our empirical investigation (section 7).

7 EVALUATION

In this section, we report on the evaluation of our contributions. First, we evaluate our specification language SB-TemPsy-DSL in

```
def: checkPatternOscillation(trace:OrderedSet(trace::TraceElement), pattern::Oscillation, tl:Real,tu:Real): Boolean =
let s : String = pattern.s in
 trace->exists(el1,el2| tl<= el1.simulationTime and el1.simulationTime < el2.simulationTime
    trace->exists(el3,el4| el2.simulationTime< el3.simulationTime and el3.simulationTime < el4.simulationTime
         trace->exists(el5 |  el4.generationTime< el5.generationTime and el5.generationTime <=tu and
                (   (isLMax(trace,el2,el1,el3,s) and isLMin(trace,el3,el2,el4,s) and isLMax(trace,el4,el3,el5,s))
                 or (isLMin(trace,el2,el1,el3,s) and isLMax(trace,el3,el2,el4,s)  and isLMin(trace,el4,el3,el5,s)))
            and checkFeatures(trace,el1,el2,el3,el4,el5,pattern))))
```

Figure 6: OCL function for the *oscillation* pattern of SB-TemPsy-DSL

terms of expressiveness, and compare with state-of-the-art specification languages. Second, we evaluate the performance of the implementation of our model-driven trace checking approach SB-TemPsy-Check, and compare it to a state-of-the-art tool for trace checking of SBTPs. In both cases, as properties to express and check, we consider the requirements of our industrial case study (see section 2). Summing up, we evaluated our contributions by answering the following research questions:

RQ1 *To which extent can SB-TemPsy-DSL express requirements of real-world, industrial CPS applications and how does it compare with state-of-the-art specification languages in terms of expressiveness?* (section 7.1)

RQ2 *Can SB-TemPsy-Check verify SBTPs on real-world execution traces within practical time and how does it compare with a state-of-the-art tool?* (section 7.2)

RQ1 focuses on the *expressiveness* of SB-TemPsy-DSL, whereas RQ2 focuses on the *applicability* (in industrial settings) of SB-TemPsy-Check.

7.1 Expressiveness of SB-TemPsy-DSL

To answer RQ1, we assessed the suitability of SB-TemPsy-DSL for expressing the requirements of our industrial case study. We also tried to express the same requirements with state-of-the-art specification languages for SBTPs, i.e., STL [36] and SFO [4], and compared the result with that of SB-TemPsy-DSL.

OBSW Requirements. We defined 101 requirements, expressed in English, through a series of meetings (cumulatively lasting about 80 hours) with a senior software engineer leading the development of the OBSW. The engineer defined the requirements and also validated the corresponding SB-TemPsy-DSL properties written by two of the authors.

Results. Out of these 101 requirements, we could express 98 in SB-TemPsy-DSL. In the vast majority of the cases, the translation from English to SB-TemPsy-DSL was straightforward; only in two cases we had to rephrase the original requirement into an equivalent form that could then be mapped to SB-TemPsy-DSL. Out of the three requirements that we could not express in SB-TemPsy-DSL, two were constraints on the number of occurrences of a certain signal pattern (e.g., spike behavior) and would have required a counting/aggregate operator [45]; the other requirement was a constraint on the signal value in two consecutive time instants, which would have required a modality for referring to the value of a signal in a previous time instant. We plan to extend SB-TemPsy-DSL with these constructs as part of future work.

Table 2 shows the occurrences of the various scopes (left-hand side) and patterns (right-hand side) of SB-TemPsy-DSL across the requirements of our case study. The pattern distribution is in line with the findings of a recent taxonomy of SBTP [11], in which data assertion (**`assert`** pattern in SB-TemPsy-DSL) is the most represented pattern type, followed by the order relationship (**`if then`** in SB-TemPsy-DSL). The **`globally`** and **`at`** scopes are the most used; we observed they are usually combined with data assertions to specify invariants that should hold during the entire simulation and conditions that should hold at specific time instants.

Using state-of-the-art specification languages, out of the 101 requirements, we could express 59 in STL and 101 in SFO. The lower number of requirements expressible in STL is due to the lack of a modality that allows for referring to (and comparing) signal values at different time instants. Such a modality would be required to specify spike, oscillation, rise/fall time, and overshoot/undershoot patterns. On the other hand, STL can express data assertions and also order relationship properties, when in the latter the "cause" and "effect" sub-properties are data assertions (as it was the case in our case study) or (recursively) other order relationship sub-properties. SFO allows us to express all 101 requirements, thanks to its support for first-order quantification. These expressiveness results for STL and SFO are in line with previous findings [11], which assessed the expressiveness of STL and SFO (and STL*) with respect to different types of SBTPs.

The answer to RQ1 is that, when using SB-TemPsy-DSL, we could express 98 out of 101 requirements of a real-world, industrial system in the satellite domain. This result shows the high expressiveness of SB-TemPsy-DSL for specifying SBTPs of CPS. When compared with state-of-the-art logic-based specification languages, SB-TemPsy-DSL could express many more requirements than STL, since STL can not express spike, oscillation, rise/fall time, and overshoot/undershoot patterns. Nevertheless, SB-TemPsy-DSL is slightly less expressive than SFO (98 vs. 101). However, there is no tool support for SFO trace-checking. While we could have implemented such a tool, it would have likely exhibited low performance since the time complexity of trace-checking SFO formulae is exponential in the number of quantifiers, function symbols and length of the SFO formula, as well as in the length of the trace [4]. On the other hand, as discussed in section 6, the time complexity of the trace-checking procedure for SB-TemPsy-DSL is polynomial in the length of the trace for the "order relationship" pattern and the "event" scope, and is linear in all other cases. Such a lower time complexity is likely to result in wider applicability in industrial contexts; we will experimentally investigate the performance of our trace checking approach for SB-TemPsy-DSL in the next section.

Table 2: Occurrences of the SB-TemPsy-DSL scopes (left) and patterns (right) in the requirements of the case study

Scope Type	#Req
`globally`	74
`before`	1
`after`	9
`at`	28
`between`	9

Pattern Type	#Req	Pattern Type	#Req
`assert`	91	`rises`	3
`becomes`	9	`falls`	7
`if then`	31	`overshoots`	3
`oscillations`	24	`undershoots`	5
`spike`	3		

7.2 Applicability of SB-TemPsy-Check

To answer RQ2, we assessed the applicability of SB-TemPsy-Check on execution traces from our industrial case study. Furthermore, we also compared—in terms of applicability of the trace checking procedure—SB-TemPsy-Check with Breach [17], a state-of-the-art (offline) trace checking tool for SBTPs expressed in STL. We chose Breach among other similar tools listed in a recent survey [5] (i.e., AMT [42] and S-TaLiRo [1]), because AMT 2.0, in contrast to Breach, is not publicly available, and also because Breach has been shown [17] to be faster than S-TaLiRo.

Dataset. Our dataset consists of 18 traces provided by our industrial partner. These traces have been obtained by simulating the behavior of the OBSW in different scenarios, with a simulation time ranging approximately from one hour to 18 h. Their size (in number of entries) ranges from 41844 to 1202241 entries (avg = 389771, $StdDev$ = 393718); the corresponding file size ranges from ≈1.7 MB to ≈58.9 MB (avg ≈17.6 MB, $StdDev$ ≈19.4 MB).

Methodology and settings. Our dataset contains traces recorded while a satellite system was tested in different environmental conditions (see section 2); the entries in each trace contain only signals whose behavior was relevant to the specific test performed. Consequently, for each trace in the dataset, our industrial partner indicated which properties had to be checked, based on the signals recorded in the trace and the signals referred to in the properties. Overall, we ran SB-TemPsy-Check over 217 distinct combinations[3] of traces and properties.

The final trace size (in number of entries) ranged from 12 to 13068 entries (avg = 6187, $StdDev$ = 4456); the corresponding file size ranged from ≈255 B to ≈4.7 MB (avg ≈0.4 MB, $StdDev$ ≈0.8 MB).

We configured SB-TemPsy-Check to use *linear interpolation* as interpolation function for the pre-processing step (see section 6.1). We chose this function since it is relatively simple and produces reasonably good approximations of the signal behavior, suitable for checking typical SBTPs (e.g., oscillation, spike).

As for the comparison with Breach, we used version 1.7 (installed on Matlab version 2018a) and only ran the tool for the 59 properties that could be expressed both in SB-TemPsy-DSL and in STL (see section 7.1). Overall, we ran Breach over 110 distinct combinations of traces and properties.

We carried out the experiments on one node (using four cores) of the HPC facilities of the University of Luxembourg [48]. Each run (checking a distinct combination of a trace and a property) was repeated 10 times, to account for variations in the performance of the HPC. We set the timeout of each run to 2 h, which is a relatively short and practical time when compared to the time (in the order of several days) taken by the OBSW simulator to generate a trace in our dataset (see section 2). The total wall-clock time to run all the experiments was ≈ 12 days, reduced to about three days by exploiting the parallelization mechanisms of the underlying HPC infrastructure.

In total, we measured the execution time over 2170 runs for SB-TemPsy-Check and over 1100 runs for Breach. The execution time of SB-TemPsy-Check was measured using the Unix `time` utility for the pre-processing step and the `System.currentTimeMillis()` method (from the Java library) for the invocation of the OCL checker; the execution time of Breach was measured using the `tic` and `toc` functions of the stopwatch timer integrated within Matlab.

Results. SB-TemPsy-Check yielded a verdict within the timeout in ≈ 87% of the runs (1884 out of 2170). On average, SB-TemPsy-Check took 5.98 s (min = 0.35 s, max = 103 s) for the pre-processing and 48.7 s (min = 0.18 s, max = 7076.8 s) for the trace checking through the OCL solver. Overall, SB-TemPsy-Check took less than 10 s to check properties (i) whose pattern is different from "order relationship", and (ii) whose scope is of type "absolute"; such properties account for ≈ 99% of the completed runs (1870 out of 1884).

In the remaining 286 runs (≈ 13%) in which SB-TemPsy-Check did not finish within the timeout, the property to be verified contained either an instance of the "order relationship" pattern or an "event" scope. As discussed in section 6.3, checking these types of properties has a time complexity that is polynomial in the length of the trace. We remark that SB-TemPsy-Check was still able to return a verdict in 14 out of the 300 runs in which the property contained the aforementioned pattern type or scope type; in all these cases, the trace did violate the property and the violation was found before the timeout.

As for the 1100 runs that checked one of the 59 properties that could be expressed both in SB-TemPsy-DSL and in STL (and thus could be checked by Breach), Breach finished within the timeout in 100% of the runs, with an average execution time of 0.01 s (min = 0.006 s, max = 0.15 s); SB-TemPsy-Check finished within the timeout in ≈ 97% of the runs (1064 out of 1100). For these runs, SB-TemPsy-Check took on average 85.28 s (min = 0.18 s, max = 7076.8 s). For the remaining ≈ 3% of the runs in which SB-TemPsy-Check did not finish within the timeout, the property to be checked contained an "order relationship" pattern; the same observations made above about the complexity of checking such properties apply also here.

In terms of execution time, when considering the ≈ 97% runs in which both SB-TemPsy-Check and Breach terminated within the timeout, though SB-TemPsy-Check was slower than Breach, it was able to yield a verdict within 10 s for all the properties (i) whose pattern is different from "order relationship", and (ii) whose scope is of type "absolute"; such properties account for the vast majority of the completed runs (1050 out of 1064, ≈ 99%). This cost is reasonable given that SB-TemPsy-Check supports the verification of a much larger set of property types than Breach.

Summing up, the answer to RQ2 is that, in 87% of the runs SB-TemPsy-Check could complete within practical time limits (i.e., the timeout determined based on the development context of our case study) the verification of SBTPs (expressed in SB-TemPsy-DSL) over industrial traces, with an average checking time of 48.7 s. We deem this time to be reasonable for practical applications, since it is

[3] We also removed 13 combinations in which the trace had less than 10 entries.

orders of magnitude lower than the time needed for simulation and trace generation (as discussed in section 2). In other words, it allows engineers to integrate SB-TemPsy-Check within the development process at a negligible cost. Furthermore, though SB-TemPsy-Check was slower than Breach, it was always able to yield a verdict within 10 s in ≈ 99% of the cases, despite supporting the verification of a much larger set of property types.

7.3 Discussion and Threats to Validity

The results of our empirical investigation show that SB-TemPsy represents a viable trade-off between an expressive specification language for SBTPs (SB-TemPsy-DSL) and an efficient trace-checking procedure (SB-TemPsy-Check). Using SB-TemPsy-DSL, we could express 98 out of 101 requirements of an industry-grade CPS. SB-TemPsy-DSL was considerably more expressive than STL, which is a temporal logic for SBTPs supported by publicly available trace-checking tools. Furthermore, SB-TemPsy-Check completed the verification of these requirements on real-world execution traces within practical time limits (determined based on the development context of our case study) in ≈ 87% of the runs. This also confirms our conjecture that a model-driven approach is a viable solution for trace checking of SBTPs.

Supporting an expressive specification language like SB-TemPsy-DSL comes with a performance loss in terms of the trace checking procedure. For the 59 requirements of our case study that could be expressed in STL, in ≈ 97% of the runs in which SB-TemPsy-Check terminated, it was slower than Breach but it was able to yield a verdict within 10 s for ≈ 99% of the cases. Furthermore, differently from SB-TemPsy-Check, Breach always terminated within the timeout.

Based on the above observations, taking advantage of the complementary strengths of both approaches, we propose to combine SB-TemPsy and Breach. SB-TemPsy-DSL properties that can also be expressed in STL (i.e., logical expressions of data assertions) should be checked with Breach, since it yields better performance. More complex SBTPs (which cannot be expressed in STL) should then be checked with SB-TemPsy-Check, since it is the only tool that can efficiently verify them. Overall, this complementary usage of the two verification tools would significantly reduce the execution time and number of timeouts of the trace checking procedure.

As mentioned in Section 6.3, although our model-driven trace checking approach is correct, the interpolation function used in the pre-processing step influences the verdicts returned by SB-TemPsy-Check and Breach. In practical scenarios, engineers may want to consider different interpolations functions depending on the expected signals' behaviors. For this reason, we plan to (i) provide additional interpolation functions, and (ii) allow the selection of a different interpolation function for each signal. Engineers can then choose the interpolation function based on the type and the domain of the signal, and on their domain knowledge. Since signals usually represent the state either of some CPS software components or of their environment, engineers usually have a precise, yet intuitive, understanding of how these signals will change over time. Therefore, they can easily select the most appropriate interpolation function given their usage scenario.

Threats to validity. In our evaluation, we used a set of requirements and traces coming from one industrial case study from the satellite domain. Though the targeted system and requirements are in many ways representative of what can be found in satellite and other cyber-physical domains—where the behavior being controlled involves convoluted physical dynamics and the system requirements are expressed as complex SBTPs—this could influence the generalization of our results.

Another threat to the validity of the evaluation results is the presence of coding errors in the implementation of SB-TemPsy-Check. We tried to mitigate it by thoroughly testing the tool.

Data availability. The supplementary material accompanying this work is available at https://github.com/SNTSVV/SB-TemPsy.

8 RELATED WORK

Our approach is related to work done in the areas of (i) specification languages for SBTPs, (ii) trace-checking methods, and (iii) model-driven approaches for trace checking.

Specification languages. STL [36] has been one of the first logic-based languages proposed for specifying SBTPs. STL* [12] is an extension of STL with the signal-value freezing operator, which binds the value of a signal at a precise time instant. SFO [4] is a first-order temporal logic for signals. Boufaied et al. [11] compared the expressiveness of STL, STL*, and SFO analytically, based on different formulations of the main types of SBTPs (see section 3.2); in contrast, we have empirically compared the expressiveness of SB-TemPsy-DSL, STL and SFO as part of our evaluation (section 7.1), using properties from a representative industrial case study. Bakhirkin and Basset [3] proposed an extension of STL with (i) a new form of *until* operator, (ii) support for computable aggregate function over a sliding window, and (iii) formulae having the possibility of producing and manipulating real-valued output signals. These extensions give the possibility of expressing some SBTP patterns (e.g., stabilization, maximum/minimum value, linear increase, spike) without the need for more expressing languages like STL* or SFO. Menghi et al. [38] have recently proposed RFOL, a logic that is more expressive than STL and less expressive than SFO. We did not consider RFOL since this work is focused on offline trace-checking, whereas RFOL is supported by an online trace checker. SB-TemPsy-DSL is also related to other DSLs for expressing temporal properties, such as PROPEL - DNL [47], Structured English Grammar for real-time specifications [33], Temporal OCL [32], OCLR [18], VISPEC - graphical formalism [31], TemPsy [19], SpeAR [28], TemPsy-AG [10], ProMoboBox - property language [39], FRETISH [29]. The majority of them is based on some property specification patterns [2, 21]; none of them support SBTPs. The contract language proposed by Bernaerts et al. [7] is the closest to SB-TemPsy-DSL, since it is pattern-based and has been developed for the CPS domain; however, it supports only STL-like properties, and thus cannot express more complex behaviors such as spikes and oscillations.

SB-TemPsy-DSL supports the main types of SBTPs identified in Boufaied et al. [11]'s taxonomy, which also provides a formalization of the properties in SFO. For SB-TemPsy, we have refined such a formalization since our goal was to develop a trace-checking procedure for SB-TemPsy-DSL properties, rather than providing a mere formalization in a temporal logic. For example, our OCL

definition of the SB-TemPsy-DSL semantics uses different predicates for determining the local extrema (see table 1), since they are more appropriate (by requiring the signal value to change according to a unimodal function) to model the signal behavior in the context of trace checking.

Trace-checking methods and tools. Among the temporal logics for SBTPs discussed above, STL is supported by tools (such as AMT [42], Breach [16], and S-Taliro [1]) for offline trace checking; a tool is also available for the STL extension proposed in [3]. No tools are available for STL* and SFO. RFOL is supported by an online trace-checking procedure (SOCRaTEs [38]). The contract language proposed by Bernaerts et al. [7] is translated into STL and then relies on Breach for verification.

Model-driven approaches for trace checking. Model-driven trace checking has been originally proposed by Dou et al. [19] for the verification of simple temporal properties. Dou et al.'s work has been extended by Boufaied et al. [10] to support service provisioning specification patterns [8, 9] using aggregate operators. In this work we have applied Dou et al.'s idea of model-driven trace checking in the context of SBTP through the development of the SB-TemPsy approach. The main differences with Dou et al. [19] are:

(i) The SB-TemPsy-DSL language has constructs specific to the domain of SBTP, based on the property types proposed in a recent taxonomy [11]; also, the scope operators, though inspired by Dwyer et al. [21]'s work, have been tailored to the domain of SBTP (e.g., to support absolute time instants). On the other hand, the TemPsy language [19] (and its predecessor OCLR [18]) are only based on Dwyer et al. [21]'s specification patterns (and thus do not support SBTPs).
(ii) SB-TemPsy-Check includes a pre-processing step, to deal with trace entries with missing signal values and recorded both at fixed and at variable sampling rates.
(iii) SB-TemPsy-Check sports an improved trace meta-model, to support trace entries recording the values of several signals at a certain time instant.
(iv) The mapping of the semantics of SB-TemPsy-DSL into OCL constraints is completely new, since it is based on the semantics presented in section 5.2.

To the best of our knowledge, SB-TemPsy is the first approach to provide model-driven trace checking of SBTPs.

9 CONCLUSION AND FUTURE WORK

In this paper, we propose SB-TemPsy, a model-driven approach for checking signal-based temporal properties (SBTPs) on execution traces of CPSs. SB-TemPsy includes SB-TemPsy-DSL, a domain-specific language for specifying SBTPs that cover the most frequent requirement types in CPSs, and SB-TemPsy-Check, an efficient trace-checking procedure that reduces the problem of checking an SB-TemPsy-DSL property over a trace to the problem of evaluating an OCL constraint on a model of the trace.

We evaluated SB-TemPsy by assessing the expressiveness of SB-TemPsy-DSL and the applicability of SB-TemPsy-Check to a representative CPS in the satellite domain. The results of our empirical investigation show that our approach—when compared, from a practical standpoint, to state-of-the-art alternatives—strikes a better trade-off between expressiveness and performance as it supports a much larger set of property types that can be checked, in most cases, within practical time limits. Moreover, the results suggest that SB-TemPsy could be combined with existing approaches efficiently supporting STL. In this way, we show we can make optimal use of a given verification budget while avoiding most time-outs by relying on the best tool option depending on the type of the checked property.

As part of future work, we plan to extend SB-TemPsy-DSL with additional constructs, based on the expressiveness results of our evaluation. Furthermore, we are going to develop alternative OCL definitions in SB-TemPsy-Check, optimized to minimize the number of time-outs when checking specific types of properties. Also, we plan to investigate how different implementations of SB-TemPsy-Check (e.g., using an SMT-based encoding or another type of logic-based encoding relying on tools like R2U2 [40]) fare with respect to the one based on OCL. Finally, we plan to extend the output returned by SB-TemPsy-Check in case of violations with diagnostic information, inspired by existing work [20, 27].

ACKNOWLEDGMENTS

This work has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277), from the Natural Sciences and Engineering Research Council of Canada (NSERC) under the Discovery and CRC programs, and from the University of Luxembourg (grant "MOVIDA").

The experiments presented in this paper were carried out using the HPC facilities of the University of Luxembourg [48] — see hpc.uni.lu.

The authors would also like to thank the anonymous referees for their valuable comments and helpful suggestions.

REFERENCES

[1] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. 2011. S-taliro: A tool for temporal logic falsification for hybrid systems. In *Proc. TACAS 2011*. Springer, Berlin, Heidelberg, 254–257.
[2] Marco Autili, Lars Grunske, Markus Lumpe, Patrizio Pelliccione, and Antony Tang. 2015. Aligning Qualitative, Real-Time, and Probabilistic Property Specification Patterns Using a Structured English Grammar. *IEEE Trans. Softw. Eng.* 41, 7 (2015), 620–638.
[3] Alexey Bakhirkin and Nicolas Basset. 2019. Specification and efficient monitoring beyond STL. In *Proc. TACAS 2019*. Springer, Cham, 79–97.
[4] Alexey Bakhirkin, Thomas Ferrère, Thomas A. Henzinger, and Dejan Ničković. 2018. The First-order Logic of Signals: Keynote. In *Proc. EMSOFT2018*. IEEE Press, Los Alamitos, CA, USA, 1:1–1:10.
[5] Ezio Bartocci, Jyotirmoy Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Ničković, and Sriram Sankaranarayanan. 2018. Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In *Lectures on Runtime Verification*. Springer, Cham, 135–175.
[6] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. Introduction to Runtime Verification. In *Lectures on Runtime Verification - Introductory and Advanced Topics*. LNCS, Vol. 10457. Springer, Cham, Switzerland, 1–33.
[7] M. Bernaerts, B. Oakes, K. Vanherpen, B. Aelvoet, H. Vangheluwe, and J. Denil. 2019. Validating Industrial Requirements with a Contract-Based Approach. In *Proc. MODELS 2019 (Companion))*. IEEE, Los Alamitos, CA, USA, 18–27.
[8] Domenico Bianculli, Carlo Ghezzi, Cesare Pautasso, and Patrick Senti. 2012. Specification patterns from research to industry: a case study in service-based applications. In *Proc. ICSE2012*. IEEE, Los Alamitos, CA, USA, 968–976.
[9] Domenico Bianculli, Carlo Ghezzi, and Pierluigi San Pietro. 2013. The Tale of SOLOIST: a Specification Language for Service Compositions Interactions. In *Proc. FACS'12 (LNCS)*, Vol. 7684. Springer, Heidelberg, Germany, 55–72.
[10] Chaima Boufaied, Domenico Bianculli, and Lionel C. Briand. 2019. A Model-driven Approach to Trace Checking of Temporal Properties with Aggregations. *Journal of Object Technology* 18, 2 (2019), 15:1–15:21. https://doi.org/10.5381/jot.2019.18.2.a15

[11] Chaima Boufaied, Maris Jukss, Domenico Bianculli, Lionel Claude Briand, and Yago Isasi Parache. 2019. Signal-Based Properties: Taxonomy and Logic-based Characterization. *CoRR* abs/1910.08330 (2019), 1–39. arXiv:1910.08330 http://arxiv.org/abs/1910.08330

[12] Lubos Brim, Petr Dluhoš, David Šafránek, and Tomas Vejpustek. 2014. STL*: Extending signal temporal logic with signal-value freezing operator. *Information and computation* 236 (2014), 52–67.

[13] Kalou Cabrera Castillos, Frédéric Dadeau, Jacques Julliand, Bilal Kanso, and Safouan Taha. 2013. A compositional automata-based semantics for property patterns. In *Proc. iFM 2013*. Springer, Heidelberg, 316–330.

[14] Christoph Czepa, Amirali Amiri, Evangelos Ntentos, and Uwe Zdun. 2019. Modeling compliance specifications in linear temporal logic, event processing language and property specification patterns: a controlled experiment on understandability. *Software and Systems Modeling* 18, 6 (2019), 3331–3371. https://doi.org/10.1007/s10270-019-00721-4

[15] C. Czepa and U. Zdun. 2018. On the Understandability of Temporal Properties Formalized in Linear Temporal Logic, Property Specification Patterns and Event Processing Language. *IEEE Trans. Softw. Eng.* 46 (2018), 1–13. doi: 10.1109/TSE.2018.2859926.

[16] Alexandre Donzé. 2010. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *Proc. CAV2010*. Springer, Berlin, Heidelberg, 167–170.

[17] Alexandre Donzé, Thomas Ferrère, and Oded Maler. 2013. Efficient Robust Monitoring for STL. In *Proc. CAV 2013*. Springer, Berlin, Heidelberg, 264–279.

[18] Wei Dou, Domenico Bianculli, and Lionel Briand. 2014. OCLR: a More Expressive, Pattern-based Temporal Extension of OCL. In *Proc. ECMFA 2014 (LNCS)*, Vol. 8569. Springer, Heidelberg, Germany, 51–66.

[19] Wei Dou, Domenico Bianculli, and Lionel Briand. 2017. A Model-Driven Approach to Trace Checking of Pattern-based Temporal Properties. In *Proc. MODELS2017*. IEEE Computer Society, Los Alamitos, CA, USA, 323–333.

[20] Wei Dou, Domenico Bianculli, and Lionel Briand. 2018. Model-Driven Trace Diagnostics for Pattern-based Temporal Specifications. In *Proc. MODELS 2018*. ACM, New York, NY, USA, 278–288. https://doi.org/10.1145/3239372.3239396

[21] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. 1999. Patterns in property specifications for finite-state verification. In *Proc. ICSE '99*. ACM, New York, NY, USA, 411–420.

[22] Eclipse. 2020. Eclipse OCL Tools. https://projects.eclipse.org/projects/modeling.mdt.ocl.

[23] ESA. 2020. Building and testing spacecraft. https://www.esa.int/Science_Exploration/Space_Science/Building_and_testing_spacecraft

[24] ESA 2020. The European Space Agency (ESA). https://www.esa.int/

[25] exactEarth 2020. exactEarth. https://www.exactearth.com/

[26] Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. 2018. A Taxonomy for Classifying Runtime Verification Tools. In *Proc. RV 2018 (Lecture Notes in Computer Science)*, Vol. 11237. Springer, Cham, 241–262.

[27] Thomas Ferrère, Oded Maler, and Dejan Ničković. 2015. Trace Diagnostics Using Temporal Implicants. In *Proc. ATVA 2015 (LNCS)*, Vol. 9364. Springer International Publishing, Cham, 241–258.

[28] Aaron W. Fifarek, Lucas G. Wagner, Jonathan A. Hoffman, Benjamin D. Rodes, M. Anthony Aiello, and Jennifer A. Davis. 2017. SpeAR v2.0: Formalized Past LTL Specification and Analysis of Requirements. In *Proc. NFM 2017*. Springer International Publishing, Cham, 420–426.

[29] Dimitra Giannakopoulou, Thomas Pressburger, Anastasia Mavridou, and Johann Schumann. 2020. Generation of Formal Requirements from Structured Natural Language. In *Proc. REFSQ 2020*. Springer International Publishing, Cham, 19–35.

[30] K. Havelund, D. Peled, and D. Ulus. 2017. First order temporal logic monitoring with BDDs. In *Proc. FMCAD 2017*. IEEE, Los Alamitos, CA, USA, 116–123.

[31] Bardh Hoxha, Nikolaos Mavridis, and Georgios Fainekos. 2015. VISPEC: A graphical tool for elicitation of MTL requirements. In *Proc. IROS2015*. IEEE, Los Alamitos, CA, USA, 3486–3492.

[32] Bilal Kanso and Safouan Taha. 2013. Temporal Constraint Support for OCL. In *Proc. SLE 2012 (LNCS)*, Vol. 7745. Springer, Berlin, Heidelberg, 83–103.

[33] Sascha Konrad and Betty H. C. Cheng. 2005. Real-time specification patterns. In *Proc. ICSE '05*. ACM, New York, NY, USA, 372–381.

[34] Jianwen Li, Moshe Y Vardi, and Kristin Y Rozier. 2019. Satisfiability checking for mission-time LTL. In *Proc. CAV2019*. Springer, Cham, 3–22.

[35] Luxspace 2020. Luxspace. https://luxspace.lu/

[36] Oded Maler and Dejan Nickovic. 2004. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Springer, Berlin, Heidelberg, 152–166.

[37] Claudio Menghi, Shiva Nejati, Lionel C. Briand, and Parache Yago Isasi. 2020. Approximation-Refinement Testing of Compute-Intensive Cyber-Physical Models: An Approach Based on System Identification. In *Proc. ICSE 2020*. ACM, New York, NY, USA, 1–12.

[38] Claudio Menghi, Shiva Nejati, Khouloud Gaaloul, and Lionel C. Briand. 2019. Generating Automated and Online Test Oracles for Simulink Models with Continuous and Uncertain Behaviors. In *Proc. ESEC/FSE 2019*. ACM, New York, NY, USA, 27–38.

[39] B. Meyers, H. Vangheluwe, J. Denil, and R. Salay. 2020. A Framework for Temporal Verification Support in Domain-Specific Modelling. *IEEE Trans. Softw. Eng.* 46, 4 (2020), 362–404.

[40] Patrick Moosbrugger, Kristin Y. Rozier, and Johann Schumann. 2017. R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems. *Formal Methods in System Design* 51 (April 2017), 31–61. https://doi.org/10.1007/s10703-017-0275-x

[41] Shiva Nejati, Khouloud Gaaloul, Claudio Menghi, Lionel C. Briand, Stephen Foster, and David Wolfe. 2019. Evaluating model testing and model checking for finding requirements violations in Simulink models. In *Proc. ESEC/FSE 2019*. ACM, New York, NY, USA, 1015–1025.

[42] Dejan Ničković, Olivier Lebeltel, Oded Maler, Thomas Ferrère, and Dogan Ulus. 2018. AMT 2.0: Qualitative and Quantitative Trace Analysis with Extended Signal Temporal Logic. In *Proc. TACAS 2018*. Springer International Publishing, Cham, 303–319.

[43] OMG. 2012. ISO/IEC 19507 (OCL v2.3.1). http://www.omg.org/spec/OCL/ISO/19507/PDF.

[44] Yago Isasi Parache, Aleix Pinardell, Antonio Márquez, Christophe Molon-Noblot, Alexander Wagner, Marc Gales, and Miroslav Brada. 2019. The ESAIL Multi-purpose Simulator. Poster at the Workshop on Simulation and EGSE for Space Programmes (SESP 2019).

[45] Nicolas Rapin. 2016. Reactive Property Monitoring of Hybrid Systems with Aggregation. In *Proc. RV2016 (LNCS)*, Vol. 10012. Springer, Cham, 447–453.

[46] César Sánchez, Gerardo Schneider, Wolfgang Ahrendt, Ezio Bartocci, Domenico Bianculli, Christian Colombo, Yliès Falcone, Adrian Francalanza, Srdan Krstic, João M. Lourenço, Dejan Nickovic, Gordon J. Pace, José Rufino, Julien Signoles, Dmitriy Traytel, and Alexander Weiss. 2019. A survey of challenges for runtime verification from advanced application domains (beyond software). *Formal Methods Syst. Des.* 54, 3 (2019), 279–335. https://doi.org/10.1007/s10703-019-00337-w

[47] R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. 2002. PROPEL: an approach supporting property elucidation. In *Pro. ICSE 2002*. IEEE, Los Alamitos, CA, USA, 11–21.

[48] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos. 2014. Management of an Academic HPC Cluster: The UL Experience. In *Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014)*. IEEE, Los Alamitos, CA, USA, 959–967.