PhD-FSTM-2021-014
The Faculty of Sciences, Technology and Medicine

DISSERTATION

Defence held on 29/04/2021 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Chaima BOUFAIED
Born on 16 August 1991 in Nabeul (Tunisia)

SPECIFICATION AND MODEL-DRIVEN TRACE CHECKING OF COMPLEX TEMPORAL PROPERTIES

DISSERTATION DEFENCE COMMITTEE

DR. DOMENICO BIANCULLI, Dissertation Supervisor
Associate Professor, University of Luxembourg, Luxembourg

DR. LIONEL CLAUDE BRIAND, Member
Professor, University of Luxembourg, Luxembourg

DR. FABRIZIO PASTORE, Chairman
Associate Professor, University of Luxembourg, Luxembourg

DR. PIERLUIGI SAN PIETRO, Member
Professor, Politecnico di Milano, Italy

DR. PAOLA INVERARDI, Member
Professor, Università dell'Aquila, Italy

Acknowledgments

I would like to thank Lionel Briand for the opportunity he gave to me to join SVV lab and to work on an interesting PhD topic.

I truly appreciate the continuous feedback, the great support, and the confidence that my advisor, Domenico Bianculli, showed to me.

I value the insights and guidance provided by Claudio Menghi as well as the fruitful collaboration I had with him.

Special thanks go to LuxSpace, our industrial partner, and especially to Yago Isasi Parache for his availability and for providing us with traces and requirements to do trace checking and trace diagnostics.

I am grateful as well for the feedback I got from my committee members Paola Inverardi and Pierluigi San Pietro.

I offer this PhD to the soul of my grandma who left me two years ago...
I also offer it to my parents who never stopped supporting me during my PhD journey! I also would like to thank my siblings and my besties for always being there for me.

In the loving memory of my grandma…♡

Abstract

Offline trace checking is a procedure used to evaluate requirement properties over a trace of recorded events. System properties verified in the context of trace checking can be specified using different specification languages and formalisms; in this thesis, we consider two classes of complex temporal properties: 1) properties defined using aggregation operators; 2) signal-based temporal properties from the Cyber Physical System (CPS) domain.

The overall goal of this dissertation is to develop methods and tools for the specification and trace checking of the aforementioned classes of temporal properties, focusing on the development of scalable trace checking procedures for such properties.

The main contributions of this thesis are:

i) the TEMPSY-CHECK-AG model-driven approach for trace checking of temporal properties with aggregation operators, defined in the *TemPsy-AG* language;

ii) a taxonomy covering the most common *types* of Signal-based Temporal Properties (SBTPs) in the CPS domain;

iii) *SB-TemPsy*, a trace-checking approach for SBTPs that strikes a good balance in industrial contexts in terms of *efficiency* of the trace checking procedure and *coverage* of the most important types of properties in CPS domains. SB-TemPsy includes: 1) *SB-TemPsy-DSL*, a DSL that allows the specification of the types of SBTPs identified in the aforementioned taxonomy, and 2) an efficient trace-checking procedure, implemented in a prototype tool called *SB-TemPsy-Check*;

iv) *TD-SB-TemPsy-Report*, a model-driven trace diagnostics approach for SBTPs expressed in SB-TemPsy-DSL. TD-SB-TemPsy-Report relies on a set of *diagnostics patterns*, i.e., undesired signal behaviors that might lead to property violations. To provide relevant and detailed information about the cause of a property violation, TD-SB-TemPsy-Report determines the diagnostics information specific to each type of diagnostics pattern.

Our technological contributions rely on *model-driven* approaches for trace checking and trace diagnostics. Such approaches consist in *reducing* the problem of checking (respectively, determining the diagnostics information of) a property ρ over an execution trace λ to the problem of evaluating an OCL (Object Constraint Language) constraint (semantically equivalent to ρ) on an instance (equivalent to λ) of a meta-model of the trace. The results — in terms of efficiency of our model-driven tools — presented in this thesis are in line with those presented in previous work, and confirm that model-driven technologies can lead to the development of tools that exhibit good performance from a practical standpoint, also when applied in industrial contexts.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context and Motivation

Trace checking is a *run-time verification (RV)* [LS09] technique that checks whether a property holds over a log of recorded events over system behaviors, represented as execution traces. An execution trace is collected by executing the system and recording the values of the variables of interest at certain time instants. Since the execution trace is obtained and checked *after* the execution of a system terminates, this technique is also called *offline trace checking* or *post-mortem analysis*, in order to distinguish it from online techniques that check the correctness of a system *while* it is executing [FHR13], possibly by processing a stream of events [DSS+05, Hal16]. Trace-checking tools (e.g., [DBB17a], [BBB19], [BMB+20]) are commonly used to check whether requirements hold over one or more execution traces. They are used for software verification & validation (V&V) activities as test oracles [MNGB19] and run-time monitors [SSA+19]. We distinguish between two main aspects that characterize trace checking approaches: 1) the type of properties to verify (as well as the specification language used to express them), and 2) the actual checking procedure to use.

System properties verified in the context of trace checking can be specified using different specification languages and formalisms, such as a temporal logic (e.g., linear temporal logic, metric temporal logic [Koy90], signal temporal logic [MN04]), regular expressions, state machines, or a combination of them [BFFR18], possibly using domain-specific languages (DSLs). In this thesis we consider two classes of quantitative temporal properties:

- properties defined using aggregation operators;
- signal-based temporal properties from the Cyber Physical System (CPS) domain.

The class of temporal properties with aggregation operators was identified in a field study [BGPS12], which analyzed more than 900 requirements specifications written in the context of service-based applications, extracted from research papers and industrial data (in the domain of banking service provisioning). More specifically, the study identified a new class of property specification patterns (inspired by Dwyer et al.'s seminal work [DAC99]) called *service provisioning patterns*, which matched the majority of the requirements specifications stated in industrial settings. More than 80% of the industrial specifications analyzed in the study could be written as temporal properties using aggregation operators, i.e., maximum and average response time. To the best of our knowledge, the only specification language supporting the service provisioning patterns identified in [BGPS12] is *SOLOIST* [BGS13]; a language based on first-order metric temporal logic extended with aggregating modalities.

Regarding Signal-based Temporal Properties (SBTPs), they are common in CPS domains. Note that a typical CPS consists of a mix of analog and digital components, such as sensors, actuators, and control units, which process input and output signals. System engineers specify the desired system behavior by defining requirements in terms of the signals obtained from these components. Such requirements can be specified using SBTPs, which characterize the expected behavior of signals. For example, a property may require that a signal must not exhibit an abrupt increase of amplitude (i.e., a spike or bump) within a certain time interval, or that the signal shall manifest an oscillatory behavior with a particular period.

Expressing requirements in terms of SBTPs poses a number of challenges for system and software engineers. First, a signal behavior (e.g., a spike) can be characterized using a number of features (e.g., amplitude, slope, width); for example, a total of 16 different features (and eight parameters) have been identified in the literature [AMM+14] to detect (and thus characterize) a spike in a signal. Engineers may decide to choose various subsets of features; without proper guidelines for selecting the features most appropriate in a certain context and without their precise characterization, the resulting specification of a signal behavior may become ambiguous or inconsistent. The second challenge is related to the expressiveness of the specification languages used for defining SBTPs. Starting from the seminal work on *STL* [MN04] (*STL* Signal Temporal Logic), there have been several proposals of languages that extend more traditional temporal logics like *LTL* (Linear Temporal Logic) to support the specification of signal-based behaviors. Such languages have different levels of expressiveness when it comes to describing certain signal behaviors. For example, *STL* cannot be used to express properties (like those related to oscillatory behaviors) that require to reference the concrete value of a signal at an instant in which a certain property was satisfied [BDŠV14]. This means that engineers need guidance to carefully select the language to use for defining SBTPs, based on the type of requirements they are going to define, the expressiveness of the candidate specification languages, and the availability of suitable tools (e.g., trace checker) for each language. To tackle these challenges, one first needs to properly define and characterize the different signal behaviors in CPS domain.

When it comes to verification of properties with aggregation, the support provided in terms of verification of such properties is limited. Moreover, trace checking algorithms for *SOLOIST* proposed in the literature [BGKSP14, BBG+14] do not scale well in terms of the length of the

trace [BGK14]. As for the trace checking of SBTPs, *STL* is supported both by *offline* tools—such as *AMT* [NLM+18] (a stand-alone GUI tool with qualitative semantics), *Breach* [Don10] and *S-Taliro* [FSUY12] (two *Matlab* plugins with quantitative semantics)—and by *online* tools, such as the `rtamt` library [NY20], which automatically generates online monitors with robustness semantics from *STL* specifications. However, there is a limited support of trace-checking procedures for the most expressive temporal logics [BDŠV14, BFHN18].

Finally, we remark that most of trace checkers rely on qualitative semantics, returning a Boolean verdict; the latter is set to *true* if the input property is satisfied by the execution trace, and to *false* in case the property is violated. However, Boolean verdicts are not sufficient to reveal information about the cause of the property violation, especially when a property violation can be due to several, distinct causes. This means that when the property is violated, in the current practice, trace diagnostics is still mostly a manual activity performed with ad-hoc solutions. For example, in the context of CPS, engineers usually plot the values the signal takes over time and try to understand the root cause of a violation; this activity is complex and time-consuming. Solutions for the trace diagnostics problem have been proposed in the literature for STL [FMN15] and for simple temporal properties based on Dwyer et al. [DAC99] specification patterns [DBB18]. However, there is no support for trace diagnostics of more complex classes of temporal properties, such as SBTPs.

1.2 Research Contributions

The overall goal of this dissertation is to develop methods and tools for the specification and trace checking of the aforementioned classes of temporal properties, addressing the limitations of the state-of-the-art discussed above. More specifically, we identify the following Research Goals (RGs):

RG1 developing an efficient trace checking procedure for temporal properties with aggregation operators;

RG2 providing a comprehensive specification framework for defining the most common *types* of SBTPs in the CPS domain as well as developing a scalable trace checking procedure for such properties, complemented by informative verdicts.

To achieve *RG1*, we propose the TEMPSY-CHECK-AG approach, which extends TEMPSY-CHECK [DBB17a] — an approach and tool for *model-driven* trace checking of a subset of LTL properties defined in the *TemPsy* (Temporal Property made easy) DSL — to support temporal properties with aggregation operators. The properties to verify with TEMPSY-CHECK-AG are expressed in *TemPsy-AG*, an extension of *TemPsy* that supports aggregation operators based on the service provisioning specification patterns [BGPS12] and the *SOLOIST* language [BGS13]. We assessed the scalability of TEMPSY-CHECK-AG and compared it with *SOLOIST*-translator, a state-of-the-art tool for (non-distributed) trace checking of *SOLOIST* specifications [BGKSP14, BBG+14].

To achieve *RG2* goal, we make the following contributions:

- We propose a taxonomy of SBTPs, based on practical experience in analyzing temporal requirements in CPS domains like the aerospace industry, and by reviewing the literature in the area of V&V of CPS. Through the taxonomy, we provide a comprehensive and detailed description of the different types of signal-based behaviors, with each property type precisely characterized in terms of a temporal logic (i.e., *STL*, *STL** [BDŠV14] and *SFO* [BFHN18]).
- To specify and check the most frequent requirement types in CPS domains, we propose SB-TemPsy, a trace-checking approach for SBTPs that strikes a good balance in industrial contexts in terms of *efficiency* of the trace checking procedure and *coverage* of the most important types of properties in CPS domains. SB-TemPsy provides:
  - *SB-TemPsy-DSL*, a DSL that allows the specification of the types of SBTPs identified in the aforementioned taxonomy;
  - an efficient trace-checking procedure, implemented in a prototype tool called *SB-TemPsy-Check*.
- To allow SB-TemPsy-Check to provide informative verdicts, we propose TD-SB-TemPsy-Report, a trace diagnostics approach for SBTPs expressed in SB-TemPsy-DSL. TD-SB-TemPsy-Report relies on a set of *diagnostics patterns*, i.e., undesired signal behaviors that might lead to property violations. To provide relevant and detailed information about the cause of a property violation, TD-SB-TemPsy-Report determines the diagnostics information specific to each type of *diagnostics pattern*.

To achieve these research goals, we rely on *model-driven* approaches for trace checking and trace diagnostics. Such approaches consist in *reducing* the problem of checking (respectively, determining the diagnostics information of) a property ρ over an execution trace λ to the problem of evaluating an OCL (Object Constraint Language) constraint (semantically equivalent to ρ) on an instance (equivalent to λ) of a meta-model of the trace. We made this choice for two reasons:

1. OCL is a standardized constraint specification language defined by OMG [OMG12] and, as a result, is supported by a mature constraint checking technology, such as the constraint checker included in Eclipse OCL [Ecl20].
2. The seminal work of Dou et al. on trace checking [DBB17a] and trace diagnostics [DBB18] of *TemPsy* properties has shown that the adoption of model-driven technologies can lead to the development of tools that exhibit good performance, also when applied in industrial contexts. In this thesis, we made the conjecture that such a level of performance could be also obtained when model-driven technologies are applied for trace checking of more complex classes of temporal properties. The evaluation of this conjecture was part of our empirical investigation.

1.3 Dissemination

Our research work has led to the following publications (listed in chronological order based on their publication date):

Published papers

- Chaima Boufaied, Domenico Bianculli, and Lionel Briand. A model-driven approach to trace checking of temporal properties with aggregations. *Journal of Object Technology*, 18(2):15:1–21, 2019. Proceedings of the 2019 European Conference on Modelling Foundations and Applications (ECMFA 2019), Eindhoven, The Netherlands.

  This paper is the basis for Chapter 3. It presents TEMPSY-CHECK-AG and an evaluation of its scalability.

- Chaima Boufaied, Maris Jukss, Domenico Bianculli, Lionel Briand, and Yago Isasi Parache. Signal-based properties of cyber-physical systems: Taxonomy and logic-based characterization. *Journal of Systems and Software*, 174:110881, April 2021.

  This paper is the basis for Chapter 4. It presents our taxonomy of SBTPs.

- Chaima Boufaied, Claudio Menghi, Domenico Bianculli, Lionel Briand, and Yago Isasi-Parache. Trace-checking signal-based temporal properties: A model-driven approach. In *Proceedings of the 2020 35th ACM/IEEE International Conference on Automated Software Engineering (ASE'20), September 21–25, 2020, Virtual Event, Australia*. ACM, September 2020.

  This paper is the basis for Chapter 5. It presents SB-TemPsy and an evaluation of its scalability.

Unpublished report

- Chaima Boufaied, Claudio Menghi, Domenico Bianculli and Lionel Briand. Trace-diagnostics signal-based temporal properties: A model-driven approach.

  This paper is the basis for Chapter 6. It presents our trace diagnostics approach TD-SB-TemPsy

1.4 Organization of the Thesis

Chapter 2 provides some background concepts that are used throughout the thesis. Chapter 3 presents the *TemPsy-AG* approach. Chapter 3 illustrates the taxonomy of signal-based temporal

properties. Chapter 5 presents SB-TemPsy, including the SB-TemPsy-DSL language and the SB-TemPsy-Check trace checking procedure. Chapter 6 presents our trace diagnostics approach TD-SB-TemPsy-Report. Chapter 7 provides conclusions and directions for future work.

Chapter 2

Background

In this chapter, we illustrate and define the different concepts we used as a background in this thesis. The chapter is structured based on the two classes of quantitative temporal properties considered for the accomplishment of this thesis:

Regarding temporal properties with aggregation, we first give a short overview of the *TemPsy* language in section 2.1. Then, we present the necessary background for the corresponding model-driven trace checking procedure realized by TEMPSY-CHECK in section 2.2. Afterwards, in section 2.3, we present the specification patterns for service provisioning supported by *TemPsy-AG*.

As for Signal-based Temporal Properties, we provide background concepts on signals in section 2.4 and we introduce temporal logics for SBTPs in section 2.5.

2.1 The *TemPsy* language

TemPsy [DBB17a] is a pattern-based, domain-specific language for the specification of temporal properties. It has been designed based on the catalogue of property specification patterns by Dwyer et al. [DAC99], with new constructs derived from a field study performed in the domain of business processes for eGovernment. The main elements of the syntax of *TemPsy* are shown on the left of Figure 2.1: terminals are enclosed in single quotes, non-terminals in angle brackets, optional elements in brackets; character '*' indicates zero or more occurrences of an element; ⟨*event*⟩s are denoted by alphanumeric strings. Temporal properties in *TemPsy* are represented through the concept of ⟨*TemPsyExpression*⟩, which is composed of a *scope* and a *pattern*: the latter represents a high-level abstraction of a formal specification while the former indicates the portion(s) of a system execution in which a certain pattern should hold. *TemPsy*

supports all the patterns ("absence", "universality", "existence", "bounded existence", "precedence", "response", "precedence chain", "response chain") and scopes ("globally", "before", "after", "between-and", "after-until") introduced in [DAC99]; patterns and scopes are denoted by an intuitive syntax. The new constructs added in *TemPsy* on top of the original patterns def-

⟨*TemPsyExpression*⟩ ::= ['`temporal`' ⟨*Id*⟩ '`:`']
⟨*Scope*⟩ ⟨*Pattern*⟩

⟨*Scope*⟩ ::= '`globally`'
| '`before`' ⟨*Boundary1*⟩
| '`after`' ⟨*Boundary1*⟩
| '`between`' ⟨*Boundary2*⟩ '`and`' ⟨*Boundary2*⟩
| '`after`' ⟨*Boundary2*⟩ '`until`' ⟨*Boundary2*⟩

⟨*Pattern*⟩ ::= '`always`' ⟨*Event*⟩
| '`eventually`' ⟨*RepeatableEventExp*⟩
| '`never`' ['`exactly`' ⟨*Int*⟩] ⟨*Event*⟩
| ⟨*EventChainExp*⟩ '`preceding`'
[⟨*TimeDistanceExp*⟩] ⟨*EventChainExp*⟩
| ⟨*EventChainExp*⟩ '`responding`'
[⟨*TimeDistanceExp*⟩] ⟨*EventChainExp*⟩

⟨*Boundary1*⟩ ::= [⟨*Int*⟩] ⟨*Event*⟩ [⟨*TimeDistanceExp*⟩]

⟨*Boundary2*⟩ ::= [⟨*Int*⟩] ⟨*Event*⟩ ['`at least`' ⟨*Int*⟩ '`tu`']

⟨*EventChainExp*⟩ ::= ⟨*Event*⟩
('`,`' ['`#`' ⟨*TimeDistanceExp*⟩] ⟨*Event*⟩)*

⟨*TimeDistanceExp*⟩ ::= ⟨*ComparingOp*⟩ ⟨*Int*⟩ '`tu`'

⟨*RepeatableEventExp*⟩ ::= [⟨*ComparingOp*⟩ ⟨*Int*⟩] ⟨*Event*⟩

⟨*ComparingOp*⟩ ::= '`at least`' | '`at most`' | '`exactly`'

⟨*Event*⟩ ::= ⟨*Id*⟩

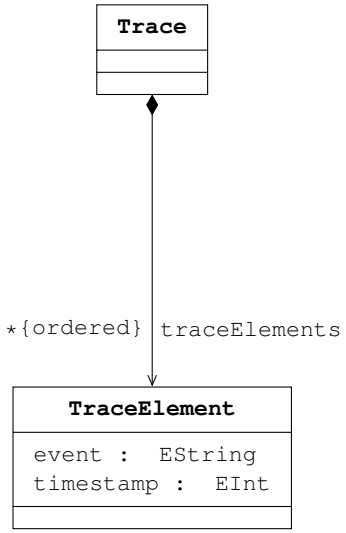


Figure 2.1: Syntax (fragment) of *TemPsy* (left) and meta-model for execution traces (right)

initions in [DAC99] include: the possibility, in the definition of a scope boundary, of i) referring to a specific occurrence of an event, and of ii) indicating a distance from the scope boundary; iii) the support for indicating a time distance between occurrences in the *precedence* and *response* patterns (as well as their chain versions); iv) additional variants for the bounded existence and absence patterns. Notice that time distances are expressed with an integer value, followed by the '`tu`' keyword, which represents a generic system time unit (i.e., any denomination of time) as suggested in [KC05]). For example, the property "Event A shall happen at least 2 time units after the fifth occurrence of event X" is expressed as "`after 5 X at least 2 tu eventually A`".

2.2 Model-driven trace checking with *TemPsy*-Check

Model-driven trace checking [DBB17a] is an approach that *reduces* the problem of checking a temporal property ρ over an execution trace λ to the problem of evaluating an OCL constraint (semantically equivalent to ρ) on an instance (equivalent to λ) of a meta-model of the trace. This reduction enables the use of standard constraint checking technology to perform trace checking; standard OCL checkers, such as Eclipse OCL[1], can be used to evaluate OCL constraints on model instances in a practical and scalable way. This trace checking technique has been proposed for adoption in contexts that rely on a model-driven development process, in which solutions must be engineered by using standard MDE technologies that are already in place in the targeted development environment. In the case of *TemPsy*, the model-driven trace checking approach has been implemented in the TEMPSY-CHECK tool [DBB17b]. This tool relies on an optimized mapping of *TemPsy* properties into OCL constraints on the meta-model of execution traces depicted in the UML class diagram shown on the right of Figure 2.1. The meta-model contains a `Trace` class composed of a sequence of `TraceElements` accessed through the association `traceElements`. Each `TraceElement` contains an attribute `event` of type `string`, which represents an event recorded in the trace, and an attribute `timestamp` of type `integer`, which indicates the time at which the event occurred. In a nutshell, TEMPSY-CHECK works as follows: given a trace and a *TemPsy* property (represented by a scope s and a pattern p), the tool evaluates an OCL invariant defined based on the type of s and p. This evaluation conceptually corresponds to applying the semantics of pattern p on a set of sub-traces; the latter is determined by the semantics of scope s. The output of the invariant evaluation is then returned as Boolean verdict of the trace checking procedure.

2.3 Specification Patterns for Service Provisioning

Specification patterns for service provisioning were identified in a field study [BGPS12], which analyzed more than 900 requirements specifications written in the context of service-based applications, extracted from research papers and industrial data. The study classified the requirements specifications according to four systems of property specification patterns: three of them were already defined in the literature [DAC99, KC05, GL06] whereas the fourth class included patterns that emerged during the study. Indeed, the majority of the requirements specifications stated in industrial settings (in the domain of banking service provisioning) could be expressed through this new set of patterns, which consists of: "average response time" (S1), "counting the number of events" (S2), "average number of events" (S3), "maximum number of events" (S4), "absolute time" (S5), "unbounded elapsed time" (S6), and "data awareness" (S7). Among these, patterns S1-S3-S4 were used in almost 82% of the specifications.

In the following, we provide an explanation of patterns S1-S3-S4, based on the semantics adopted by the *SOLOIST* language [BGS13], which is a temporal logic tailored to the specifica-

[1] https://projects.eclipse.org/projects/modeling.mdt.ocl

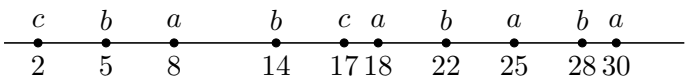


Figure 2.2: Sample trace

tion of properties based on the service provisioning patterns. Figure 2.2 depicts a sample trace (where letters in the upper part of the timeline correspond to events, and numbers in the lower part of the timeline indicate timestamps in seconds) that will be used to explain the patterns.

***Average response time* (S1).** This pattern (also known as *average elapsed time*) is a variant of the bounded response time proposed in [KC05], in which the distance between pairs of events (i.e., the response time) is aggregated over a time window of length K using the average operator. It can be used to express a property like "P1: *The average distance between events a and b in the last 20 seconds should be less than 3*", where $K = 20$. The evaluation of this property, when done in the position corresponding to the last element of the trace in Figure 2.2 (with timestamp $\tau_{end} = 30$), will consider the time window of length K that includes the events with a timestamp τ such that $\tau_{end} - K < \tau \leq \tau_{end}$: $(b, 14)$, $(c, 17)$, $(a, 18)$, $(b, 22)$, $(a, 25)$, $(b, 28)$, $(a, 30)$. The average distance is then computed by summing the differences between the timestamps of each pair[2] of events (a, b) and dividing the result by the number of the selected events pairs (2 in the example). In this case the average elapsed time is $\frac{(22-18)+(28-25)}{2} = 3.5$, which is greater than the bound 3, thus violating the property.

***Average number of events* (S3).** This pattern aggregates (using the average operator) the number of events that occurred in an observation interval h within a time window K. It can be used to express a property like "P3: *Within the last 20 seconds, in each 6-second interval, the average number of occurrences of event a should be less than 3*", where $K = 20, h = 6$. The evaluation of this property, when done in the position corresponding to the last element of the trace in Figure 2.2 (with timestamp $\tau_{end} = 30$), will consider the time window that includes the events with a timestamp τ such that $\tau_{end} - \left\lfloor \frac{K}{h} \right\rfloor h < \tau \leq \tau_{end}$. Notice that the left boundary of the time window is determined by taking into account the possibility that K may not be an exact multiple of h; if this is the case, the tail interval (with length shorter than the observation interval) is discarded. This time window is then split in $\left\lfloor \frac{K}{h} \right\rfloor$ adjacent, non-overlapping observation intervals (open to the left and closed to the right) of length $h = 6$; in the example, we have $\left\lfloor \frac{20}{6} \right\rfloor = 3$ observation intervals, delimited by the following timestamp boundaries: $(12, 18]$, $(18, 24]$, and $(24, 30]$. The average number of occurrences is then computed by summing all the occurrences of event a in each observation interval, and dividing the result by the number of observation intervals. In the example, the average number of occurrences of a is $\frac{1+0+2}{3} = 1$, which is less than the bound 3: the property is satisfied.

[2]Based on the semantics in [BGS13], the event $(a, 30)$ is ignored for computing the (average) distance, since it is not matched by a corresponding b event within the selected time window; a similar reasoning applies to event $(b, 14)$, which does not follow any event a in the selected time window. Furthermore, as proposed in [BGS13], we require that between two occurrences of event a (respectively, event b), there is an occurrence of event b (respectively, event a); because of this assumption, fragments of traces of the form $aabb$ will be collapsed into ab in a pre-processing step.

***Maximum number of events* (S4).** This pattern is a variant of the previous one, in which the events are aggregated using the maximum operator. It can be used to express a property like "P4: *Within the last 20 seconds, in each 6-second interval, the maximum number of occurrences of event a should be less than 3*"; also in this case, $K = 20$ is the time window considered for the aggregation, and $h = 6$ is the observation interval. Differently from the case of pattern S3, the semantics of this pattern (as defined in [BGS13]) takes also into account the events occurring in the tail interval, even if its length is shorter than the one of the observation interval h. The evaluation of property P4, when done in the position corresponding to the last element of the trace in Figure 2.2, will thus consider $\lceil \frac{20}{6} \rceil = 4$ observation intervals, delimited by the following timestamp boundaries: $(10, 12]$, $(12, 18]$, $(18, 24]$, and $(24, 30]$. The application of the maximum operator will yield the value 2, which is less than the bound 3: the property is satisfied.

2.4 Signals

A finite length signal s over a domain $\mathbb{D}$ is a function $s\colon \mathbb{T} \to \mathbb{D}$, where $\mathbb{T}$ is the time domain and $\mathbb{D}$ is an application-dependent value domain. In the context of CPSs, we need to differentiate between *analog*, *discrete*, and *digital* signals [JBGN16].

An analog signal is a signal that is continuous both in the time and in the value domains. The time domain $\mathbb{T}$ of an analog signal is thus the set of non-negative real numbers $\mathbb{R}_{\geq 0}$ and the value domain $\mathbb{D}$ is the set of real numbers $\mathbb{R}$. More formally, we define an analog signal s_a as $s_a\colon \mathbb{T} \to \mathbb{R}$. The domain of definition of s_a is the interval $I_{s_a} = [0, r)$, with $r \in \mathbb{Q}_{\geq 0}$; the length of s_a is defined as $|s_a| = r$; undefined signal values are denoted by $s_a(t) = \perp, \forall t \geq |s_a|$.

In a discrete signal, the value domain is continuous whereas the time domain is the set of natural numbers $\mathbb{N}$. More specifically, a discrete signal can be obtained from an analog signal through *sampling*, which is the process of converting the continuous-time domain of a signal to a discrete-time domain. Throughout this process, the analog signal is read at a regular time interval Δ called the *sampling interval*. The resulting discretized signal s_{dsc} can be represented by the values of an analog signal s_a read at the following time points: $0, \Delta, 2 \times \Delta, \ldots, k \times \Delta$. A digital signal has the set of natural numbers $\mathbb{N}$ as time domain and a finite discrete set as value domain. Such a signal can be obtained from a discrete signal by *quantization*, which is the process of transforming continuous values into their finite discrete approximations.

In the rest of the thesis we will consider analog signals, simply denoted by s, unless a specific signal type is explicitly mentioned. This choice is motivated by the context in which this work has been developed, which is the domain of Cyber-physical systems (CPSs) [MNBB18]. CPSs are systems characterized by a complex interweaving of hardware and software [LS16]. They are widely used in many safety-critical domains (e.g., aerospace, automotive, medical) where validation and verification (V&V) activities [BDD+18] of the system's intended functionality play a crucial role to guarantee the reliability and safety of the system. In such a domain, model-driven engineering is used throughout the development process and *simulation* is used for design-time testing of system models; simulation models (e.g., those defined in *Simulink*®)

capture both continuous and discrete system behaviors and, when executed, produce traces containing analog signals [GPVN+18].

2.5 Temporal Logics for Signal-based Properties

In this section, we provide a brief introduction to the main temporal logics that have been proposed in the literature for specifying signal-based temporal properties. They will be used in the next section to present the formalization of signal-based properties.

2.5.1 Signal Temporal Logic (*STL*)

STL [MN04] has been one of the first proposals of a temporal logic for the specification of temporal properties over dense-time (i.e., $\mathbb{T} = \mathbb{R}_{\geq 0}$), real-valued signals.

Let Π be a finite set of atomic propositions, X be a finite set of real variables, and $\mathcal{I}$ be an interval[3] $[a, b]$ over $\mathbb{R}$ with $a, b \in \mathbb{Q}_{\geq 0}$ such that $0 \leq a < b$. The syntax of *STL* with both *future* and *past* operators [MN13] is defined by the following grammar:

$$\varphi ::= p \mid x \sim c \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \,\mathsf{U}_{\mathcal{I}}\, \varphi_2 \mid \varphi_1 \,\mathsf{S}_{\mathcal{I}}\, \varphi_2$$

where $p \in \Pi$, $x \in X$, $\sim \in \{<, \leq, =, \geq, >\}$, $c \in \mathbb{R}$, $\mathsf{U}_{\mathcal{I}}$ is the metric "*Until*" operator, and $\mathsf{S}_{\mathcal{I}}$ is the metric "*Since*" operator. Additional temporal operators can be derived using the usual conventions; for example, "*Eventually*" $\mathsf{F}_{\mathcal{I}}\varphi \equiv \top \,\mathsf{U}_{\mathcal{I}}\, \varphi$; "*Globally*" $\mathsf{G}_{\mathcal{I}}\varphi \equiv \neg\mathsf{F}_{\mathcal{I}}\neg\varphi$; "*Once (Eventually in the Past)*" $\mathsf{P}_{\mathcal{I}}\varphi \equiv \top \,\mathsf{S}_{\mathcal{I}}\, \varphi$; "*Historically*" $\mathsf{H}_{\mathcal{I}}\varphi \equiv \neg\mathsf{P}_{\mathcal{I}}\neg\varphi$.

The semantics of *STL* is defined through a satisfaction relation $(s, t) \models_{STL} \varphi$, which indicates that signal s satisfies formula φ starting from position t in the signal. The satisfaction relation is defined inductively as follows:

$$\begin{aligned}
(s,t) &\models_{STL} p \text{ iff } p \text{ holds on } s \text{ in } t, \text{ for } p \in \Pi \\
(s,t) &\models_{STL} x \sim c \text{ iff } x \sim c \text{ holds on } s \text{ in } t, \text{ for } x \in X \text{ and } c \in \mathbb{R} \\
(s,t) &\models_{STL} \neg\varphi \text{ iff } (s,t) \not\models_{STL} \varphi \\
(s,t) &\models_{STL} \varphi_1 \vee \varphi_2 \text{ iff } (s,t) \models_{STL} \varphi_1 \text{ or } (s,t) \models_{STL} \varphi_2 \\
(s,t) &\models_{STL} \varphi_1 \,\mathsf{U}_{[a,b]}\, \varphi_2 \text{ iff } \exists t'.(t' \in [t+a, t+b] \text{ and } (s,t') \models_{STL} \varphi_2 \\
&\qquad \text{and } \forall t''.(t'' \in [t, t'] \text{ and } (s,t'') \models_{STL} \varphi_1)) \\
(s,t) &\models_{STL} \varphi_1 \,\mathsf{S}_{[a,b]}\, \varphi_2 \text{ iff } \exists t'.(t' \in [t-a, t-b] \text{ and } (s,t') \models_{STL} \varphi_2 \\
&\qquad \text{and } \forall t''.(t'' \in [t, t'] \text{ and } (s,t'') \models_{STL} \varphi_1))
\end{aligned}$$

We say that a signal s satisfies an *STL* formula φ iff $(s, 0) \models_{STL} \varphi$.

Several extensions of *STL* have been proposed in the literature. For example, STL/PSL [NM07] adds an analog layer to STL that enables the application of (low-level) signal operations; xSTL [NLM+18]

[3] The restriction on the non-punctual interval $\mathcal{I}$ for *STL* has been lifted in reference [MN13].

adds support for Timed Regular Expressions [ACM02]. The *STL* expressions that we will present in this thesis can be written in the same form also in STL/PSL or xSTL since they only rely on the core operators of *STL*.

2.5.2 *STL**

*STL** [BDŠV14] is an extension of *STL* that adds a signal-value *freezing* operator that binds the value of a signal to a precise instant of time.

Let $\mathcal{J}$ be a finite index set (e.g., the set $\{1, \ldots, n\}, n \in \mathbb{N}$) and let the function $t^*\colon \mathcal{J} \to [0, |s|]$ be the *frozen time vector*; the i-th frozen time can then be referred to with $t_i^* = t^*(i)$. As in the case of *STL*, let Π be a finite set of atomic propositions, X be a finite set of real variables, and $\mathcal{I}$ be an interval $[a, b]$ over $\mathbb{R}$ with $a, b \in \mathbb{Q}_{\geq 0}$ such that $0 \leq a < b$. The syntax of *STL** is defined by the following grammar:

$$\varphi ::= p \mid x \sim c \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \,\mathsf{U}_{\mathcal{I}}\, \varphi_2 \mid *_i[\varphi]$$

where $p \in \Pi$, $x \in X$, $\sim \in \{<, \leq, =, \geq, >\}$, $c \in \mathbb{R}$, $\mathsf{U}_{\mathcal{I}}$ is the metric "*Until*" operator, and $*_i$ is the unary signal-value freezing operator for all $i \in \mathcal{J}$. Additional operators like *Eventually* and *Globally* can be defined as done above for *STL*.

The semantics of *STL** is defined through a satisfaction relation $(s, t, t^*) \models_{STL*} \varphi$, which indicates that signal s satisfies formula φ starting from position t in the signal, taking into account the frozen time vector $t^* \in [0, |s|]^{\mathcal{J}}$. The satisfaction relation is defined inductively as follows:

$$\begin{aligned}
(s,t,t^*) &\models_{STL*} p \text{ iff } p \text{ holds on } s \text{ in } t, \text{ for } p \in \Pi, \text{ with the frozen time vector } t^* \\
(s,t,t^*) &\models_{STL*} x \sim c \text{ iff } x \sim c \text{ holds on } s \text{ in } t, \text{for } x \in X \text{ and } c \in \mathbb{R}, \text{with the frozen} \\
&\qquad \text{time vector } t^* \\
(s,t,t^*) &\models_{STL*} \neg\varphi \text{ iff } (s,t,t^*) \not\models_{STL*} \varphi \\
(s,t,t^*) &\models_{STL*} \varphi_1 \vee \varphi_2 \text{ iff } (s,t,t^*) \models_{STL*} \varphi_1 \text{ or } (s,t,t^*) \models_{STL*} \varphi_2 \\
(s,t,t^*) &\models_{STL*} \varphi_1 \,\mathsf{U}_{\mathcal{I}}\, \varphi_2 \text{ iff } \exists t'.(t' \in [t+a, t+b] \text{ and } (s,t,t^*) \models_{STL*} \varphi_2 \\
&\qquad \text{and } \forall t''.(t'' \in [t, t'] \text{ and } (s, t'', t^*) \models_{STL*} \varphi_1)) \\
(s,t,t^*) &\models_{STL*} *_i[\varphi] \text{ iff } (s,t,t^*[i \leftarrow t]) \models_{STL*} \varphi
\end{aligned}$$

where $[i \leftarrow t]$ is the operator substituting t with the i-th position in the frozen time vector, defined as $t^*[i \leftarrow t] = \begin{cases} t, & i = j \\ t^*(j), & i \neq j \end{cases}$.

We say that a signal s satisfies the *STL** formula φ iff $(s, 0, \mathbf{0}) \models_{STL*} \varphi$.

2.5.3 Signal First-Order Logic (*SFO*)

SFO [BFHN18] is a formalism that combines first order logic with linear real arithmetic and uninterpreted unary function symbols; the latter represent real-valued signals evolving over time.

Let F be a set of function symbols and let $X = T \cup R$ be a set of variables, where T is the set of *time* variables and R is the set of *value* variables. Let $\Sigma = \langle f_1, f_2, \ldots, \mathbb{Z}, -, +, < \rangle$ be a (first-order) signature where $f_1, f_2, \cdots \in F$ are uninterpreted unary function symbols, $\mathbb{Z}$ are integer constants, and $-, +, <$ are the standard arithmetic functions and order relation. The syntax of *SFO* over Σ is defined by the following grammar:

$$\begin{aligned}
\varphi &::= \theta_1 < \theta_2 \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \exists r \colon \varphi \mid \exists t \in \mathcal{I} \colon \varphi \\
\theta &::= \tau \mid \rho \\
\tau &::= t \mid n \mid \tau_1 - \tau_2 \mid \tau_1 + \tau_2 \\
\rho &::= r \mid f(\tau) \mid n \mid \rho_1 - \rho_2 \mid \rho_1 + \rho_2
\end{aligned}$$

where $r \in R$, $t \in T$, $n \in \mathbb{Z}$, $f \in F$, $\mathcal{I}$ is a time interval with bounds in $\mathbb{Z} \cup \{\pm\infty\}$. Notice that a term θ can be either a time term τ or a value term ρ. Additional logical connectors can be derived using the usual conventions; for example, $\forall r \colon \varphi \equiv \neg\exists r \colon \neg\varphi$.

Let a trace ω be an interpretation of a function symbol $f \in F$ as a signal, denoted by $[\![f]\!]_\omega$; let a valuation v be an interpretation of a variable $x \in X$ as a real number, denoted by $[\![x]\!]_v$. The valuation function for a term θ over the trace ω and the valuation v, denoted as $[\![\theta]\!]_{\omega,v}$ is defined inductively as follows: $[\![x]\!]_{\omega,v} = [\![x]\!]_v$, $[\![n]\!]_{\omega,v} = n$ for all $n \in \mathbb{Z}$, $[\![f(\tau)]\!]_{\omega,v} = \left[\!\!\left[f\left([\![\tau]\!]_{\omega,v} \right) \right]\!\!\right]_\omega$, $[\![\theta_1 - \theta_2]\!]_{\omega,v} = [\![\theta_1]\!]_{\omega,v} - [\![\theta_2]\!]_{\omega,v}$, $[\![\theta_1 + \theta_2]\!]_{\omega,v} = [\![\theta_1]\!]_{\omega,v} + [\![\theta_2]\!]_{\omega,v}$. The semantics of *SFO* is defined through a satisfaction relation $(\omega, v) \models_{SFO} \varphi$, which indicates the satisfaction of formula φ over the trace ω and the valuation v. The satisfaction relation is defined inductively as follows:

$$\begin{aligned}
(\omega, v) &\models_{SFO} \theta_1 < \theta_2 \text{ iff } [\![\theta_1]\!]_{\omega,v} < [\![\theta_2]\!]_{\omega,v} \\
(\omega, v) &\models_{SFO} \neg\varphi \text{ iff } (\omega, v) \not\models_{SFO} \neg\varphi \\
(\omega, v) &\models_{SFO} \varphi_1 \vee \varphi_2 \text{ iff } (\omega, v) \models_{SFO} \varphi_1 \vee (\omega, v) \models_{SFO} \varphi_2 \\
(\omega, v) &\models_{SFO} \exists r \colon \varphi \text{ iff } (\omega, v[r \leftarrow a]) \models_{SFO} \varphi \text{ for some } a \in \mathbb{R} \\
(\omega, v) &\models_{SFO} \exists t \in \mathcal{I} \colon \varphi \text{ iff } (\omega, v[t \leftarrow a]) \models_{SFO} \varphi \text{ for some } a \in \mathbb{R}
\end{aligned}$$

Variants of *SFO* can be defined by opportunely changing the underlying signature Σ.

The variant of *SFO* we use for all the formalizations in chapter 4show the following signature $\Sigma = \langle F, A, Rel, \mathbb{Z}, \mathbb{R} \rangle$, where:

- $F = Sig \cup Aux$ is the set of function symbols, composed of signal functions $Sig = \{s, s_1, s_2, s_{tr}\}$ and auxiliary functions and predicates $Aux = \{\sigma^{\mathbb{B}e}_{s,P}, \sigma^{\mathbb{B}s}_{s,P}, \xi, checkOsc, local\_min, local\_max\}$;
- A is the set of (non-linear) arithmetic functions $A = \{+, -, \times, \div, \mathrm{abs}\}$, where abs represents the absolute value operator;
- Rel is the set of relational operators $Rel = \{<, >, \geq, \leq, =, \neq\}$;
- $\mathbb{Z}$ and $\mathbb{R}$ are integer and real constants, respectively.

Chapter 3

A Model-driven Approach to Trace Checking of Temporal Properties with Aggregations

3.1 Overview

In this chapter, we consider temporal properties with aggregating operators. An example of a property of this specific class of temporal properties is: "the average number of client requests per hour computed over the daily business hours (from 7.30AM to 7PM) should be less than 10000", where the operator "average" is used to aggregate the number of events of type "client request" over one-hour intervals, in an observation time window ranging "from 7.30AM to 7PM". As detailed in , this class of temporal properties was identified in a field study [BGPS12] that identified *service provisioning patterns*; a new class of property specification patterns inspired by Dwyer et al.'s seminal work [DAC99] (for more details, refer to section 1.1).

To the best of our knowledge, the only specification language that supports the service provisioning patterns identified in [BGPS12] is *SOLOIST* [BGS13], which is a language based on first-order metric temporal logic extended with aggregating modalities.

In terms of checking procedure, in this chapter, we consider a *model-driven trace checking* approach [DBB17a]. Such an approach consists in *reducing* the problem of checking a property ρ over an execution trace λ to the problem of evaluating an OCL (Object Constraint Language) constraint (semantically equivalent to ρ) on an instance (equivalent to λ) of a meta-model of the trace. Model-driven trace checking has been proposed in the literature [DBB17a] *as a viable solution to the trace checking problem, to be adopted in software development contexts that rely*

on a model-driven engineering (MDE), a common practice in many domains [BCW17]. TEMPSY-CHECK [DBB17b] is a state-of-the-art tool implementing model-driven trace checking; it has been shown [DBB17a] to be highly scalable with respect to the length of the trace, exhibiting performance that is comparable to (and in some cases better than) state-of-the-art alternative technologies based on temporal logic. The properties to verify with TEMPSY-CHECK are expressed in *TemPsy* (Temporal Property made easy) [DBB17a], a pattern-based DSL for the specification of temporal properties. This language is based on the catalogue of property specification patterns by Dwyer et al. [DAC99], with new constructs derived from a field study performed in the domain of business processes for eGovernment. *TemPsy* is suitable for adoption by practitioners, since it is a high-level specification language that does not require a strong theoretical and mathematical background. Furthermore, *TemPsy* is pattern-based and inherits the benefits of pattern-based languages: for example, a recent empirical study [CZss] has shown that pattern-based temporal property specifications are easier to understand than specifications written using Linear Temporal Logic and the Event Processing Language.

The work proposed in this chapter is motivated by two observations. On one hand, though requirements specifications based on temporal properties with aggregation operators (i.e., those based on the "service provisioning" specification patterns) are common, the support provided in terms of verification (more specifically, trace checking) of such properties is limited. Indeed, the two non-distributed[1] trace checking algorithms for *SOLOIST* proposed in the literature [BGKSP14, BBG+14] do not scale well in terms of the length of the trace [BGK14]. On the other hand, there is TEMPSY-CHECK, a scalable and effective model-driven trace checking solution, which only supports temporal properties based on Dwyer et al.'s specification patterns.

The goal of this chapter is *to provide a solution for scalable model-driven trace checking of temporal properties with aggregation operators*. The main idea is to bridge the gap between property specifications based on service provisioning patterns and model-driven trace checking. More specifically, we extend the approach presented in [DBB17a] by proposing: 1) an extension of the *TemPsy* language called *TemPsy-AG*, which supports the most used service provisioning patterns identified in the study in [BGPS12]; 2) an extension of the *TemPsy* trace checking procedure, realized through an optimized mapping into OCL constraints (on a meta-model of execution traces) of the new types of properties included in *TemPsy-AG*.

We have implemented our approach in TEMPSY-CHECK-AG, a prototypical extension of TEMPSY-CHECK. We evaluated its scalability (in terms of execution time) with respect to the length of the trace and other parameters used in the specification of *TemPsy-AG* properties; we also compared its performance with respect to *SOLOIST*-translator, the state-of-the-art tool for (non-distributed) trace checking of *SOLOIST* specifications [BGKSP14, BBG+14]. The results show that TEMPSY-CHECK-AG scales linearly with respect to the parameters considered in the analysis; in particular its execution time for processing a large trace with one million events ranges between $6250\,\mathrm{ms}$ and $15339\,\mathrm{ms}$, depending on the aggregation operator used in

[1]The issue of distributed trace checking using Big Data technologies is out of the scope of this chapter (which restricts itself to the non-distributed case) and is left for future work; a distributed algorithm for *SOLOIST* has been presented in [BGK14].

the property. Furthermore, it can deal with much larger traces than *SOLOIST-Translator*, which could only handle traces up to 1500 events.

To summarize, the two main contributions of this chapter are: i) a model-driven approach for trace checking of temporal properties with aggregation operators; ii) an evaluation of the scalability of such an approach when implemented in the TEMPSY-CHECK-AG tool, and the comparison with a state-of-the-art alternative technology. In addition, the work presented in this chapter can be seen as a *successful case study* on the feasibility and viability of extending model-driven trace checking [DBB17a], i.e., a verification technique enabled by MDE technologies, with support for a larger class of properties, while retaining acceptable performance from a practical standpoint.

The rest of the chapter is structured as follows. Section 3.2 presents *TemPsy-AG*. Section 3.3 illustrates our approach for model-driven checking of temporal properties based on service provisioning patterns. Section 3.4 reports on the evaluation of the scalability of our implementation. Section 3.5 discusses related work.

3.2 Specifying temporal properties with aggregation operators through *TemPsy-AG*

As a preliminary step towards our model-driven approach for trace checking of temporal properties with aggregations, we extended the *TemPsy* language (see syntax in 2.1) to support the most used service provisioning patterns (i.e., S1, S3, and S4); the new version of the language is called *TemPsy-AG*. We modified the syntax of *TemPsy* by adding new rules corresponding to the constructs needed for the new patterns; the main additions to the grammar are shown in Figure 3.1. More specifically, *TemPsy-AG* sports three new, intuitive keywords (‘`avgRT`’, ‘`average`’, ‘`maximum`’) indicating, respectively, patterns S1, S3, and S4. In all these patterns, the time window used for aggregation is denoted through the ‘`within`’ keyword; the observation interval for patterns S3 and S4 is represented with the ‘`every`’ keyword; the bound is expressed through the non-terminal ⟨*Bound*⟩, supporting the usual relational operators.

The example properties presented in section 2.3 can be written in *TemPsy-AG* as follows (assuming a *globally* scope):

- *P1*: `globally avgRT(a,b) within 20 tu < 3`

⟨*Pattern*⟩ ::= ...
| ‘`avgRT`’ ‘`(`’⟨*Event*⟩ ‘`,`’ ⟨*Event*⟩‘`)`’ ‘`within`’ ⟨*Int*⟩ ‘`tu`’ ⟨*Bound*⟩
| ‘`average`’ ⟨*Event*⟩ ‘`within`’⟨*Int*⟩ ‘`tu`’ ‘`every`’⟨*Int*⟩ ‘`tu`’ ⟨*Bound*⟩
| ‘`maximum`’ ⟨*Event*⟩ ‘`within`’⟨*Int*⟩ ‘`tu`’ ‘`every`’⟨*Int*⟩ ‘`tu`’ ⟨*Bound*⟩
...
⟨*Bound*⟩ ::= (‘>’ | ‘>=’ | ‘<’ | ‘<=’ | ‘==’ | ‘!=’) ⟨*Int*⟩

Figure 3.1: Syntactic extension included in *TemPsy-AG*

- *P3*: `globally average a within 20 tu every 6 tu < 3`
- *P4*: `globally maximum a within 20 tu every 6 tu < 3`

In the same spirit as *TemPsy*, by design *TemPsy-AG* does not aim at being as expressive as a full-fledged temporal logic. Instead, its goal is to make as easy as possible the specification of common types of temporal properties through a pattern-based language.

The formal definition of the semantics of *TemPsy-AG* extends the one of *TemPsy* (available in [Dou16]), and is based on the concept of temporal linear traces. The semantics of the new three operators added in *TemPsy-AG* largely mirrors the formalization of the corresponding service provisioning patterns provided in [BGS13]. The main difference from the definitions in [BGS13] is that the operators corresponding to the service provisioning patterns are always evaluated in correspondence of the last element of a (sub-)trace; in other words, the time window defined by the parameter K is always computed with respect to the timestamp of the last element of the (sub-)trace (as done in section 2.3).

3.3 Model-driven trace checking of *TemPsy-AG* properties

Our approach for model-driven trace checking of temporal properties with aggregation operators is based on the existing trace checking procedure available in TEMPSY-CHECK [DBB17b, DBB17a] (see approach definition in 2.2). In this section we present the extension of this procedure to support the new types of properties included in *TemPsy-AG*.

Our main contribution is the operationalization, in OCL, of the semantics of the service provisioning patterns included in *TemPsy-AG*. This is a challenging task since this mapping has to be optimized, based on the structure of the properties to check, in order to achieve better performance.

As mentioned in section 2.2, the idea at the basis of model-driven trace checking, given as input a *TemPsy-AG* property represented by a scope s and a pattern p, is to evaluate an OCL invariant defined based on the type of s and p. This evaluation conceptually corresponds to applying the semantics of pattern p on the set of sub-traces that is determined by the semantics of scope s.

We extend the definition of this invariant to support the aggregation operators available in *TemPsy-AG*; a snippet of its OCL pseudocode is shown in Figure 3.2. The body of the invariant expression is a multi-way branch, which selects a certain branch based on the specific scope type used within the property (line 4 shows the case for the "globally" scope). In each branch, after determining the collection of sub-traces (as determined by the scope semantics) with a call to a function of the form `applyScope*` (as in line 5, invoking `applyScopeGlobally`), there is another multi-way branch, which selects a certain branch based on the specific pattern type used within the property. Lines 7–13 show the branches corresponding to the three new aggregation operators of *TemPsy-AG*. In each branch there is a function of the form `checkPattern*` that checks whether the pattern used in the property holds on each sub-trace. The core of our

```
1 context Trace
2 inv: self.properties->forAll(property:TemPsy::TemPsyExpression |
3   let scope:TemPsy::Scope = property.scope, pattern:TemPsy::Pattern = property.
       pattern in
4   if scope.type = TemPsy::GLOBALLY then
5    let subtraces:Sequence(OrderedSet(TraceElement)) =    applyScopeGlobally(scope)
         in
6    [...code related to the other patterns omitted...]
7    if pattern.type = TemPsy::MAX then
8      subtraces->forAll(subtrace | checkPatternMAX(subtrace, pattern))
9     else  if pattern.type = TemPsy::AVG then
10     subtraces->forAll(subtrace | checkPatternAVG(subtrace, pattern))
11    else if pattern.type = TemPsy::AVGRT then
12     subtraces->forAll(subtrace | checkPatternAVGRT(subtrace, pattern)
13    endif endif endif
14  else if scope.type = TemPsy::BEFORE then [...])
```

Figure 3.2: OCL invariant for checking *TemPsy-AG* properties on a trace

extension lies in the definition of three new OCL functions—`checkPatternAVGRT` (for pattern S1, "average response time"), `checkPatternAVG` (for pattern S3, "average number of events"), and `checkPatternMAX` (for pattern S4, "maximum number of events")—that contain the operational definition (in OCL) of the semantics of each pattern. The rest of the invariant definition (e.g., returning the verdict) is the same as in the original version [DBB17a].

In the rest of this section we illustrate the definition of the three new `checkPattern*` aforementioned functions, corresponding to the S1, S3, S4 service provisioning patterns; to ease readability, the algorithms are written using OCL pseudocode.

3.3.1 Checking the "average response time" pattern

Function `checkPatternAVGRT`, whose pseudocode is shown in Algorithm 1, takes as input a sub-trace and the parameters of an object representing an *average response time* pattern in *TemPsy-AG*: the pair of events (a, b), the length of the time window K, and a bound expressed with a relational operator $\bowtie$ and a numeric constant n. The function returns a Boolean value indicating whether the pattern holds on the input sub-trace, i.e., whether the cumulative sum (over all pairs of events (a, b) in the time window) of the time distance between each occurrence of event b and the corresponding occurrence of event a, divided by the number of (a, b) events pairs in the time window, satisfies the given bound.

The algorithm uses four auxiliary variables: $accDist$ represents the cumulative sum (over all pairs of events (a, b)) of the time distance between each occurrence of event b and the corresponding occurrence of event a; $numPairs$ is a counter keeping track of the number of (a, b) events pairs found; $inPair$ is a Boolean flag that is true when the event a has been seen and the corresponding event b has not been seen yet; $lastSeenTS$ is the timestamp of the last-seen occurrence of event a.

First (line 2), the function determines the timestamps corresponding to the left and right boundaries of the sub-trace to consider. The right boundary RB is the timestamp of the last element of the sub-trace, whereas the left boundary LB is determined by the value of the time window and is equal to $RB - K$; notice that parameter K is assumed to be greater than the sub-trace length.

The central block of the function is a loop (lines 3–11) that iterates over all elements of the input sub-trace whose timestamp is comprised between the left and the right boundaries of the time window[2]. For each element, we check whether the corresponding event is a match for either a or b. If we match an occurrence of event a, we set the flag $inPair$ to `true` and save the corresponding timestamp in variable $lastSeenTS$. Notice that, as discussed in the footnote on page 10, we assume that between two occurrences of event a there is an occurrence of event b. If we match an occurrence of event b, if the flag $inPair$ is `true` it means that this event is the one corresponding to the last occurrence of event a previously matched. In this case, we compute the distance between this element and the timestamp of the last-seen occurrence of event a, update the value of $accDist$ accordingly, increase by 1 the value of the $numPairs$ counter, and reset the value of $inPair$ to `false`. By construction, following the informal semantics of the pattern presented in section 2.3, the algorithm will ignore the occurrences of event a that are not matched by a corresponding b event within the time window, as well as the occurrences of event b that do not follow any event a in the time window.

The average response time is then computed by dividing the value of the variable $accDist$ (i.e., the cumulative time distance) by the value of variable $numPairs$ (i.e., the number of matched pairs). This value is passed to function $evalBound$, together with the value of parameters $\bowtie$ and n. This function evaluates the bound stated in the property, according to the relational operator $\bowtie$ and the numeric constant n; the result is then returned by the algorithm, representing the Boolean verdict of the trace checking procedure.

3.3.2 Checking the "average number of events" pattern

Function `checkPatternAVG`, whose pseudocode is shown in Algorithm 2, takes as input a sub-trace and the parameters of an object representing an *average number of events* pattern in *TemPsy-AG*: the event a, the length of the time window K, the length of the observation interval h, a bound expressed with a relational operator $\bowtie$, and a numeric constant n. The function returns a Boolean value indicating whether the pattern holds on the input sub-trace, i.e., whether the average number of occurrences of event a, aggregated over the observation intervals that are included in the time window, satisfies the given bound.

First, the algorithm computes the temporal boundaries of the sub-trace to consider: the right boundary RB is the timestamp of the last element of the sub-trace; the left boundary LB depends on the value of the time window and on the value of the observation interval and is equal to $RB - \lfloor \frac{K}{h} \rfloor * h$.

[2] We consider the time window interval closed to the right, open to the left.

Algorithm 1: checkPatternAVGRT

Input: a trace segment *subtrace* and the parameters of an instance of the *average response time* pattern of the form `avgRT`(a,b) `within` K `tu` $\bowtie n$:
Output: `true` if the pattern holds on *subtrace*; `false` otherwise

```
1  accDist ← 0, numPairs ← 0
   inPair ← false, lastSeenTS ← null
2  RB ← subtrace.last().timestamp, LB ← RB − k
3  foreach elem ∈ subtrace such that
   LB < elem.timestamp ≤ RB do
4      if elem.event = a then
5          inPair ← true
6          lastSeenTS ← elem.timestamp
7      else if elem.event = b then
8          if inPair = true then
9              accDist ← accDist + (elem.timestamp − lastSeenTS)
10             numPairs ← numPairs + 1
11             inPair ← false
12 return evalBound (accDist/numPairs, ⋈, n)
```

Algorithm 2: checkPatternAVG

Input: a trace segment *subtrace* and the parameters of an instance of the *average number of events* pattern of the form `average` a `within` K `tu every` h `tu` $\bowtie n$:
Output: `true` if the pattern holds on *subtrace*; `false` otherwise

```
1  RB ← subtrace.last().timestamp, LB ← RB − ⌊K/h⌋ ∗ h
2  totalOccurrences ← count (subtrace,(LB,RB),a)
3  numIntervals ← ⌊K/h⌋
4  return evalBound (totalOccurrences/numIntervals, ⋈, n)
```

Then the algorithm computes two values:

- the number of occurrences of event a occurring in the interval $(LB, RB]$, computed using the auxiliary function *count* and stored in variable *totalOccurrences*;
- the number of observation intervals over which to compute the aggregate value, which is equal to $\lfloor \frac{K}{h} \rfloor$ according to the semantics of the pattern (see section 2.3); this value is stored in variable *numIntervals*.

The average number of events is then computed by dividing the value of *totalOccurrences* by the value of *numIntervals*. The resulting value is passed to function *evalBound*, together with the value of parameters $\bowtie$ and n, to evaluate the bound stated in the property and determine

Algorithm 3: checkPatternMAX

Input: a trace segment *subtrace* and the parameters of an instance of the *maximum number of events* pattern of the form
`maximum` a `within` K `tu every` h `tu` $\bowtie$ n

Output: `true` if the pattern holds on *subtrace*; `false` otherwise

1 $RB \leftarrow$ *subtrace*.`last()`.*timestamp*
2 *intervals* $\leftarrow$ *getIntervals*`(`RB,K,h`)`
3 **foreach** $itv \in$ *intervals* **do**
4 $\quad$ *numOccurrences*.`append(`*count*`(`*subtrace*,*itv*,a`))`
5 **return** *evalBound* $(\max(numOccurrences), \bowtie, n)$

the verdict of the trace checking procedure.

3.3.3 Checking the "maximum number of events" pattern

Function `checkPatternMAX`, whose pseudocode is shown in Algorithm 3, takes as input a sub-trace and the parameters of an object representing a *maximum number of events* pattern in *TemPsy-AG*: the event a, the length of the time window K, the length of the observation interval h, a bound expressed with a relational operator $\bowtie$, and a numeric constant n. The function returns a Boolean value indicating whether the pattern holds on the input sub-trace, i.e., whether the maximum number of occurrences of event a, aggregated over the observation intervals that are included in the time window, satisfies the given bound.

First, the algorithm computes in variable RB the temporal right boundary of the trace, i.e., the timestamp of the last element. Then it determines—by calling the auxiliary function *getIntervals*—the left and right temporal boundaries of each observation interval in the sub-trace, based on the values of RB, K, and h. Function *getIntervals* will return a list with $\lceil \frac{K}{h} \rceil$ intervals; these intervals are open to the left and closed to the right. In this list, stored in variable *intervals*, the first $\lceil \frac{K}{h} \rceil - 1$ intervals have length h and have the form $(RB - (m+1)h, RB - mh]$ for $0 \leq m \leq \left(\lfloor \frac{K}{h} \rfloor - 1\right) h$; the last interval (i.e., the left-most on the timeline) in the list will be $\left(\max\left(RB - K, RB - \lceil \frac{K}{h} \rceil\right), RB - \lfloor \frac{K}{h} \rfloor h\right]$, to take into account the possibility of having a tail interval shorter than h, according to the semantics of the pattern described in section 2.3.

Afterwards, the loop at lines 3–4 computes (through the auxiliary function *count*), for each interval *itv* in the list *intervals*, the number of occurrences of event a in *itv* and stores this value in the set *numOccurrences*.

The maximum number of events is then determined by computing the maximum value over the set *numOccurrences*. This value is passed to function *evalBound*, together with the value of parameters $\bowtie$ and n, to evaluate the bound stated in the property and determine the verdict of the trace checking procedure.

3.3.4 Tool implementation

We have implemented our approach in TEMPSY-CHECK-AG, as an extension of the publicly available TEMPSY-CHECK [DBB17b] tool. The extension includes the OCL code to deal with the service provisioning patterns; we have also extended the *TemPsy* DSL editor to support the new expressions in the *TemPsy-AG* language.

Our extension uses the same toolchain as TEMPSY-CHECK: it takes as input a trace in CSV format and a text file following the DSL syntax, with the properties to check on the trace; the evaluation of the OCL constraints corresponding to the input properties to check is performed using the OCL checker included in the Eclipse OCL distribution.

3.4 Evaluation

We evaluated the scalability of our approach—in terms of the execution time—with respect to the length of the trace and other parameters used in the specification of *TemPsy-AG* properties. We also compared the performance of our approach with *SOLOIST-Translator*, the state-of-the-art tool for (non-distributed) trace checking of *SOLOIST* specifications [BGKSP14, BBG+14]. More specifically, we evaluated our approach implemented in TEMPSY-CHECK-AG by answering the following research questions:

RQ1: *How does* TemPsy-AG *scale with respect to the trace length when checking properties expressed using the three main service provisioning patterns (S1, S3, S4)?* (section 3.4.2.1)

RQ2: *How does* TemPsy-AG *scale with respect to the number of observation intervals induced by the values of the parameters K and h, when checking properties expressed using patterns S3 and S4?* (section 3.4.2.2)

RQ3: *How does* TemPsy-AG *fare with respect to* SOLOIST-Translator*, a state-of-the-art tool for checking properties expressed using the three main service provisioning patterns (S1, S3, S4)?* (section 3.4.2.3)

3.4.1 Evaluation Settings

3.4.1.1 Temporal Properties

We used the three following property templates to answer all three research questions, one for each type of service provisioning pattern:

- *P1*: `globally avgRT(a,b) within` K `tu < 5` (for pattern S1)
- *P3*: `globally average a within` K `tu every` h `tu < 5` (for pattern S3)
- *P4*: `globally maximum a within` K `tu every` h `tu < 5` (for pattern S4)

where a and b are event names and K, h are parameters that are varied according to the evaluation methodology (described below). Notice that all properties are expressed using the "globally" scope: we made this choice following the evaluation methodology proposed in existing work on model-driven trace checking [DBB17a]. Indeed, properties with the "globally" scope are the most challenging in terms of scalability, since the semantics of this scope guarantees that the pattern (used in the property to check) will be evaluated throughout the entire length of the trace.

3.4.1.2 Trace Generation Strategy

Following the evaluation guidelines proposed in existing work [BBG+16, DBB17a] on trace checking, we used *synthesized* traces for the evaluation. The use of synthesized traces over real ones allows us to systematically control the factors (e.g., the trace length, the number of intervals) that are relevant for our research questions, while setting other factors randomly, to avoid any bias.

We extended the trace generator program included in the TEMPSY-CHECK distribution with new generator strategies specific to the service provisioning patterns. The generator program takes as input a *TemPsy-AG* property, the desired length of the trace to generate, and additional parameters depending on the type of property given in input and the factors one wants to control. The position and the order of events are generated randomly taking into account the temporal and timing constraints prescribed by the semantics of the pattern used in the input property. Positions in the trace that are deemed not relevant for the evaluation of the property are filled with a dummy event, so that *the number of events in the trace is equal to the parameter K* used in patterns S1, S3, S4. In other words, between two adjacent events in the trace we assume a time difference of 1 time unit, possibly indicated by the presence of a dummy event. Given the semantics of the service provisioning patterns and taking into account their formalization in *TemPsy-AG* (see section 3.2), this case corresponds to the worst-case scenario (from a scalability point of view), in which the time window over which the properties with aggregations are evaluated includes all the elements of the trace. Below we sketch the trace generation strategies for the three new patterns.

Average response time **(P1).** For a given value of the parameter K we generate a trace of length K, containing X pairs of events (a, b), where X is a random value between 2 and $\frac{K}{2}$. We require that these pairs are distributed over the trace such that the average time distance between the individual occurrences of events a and the corresponding occurrences of b satisfies the bound indicated in the property. We use the Z3 constraint solver [DMB08] to get the value of the X distances (one for each events pair) that satisfy the property bound. Then we randomly allot the pairs of (a, b) events over the trace according to a uniform distribution, while maintaining for each pair the distance determined by the solver.

Average number of events **(P3).** For a given value of the parameters K and h, we generate a trace of length K; the number of observation intervals I is computed as $I = \lfloor \frac{K}{h} \rfloor$. We then need to determine the number of occurrences of event a in each of these I intervals, such that their

distribution on the trace satisfies the property bound. We use the Z3 constraint solver to find an assignment for the I variables that represent the number of event occurrences in each interval. Finally, within each interval, we randomly generate (with a uniform distribution on the range induced by h) the required number of occurrences of events a in that interval.

***Maximum number of events* (P4).** For a given value of the parameters K and h, we generate a trace of length K; the number of observation intervals I is computed as $I = \lceil \frac{K}{h} \rceil$. We then need to determine the number of occurrences of event a in each of these I intervals, such that their distribution on the trace satisfies the property bound. For example, if the bound used in the property is "`< n`", in each observation interval we will generate occurrences of event a with a uniform distribution on the range $[0, \mathtt{n}]$.

3.4.1.3 Computer Settings

The results reported in this section have been measured using the Unix `time` program on a desktop computer with a $2.5\,\mathrm{GHz}$ Intel Core i7 CPU and $16\,\mathrm{GB}$ of memory, running Eclipse DSL Tools v. 4.6.2 (Neon Milestone 2), JavaSE-1.7 (Java SE v. 1.8.0_121, Java HotSpot (TM) 64-Bit Server VM v. 25.121-b13, mixed mode), Eclipse OCL v. 6.1.2, and *SOLOIST-Translator* (most recent version, commit `65684d1`). All measurements reported correspond to the average value over 5 runs of the trace checking procedure (on the same trace, for the same property).

3.4.2 Evaluation Results

3.4.2.1 Scalability with respect the to trace length

Methodology. To answer RQ1, we ran TEMPSY-CHECK-AG on traces of different length (i.e., value of the parameter K in the input property), ranging from 100K to 1M in steps of 100K; on each trace, we checked the three properties shown above. For P3 and P4 properties, we varied h so that the number of intervals was fixed to 10.

Results. The execution time measured for our approach is shown in Figure 3.3a. Overall, the execution time varies from $2218\,\mathrm{ms}$, for checking property P1 (with the "average response time" pattern) on a 100K trace, to $15339\,\mathrm{ms}$, for checking property P3 (with the "average number of events" pattern) on a 1M trace. We also notice that checking properties with the "average/maximum number of events" patterns requires longer than checking properties with the "average response time" pattern.

The answer to RQ1 is that our approach *scales linearly* with respect to the length of the trace for all three types of service provisioning patterns.

3.4.2.2 Scalability with respect to the number of observation intervals

Methodology. To address RQ2, we generated traces with length varying from 100K to 500K in steps of 100K; for each of these trace lengths, we considered 10 different values for the number

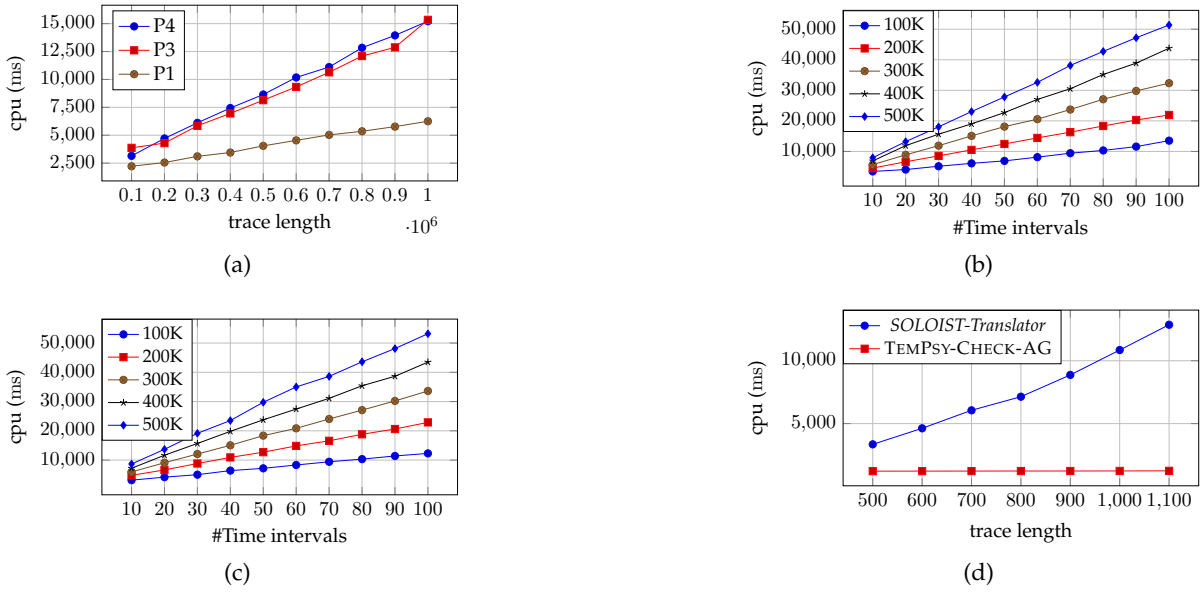


Figure 3.3: Scalability in terms of execution time with respect to the trace length (a); scalability in terms of the number of observation intervals for P3 properties (b) and P4 properties (c); comparison between the execution time of TEMPSY-CHECK-AG and of *SOLOIST-Translator* for checking P1 properties (d)

of observation intervals (used in the context of patterns S3 and S4), ranging from 10 to 100 in steps of 10. In practice, to vary the number of observation intervals, we varied the value of parameter h in properties P3 and P4 such that, when combined with the value of parameter K, it yields the desired value for the number of observation intervals. For example, in the case of a property with the "average number of events" pattern, the value of h for obtaining 30 intervals on a 100K long trace is $h = \lfloor \frac{100000}{30} \rfloor = 3333$. Notice that in the case of the "maximum number of events" pattern, when $K \mod h \neq 0$, the tail interval will have a length shorter than h and there is a range of values for h that yield the same number of observation intervals; in this case, we chose the lowest value for h to get the largest possible tail interval. We executed TEMPSY-CHECK-AG on all the generated traces, for the different values of K and h, to check properties P3 and P4.

Results. The execution time for checking properties with pattern S3 "average number of events" and with pattern S4 "maximum number of events" is shown in Figure 3.3b and Figure 3.3c, respectively. Overall, the execution time ranges from $3138\,\mathrm{ms}$, for checking a property with pattern S3 on a 100K long trace with 10 time intervals, to $53183\,\mathrm{ms}$, for checking a property with pattern S3 on a 500K long trace with 100 time intervals. In line with the results discussed for RQ1, the execution time for checking properties P3 and P4 is similar, i.e., checking patterns S3 and S4 has a similar cost.

The answer to RQ2 is that TEMPSY-CHECK-AG scales linearly with respect to the number of time intervals for patterns S3 and S4. With the same number of observation time intervals, the higher the length of the trace, the longer it takes to complete the trace checking procedure.

3.4.2.3 Comparison with *SOLOIST-Translator*

Methodology. To answer RQ3, we compared the performance (in terms of execution time) of TEMPSY-CHECK-AG with *SOLOIST-Translator*, the only publicly available tool for non-distributed trace checking of properties written in *SOLOIST*, a language that supports the same service provisioning patterns as *TemPsy-AG*.

However, the two non-distributed trace checking algorithms for *SOLOIST* proposed in the literature [BGKSP14, BBG+14], implemented in *SOLOIST-Translator*, do not scale well in terms of the length of the trace [BGK14]. A preliminary set of experiments that we conducted reported similar results to those published in [BGKSP14, BBG+14]: *SOLOIST-Translator* cannot handle traces longer than 1500 for checking properties with patterns S3 and S4, and traces longer than 1200 for checking properties with pattern S1. This is due to the internal translation of input traces done by the tool, which takes into account the granularity of the timestamps of the trace elements. For these reasons, we could only use small traces to compare the two tools:

- For properties with pattern S1, we generated traces with length varying from 500 to 1100 in steps of 100.
- For properties with patterns S3 and S4, we generated traces with length varying from 1000 to 1500 in steps of 100, and varied the number of observation intervals (by setting the parameter h) through the values 2, 10, 50, 100.

Results. The results indicate that *SOLOIST-Translator* displays a steep linear growth of the execution time for traces with a short length, whereas our approach takes almost a constant time. For space reasons, we only show (in Figure 3.3d) the results for the case of properties with pattern S1 "average response time". For a trace with length 1100 (the largest length we considered), *SOLOIST-Translator* took $12864\,\mathrm{ms}$, whereas TEMPSY-CHECK-AG took $1243\,\mathrm{ms}$. We remark that the execution time taken by *SOLOIST-Translator* for checking pattern S1 on a trace with length 1100 is *more than twice* the time ($6250\,\mathrm{ms}$) taken by TEMPSY-CHECK-AG for checking a similar property on a 1M trace (see Figure 3.3a).

For the case of properties with the "average number of events" pattern, when varying the number of time intervals from 2 to 100, the execution time of *SOLOIST-Translator* ranges from $7929\,\mathrm{ms}$ to $8285\,\mathrm{ms}$ for a trace with length 1000, and from $17614\,\mathrm{ms}$ to $18\,254\,\mathrm{ms}$ for a trace with length 1500. On the other hand, TEMPSY-CHECK-AG takes from $1260\,\mathrm{ms}$ to $1446\,\mathrm{ms}$ for a trace with length 1000, and from $1247\,\mathrm{ms}$ to $1574\,\mathrm{ms}$ for a trace with length 1500. We observed similar values for the case of properties with the "maximum number of events".

The answer to RQ3 is that TEMPSY-CHECK-AG can handle much larger traces than *SOLOIST-Translator* (with up to a 1000x increase in length) and exhibits faster execution times.

3.4.3 Discussion

The results presented above indicate that the execution time of TEMPSY-CHECK-AG is acceptable from a practical standpoint: temporal properties with aggregation operators can be checked on a trace with millions of events within seconds. Furthermore, the comparison with *SOLOIST-Translator*, a state-of-the-art tool that supports the same service provisioning patterns (S1, S3, S4) as TEMPSY-CHECK-AG, shows that our approach can handle much larger traces than *SOLOIST-Translator* (with up to a 1000x increase in length) and exhibits faster execution times. Overall, these results show that extending a model-driven trace checking approach [DBB17a] with support for a larger range of properties yield a scalable and viable solution for verifying, in offline settings, temporal properties with aggregation operators.

Threats to validity. One of the main threats is the use of synthesized traces in the evaluation. Real execution traces might be different, in terms of events occurrences and time distances. However, this threat does not affect our research questions on scalability, as we want to analyze the execution time as a function of a number of parameters (e.g., trace length), while varying randomly other aspects. Another threat is the representativeness of the properties templates used for the evaluation. These property templates, although simple, represent the type of properties (with service provisioning patterns) that can be written in realistic scenarios, since they are based on those extracted in a study [BGPS12] of specifications written in industrial settings; furthermore, similar templates have been used in existing work [BGKSP14, BBG+14, BGS13] on trace checking of the same service provisioning patterns. Another threat is given by the use of Eclipse OCL; one could get different results by using another OCL checker, with lower performance. We chose Eclipse OCL for its scalability. Finally, as for the comparison with *SOLOIST-Translator*, we remark that its specification language (*SOLOIST*) is more expressive than *TemPsy-AG* (e.g., by supporting first-order quantification and the full set of metric temporal logic modalities); hence the performance of *SOLOIST-Translator* could have been negatively affected by the more complex implementation needed to support a richer specification language.

3.5 Related work

Besides the work revolving around the *SOLOIST* language [BGKSP14, BBG+14, BGS13], there are other approaches that focus on run-time verification of properties with aggregation operators [SW95, FSS05, DSS+05, Rap16, BKMZ15]. Among the most recent, Basin et al. [BKMZ15] present an extension of the MFOTL logic that supports aggregation on data, i.e., terms used in the logical predicates, and the corresponding monitoring algorithm; Rapin [Rap16] proposes a dense-time specification language (and the monitoring algorithm), in which aggregation operators can be used to specify invariants of hybrid, signal-based systems. The main difference of these approaches from ours is the specific type of aggregation operators considered: in all the aforementioned approaches, the aggregation is done on the values of data/signals whereas the

service provisioning patterns supported by *TemPsy-AG* aggregate events. LarvaStat [CGP10] is an extension of the Larva monitoring tool [CP17] with support for collecting statistical data of the execution, through point- and interval-statistics operators; however, the definition of these operators is quite operational, requiring explicitly to specify the update rules for the aggregations and the conditions characterizing the intervals. In contrast, *TemPsy-AG* provides high-level aggregation operators, with pre-defined semantics.

One of the key features of Complex Event Processing (CEP) systems [Luc01] is to aggregate data from multiple sources, using aggregating operators similar to those included in *TemPsy-AG*. One of the main difference between CEP approaches and RV ones is that the former compute the result of a query on a event trace, whereas the latter evaluate a property on a trace [Hal16]. Approaches like BeepBeep [Hal16] combine CEP and RV, allowing the evaluation of properties (possibly temporal) over event streams processed (e.g., by means of aggregating operators) through a CEP-like pipeline. Conceptually, *TemPsy-AG* adopts a similar approach, since in the OCL constraints the events are aggregated according to the pattern semantics before the evaluation of the relational expression in the property.

3.6 Summary

The verification of complex software systems often requires to check temporal properties that contain aggregation operators. When specifying such properties, a software engineer can leverage an existing catalogue of property specification patterns, called "service provisioning patterns" [BGPS12]. Nevertheless, existing solutions for trace checking of these properties suffer from scalability limitations. In this chapter we have presented our solution for scalable trace checking of temporal properties with aggregation operators, extending an existing model-driven approach for trace checking. Our approach is based on an optimized mapping into OCL constraints (on a meta-model of execution traces) of the main service provisioning patterns. We have implemented our approach in the tool TEMPSY-CHECK-AG and evaluated its performance: the results show that our approach can check temporal properties with aggregation operators on a trace with millions of event within seconds, scales linearly with respect to the length of the input trace, and can deal with much larger traces than a state-of-the-art tool. Furthermore, the results indicate the feasibility and viability of extending model-driven trace checking [DBB17a], a promising run-time verification approach enabled by MDE technologies, with support for a larger class of properties, while retaining acceptable performance from a practical standpoint.

Chapter 4

Signal-Based Properties: Taxonomy and Logic-based Characterization

4.1 Overview

Expressing requirements in terms of SBTPs poses a number of challenges for system and software engineers; 1) the possibility to detect and characterize signal behaviors (e.g., a *spike*) using different features (and parameters), without a proper guideline for selecting some of these features/parameters and 2) the availability of specification languages with different levels of expressiveness, which also requires a guideline to properly choose the appropriate language for the specification of SBTPs, while taking into account the corresponding trace checking tools, if any (for more details about these challenges, refer to section 1.1).

In this chapter, we tackle these challenges by proposing a taxonomy of the most common types of SBTPs and a logic-based characterization of such properties. Based on industrial experience and a thorough review of the literature, our goal is to provide system and software engineers, as well as researchers working on CPSs, with a reference guide to systematically identify and characterize signal behaviors, to support both requirements specification and V&V activities. More specifically, we address the first challenge by providing, through the taxonomy, a comprehensive and detailed description of the different types of signal-based behaviors, with each property type precisely characterized in terms of a temporal logic. As a result, an engineer can be guided by the precise characterization of the property types included in our taxonomy, to derive—from an informal requirements specification—a formal specification of a property, which can then be used in the context of V&V activities (e.g., as test oracle). We take on the second challenge by reviewing the expressiveness of the main temporal logics that

have been proposed in the literature for specifying Signal-based Temporal Properties (i.e., *STL*, *STL** [BDŠV14], *SFO* [BFHN18] - Signal First-Order Logic), in terms of the property types identified in the taxonomy. In this way, we can guide engineers to choose a specification formalism based on their needs in terms of property types to express.

We developed our taxonomy of SBTPs based on practical experience in analyzing temporal requirements in CPS domains like the aerospace industry, and by reviewing the literature in the area of verification of cyber-physical systems, starting from the recent survey of specification formalisms in reference [BDD+18]. We identified and included in our taxonomy seven property types:

For each of these types, we provide a logic-based characterization using *SFO* and also discuss alternative formalizations—when applicable—using also *STL* and *STL**. In this way, we are able to report on the expressiveness of state-of-the-art temporal logics with respect to the property types included in our taxonomy: *SFO* is the only language among the three we considered in which we can express *all the property types of our taxonomy*.

We also report on the application of our taxonomy to classify the requirements specifications of an industrial case study in the aerospace domain. Through this case study we show:

- The feasibility of expressing requirements specifications of a real-world CPS using the property types included in our taxonomy. Indeed, in the vast majority of the cases, the mapping from a specification written in English to its corresponding property type defined in the taxonomy was straightforward.
- The completeness of our taxonomy: all requirements specifications of the case study could be defined using the property types included in our taxonomy.

To summarize, the main contributions of this chapter are:

- a taxonomy of SBTPs;
- a logic-based characterization of the various property types included in the taxonomy;
- a discussion on the expressiveness of state-of-the-art temporal logics with respect to the property types included in our taxonomy;
- the application of our taxonomy to classify the requirements specifications of an industrial case study in the aerospace domain.

The rest of this chapter is structured as follows. Section 4.2 illustrates our taxonomy of SBTPs and provides a logic-based characterization of each property type. In section 4.3 we discuss the expressiveness of state-of-the-art temporal logics with respect to the property types included in our taxonomy. Section 4.4 presents the application of our taxonomy to an industrial case study. Section 4.5 discusses how this chapter contributions can support the research community and practitioners. Section 4.6 discusses related work.

4.2 Taxonomy of signal-based Temporal properties

One of the main challenges in using Signal-based Temporal Properties for expressing requirements of CPSs is the lack of precise descriptions of signal behaviors. First, a signal behavior (e.g., a spike or an oscillation) can be "described" in different ways, i.e., it can be characterized using various features; for example, a total of 16 different features (and eight parameters) have been identified in the literature [AMM+14] to detect a spike in a signal. Given the large variety of options, (software and system) engineers may choose various subsets of features for characterizing the same type of signal behavior, leading to ambiguity and inconsistency in the specifications. In addition, slightly different features may have similar names (e.g., "peak amplitude" and "peak-to-peak amplitude"), potentially leading to mistakes when writing specifications. It is then important to define proper guidelines for selecting the features most appropriate in a certain context, and provide engineers with a precise characterization of such features.

In this section, we tackle this challenge by proposing a taxonomy of the most common types of Signal-based Temporal Properties and a logic-based characterization of such properties. Our goal is to provide system and software engineers, as well as researchers working on CPSs, with a reference guide to systematically identify and characterize signal behaviors, so that they can be defined precisely and used correctly during the development process of CPSs, in particular during the activities related to requirements specification and V&V.

Our taxonomy provides a comprehensive and detailed description of the different types of signal-based behaviors, with each property type precisely characterized in terms of a temporal logic. As a result, an engineer can be guided by the precise characterization of the property types included in our taxonomy, to derive—from an informal requirements specification—a formal specification of a property, which can be used in other development activities (e.g., V&V).

We developed this taxonomy based on our general understanding of temporal requirements in CPS domains like the aerospace industry, and by reviewing the literature in the area of verification of cyber-physical systems, starting from the recent survey in reference [BDD+18]. The taxonomy focuses on properties specified in the time domain; we purportedly leave out properties specified in the frequency domain [NKJ+17, DMB+12] because in our context (V&V of CPS) the properties of interest are mainly specified in the time domain.

The taxonomy with the acronyms of signal-based property types is shown in figure 4.1. At the top level, it includes three main signal-based property types:

Data assertion (DA): properties expressing constraints on the value of a signal.

Signal behavior (SB): properties on the behavior represented by a signal shape. We further distinguish among two property subtypes:

- properties on signals exhibiting spikes (SPK) ;
- properties on signals manifesting oscillatory behaviors (OSC).

Relationship between signals (RSH): properties characterizing relationships between signals. This type includes two further property subtypes:

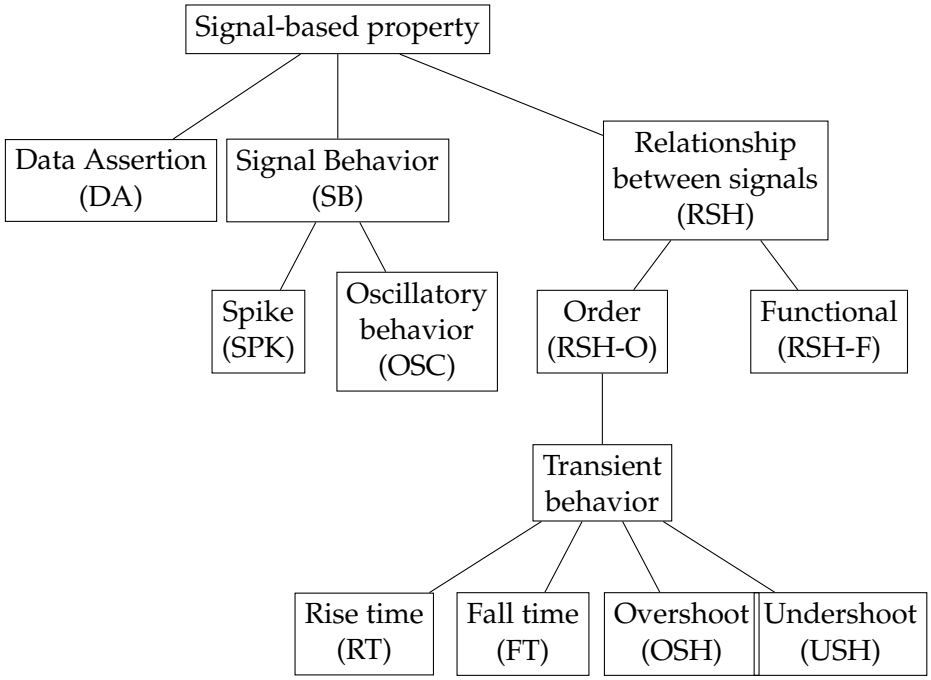


Figure 4.1: Taxonomy of Signal-based Temporal Properties

- *functional*, based on the application of a signal transforming function (RSH-F);
- *order*, describing sequences of events/states related to signal behaviors (RSH-F). In this category we also include properties of transient behaviors of a signal when changing from the current value to a new target value, such as:
  - properties on signals exhibiting a *rising* (Rise Time - RT) or a *falling* (Fall Time - FT) behavior;
  - properties on signals exhibiting an overshoot (OSH) or an undershoot (USH) behavior.

In the following subsections we provide the detailed description of each property type, including a mathematical formalization and examples. We use (a variant of) *SFO* to formalize the various property types; anticipating the results of section 4.3, the reason for the adoption of *SFO* is its expressiveness, which allows us to express *all the property types considered in this chapter*. We also provide examples of properties in *STL* and *STL** (when applicable).

4.2.1 Data assertion

A data assertion specifies a constraint on the value of a signal. This constraint is expressed through a *signal predicate* of the form $s \bowtie expr$, where $expr$ is an *SFO* value term defined over the value domain of the signal s and $\bowtie \in Rel$. A data assertion property holds on the signal if the

assertion predicate evaluates to $true$. Data assertions can be combined to form more complex expressions through the standard logical connectives. We distinguish between *untimed* data assertions, which are evaluated through the entire domain of definition I_s of a signal s, and *time-constrained* data assertions, which are evaluated over one or more distinct sub-intervals of the signal domain of definition.

More formally, let H be a set of time intervals $H = \{\mathcal{I}_1, \ldots, \mathcal{I}_K\}$, such that $\mathcal{I}_k \subseteq I_s, 1 \leq k \leq K$, and for all $i, j \in \{1, \ldots K\}, i \neq j$ implies $\mathcal{I}_i \cap \mathcal{I}_j = \emptyset$. A data assertion defined over the time intervals in H holds on a signal s if and only if (iff) the *SFO* formula $\bigwedge_{h \in H} \forall i \in h \colon s(i) \bowtie expr$ evaluates to *true*. Notice that an *untimed* data assertion over a signal s is defined by having $H = \{I_s\}$.

For example, let us consider the property *pDA*: "The signal value shall be less than 3 between $2\,\mathrm{tu}$ and $6\,\mathrm{tu}$ and between $10\,\mathrm{tu}$ and $15\,\mathrm{tu}$", where "tu" is a generic time unit (which has to be set according to the application domain, e.g., seconds). This property is a *time-constrained* data assertion over the two intervals $[2, 6]$ and $[10, 15]$; it can be expressed in *SFO* as:

$$\boxed{\text{SFO}}\ pDA \quad \forall t \in [2, 6] : s(t) < 3 \wedge \forall t \in [10, 15] : s(t) < 3$$

Figure 4.2 shows two signals, s_1 plotted with a thick line (—), and s_2 plotted with a thin line (—); the threshold on the signal value specified by the property is represented with a dashed horizontal line. Property *pDA* does not hold for s_2 as its value is above the threshold of 3 in the intervals $[2, 6]$ and $[10, 15]$; however, it holds for s_1 because its value is below the threshold in both intervals.

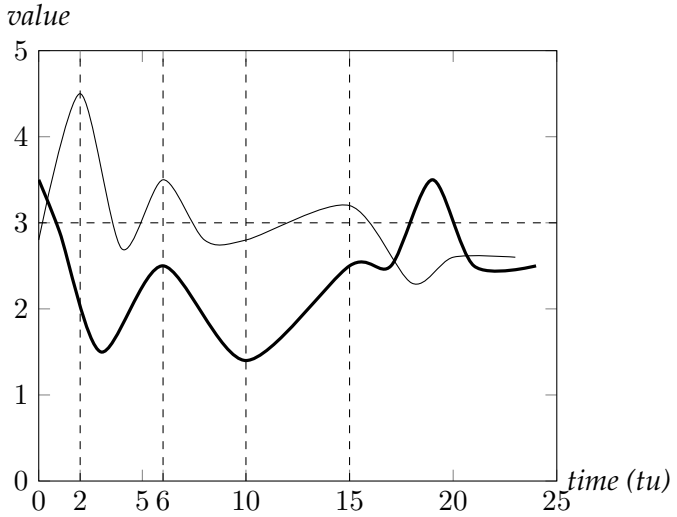


Figure 4.2: Two signals used to evaluate property *pDA*: signal s_1 (—) satisfies the property whereas signal s_2 (—) violates it.

Figure 4.3: (a) Main features used to define a spike based on [DRS82]. (b) two signals used to evaluate property *pSPK1*: signal s_1 (—) satisfies the property, whereas s_2 (—) violates it.

4.2.1.1 Alternative formalizations

Data assertion properties like *pDA* can be also expressed in *STL* and *STL**:

$$\boxed{\text{STL}\ pDA} \equiv \boxed{\text{STL}^*\ pDA} \quad \mathsf{G}_{[2,6]}(s < 3) \wedge \mathsf{G}_{[10,15]}(s < 3)$$

4.2.2 Spike

A spike[1] can be informally defined as a short-lived, (relatively) large increase or decrease of the value of a signal. Such a signal behavior is typically undesirable [BDD+18]. However, there are situations in which a spike characterized by a set of specific features is desirable, as it is the case for the discovery pulse [Nič15] in the discovery mode of the DSI3 protocol [DSI11]. Inspired by the definitions in the bio-medical domain [DRS82], we consider

four main features to characterize a spike, based on three extrema of the function corresponding to the signal shape, which are local extrema with respect to an observation interval $[f, g] \subset I_s$. These three points (with their respective coordinates) are: the peak point $(PP, s(PP))$ representing the local maximum of the signal and characterizing the actual spike[2], and the two surrounding valley points $(VP_1, s(VP_1))$ and $(VP_2, s(VP_2))$ representing the local minima (closest to the peak point) of the first and second half of the spike, respectively. These three local extrema are shown in figure 4.3a; we refer the reader to reference [DRS82] for a detailed description of how to detect these points. The four features (also shown in figure 4.3a) characterizing a spike are:

- Amplitude a of the spike, defined as $a = \psi(a_1, a_2)$, where a_1 is the amplitude of the first-half of the spike shape $a_1 = \text{abs}\,(s(PP) - s(VP_1))$, a_2 is the amplitude of the second-half

[1] A spike is also called bump, peak, or pulse in the literature.

[2] In the following we only characterize and formalize spikes corresponding to an increase of the signal value; the case of a decrease of the signal value is the dual.

of the spike shape $a_2 = \text{abs}\,(s(PP) - s(VP_2))$, and ψ is a generic amplitude function[3];

- slope sp_1 between the peak point and the valley point of the first half of the spike shape, $sp_1 = \text{abs}\left(\frac{s(PP)-s(VP_1)}{PP-VP_1}\right)$;
- slope sp_2 between the peak point and the valley point of the second half of the spike shape, $sp_2 = \text{abs}\left(\frac{s(PP)-s(VP_2)}{PP-VP_2}\right)$;
- spike width w between the two consecutive valley points, $w = VP_2 - VP_1$. Note that the width w can be also defined as $w = w_1 + w_2$, where $w_1 = PP - VP_1$ and $w_2 = VP_2 - PP$.

The four features a, sp_1, sp_2, and w can be opportunely combined to define a spike of a particular shape[4].

A spike property specifies a constraint on the existence of a spike with certain features; it evaluates to true when the signal exhibits a spike whose features satisfy certain criteria. More specifically, when defining a spike property, an engineer has to specify—for each feature—a predicate with a *threshold criterion* whose value depends on the application context. The signal predicates of each feature are then logically conjoined for characterizing the spike.

Formally, given the threshold criteria for the four features (specified as *SFO* terms over the value domain of signal s) $\Gamma_a, \Gamma_{sp_1}, \Gamma_{sp_2}, \Gamma_w$, a spike property holds on a signal s iff the following *SFO* formula evaluates to true:

$$\begin{aligned} \exists VP_1, PP, VP_2 \in [f, g]\colon\ & \mathit{local\_min}(VP_1, f, PP) \wedge \\ & \mathit{local\_max}(PP, VP_1, g) \wedge \\ & \mathit{local\_min}(VP_2, PP, g) \wedge \\ & a \bowtie \Gamma_a \wedge sp_1 \bowtie \Gamma_{sp_1} \wedge sp_2 \bowtie \Gamma_{sp_2} \wedge w \bowtie \Gamma_w \end{aligned} \tag{4.1}$$

where $\bowtie\ \in \mathit{Rel}$, $\mathit{local\_min}$ and $\mathit{local\_max} \in \mathit{Aux}$ are predicates identifying local extrema, and a, sp_1, sp_2, w are *SFO* terms defined as shown above using the three variables VP_1, VP_2, and PP.

In essence, formula (4.1) requires a) the existence of the three local extrema in a proper order characterizing the spike shape (i.e., a local minimum followed by a local maximum, followed by another local minimum), and b) the satisfaction of the constraints for all the features. More relaxed formulations can be obtained by omitting some of the spike features from the above definition.

[3] This function depends on the application domain; for example, in the context of bio-medical systems [DRS82], ψ is the minimum function.

[4] Although other spike features have been proposed in the spike detection literature—such as different types of width, amplitude, and slope [AG04, Acı05, AOK+05, LZY02, DJCF93], as well as the area under the curve [Häg95]—we decided not to adopt them since the features we have selected are sufficient to describe (and specify) the spike behaviors we consider in this paper.

The predicate $local\_min(x, y, z)$ (respectively, $local\_max(x, y, z)$) returns true if the time point x is a local minimum (respectively, local maximum) with respect to the interval $[y, z]$. These predicates can be defined in several ways; below we provide three possible definitions.

Definition 1 (local extrema through punctual derivatives) *Some specification languages allow for defining expressions corresponding to punctual derivatives. For example, in* SFO *the punctual derivatives can be defined as language terms as follows:*

$$s'_p(t) \equiv \frac{s(t+\epsilon) - s(t)}{\epsilon} \text{ and } s''_p(t) \equiv s'_p(s'_p(t))$$

with ϵ being an arbitrary, small constant[5]*. The local extrema predicates can then be defined in* SFO *as follow:*

$$local\_min(x, y, z) \equiv \exists x \in [y, z] \colon s'_p(x) = 0 \wedge s''_p(x) > 0$$
$$local\_max(x, y, z) \equiv \exists x \in [y, z] \colon s'_p(x) = 0 \wedge s''_p(x) < 0$$

Definition 2 (local extrema - analytical formulation) *Another way to characterize local extrema is to write a logical expression corresponding to their analytical definition; in* SFO *we have*

$$local\_min(x, y, z) \equiv \exists x \in [y, z] \colon \forall t \in [y, z], x \neq t \colon s(x) \leq s(t)$$
$$local\_max(x, y, z) \equiv \exists x \in [y, z] \colon \forall t \in [y, z], x \neq t \colon s(x) \geq s(t)$$

Definition 3 (local extrema through pre-computed derivatives) *When the first and second order derivatives of a signal are available as* (pre-computed), separate signals*, the local extrema can be characterized using such signals. Let s'_c and s''_c be the first and second order derivatives of signal s; the local extrema predicates can defined in* SFO *as follow:*

$$local\_min(x, y, z) \equiv \exists x \in [y, z] \colon s'_c(x) = 0 \wedge s''_c(x) > 0$$
$$local\_max(x, y, z) \equiv \exists x \in [y, z] \colon s'_c(x) = 0 \wedge s''_c(x) < 0$$

The choice of which definition to use for defining local extrema predicates depends on the specification language and the application context; as shown above, all three definitions can be used with *SFO*.

For example, let us characterize spikes through features width w and amplitude a, with the latter defined by using the maximum function as the amplitude function ψ; let us consider the evaluation of property *pSPK1*: "In a signal, there is a spike with a maximum width of $20\,\mathrm{tu}$ and a maximum amplitude of 1". For this property, the parameters of an instance of specification (4.1)

[5] In the context of a discrete signal, the ϵ constant can be replaced with the sampling interval Δ.

are $\Gamma_a = 1$ and $\Gamma_w = 20$; the resulting *SFO* formula is:

SFO *pSPK1* $\exists t, t', t'' \in [f, g]\colon local\_min(t, f, t') \wedge$
$local\_max(t', t, g) \wedge$
$local\_min(t'', t', g) \wedge$
$\max(\text{abs}\,(s(t') - s(t)), \text{abs}\,(s(t'') - s(t'))) \leq 1 \wedge \text{abs}\,(t'' - t) \leq 20$

In figure 4.3b, we show two signals, s_1 plotted with a thick line (—) and s_2 plotted with a thin line (–). To evaluate property *pSPK1* on these signals, we first need to evaluate the local extrema predicates in specification (4.1) (according to one of the three definitions above): signal s_1 exhibits a spike where $VP_1 = 10$, $PP = 20$, and $VP_2 = 30$, while s_2 exhibits a spike where $VP_1 = 10$, $PP = 25$, and $VP_2 = 35$. In both cases, the three points satisfy the local extrema predicates. The second step is to evaluate the threshold criteria of the spike features. We calculate the amplitude a_{s_1} and the width w_{s_1} of the spike in s_1 as:
$a_{s_1} = \max(\text{abs}\,(s_1(PP) - s_1(VP_1)), \text{abs}\,(s_1(PP) - s_1(VP_2)))$=
$\max(\text{abs}\,(s_1(20) - s_1(10)), \text{abs}\,(s_1(20) - s_1(30))) = \max(\text{abs}\,(2-1), \text{abs}\,(2-1)) = 1$
and $w_{s_1} = VP_2 - VP_1 = 30 - 10 = 20$. Signal s_1 satisfies property *pSPK1* because the expression $a_{s_1} \leq 1 \wedge w_{s_1} \leq 20 \equiv 1 \leq 1 \wedge 20 \leq 20$ evaluates to true. Following a similar computation, the amplitude a_{s_2} and the width w_{s_2} of the spike in s_2 are $a_{s_2} = \max(1.5, 1) = 1.5$ and $w_{s_2} = 25$; signal s_2 violates property *pSPK1* because the expression $a_{s_2} \leq 1 \wedge w_{s_2} \leq 20 \equiv 1.5 \leq 1 \wedge 25 \leq 20$ evaluates to false.

Another definition, proposed in the context of automotive control applications [KDJ+16], characterizes a spike using two parameters, w and $m = \frac{a}{w}$, where w is the spike width and a the spike amplitude. Formally, a signal s exhibits a spike with parameters m and w (defined as numerical constants) iff the following *SFO* formula evaluates to true:

$$\exists t \in I_s\colon s'(t) > m \wedge \exists t' \in [t, t + w]\colon s'(t') < -m \tag{4.2}$$

where s', denoting the first order derivative of s, can be either a pre-computed, separated signal s'_c or the punctual derivative s'_p introduced above. This characterization identifies two time instants: the first in which the signal derivative is greater than parameter m and another one in which the signal derivative is less than $-m$; the distance between these two points is the spike width w.

The main limitation of this formulation is that it does not allow to express precise constraints on the absolute value of the amplitude of a spike; instead, it uses parameter m that is a quotient between amplitude and width. We illustrate this with the example in figure 4.4, with the signals s_1 plotted with a thick line (—) and s_2 plotted with a thin line (–). Let us consider the evaluation of property *pSPK2*: "In a signal, there exists a spike with a maximum width of $20\,\text{tu}$ and an amplitude greater than 2". This property cannot be captured by an instance of specification (4.2), since the latter does not take into account the concept of amplitude; the property needs to be adapted. Based on the desired values of width and amplitude in property *pSPK2*,

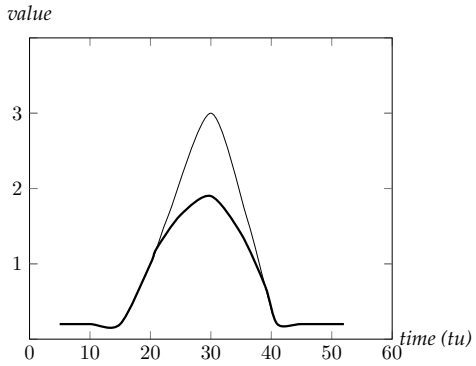


Figure 4.4: Characterization of the spike in two signals s_1 (—) and s_2 (—) based on the definition in [KDJ+16], with parameters $m = 0.1$, $w = 20$.

the parameters of an instance of specification (4.2) would be $m = 0.1$, $w = 20$. Therefore, instead of property *pSPK2*, one can consider the following alternative *pSPK3*: "In a signal, there exists a spike with a maximum width of $20\,\mathrm{tu}$ and parameter m equal to 0.1", which can be captured by an instance of specification (4.2); the corresponding *SFO* formula is:

$$\boxed{\text{SFO}}\ \textit{pSPK3} \quad \exists t \in I_s \colon s'(t) > 0.1 \wedge \exists t' \in [t, t+20] \colon s'(t') < -0.1$$

This formula will evaluate to true for both s_1 and s_2. However, signal s_1 should not satisfy the property, since its peak point does not reach a magnitude (amplitude) of 2 as was required in the original formulation of the property (*pSPK2*). This spurious spike characterization happens with specification (4.2) because signal s_1 follows the same shape as signal s_2 in the points in which the signal derivative s' is compared to m. We remark that the application of specification (4.1) to the evaluation of property *pSPK2* would correctly characterize the spike only in signal s_2. Given a lack of precision in specification (4.2), in the following we will consider spikes defined according to specification (4.1).

4.2.2.1 Alternative formalizations

STL Our characterization of a spike through the *SFO* formulation (4.1) relies on the existence of three extrema in the function corresponding to the signal shape. In *STL*, the existence of these extrema could be formalized through proper nesting of the "eventually" and "once" operators, in conjunction with a constraint on the width of the spike. However, it would not be possible to include in such a formulation a constraint on the amplitude or on the slope, since in *STL* one cannot refer to the value of the signal at an arbitrary time point. For all these reasons, we cannot express a property like *pSPK1* in *STL*.

On the other hand, spike properties characterized through the *SFO* formulation (4.2) can be expressed in *STL* when the pre-computed signal derivatives are available. For example, property *pSPK3* can be expressed as

$$\boxed{\text{STL}}\ \textit{pSPK3} \quad \mathsf{F}_{[0,|s|)}(s' > 0.1 \wedge \mathsf{F}_{[0,20]} s' < -0.1)$$

STL* Differently from *STL*, *STL** can refer to the value of the signal at a certain time point in which a local formula holds thanks to the freeze operator; below we discuss how it can be used to express properties *pSPK1* and *pSPK3*.

(Using local extrema expressed through punctual derivatives) Definition 1 for local extrema uses the values of the signal at two consecutive time points, within a small distance ϵ. However, in *STL** it is not possible to explicitly reference the signal value at time points that are not associated with the evaluation of a local (sub-)formula; hence, properties defined using punctual derivatives cannot be specified using *STL**[6].

(Using local extrema expressed through the analytical formulation) We can characterize local extrema using the analytical formulation (definition 2) by assuming a variant of *STL** with past operators[7] and using a 3D frozen time vector.

STL* *pSPK1*

$$\begin{aligned}
&\mathsf{F}_{[f,g]} *_1 (\mathsf{G}_{[0,w_1]}(s > s^{*_1}) \\
&\wedge \mathsf{F}_{[0,w_1]} *_2 (\mathsf{H}_{[0,w_1]}(s < s^{*_2}) \\
&\wedge \mathsf{F}_{[0,w_2]} *_3 (\mathsf{H}_{[0,w_2]}(s > s^{*_3}) \\
&\wedge \max(\mathrm{abs}(s^{*_1} - s^{*_2}), \mathrm{abs}(s^{*_2} - s^{*_3})) \leq 1 \wedge w_1 + w_2 \leq 20)))
\end{aligned}$$

In the formula above, the expression in the first row states the existence of the first local minimum by checking for the existence, within the observation interval $[f, g]$, of a point (whose time instant is frozen in the first component of the frozen time vector) for which the corresponding signal value is smaller than all other signal values in the interval $[0, w_1]$; this condition is captured by the sub-formula with the "globally" operator. The expression on the second row, nesting the "historically" operator within the "eventually", states the existence of the local maximum (whose time instant is frozen in the second component of the frozen time vector), such that all the signal values between the first local minimum and such a point are indeed smaller than the local maximum. Notice that the distance between the first local minimum and the local maximum is equal to w_1[8]. The expression on the third row checks in a similar way for the existence of the second local minimum within an interval $[0, w_2]$ from the local maximum. The expression on the fourth row checks the constraints on the spike amplitude and on the spike width. For the former, it uses the values of the signal in correspondence of the first local minimum (s^{*_1}), of the local maximum (s^{*_2}), and of the second local minimum (s^{*_3}).

Note that this property relies on a particular sequence of local extrema (i.e., valley-peak-valley); other variants of this property can be specified by changing the order of the sub-

[6]Such a restriction could be lifted when using discrete signals, since the distance between two consecutive time points is known and is equal to the sampling interval Δ.

[7]Although the version of *STL** presented in [BDŠV14] does not use past operators, the addition of such operators would be done along the lines of the definition of *STL* with past operators in [MN13].

[8]If the spike shape is symmetrical, the distance between all local extrema is equal to $\frac{w}{2}$.

formulae stating the existence of a certain extremum. Furthermore, we remark that the specification of this property assumes the knowledge of the signal shape, since it uses the two components of the width w_1 and w_2 as defined on page 36. However, making such an assumption in practice is not reasonable because typically the shape of a spike is unknown.

(Using local extrema defined through pre-computed derivatives) Property *pSPK1* can be expressed using definition 3 for local extrema, assuming the existence of signals s' and s'' and a 3D frozen time vector.

STL* *pSPK1*

$$\mathsf{F}_{[f,g]} *_1 \big(s' = 0 \wedge s'' > 0 \wedge \mathsf{F}_{[0,w_1]} *_2 (s' = 0 \wedge s'' < 0 \wedge \mathsf{F}_{[0,w_2]} *_3 (s' = 0 \wedge s'' > 0 \wedge \max(\mathrm{abs}(s^{*_1} - s^{*_2}), \mathrm{abs}(s^{*_2} - s^{*_3})) \leq 1 \wedge w_1 + w_2 \leq 20))\big)$$

The structure of the formula above is similar to the one for the case of using definition 2 for local extrema, except for the direct use of the first and second order derivatives, available as pre-computed signals. The same remarks made above in terms of assuming the knowledge of the signal shape also apply in this case.

Furthermore, pre-computed derivative signals can be used to specify property *pSPK3* in *STL** in the same way as it was done above using *STL*.

4.2.3 Oscillation

An oscillation can be informally described as a repeated variation over time of the value of a signal, possibly with respect to a reference value; often, in the context of CPS, oscillations represent an undesirable signal behavior.

Figure 4.5 depicts an analog signal s exhibiting an oscillatory behavior with respect to a reference value ref, within an observation interval $oscI = [a, b] \subset I_s$. Such a behavior is characterized by the existence, within the observation interval, of M extrema of the function corresponding to the signal shape; these points are marked with blue squares (□) in the figure. A *cycle* (i.e., a *complete oscillation*) occurs when the signal value swings from one extremum to the adjacent extremum of the same type, by traversing an extremum of the other type; for example, in the figure there is one complete oscillation when the signal goes from p_3 to p_5 (two peak points) through p_4 (a valley point). The figure also shows two additional features typically used to characterize oscillations:

- the *(peak) amplitude*, denoted by $oscA$, is the distance between the maximum magnitude of the signal and its reference value;
- the *period*, denoted by $oscP$, is the time required to complete one cycle. Its reciprocal, called *frequency*, represents the number of complete oscillations occurring in a unit of time.

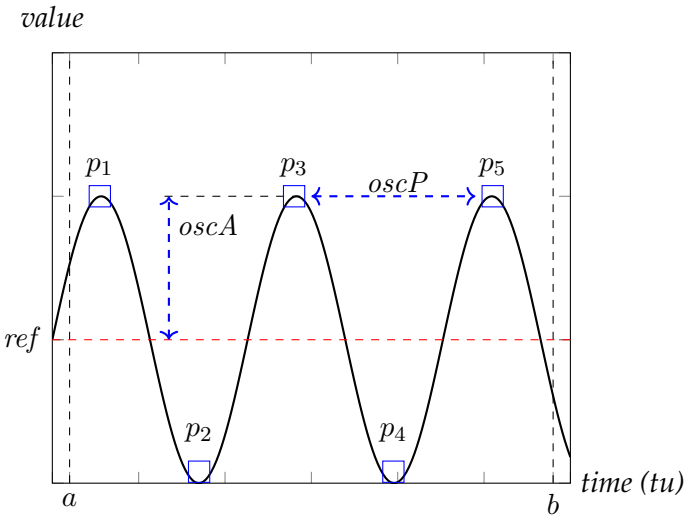


Figure 4.5: A signal exhibiting an oscillatory behavior; the reference value *ref* is shown in red.

An oscillation property specifies a constraint on the existence, in a signal, of an oscillatory behavior with certain features; it evaluates to true when the signal exhibits an oscillatory behavior whose features satisfy certain criteria. More specifically, these criteria are expressed as relational expressions, on the oscillation amplitude and/or period, with an application-specific threshold. More formally, given the *SFO* terms representing the threshold criteria Γ_{oscP} (for the period) and Γ_{oscA} (for the amplitude), an oscillation property holds on a signal s in the observation interval $[a, b]$ iff the following *SFO* formula evaluates to true:

$$
\begin{aligned}
\forall t \in [a,b]\colon\ & (\exists t', t'' \in [t,b]\colon \\
& \mathit{local\_min}(t, a, t') \rightarrow \\
& (\mathit{local\_max}(t', t, b) \wedge \mathit{local\_min}(t'', t', b) \\
& \wedge \mathit{checkOsc}(t, t', t'', \bowtie_P, \Gamma_{oscP}, \bowtie_A, \Gamma_{oscA})) \\
& \wedge \mathit{local\_max}(t, a, t') \rightarrow \\
& (\mathit{local\_min}(t', t, b) \wedge \mathit{local\_max}(t'', t', b) \\
& \wedge \mathit{checkOsc}(t, t', t'', \bowtie_P, \Gamma_{oscP}, \bowtie_A, \Gamma_{oscA})))
\end{aligned}
\tag{4.3}
$$

where $\mathit{local\_min}(x, y, z)$ (respectively, $\mathit{local\_max}(x, y, z)$) is a predicate that returns true if the time point x is a local minimum (respectively, local maximum) with respect to the interval $[y, z]$ (see section 4.2.2); $\mathit{checkOsc}(t, t', t'', \bowtie_P, \Gamma_{oscP}, \bowtie_A, \Gamma_{oscA})$ is a predicate that returns whether the expression $oscA \bowtie_A \Gamma_{oscA} \wedge oscP \bowtie_P \Gamma_{oscP}$ evaluates to true for the oscillation (with amplitude $oscA$ and period $oscP$) determined by its first three arguments t, t', t''; $\bowtie_P$ and $\bowtie_A$ are relational operators in Rel of Σ.

In essence, formula (4.3) requires a) the existence of the three local extrema in a proper order characterizing the complete oscillation (i.e., either a local minimum followed by a local maxi-

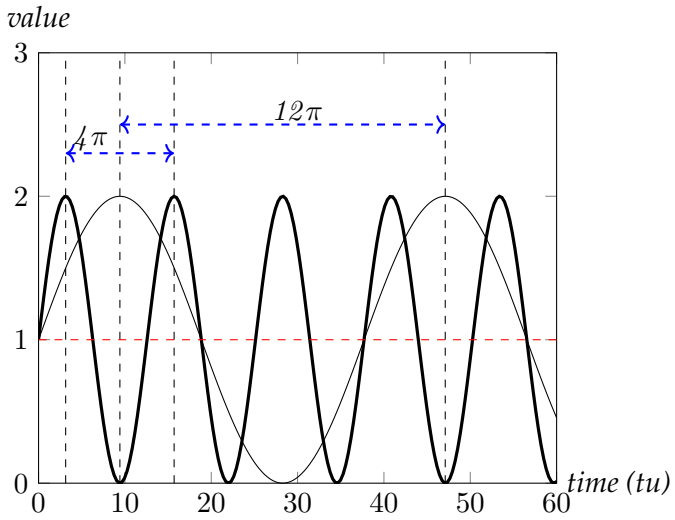


Figure 4.6: Two signals used to evaluate property *pOSC*: signal s_1 (—) satisfies the property, whereas s_2 (—) violates it.

mum followed by another local minimum, or a local maximum followed by a local minimum followed by another local maximum), and b) the satisfaction of the constraints on the oscillation features evaluated in the *checkOsc* predicate.

As an example, let us consider property *pOSC*: "Within an observation interval of 60 time units (starting from the beginning of the signal), in the signal there exist oscillations with a period less than 20 and an amplitude less than 3". For this property the parameters of an instance of specification (4.3) are $a = 0, b = 60, \Gamma_{oscP} = 20$, $\Gamma_{oscA} = 3$, $\bowtie_A = \bowtie_P = <$. For evaluating the property, we show two signals in figure 4.6: s_1 (drawn with a thick line) corresponds to a sine wave defined as $y = \sin(\frac{x}{2}) + 1$; s_2 (drawn with a thin line) is defined by $y = \sin(\frac{x}{6}) + 1$. In both signals, oscillations have a peak amplitude equal to 1, which satisfies the constraint on the amplitude. The period of signal s_1, calculated from its sine definition, is equal to 4π; similarly, the period of s_2 is equal to 12π (see figure 4.6). Signal s_1 satisfies property *pOSC* because it oscillates by exhibiting alternating local minima and maxima, with a period and an amplitude satisfying the thresholds ($4\pi < \Gamma_{oscP}$ and $1 < \Gamma_{oscA}$). However, signal s_2 violates the property because its period is greater than the threshold value of 20 ($12\pi > \Gamma_{oscP}$).

The pure sine wave shown in Figure 4.5 is characterized by a constant period and by a constant amplitude. However, in the context of CPSs, signals may be noisy; this means that the amplitude and the period of their oscillatory behaviors may vary over time. Furthermore, a reference value may be unknown, making the computation of the oscillation amplitude challenging. In such cases one may use an aggregation function (e.g., average, maximum, minimum) over different amplitude values (e.g., peak-to-peak). In the following, we introduce the concepts of *average amplitude* and *average period*; these definitions can easily be adapted to take into account other aggregation functions.

To deal with situations in which the reference value is not known, we will consider the peak-to-peak amplitude, i.e., the difference between two adjacent extrema, denoted by $oscA_{PP}$. The *average peak-to-peak amplitude* $\overline{oscA_{PP}}$ can then be computed as the arithmetic mean of the peak-to-peak amplitude between adjacent extrema. More formally, given the sequence $p_1, \ldots, p_{M-1}, p_M$ of local extrema, $\overline{oscA_{PP}} = \frac{\sum_{i=1}^{M-1} \text{abs}\,(s(p_i) - s(p_{i+1}))}{M-1}$. Other definitions of amplitude (such as the root mean square) can be used too, depending on the application domain.

The *average period* can be defined as the arithmetic mean of the period of each complete oscillation of the signal, computed over pairs of extrema of the same type. More formally, given the sequence $p_1, \ldots, p_{M-1}, p_M$ of local extrema, we define the number $oscN$ of complete oscillations within the observation interval of the signal as $oscN = \lfloor \frac{M-1}{2} \rfloor$; the *average period* $\overline{oscP}$ is then defined as $\overline{oscP} = \frac{\sum_{i=1}^{oscN} \text{abs}\,(p_{2i-1} - p_{2i+1})}{oscN}$.

When the concepts of average amplitude and average period are used to characterize an oscillatory behavior, specification (4.3) has to be adapted accordingly; more precisely, predicate $checkOsc$ has to be redefined to consider the average amplitude $\overline{oscA_{PP}}$ and the average period $\overline{oscP}$.

Damped/Driven oscillations In the real world, oscillatory behaviors may be subject to various forces that reduce or increase their amplitude. More precisely, we distinguish between *damped* and *driven* oscillations: for the former the amplitude decays monotonically, whereas for the latter the amplitude increases monotonically.

The characterization of these specific behaviors can be done by constraining the change of the amplitude of the oscillatory signal. For example, given the sequence $p_1, \ldots, p_{M-1}, p_M$ of local extrema, we say that an oscillatory signal s (formalized according to specification (4.3)) exhibits damped oscillations iff the following *SFO* formula evaluates to *true*:

$$\forall j \in [1, M-2]\colon \text{abs}\,(s(p_j) - s(p_{j+1})) \geq \text{abs}\,(s(p_{j+1}) - s(p_{j+2})) \tag{4.4}$$

The case for driven oscillations is similar and can be obtained from the expression above by replacing the relational operator with its dual.

The amplitude of signals may not change monotonically; in such cases, statistical trends (e.g., a linear trend) in amplitude changes may be observed. We could account for statistical trends by specifying that, on average, the difference in amplitude tends to decrease/increase; such a constraint would then be included in the formula above.

4.2.3.1 Alternative formalizations

STL Similar to the case of spike properties (see section 4.2.2), our formalization in *SFO* of oscillation properties relies on the existence of local extrema in the signal. Converting such formalization to *STL* would rely on the use of properly nested "eventually" and "once" operators,

in conjunction with a constraint on the oscillation period. However, a constraint on the amplitude could not be expressed because in *STL* one cannot refer to the value of the signal at an arbitrary time point.

STL* The specification of oscillatory behaviors is one of the main motivations behind the definition of *STL**. Below, we discuss how to specify property *pOSC1* in *STL** using the three local extrema characterization approaches introduced in section 4.2.2.

(Using local extrema expressed through punctual derivatives) As discussed for the case of spike properties (see page 41), properties referring to local extrema expressed according to definition 1 cannot be specified using *STL** because they would require to explicitly reference the signal value at time points that are not associated with the evaluation of a local (sub-)formula.

(Using local extrema expressed through the analytical formulation) We can express local extrema using their analytical formulation (definition 2) by assuming a variant of *STL** with past operators. Property *pOSC* can be specified in the following way using a 3D frozen time vector:

STL* *pOSC*

$$
\begin{aligned}
&\mathsf{G}_{[a,b]}(\mathsf{F}_{[0,b]*_1}(\mathsf{G}_{[0,\frac{\Gamma_{oscP}}{2}]}(s > s^{*_1}) \rightarrow \\
&\quad \mathsf{F}_{[0,\frac{\Gamma_{oscP}}{2}]*_2}(\mathsf{H}_{[0,\frac{\Gamma_{oscP}}{2}]}(s < s^{*_2}) \\
&\quad \wedge \mathsf{F}_{[0,\frac{\Gamma_{oscP}}{2}]*_3}(\mathsf{H}_{[0,\frac{\Gamma_{oscP}}{2}]}(s > s^{*_3}) \\
&\quad \wedge \mathrm{abs}\,(s^{*1} - s^{*2}) < 3))) \\
&\quad \wedge \mathsf{F}_{[0,b]*_1}(\mathsf{G}_{[0,\frac{\Gamma_{oscP}}{2}]}(s < s^{*_1}) \rightarrow \\
&\quad \mathsf{F}_{[0,\frac{\Gamma_{oscP}}{2}]*_2}(\mathsf{H}_{[0,\frac{\Gamma_{oscP}}{2}]}(s > s^{*_2}) \\
&\quad \wedge \mathsf{F}_{[0,\frac{\Gamma_{oscP}}{2}]*_3}(\mathsf{H}_{[0,\frac{\Gamma_{oscP}}{2}]}(s < s^{*_3}) \\
&\quad \wedge \mathrm{abs}\,(s^{*1} - s^{*2}) < 3))))
\end{aligned}
$$

In the formula above, the expression on the first row prescribes the existence of the first local minimum, by checking all points within the observation interval $[a, b]$ for the existence of a point (whose time instant is frozen in the first component of the frozen time vector) for which the corresponding signal value is smaller than all other signal values in the interval $[0, \frac{\Gamma_{oscP}}{2}]$; this condition is captured by the sub-formula with the second "globally" operator. The expression on the second row, nesting the "historically" operator within the "eventually", states the presence of a local maximum (whose time instant is frozen in the second component of the frozen time vector), such that all the signal values between the first local minimum and such a point are indeed smaller than the local maximum. Notice that the distance between

two neighboring extrema for an oscillation with period Γ_{oscP} is equal to $\frac{\Gamma_{oscP}}{2}$. The expression on the third row checks for the existence of the second local minimum in a similar way; the expression on the fourth row checks the constraint on the peak-to-peak amplitude using the values of the signal in correspondence of the first local minimum and of the local maximum. The remaining part of the formula has the same structure and considers the dual case, in which the first extremum in the oscillatory behavior is a local maximum.

We remark that this specification assumes that the oscillation is regular, i.e., its period is constant and the constraint on the period is specified as "*oscP*=Γ_{oscP}". However, making such an assumption in practice is not reasonable because typically the shape of oscillations is unknown.

(Using local extrema defined through pre-computed derivatives) Property *pOSC* can be expressed using definition 3 for local extrema, assuming the existence of pre-computed derivatives as separate signals s'_c and s''_c and a 3D frozen time vector.

STL* *pOSC*

$$\begin{aligned}
\mathsf{G}_{[a,b]}(\mathsf{F}_{[0,b]*_1}((s' = 0 \wedge s'' > 0) \rightarrow \\
\mathsf{F}_{[0,\frac{\Gamma_{oscP}}{2}]*_2}((s' = 0 \wedge s'' < 0) \\
\wedge \mathsf{F}_{[0,\frac{\Gamma_{oscP}}{2}]*_3}((s' = 0 \wedge s'' > 0) \\
\wedge \mathrm{abs}\,(s^{*1} - s^{*2}) < 3))) \\
\wedge \mathsf{F}_{[0,b]*_1}((s' = 0 \wedge s'' < 0) \rightarrow \\
\mathsf{F}_{[0,\frac{\Gamma_{oscP}}{2}]*_2}((s' = 0 \wedge s'' > 0) \\
\wedge \mathsf{F}_{[0,\frac{\Gamma_{oscP}}{2}]*_3}((s' = 0 \wedge s'' < 0) \\
\wedge \mathrm{abs}\,(s^{*1} - s^{*2}) < 3))))
\end{aligned}$$

The structure of the formula above is similar to the one for the case of using definition 2 for local extrema, except for the direct use of the first and second order derivatives, available as pre-computed signals. The signal values frozen at the local extrema points are used to compute the peak-to-peak amplitude of the oscillations. The same remarks made above in terms of assuming the knowledge of the signal shape also apply in this case.

4.2.4 Relationship between signals

The property types illustrated in the previous sections deal with only one signal; in this section we present property types characterizing *relationships* between two (or more) signals. We consider two types of signal relationships:

- *functional*, based on the application of a signal transforming function;
- *order*, describing sequences of events/states related to signal behaviors.

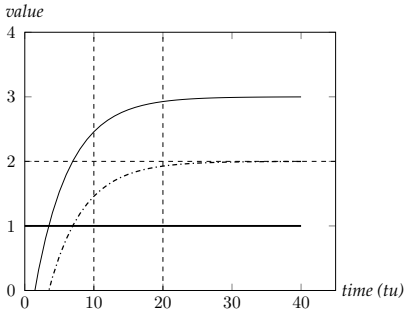


Figure 4.7: Signals used to evaluate property *pRSH-F*: the source signals are s_1 (—) and s_2 (-··), the target signal is s_T (—); Signal s_T satisfies the property.

4.2.4.1 Functional Relationship

The concept of a functional relationship between two (or more) signals is captured by the application of a signal transforming function to the signals, which yields a new signal based on the semantics of the function. Formally, let $\xi\colon \mathbb{D}_1 \times \mathbb{D}_2 \to \mathbb{D}_3$ (with $\xi \in Aux$) be an application-dependent signal transforming function[9] and let s_1 and s_2 be two signals (called *source* signals), with value domains $\mathbb{D}_1$ and $\mathbb{D}_2$ respectively, and domains of definition $I_{s_1} = I_{s_2} = I_s$; the application of ξ to s_1 and s_2 yields a *target* signal s_T over the value domain $\mathbb{D}_3$ defined as $s_T(t) = \xi\left(s_1(t), s_2(t)\right), \forall t \in I_s$. The target signal can then be referred to in the specification of other properties. More precisely, let P be an instance of one of the property types seen in the previous subsections (e.g., a data assertion), with ξ the signal transforming function defined above for the source signals s_1 and s_2. We say that property P holds on the signal representing the functional relationship between s_1 and s_2 captured by ξ iff P holds on the target signal s_T returned by the application of ξ.

For example, let us consider property *pRSH-F*:"The difference between the values of signal s_1 and signal s_2 shall be equal to 1", which contains two parts: a functional relationship part "The difference between the values of signal s_1 and signal s_2..." and a data assertion part "The [difference ...] shall be equal to 1". This property is expressed in *SFO* as follows:

$$\boxed{\text{SFO}\ \textit{pRSH-F}} \quad \forall t \in [0, |s|) : \mathrm{abs}(s_1(t) - s_2(t)) = 1 \tag{4.5}$$

Figure 4.7 shows the two source signals, s_1 plotted with a continuous line (—) and s_2 plotted with a dash-dotted line (-··), as well as the target signal s_T, plotted with a thick line (—). Signal s_T is obtained by the application of the signal transforming function ξ defined as $\xi((s_1(t), s_2(t)) \equiv \mathrm{abs}\left(s_1(t) - s_2(t)\right), \forall t \in I_s$. This signal is then used for the actual evaluation of the data assertion contained in property *pRSH-F*, as if the latter was rewritten as "The value of signal s_T shall be equal to 1"; since signal s_T is equal to 1 across its domain of definition, property *pRSH-F* evaluates to *true*.

[9] To keep the notation light and without loss of generality, we only consider a signal transforming function with arity 2.

Figure 4.8: (a) A signal being in the state characterized by property *pDAs* in the interval [*b*,*c*]. (b) A signal changing its value to 2 at time instant *c*, satisfying property *pDAe*.

4.2.4.2 Order Relationship

This type of signal relationships prescribes a sequence of events/states corresponding to signal behaviors; in practice, it captures the *precedence* and *response* temporal specification patterns proposed in the literature [DAC99], including their real-time extension [KC05]. More specifically, a precedence property specifies that an event/state (cause) *precedes* another event/state (effect); dually, a response property requires that an event/state (effect) *responds to* the occurrence of another event/state (cause). Notice that a response property allows effects to occur without causes, whereas a precedence property allows causes to occur without subsequent effects. Furthermore, in the context of real-time systems, both a precedence and a response property can include an additional constraint on the temporal distance between a cause and an effect.

When dealing with signals, the events/states used to express order relationships correspond to specific signal behaviors, which can be further expressed (and identified) using one of the property types seen above. More specifically, we define a *signal event* as a change in the signal value [CP99] occurring at a specific time instant, whereas a *signal state* is a signal behavior that holds over an interval delimited by two time boundaries or by the occurrence of two events. In the following, we discuss the concepts of signal events/states in the context of the property types described in the previous sections.

Data assertions The typical use of data assertions[10] is to represent signal states, as in property *pDAs*: "The signal value shall be greater than or equal to 2". For example, figure 4.8a shows a signal that satisfies this property in the interval [*b*,*c*].

Another formulation of this type of properties corresponds to signal events. As an example, let us consider property *pDAe*: "The signal value shall become equal to 2". Informally, this property corresponds to a predicate that captures the event of the signal *becoming* equal to 2,

[10]For simplicity, in the following we consider data assertion properties defined on one time interval.

i.e., changing from a value different from 2 to the actual value of 2. This behavior can be seen in the signal plotted in figure 4.8b: property *pDAe* holds at time instant c.

Notice that signal events can be used to characterize the boundaries of a signal state: for example, the time instants delimiting the interval in which the state represented by property *pDAs* holds correspond to the time instants in which the event represented by property *pDAe* and by its negation (i.e., "signal s becoming different from 2") occur.

Spike When a signal satisfies a spike property following the specification template (4.1) on page 37, the spike behavior of the signal can be associated with three different events, corresponding to the time instants in which the peak point and the two valley points of the spike shape (see section 4.2.2) occur. The actual choice of the most relevant event among these three is application-specific. Furthermore, the state induced by such a property type is defined over the interval $[VP_1, VP_2]$; such a state lasts for a duration corresponding to the spike width w.

Oscillation When a signal satisfies an oscillation property following the specification template (4.3) in section 4.2.3, the oscillatory behavior of the signal can be associated with distinct events, corresponding to the time instants in which the extrema points of the oscillations occur. The choice among these events is application-specific. Moreover, the state induced by such a property type is defined over the interval bounded by the first and last observed extrema of the oscillation.

Functional relationship between signals Similar to data assertions, functional relationship between signals can represent either signal events (captured by a predicate "*becomes*") or signal states.

Formalization After defining the concepts of events and states associated with signal property types, we are now ready to formalize the concept of order relationship between signal behaviors.

Given a signal s and an instance P of one of the signal property types described above, we define the *signal event boolean projection* of P on s as the predicate $\sigma^{\mathbb{B}e}_{s,P}(t)$, which evaluates to true iff the event associated with the signal behavior specified in P occurs in signal s at time instant t; similarly, we define the *signal state boolean projection* of P on s as the predicate $\sigma^{\mathbb{B}s}_{s,P}(t)$, which evaluates to true iff the state associated with the signal behavior specified in P holds on signal s at time instant t.

Given two signals s_1 and s_2 with domains of definition $I_{s_1} = I_{s_2} = [0, r)$ and lengths $|s_1| = |s_2|$ denoted with $|s|$, and two signal-based properties P_1 and P_2, we say that the event captured by P_2 in s_2 *responds to* (following the "response" pattern in [DAC99]) the event captured by P_1 in s_1 iff the following *SFO* formula evaluates to *true*:

$$\forall t \in [0, |s|) \colon \uparrow \sigma^{\mathbb{B}e}_{s_1,P_1}(t) \rightarrow \Big(\exists k \in (t, |s|) \colon \uparrow \sigma^{\mathbb{B}e}_{s_2,P_2}(k)\Big) \tag{4.6}$$

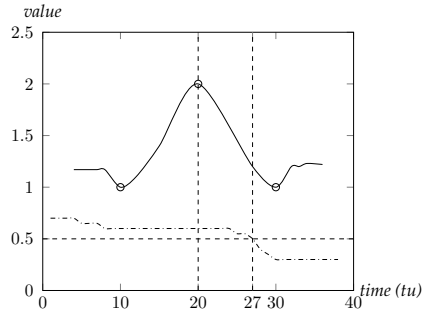


Figure 4.9: Signals s_1 (—) and s_2 (-··) used to evaluate property *pRSH-O*; the property holds.

where $\uparrow$ denotes the rising edge operator, defined as $\uparrow s(t) \equiv s(t) = 1 \wedge \exists c \in (0,t) : \forall c' \in (0,c) : s(t-c') = 0$.

If the relevant behavior captured by a property results in a state instead of an event, the formula above becomes:

$$\forall t \in [0,|s|) \colon \sigma^{\mathbb{B}s}_{s_1,P_1}(t) \rightarrow \Big(\exists k \in (t,|s|) \colon \sigma^{\mathbb{B}s}_{s_2,P_2}(k)\Big) \tag{4.7}$$

Similarly, we say that the event captured by P_1 in s_1 *precedes* (following the "precedence" pattern in [DAC99]) the event captured by P_2 in s_2 iff the following formula evaluates to *true*:

$$\forall t \in [0,|s|) \colon \uparrow \sigma^{\mathbb{B}e}_{s_2,P_2}(t) \rightarrow \Big(\exists k \in [0,t) \colon \uparrow \sigma^{\mathbb{B}e}_{s_1,P_1}(k)\Big) \tag{4.8}$$

When the relevant behavior captured by a property results in a state instead of an event, the formula above becomes:

$$\forall t \in [0,|s|) \colon \sigma^{\mathbb{B}s}_{s_2,P_2}(t) \rightarrow \Big(\exists k \in [0,t) \colon \sigma^{\mathbb{B}s}_{s_1,P_1}(k)\Big) \tag{4.9}$$

In some cases, an order relationship may prescribe a temporal distance between the cause and the effect. We assume this distance to be specified as a bound of the form $\bowtie\ n$, where $\bowtie\ \in Rel$ and $n \in \mathbb{R}$. In this case the formulae above have to be extended to take the distance into account, by conjoining the clause $\mathrm{abs}\,(k-t) \bowtie n$ to the consequent. For example, formula (4.6) will become:

$$\forall t \in [0,|s|) \colon \uparrow \sigma^{\mathbb{B}e}_{s_1,P_1}(t) \rightarrow \Big(\exists k \in (t,|s|) \colon \uparrow \sigma^{\mathbb{B}e}_{s_2,P_2}(k) \wedge \mathrm{abs}\,(k-t) \bowtie n\Big) \tag{4.10}$$

Notice that when one property induces a state and the other induces an event, the resulting formula for the corresponding order relationship is obtained by opportunely combining the occurrences of the signal boolean projection functions for states and events, following one of the above templates.

Order relationship properties can be defined recursively, i.e., when the cause and/or effect sub-property is also an order relationship. In these cases, we consider an event-based interpretation of the cause/effect sub-property.

As an example of order relationship property, let us consider the following response property *pRSH-O*: "If in signal s_1 there is a spike with a maximum width of $30\,\mathrm{tu}$ and a maximum amplitude of 1, then—within $10\,\mathrm{tu}$—the value of signal s_2 shall become less than 0.5". Assuming we use an event-based interpretation of both cause and effect sub-properties, we can rewrite the property as *pRSH-O*′ : "If there is an event corresponding to [signal s_1 having a spike with a maximum width of $30\,\mathrm{tu}$ and a maximum amplitude of 1] then—within $10\,\mathrm{tu}$—there shall be an event corresponding to [signal s_2 becoming less than 0.5]". In this instance of the response pattern, the cause is represented by the spike property "In signal s_1 there is a spike with a maximum width of $30\,\mathrm{tu}$ and a maximum amplitude of 1", whereas the effect is represented by the data assertion property "Signal s_2 shall become less than 0.5"; furthermore, the temporal distance between the cause and the effect can be at most $10\,\mathrm{tu}$. We refer to the cause and effect sub-properties as P_1 and P_2, respectively.

The specification of property *pRSH-O* in *SFO* is the following:

$$\text{SFO } \textit{pRSH-O} \quad \forall t \in [0, |s_1|) \colon \uparrow \sigma^{\mathbb{B}e}_{s_1,P_1}(t) \rightarrow \Big(\exists k \in (t, |s_2|) \colon \uparrow \sigma^{\mathbb{B}e}_{s_2,P_2}(k) \wedge \mathrm{abs}\,(k-t) \leq 10\Big) \tag{4.11}$$

where $\sigma^{\mathbb{B}e}_{s_1,P_1}$ and $\sigma^{\mathbb{B}e}_{s_2,P_2}$ are the signal event boolean projection predicates.

We evaluate the property with respect to the two signals shown in figure 4.9, s_1 plotted with a continuous line (—) and s_2 plotted with a dash-dotted line (-··). In this example, we assume that the signal boolean projection predicate for spike properties (used for the evaluation of the cause sub-property) is defined such that it is true at the actual time instant at which the spike peak point occurs (i.e., $20\,\mathrm{tu}$). By looking at figure 4.9, we see that property *pRSH-O* holds on s_1 and s_2 because the event captured by the effect sub-property (the change of value of s_2 happening at time instant $27\,\mathrm{tu}$) responds to the occurrence of the event associated with the cause sub-property within the prescribed time bound (since $\mathrm{abs}\,(27\,\mathrm{tu} - 20\,\mathrm{tu}) = 7\,\mathrm{tu} < 10\,\mathrm{tu}$).

4.2.4.3 Transient Behaviors

We consider transient signal behaviors (i.e., behaviors of a signal when changing from the current value to its target value) as a special case of order relationship. This category includes *rise time* (and *fall time*) and *overshoot* (and *undershoot*) properties.

***Rise time* (*Fall time*)** We say that a signal exhibits a *rising* (dually, *falling*) behavior when its value increases (decreases) towards a target value. Informally speaking, a property on the *rise (fall) time* defines a constraint on the time by which the signal reaches the target value. More specifically, it defines a constraint on the temporal distance between two events: 1) a (generic) cause event, also called *trigger event*, that coincides with the signal starting to manifest a transient behavior; 2) an effect event that represents the signal reaching the target value.

Figure 4.10a depicts a signal exhibiting a rising behavior starting from time instant st. The signal rises monotonically from the value $s(st)$ and reaches the target value s_{target} at time instant

c; the time interval $[st, c]$ is called *rise interval*. The left bound of the rise interval, also called *trigger time*, corresponds to the time instant at which the trigger event occurs. The right bound of the rise interval corresponds to the occurrence of the effect event, in which signal s reaches the target value. The trigger time can also be expressed in terms of an absolute time reference value; in such a case, the trigger event is the event in which a special *clock* signal reaches a certain value.

A *rise time* property defines a constraint on the right bound of the rise interval. More formally, given two signals s_{tr} and s with domains of definition $I_{s_{tr}} = I_s = [0, r)$, let P_{tr} and P be two signal-based properties. Property P_{tr} captures the trigger event defined in terms of the behavior of s_{tr}; property P captures the event of s reaching the target value. A *rise time* property bounds the *rise time* of s by a threshold $RT \in \mathbb{N}$ (indicated by the end-user); such a property holds iff the following *SFO* formula evaluates to *true*:

$$\forall st \in [0, |s_{tr}|) \colon \uparrow \sigma^{\mathbb{B}e}_{s_{tr}, P_{tr}}(st) \rightarrow \Big(\exists k \in [st, st + RT] \colon \uparrow \sigma^{\mathbb{B}e}_{s,P}(k)\Big) \tag{4.12}$$

A stricter definition requiring signal s to rise (strictly) monotonically can be expressed by adding the conjunct $\forall j \in [st, st + k) \colon \forall j' \in (j, st + k] : s(j) < s(j')$ to the consequent in the formula above.

A *fall time* constraint can be expressed in a similar way, replacing the relational operators with their duals.

As an example, let us consider the *rise time* property *pRT*: "If signal s_{tr} becomes greater than 1, then signal s shall reach the target value of 2 within at most $8\,\mathrm{tu}$". The trigger event in this property is represented by the data assertion property P_{tr}: "The value of signal s_{tr} becomes greater than 1". The effect sub-property of this order relationship property can be specified with the data assertion property P: "The value of signal s shall become greater than 2". The constraint on the rise time is $8\,\mathrm{tu}$. Property *pRT* can be expressed in *SFO* as:

$$\boxed{\text{SFO}\ \textit{pRT}} \quad \forall st \in [0, |s_{tr}|) \colon \uparrow \sigma^{\mathbb{B}e}_{s_{tr}, P_{tr}}(st) \rightarrow \Big(\exists k \in [st, st + 8] \colon \uparrow \sigma^{\mathbb{B}e}_{s,P}(k)\Big) \tag{4.13}$$

We evaluate property *pRT* with respect to signal s on the two signals shown in Figure 4.10b: s_1 plotted with a thick line (—) and s_2 plotted with a thin line (—). In the figure, an arrow at timestamp $4\,\mathrm{tu}$ denotes the trigger time st corresponding to the trigger event captured by property P_{tr} for signal s_{tr} drawn with a dash-dotted line (-·-). The maximum allowed value for the right bound of the rise interval ($st + RT = 4 + 8 = 12\,\mathrm{tu}$) is indicated with a red, vertical dashed line. Signal s_1 satisfies the property because it reaches the target value (2) at time instant $9\,\mathrm{tu} < st + RT$. Signal s_2 violates the property because it does not reach the target value by time instant $st + RT = 12\,\mathrm{tu}$.

The variant *pRT-monot* of property *pRT* with a monotonicity constraint can be expressed in *SFO* as:

$$\begin{aligned} &\boxed{\text{SFO}\ \textit{pRT-monot}} \quad \forall st \in [0, |s_{tr}|) \colon \uparrow \sigma^{\mathbb{B}e}_{s_{tr}, P_{tr}}(st) \rightarrow \\ &\Big(\exists k \in [st, st + 8] \colon \uparrow \sigma^{\mathbb{B}e}_{s,P}(k) \wedge \forall j \in [st, st + k) \colon \forall j' \in (j, st + k] : s(j) < s(j')\Big) \end{aligned} \tag{4.14}$$

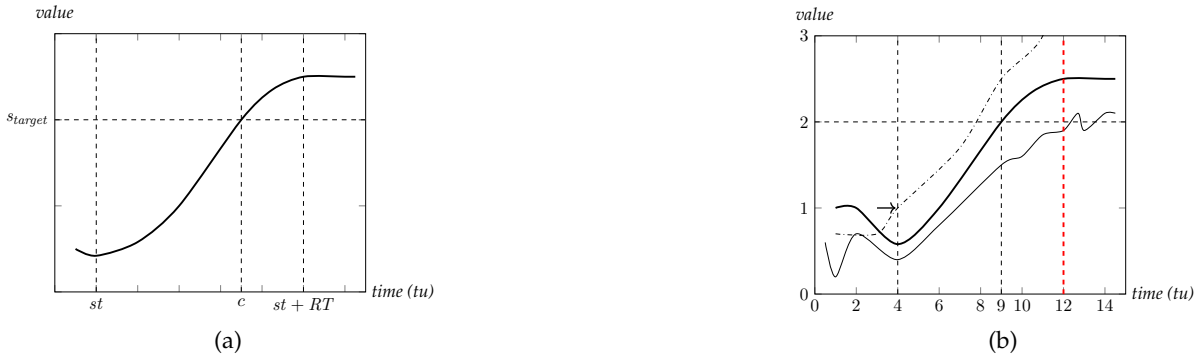


Figure 4.10: (a) Main concepts related to the specification of *rise time*. (b) two signals used to evaluate property *pRT*: signal s_1 (—) satisfies the property, whereas s_2 (—) violates it.

***Overshoot* (*Undershoot*)** We say that a signal exhibits an *overshoot* (dually, *undershoot*) behavior when it exceeds (goes below) its target value[11]. Informally speaking, an overshoot property specifies the maximum signal value, above the target value, that a signal can reach when overshooting within a certain time interval; an undershoot property is defined dually.

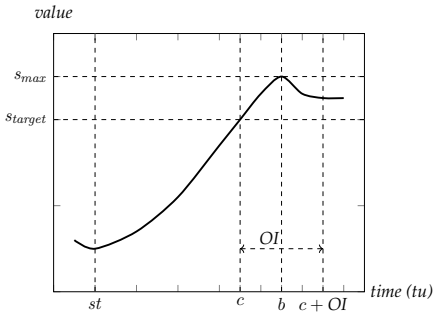
Figure 4.11a depicts a signal exhibiting an overshoot behavior starting from time instant st. This time instant is the *trigger time* and can be specified in different ways, as discussed above in the context of rise time properties. The signal rises from the value $s(st)$ and overshoots the target value s_{target} after time instant c, reaching the maximum magnitude s_{max} at time instant b. The time interval $[c, c + OI]$ is called *overshoot interval*; its width OI is specified by the end-user. This signal overshoots the target value s_{target} by an *overshoot value* $O_s = s_{max} - s_{target}$. An overshoot property defines a boundary on the overshoot value within the overshoot interval; such a boundary is expressed either with an absolute value or with a relative value with respect to the target value.

Similarly to the case of rise time specification, given two signals s_{tr} and s, let P_{tr} and P be two signal-based properties. Property P_{tr} captures the trigger event defined in terms of the behavior of s_{tr}; property P captures the event of signal s reaching the target value. An *overshoot* property bounds the *overshoot* of s by a threshold $OI \in \mathbb{N}$; such a property holds iff the following *SFO* formula evaluates to *true*:

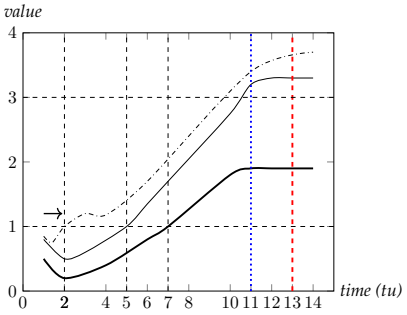
$$\begin{aligned} \forall st \in [0, |s_{tr}|) \colon \uparrow \sigma^{\mathbb{B}e}_{s_{tr},P_{tr}}(st) \rightarrow (\exists k \in [st, |s|) \colon \uparrow \sigma^{\mathbb{B}e}_{s,P}(k) \\ \wedge \forall i \in [k, k + OI] : s(i) \leq s_{max}) \end{aligned} \tag{4.15}$$

A monotonicity constraint can be added to the formula above in the same way as done for the case of rise time properties. An *undershoot* constraint can be expressed in a similar way, replacing the relational operators with their duals.

[11]Other definitions of overshoot also constrain the behavior of the signal after it exceeds (goes below) the target value, e.g., by requiring it to converge back to the target value.



(a)



(b)

Figure 4.11: (a) Main concepts related to the specification of *overshoot*. (b) two signals used to evaluate property *pOSH*: signal s_1 (—) satisfies the property, whereas s_2 (—) violates it.

As an example, let us consider property *pOSH*: "If signal s_{tr} becomes greater than 1, then signal s may overshoot the target value of 1 by at most 2 within an overshoot interval of at most $6\,\text{tu}$". As we did above for the *pRT* property, the trigger event in *pOSH* is represented by the data assertion property P_{tr}. The remaining part of the property represents the effect sub-property. The corresponding *SFO* formula is the following:

$$\begin{aligned} \text{SFO } \textit{pOSH} \quad & \forall st \in [0, |s_{tr}|)\colon \uparrow \sigma^{\mathbb{B}e}_{s_{tr},P_{tr}}(st) \rightarrow (\exists k \in [st, st + |s|)\colon \uparrow \sigma^{\mathbb{B}e}_{s,P}(k) \\ & \wedge \forall i \in [k, k + 6] : s(i) \leq 3) \end{aligned} \tag{4.16}$$

The variant of property *pOSH-monot* with a monotonicity constraint can be expressed in *SFO* as:

$$\begin{aligned} \text{SFO } \textit{pOSH-monot} \quad & \forall st \in [0, |s_{tr}|)\colon \uparrow \sigma^{\mathbb{B}e}_{s_{tr},P_{tr}}(st) \rightarrow (\exists k \in [st, st + |s|)\colon \uparrow \sigma^{\mathbb{B}e}_{s,P}(k) \\ & \wedge \forall i \in [k, k + 6] : s(i) \leq 3 \wedge \forall j \in [st, st + k)\colon \forall j' \in (j, st + k] : s(j) < s(j')) \end{aligned} \tag{4.17}$$

We evaluate property *pOSH* with respect to signal s on the two signals shown in figure 4.11b: s_1 plotted with a thick line (—) and s_2 plotted with a thin line (—). In the figure, an arrow at timestamp $2\,\text{tu}$ denotes the trigger time st corresponding to the trigger event captured by property P_{tr} for signal s_{tr}, drawn with a dash-dotted line (-·-). After this time instant, both s_1 and s_2 rise reaching the target value of 1 at time instants $7\,\text{tu}$ and $5\,\text{tu}$, respectively. We consider a threshold expressed as a relative value with respect to the target value; i.e., $s_{max} = s_{target} + 2 = 1 + 2 = 3$. The maximum allowed value for the right bound of the overshoot interval for s_1 ($7\,\text{tu} + OI = 7\,\text{tu} + 6\,\text{tu} = 13\,\text{tu}$) is indicated with a red, vertical dashed line. Similarly, in the case of s_2, the right bound for the overshoot interval ($5\,\text{tu} + OI = 5\,\text{tu} + 6\,\text{tu} = 11\,\text{tu}$) is drawn with a blue, dotted vertical line. Signal s_1 satisfies the property because its overshoot value is below the threshold within the overshoot interval $[7\,\text{tu}, 13\,\text{tu}]$; signal s_2 violates the property as its overshoot value exceeds the threshold within the overshoot interval $[5\,\text{tu}, 11\,\text{tu}]$.

4.2.4.4 Alternative formalizations

The capability of expressing functional relationship properties in *STL* and *STL** depends on the possibility, in the chosen language, of expressing a certain property type on the target signal resulting from the transforming function.

Similarly, expressing order relationship properties in *STL* and *STL** requires that the cause and effect sub-properties can be expressed in the chosen formalism. For example, the cause sub-property of property *pRSH-O* cannot be expressed in *STL*; however, it can be expressed in *STL** as explained in section 4.2.2 (page 41).

The same remarks made above for the general case of order relationships apply also to the case of rise time and overshoot properties. In addition, we remark that the specification of such properties containing a monotonicity constraint requires keeping track of the signal values seen throughout the rise/overshoot interval; this is not supported in *STL* but can be expressed in *STL** using the freeze operator.

4.3 Expressiveness

Another challenge in using Signal-based Temporal Properties for expressing requirements of CPSs is the expressiveness of the specification languages used for defining such properties. Starting from the seminal work on *STL*, there have been several proposals of languages that extend more traditional temporal logics like LTL to support the specification of signal-based behaviors. For example, in the previous section, we formally specified all property types included in our taxonomy using *SFO* and, when applicable, also using *STL* and *STL**. All these languages have different levels of expressiveness when it comes to describing certain signal behaviors.

In this section, we summarize and discuss the expressiveness of these state-of-the-art temporal logics *with respect to the property types included in our taxonomy*. We remark that we do not aim to provide a complete and formal treatment of the expressiveness of these temporal logics; our main goal is to guide engineers to choose a specification formalism based on their needs in terms of the property types to express.

Table 4.1 provides an overview of the expressiveness of *STL*, *STL**, and *SFO* with respect to the property types included in the taxonomy. The "$+$" and "$-$" symbols denote, respectively, support (or lack of support) for a certain property type; the "$\pm$" symbol indicates that the property type can be expressed under certain assumptions. Note that in the table, we also list property subtypes based on a particular feature. For example, "SPK with amplitude" indicates a spike property type (see figure 4.1 for the acronyms) with a constraint on the amplitude. In addition, we list as property subtypes (e.g., "SPK pre-computed derivatives") the three definitions to express the predicates for local extrema for spikes and oscillations (introduced in section 4.2.2, page 38). In the second column, we provide examples of properties corresponding to the property (sub)type indicated in the first column.

Table 4.1: Expressiveness of *STL*, *STL**, and *SFO* with respect to the property types included in the taxonomy in Fig. 4.1

Property Type	Example	Formalism		
		STL	*STL**	*SFO*
Data assertions (DA)	*pDA*	+	+	+
Spikes				
SPK with amplitude	*pSPK1*	−	+	+
SPK with slope	*n/a*	−	+	+
SPK with width	*pSPK1*	−	±	+
SPK - punctual derivatives		−	−	+
SPK analytical formulation		−	+	+
SPK pre-computed derivatives	*pSPK3*	+	+	+
Oscillations				
OSC with amplitude	*pOSC*	−	±	+
OSC with period	*pOSC*	±	±	+
OSC punctual derivatives		−	−	+
OSC analytical formulation		−	+	+
OSC pre-computed derivatives		+	+	+
Relationship between signals				
RSH-F	*pRSH-F*	±	±	+
RSH-O	*pRSH-O*	±	±	+
Transient Behaviors				
RT (FT) with monotonicity	*pRT-monot*	−	+	+
RT (FT)	*pRT*	+	+	+
OSH (USH) with monotonicity	*pOSH-monot*	−	+	+
OSH (USH)	*pOSH*	+	+	+

At a glance, the table shows that *SFO* can be used to express all the property types considered in this paper. *STL** can be used to express most of the property types included in our taxonomy, provided that some assumptions are made (see below). *STL* cannot be used to express all the property types; this is due to the lack of support for referring to signal values at an instant in which a certain property was satisfied. This limitation impacts on the specification of properties that constrain signal values at different time instants, such as spike and oscillation properties. In the following, we discuss the expressiveness for the various property types in details, mainly focusing on *STL* and *STL**.

Data assertion All three formalisms can express data assertion properties. This is expected since the three logics we have considered were proposed with the goal of expressing predicates on a signal value.

Spike A formalism supports our definition of spike properties if it allows for the definition of 1) two predicates for detecting local extrema, and 2) constraints on features of the signal shape (e.g., amplitude).

STL can be used to define the predicates for detecting local extrema only through definition 3 (as indicated with the "+" mark in the table), which assumes the availability of the first and second order derivatives of a signal. Furthermore, it cannot be used to express spike properties that constrain the spike amplitude or slope, since they refer to signal values at different points in the signal timeline. For example, the only spike property among those presented in the previous section that can be expressed in *STL* is *pSPK3*, because it uses pre-computed derivative signals and does not constrain the spike amplitude.

*STL** can be used to define the predicates for detecting local extrema using two of the definitions we propose (definition 2 - analytical formulation, and definition 3 - pre-computed derivatives). Furthermore, it can be used to express constraints on the different features of the signal shape. However, to do so, one has to assume the knowledge of the signal shape, since it uses the two components of the width w_1 and w_2 as defined on page 37. However, making such an assumption in practice is not reasonable because typically the shape of a spike is unknown. Finally, since *STL** (and *STL*) cannot refer to the value of the signal at arbitrary time points, properties defined using local extrema expressed according to definition 1 (punctual derivatives) cannot be specified.

Oscillation The expressiveness results in terms of oscillation properties mirror those for spike properties, since the former property type can be seen as an extension of the latter.

STL can be used to express oscillation properties when the oscillatory behavior is defined through the sequence of alternating local extrema, in which the latter are expressed using definition 3. However, as in the case of spike properties, *STL* cannot be used to express constraints on the oscillation amplitude.

Again, similarly to the case of spike properties, *STL** supports definition 2 and definition 3 for defining local extrema and can be used to express constraints on the different features of an oscillatory behavior. However, such formulations (including the one based on definition 3 for *STL*) require to assume that 1) the oscillation is regular; 2) its period is known a priori. These assumptions are required to express distance constraints between local extrema. Once again, in practice these assumptions are not realistic because typically the shape of an oscillatory behavior is unknown.

Relationship between signals Expressing functional relationship properties boils down to expressing a certain property type on the target signal resulting from the transforming function. The type of the property in which the target signal is used ultimately affects (e.g., in case of a spike property) the expressiveness for this type of properties. Furthermore, one has to consider whether the transformed (target) signal is available as a pre-computed signal or as function of other signals; in the latter case, only SFO supports function symbols.

A necessary requirement to express order relationship properties is the support for temporal operators that can capture the *precedence* and *response* temporal specification patterns [DAC99]. This is possible in *STL* and *STL** through the "*Until*" operator and in *SFO* by means of explicit quantification on the time variable. Another requirement is that the properties corresponding to the "cause" and "effect" of an order relationship can be expressed in the chosen formalism; as shown in Table 4.1, only SFO fulfills such a requirement.

Transient behaviors Transient behavior properties *without monotonicity constraints* can be expressed with all three formalisms, assuming the trigger property can be expressed in the chosen formalism. When a monotonicity constraint is used (as it is the case in properties *pRT-monot* and *pOSH-monot*), properties cannot be expressed in *STL* because one cannot compare the value of the signals at two different time instants.

Monitoring algorithms and tools When discussing the expressiveness of specification languages, it is also important to review the complexity of the corresponding verification algorithms and the availability of tools implementing them. Below we discuss the computational complexity of tools for *(offline) monitoring* of *STL*, *STL**, and *SFO* properties; we focus on monitoring because it is one of the most used V&V techniques for CPSs [BDD+18].

The complexity of monitoring *STL* is $O(k \cdot n)$ where k is the number of sub-formulae and n is the number of intervals on which the signal is defined [MN04]. For *STL**, the monitoring complexity is (similarly to *STL*) polynomial in the number of intervals on which the signal is defined and the size of the syntactic parse tree of the formula; however, it is exponential in the number of nested freeze operators in the formula [BDŠV14]. The monitoring complexity of *SFO* is $2^{(m+n)^{2^{O(k+l)}}}$, where n is the length of the trace, m is the length of the formula, k is the number of quantifiers in the formula, and l is the number of occurrences of function symbols in the formula; for a fragment of *SFO* in which intervals have bounded duration, the complexity is $n \cdot 2^{(m+j)^{2^{O(k+l)}}}$, where n, m, k, l are defined as above, and j is the maximum number of linear segments in the trace during any time period as long as the sum of the absolute values of all time constants in the formula [BFHN18]. In general, one can see that the complexity of the monitoring problem becomes harder for more expressive languages like *STL** and *SFO*.

In terms of monitoring tools, *STL* is supported both by *offline* tools—such as *AMT* [NLM+18, NM07] (a stand-alone GUI tool with qualitative semantics), *Breach* [Don10] and *S-Taliro* [FSUY12] (two *Matlab*® plugins with quantitative semantics)—and by *online* tools, such as the `rtamt` library [NY20], which automatically generates online monitors with robustness semantics from *STL* specifications.

For *STL**, a prototype implementation in *Matlab* is mentioned in the original paper [BDŠV14] but it has not been made available; furthermore, robustness analysis is supported by an extension of the *Parasim* tool [BVvF13]. No tool implementation is available for *SFO* at the time of writing this paper.

Recently, some of the authors have developed *SB-TemPsy* [BMB+20], a model-driven trace checking approach for the property types included in the taxonomy proposed in this chapter. *SB-TemPsy* includes *SB-TemPsy-DSL*, a domain-specific specification language for SBTPs, as well as the corresponding monitoring algorithm and tool, called *SBTemPsy-Check*. The complexity of the pattern-specific trace checking algorithm implemented in *SBTemPsy-Check* is polynomial in the size of the trace for all property types included in this taxonomy except for data assertions, for which the complexity is linear (in the size of the trace).

In conclusion, *with respect to the property types identified in our taxonomy*, *STL* has limited expressiveness, restricting its application in practice to simple property types (e.g., data assertion); nevertheless, it has a good support from a number of tools. *STL** is more expressive than *STL* provided that some assumptions (e.g., on the signal shape) are made; however, such assumptions are impractical. In addition, *STL** suffers from the limited tool support. *SFO* is the most expressive language for the property types defined in our taxonomy; however, its application in V&V activities is still challenging given the computational complexity of associated monitoring algorithms and the lack of tools.

4.4 Application to an Industrial Case Study

We applied our taxonomy of SBTPs to classify the requirements specifications of a case study provided by our industrial partner *LuxSpace Sàrl*[12], a system integrator of micro-satellites. Our goal is to show (1) the feasibility of expressing requirements specifications of a real-world CPS using the property types included in our taxonomy; (2) the completeness of our taxonomy, so that all requirements specifications of the case study can be defined using the property types included in our taxonomy.

The case study deals with a satellite sub-system called *Attitude Determination and Control System* (ADCS), which is responsible for autonomously controlling the attitude of the satellite, i.e., its orientation with respect to some reference point. The ADCS is mainly composed of sensors (e.g., gyroscope, sun sensors), actuators (e.g., reaction wheels, magnetic torquer), and on-board software (e.g., control algorithms). During flight, the ADCS can be in four different modes (represented with an enumeration as integer values), which determine the capabilities of the satellite: *idle* (IDLE), *Safe Mode* (SM), *Normal Mode Coarse* (NMC), and *Normal Mode Fine* (NMF); the logic controlling the switch among modes is encoded in a state machine. Overall, this sub-system has the typical characteristics of a CPS, with a deep intertwining of hardware and software.

The documentation of the ADCS includes 41 specifications written in English. Two of the authors carefully analyzed these specifications, discussed and (in some cases) refined them with a domain expert, and finally classified them using one of the property types in our taxonomy; the resulting classification was then validated by the domain expert. Table 4.2 shows the number of specifications classified for each property type (column "Total (Main)"); since properties

[12]https://luxspace.lu/

Table 4.2: Distribution of property types in the case study

Property Type	Total (Main)	Total (Sub)
Data assertion	7	49
Spike	1	1
Oscillation	1	0
Functional relationship	17	0
Order relationship	15	0
▷ Fall Time	0	1

Table 4.3: Data assertion properties in the case study

ID	Property
	Untimed Data Assertions
P1	The value of signal *currentADCSMode* shall be equal to *NMC*, *NMF* or *SM*
P2	The value of signal *pointing_error_above_20* shall be equal to 0 or 1
P3	The value of signal *pointing_error_under_15* shall be equal to 0 or 1
P4	The value of signal *RWs_angular_velocity* shall be equal to $816.814\,\mathrm{rad/s}$
	Time-Constrained Data Assertions
P5	Starting from $2000\,\mathrm{s}$, the value of signal *pointing_error* shall be less than $2°$
P6	Between $1500\,\mathrm{s}$ and $2000\,\mathrm{s}$, the value of signal *RWs_angular_momentum* shall be less than $0.35\,\mathrm{N}\cdot\mathrm{m}\cdot\mathrm{s}$
P7	At $2000\,\mathrm{s}$ the value of signal *pointing_error* shall be between $0°$ and $\delta°$

of type functional and order relationship include additional properties as sub-properties (e.g., the type of the "cause" or "effect" sub-property in an order relationship), we indicate their number separately under column "Total (Sub)". From the table we can conclude that *all requirements specifications of the case study could be classified using the property types included in our taxonomy*; this is an indication of the completeness of our taxonomy. In the following we provide some insights for each property type, derived from our classification exercise. We remark that the signal names used in the specifications correspond to the signals of a FES (Functional Engineering Simulator) in Matlab; when possible, we preserved the original signal name.

Table 4.4: Spike and oscillation properties in the case study

ID	Property
	Spike
P8	Between $2000\,\mathrm{s}$ and $7400\,\mathrm{s}$, in signal *pointing_error* there shall exist a spike with a maximum width of $20\,\mathrm{s}$
	Oscillation
P9	Between $2000\,\mathrm{s}$ and $7400\,\mathrm{s}$, signal *pointing_error* shall exhibit oscillations with a period greater than or equal to $0.01\,\mathrm{s}$

Data assertion properties (Table 4.3) This is the most represented category, if one considers the sub-properties included in the properties of type functional and order relationship. The three time-constrained data assertions show different interval types used in such properties. For example, in property P6 both boundaries of the interval are explicitly mentioned. In property P5, only the left boundary is explicitly indicated (with the expression "Starting from $2000\,\mathrm{s}$"), whereas the right boundary is implicit and is assumed to be the end of the (finite) signal. Finally, in property P7 the interval is singular (i.e., the two boundaries coincide) and corresponds to a single time point (as in the expression "At $2000\,\mathrm{s}$"). To express the latter using one of the logic-based formalizations illustrated above, which does not allow singular intervals (e.g., *STL*), one has to rewrite a singular interval $[a, a]$ as $[a - \epsilon, a + \epsilon]$, for a small $\epsilon > 0$.

We remark that time-constrained data assertions can be used to specify system-level properties such as system stabilization. For example, property P5 was originally expressed as "The stabilization time of signal *pointing_error*, when stabilizing below $2\,\mathrm{degrees}$, shall be under $2000\,\mathrm{s}$"; through the interaction with the domain expert, we further refined it into the version shown in Table 4.3. The refinement step was straightforward and consisted of rewriting the system-level property (i.e., stabilization) into a low-level one (of type "data assertion"), by expanding the definitions of domain concepts.

Spike and oscillation properties (Table 4.4) We identified one spike property (P8); furthermore an additional spike property is included in an order relationship property (P41). Both spike properties refer to one feature ("width").

We also identified one oscillation property (P9), which refers to the "period" feature. Initially, the property was defined in the frequency domain (which we did not discuss in this paper). After discussing it with the domain expert, we converted it into a property defined on the time domain by changing the corresponding constraint. This type of transformation is straightforward as it only requires to convert the units in the property (e.g., a $100\,\mathrm{Hz}$ frequency is converted into a $0.01\,\mathrm{s}$ period).

Table 4.5: Properties of type "functional relationship" in the case study

ID	Property	Subtype
P10	The modulus of signal *sat_init_angular_velocity_degree* shall be less than or equal to $3\,°/\mathrm{s}$	DA
P11	After $2000\,\mathrm{s}$, the modulus of signal *sat_real_angular_velocity* shall be less than or equal to $1.5\,°/\mathrm{s}$	DA
P12	The modulus of signal *sat_target_attitude* shall be equal to 1	DA
P13	After $2000\,\mathrm{s}$, the modulus of signal *sat_target_angular_velocity* shall be less than or equal to $1.5\,°/\mathrm{s}$	DA
P14	The modulus of signal *sat_estimated_attitude* shall be equal to 1	DA
P15	After $2000\,\mathrm{s}$, the modulus of signal *sat_estimated_angular_velocity* shall be less than or equal to $1.5\,°/\mathrm{s}$	DA
P16	The modulus of signal *sat_angular_velocity_measured* shall be less than or equal to $1.5\,°/\mathrm{s}$	DA
P17	The modulus of signal *earth_mag_field_in_body_measured* shall be less than or equal to $60\,000\,\mathrm{nT}$	DA
P18	The modulus of signal *sun_direction_ECI* shall be equal to 1	DA
P19	After $2000\,\mathrm{s}$, the modulus of signal *sat_target_angular_velocity_safe_spin_mode* shall be less than or equal to $1.5\,°/\mathrm{s}$	DA
P20	The modulus of signal *RWs_torque* shall be less than or equal to $0.015\,\mathrm{N}\cdot\mathrm{m}$	DA
P21	The elements sum of vector *sun_sensor_availability* elements shall be at most 3	DA
P22	At $2000\,\mathrm{s}$, the angular difference between signals *q_real* and *q_estimate_attitude* shall be between $0°$ and $\delta°$	DA
P23	At $2000\,\mathrm{s}$, the angular difference between signals *q_target_attitude* and *q_estimate* shall be between $0°$ and $\delta°$	DA
P24	The difference between signal *sat_estimated_angular_velocity* and signal *sat_real_angular_velocity* shall be between $0\,°/\mathrm{s}$ and $\delta°/\mathrm{s}$	DA
P25	The difference between signal *sat_angular_velocity_measured* and signal *sat_real_angular_velocity* shall be between $0\,°/\mathrm{s}$ and $\delta°/\mathrm{s}$	DA
P26	The difference between signal *RWs_torque* and the derivative of signal *RWs_angular_momentum* shall be equal to $0\,\mathrm{N}\cdot\mathrm{m}$	DA

Table 4.6: Properties of type "order relationship" in the case study

ID	Property	SubType
P27	If the value of signal *not_Eclipse* is equal to 0, then the value of signal *sun_currents* shall be equal to 0	DA-DA
P28	If the value of signal *pointing_error_under_15* is equal to 1, then the value of signal *pointing_error_above_20* shall be different from 1	DA-DA
P29	If the value of signal *pointing_error_above_20* is equal to 1, then the value of signal *pointing_error_under_15* shall be different from 1	DA-DA
P30	If the value of signal *RWs_command* is equal to 0, then the value of signal *RWs_angular_velocity* shall monotonically decrease to $0\,\mathrm{rad/s}$ within $60\,\mathrm{s}$	DA-FT
P31	If the value of signal *RWs_angular_momentum* is greater than $0.35\,\mathrm{N\cdot m\cdot s}$, then the value of signal *RWs_torque* shall be equal to $0\,\mathrm{N\cdot m}$	DA-DA
P32	If the value of signal *currentADCSMode* is equal to *NMC*, then the value of signal *control_error* shall be greater than or equal to $10°$	DA-DA
P33	If the value of signal *control_error* is less than $10°$, then the value of signal *currentADCSMode* shall be equal to *NMF*	DA-DA
P34	If the value of signal *currentADCSMode* is equal to *NMF*, then the value of signal *control_error* shall be less than or equal to $15°$	DA-DA
P35	If the value of signal *currentADCSMode* is equal to *NMF*, then if the value of signal *RWs_command* becomes greater than 0, then the value of signal *pointing_error* shall be less than $2°$ within $180\,\mathrm{s}$	DA-DA-DA
P36	If the value of signal *currentADCSMode* is equal to *NMF*, then if the value of signal *RWs_command* becomes greater than 0, then the value of signal *control_error* shall be less than $0.5°$ within $180\,\mathrm{s}$	DA-DA-DA
P37	If the value of signal *currentADCSMode* is equal to *NMF*, then if the value of signal *Not_eclipse* becomes 1, then the value of signal *knowledge_error* shall be less than 1 within at most $900\,\mathrm{s}$	DA-DA-DA
P38	If the value of signal *currentADCSMode* is equal to *SM*, then if the value of signal *RWs_command* becomes greater than 0, then the value of signal *RWs_angular_momentum* shall be less than $0.25\,\mathrm{N\cdot m\cdot s}$ within at most $900\,\mathrm{s}$	DA-DA-DA
P39	If the value of signal *currentADCSMode* is equal to *SM*, then the difference between signal *real_Omega* and signal *target_Omega* shall be equal to 0 within at most $10\,799\,\mathrm{s}$	DA-DA
P40	If the value of signal *not_Eclipse* is equal to 1, then the value of signal *sun_angle* shall be less than $45°$	DA-DA
P41	If, starting from $16\,200\,\mathrm{s}$, the value of signal *pointing_error* goes below the pointing accuracy threshold of $2°$, then in signal *pointing_error* there shall exist a spike with a maximum width of $600\,\mathrm{s}$ in an interval of $5400\,\mathrm{s}$	DA-SPK

All three properties include an observation interval. In properties P8 and P9, it is defined explicitly using absolute time boundaries (with the expression "between $2000\,\mathrm{s}$ and $7400\,\mathrm{s}$"). In property P41, the observation interval is defined through the event representing the left boundary (denoted with "the value of signal *pointing_error* after $16\,200\,\mathrm{s}$ goes below the pointing accuracy threshold of $2°$") and the duration ($5400\,\mathrm{s}$) representing the right boundary.

Functional relationship properties (Table 4.5) These properties were expressed using several signal transforming functions, such as modulus (P10–P20), vector elements sum (P21), angular difference (P22–P23), scalar difference (P24–P26), and differentiation (P26). Notice that property P26 contains nested applications of signal transforming functions (i.e., the second operand of the scalar difference is the result of the application of the derivative).

In all properties, the signal resulting from the application of the transforming function is used in a data assertion property (see column "Subtype" in Table 4.5[13]).

Order relationship properties (Table 4.6) All the order relationship properties we classified were instances of the "response" pattern (see section 4.2.4.2); we did not encounter any instance of the "precedence" pattern.

Some properties (P35–P38) contain nested properties of type "order relationship", meaning that the effect of the response pattern is represented by another property of type "order relationship". For example, in property P36, the top-level response property has "the value of signal *currentADCSMode* is equal to *NF*" as cause and "if the value of signal *RWs_command* becomes greater than 0, then the value of signal *pointing_error* shall be less than $2°$" as effect. The latter is another response property that can be further decomposed into the cause "the value of signal *RWs_command* becomes greater than 0" and the effect "the value of signal *pointing_error* shall be less than $2°$". The same group of properties also includes a temporal distance constraint (expressed with "within") as part of the nested response property.

As shown in column "Subtype" of Table 4.6, all the sub-properties used as "cause" and the vast majority of the sub-properties used as "effect" were data assertions. For example, in property P27 both the cause "the value of signal *not_Eclipse* is equal to 0" and the effect "the value of signal *sun_currents* shall be equal to 0" are data assertions. This is reflected in the third column of Table 4.6, with the notation "DA-DA".

Regarding transient behaviors, we only encountered one property of type "fall time", used as effect of the response property P30. Other types of properties (e.g., rise time, overshoot) were not present in this case study.

Summing up, through this case study we have shown the *feasibility* of expressing requirements specifications of a real-world CPS using the property types included in our taxonomy. In the vast majority of the cases, the mapping from a specification written in English to its corresponding property type defined in the taxonomy was straightforward. In two cases, the specifications had to be refined, either by expressing a system-level property into a low-level one (e.g.,

[13]See Figure 4.1 for the acronyms used in column "Subtype" of Tables 4.5 and 4.6.

stabilization being expressed as a (time-constrained) data assertion) or by converting a property defined in the frequency domain into the corresponding one defined in the time domain (e.g., in the case of an oscillation property); both types of refinement were simple and intuitive (with the help of a domain expert). Furthermore, the case study has shown the *completeness* of our taxonomy, since all requirements specifications of the case study could be classified using the property types included in our taxonomy.

Guided by the mapping to one of the property types included in our taxonomy, and by means of the formalization presented in section 4.2, an engineer can obtain a formal specification of a property (e.g, in *SFO*), which can then be used in the context of V&V activities (e.g., as test oracle).

Threats to validity The results regarding the *feasibility* of expressing requirements specifications of a real-world CPS and the *completeness* of our taxonomy, have been obtained through one large industrial case study, involving a domain expert; this is a threat to the generalization of the results. We tried to mitigate this threat by selecting a case study with a rich set of requirements extracted from the documentation of a complex, production-grade system. Such requirements are representative, in many ways, of those defined in the satellite and other cyber-physical domains. Nevertheless, some CPS domains (e.g., healthcare) may have specific types of requirements (e.g., supporting frequency-domain in the temporal specifications), which could lead to different results.

4.5 Applications

In this section, we discuss how the main contributions of the papers can support the research community and practitioners working in the CPS domain.

Application of the taxonomy The taxonomy of signal-based temporal properties can be used by researchers to *design new specification languages*, whose constructs can be directly mapped to the main property types identified in the taxonomy. This type of impact has been already observed for similar contributions in the literature, such as the seminal work of [DAC99] on temporal specification patterns, which has influenced the design of many domain-specific languages for temporal specifications (e.g., Temporal OCL [KT13], OCLR [DBB14], VISPEC - graphical formalism [HMF15], TemPsy [DBB17a], ProMoboBox - property language [MVDS20], FRETISH [GPMS20]), and the work on service provisioning patterns [BGPS12], which has led to the design of new specification languages and tools [BGS13, BBG+14, BGK14, BGKSP14, BBB19]. For instance, as mentioned in section 4.3, some of the authors have already developed *SB-TemPsy-DSL* [BMB+20], a domain-specific specification language for SBTPs based on the taxonomy proposed in this chapter.

The property types included in our taxonomy can also be used to *assess the expressiveness* of existing languages, in a way similar to what we have done in section 4.3. By doing so, re-

searchers can identify expressiveness gaps in existing languages, which could then be extended to support specific constructs. For instance, the motivating example for the development of *STL** [BDŠV14] was the impossibility of expressing oscillatory behaviors in *STL*.

Furthermore, practitioners can use the taxonomy as a reference guide to systematically *identify and characterize signal behaviors*, so that the latter can be defined precisely and used correctly during the development process of CPSs (e.g., when defining system requirements or test oracles).

Application of the logic-based characterization Researchers can leverage the logic-based characterization of the property types included in our taxonomy to *define the formal semantics* of the constructs of a new language, which has been inspired by the taxonomy itself. In this sense, the logic-based characterization can *guide the implementation* of the core, pattern-specific algorithms of a verification tool, which can be used for checking properties expressed in a language containing constructs derived from the property types included in our taxonomy.

For instance, the formal semantics of the aforementioned *SB-TemPsy-DSL* language and the corresponding trace checking algorithm implemented in *SBTemPsy-Check* [BMB+20] have been developed based on the logic-based characterization introduced in this chapter.

Expressiveness results The expressiveness results of state-of-the-art temporal logics with respect to the property types included in our taxonomy, presented in section 4.3, can be used by practitioners to carefully *select the language to use* for defining SBTPs, based on the type of requirements they are going to define, the expressiveness of the candidate specification language(s), and the availability of suitable tools.

4.6 Related Work

To the best of our knowledge, this is the first work that presents a comprehensive taxonomy of Signal-based Temporal Properties describing signal behaviors in the CPS domain. The closest work is the taxonomy of automotive controller behaviors presented in [KJD+16], in which behaviors are captured in ST-Lib, a catalogue of formal requirements written in *STL*. Although the ST-Lib catalogue contains several types of Signal-based Temporal Properties (e.g., spike, overshoot, rise time), the treatment of some property types is limited (e.g., oscillatory behaviors are only discussed for the case of short-period behaviors, i.e., ringing). Furthermore, as we have shown in section 4.2.2, the formalization of spike properties proposed in [KJD+16] has some limitations. A specific type of Signal-based Temporal Properties (i.e., oscillations) is discussed in [BDŠV14] and used as a motivation for introducing *STL**.

Similarly to what we did in section 4.2, most of the papers dealing with the specification or verification of signal-based temporal properties also include examples of such properties written using a specific temporal logic. We systematically reviewed the example properties, used throughout this chapter, dealing with specification, verification, and monitoring of CPS,

cited in a recent survey on these topics [BDD+18]; we excluded papers using spatio-temporal and frequency domain properties since they are out of the scope of this work. Table 4.7 shows, for each of the reviewed papers, the property types (from our taxonomy) to which the examples included in this chapter correspond, as well as the temporal logic used for their specification; treatment or lack thereof of a property type is denoted by a "+" or "-" symbol, respectively. One can see that data assertion and relationship between signals are the most common property types covered in the literature, whereas transient behaviors (e.g., *rise time*, *overshoot*) properties are the least common; spike and oscillation properties have a similar coverage.

To summarize, we propose in this chapter the first comprehensive taxonomy of SBTPs, formalized in a consistent and precise manner, which accounts for all reported property types in the literature.

4.7 Summary

Requirements of cyber-physical systems are usually expressed using Signal-based Temporal Properties, which characterize the expected behaviors of input and output signals processed by sensors and actuators. Expressing such requirements is challenging because of the many ways to characterize a signal behavior (e.g., using certain features). To avoid ambiguous or inconsistent specifications, we argue that engineers need precise definitions of such features and proper guidelines for selecting the features most appropriate in a certain context. Furthermore, given the broad variation in expressiveness of the specification languages used for defining Signal-based Temporal Properties, our experience indicates that engineers need guidance for selecting the most appropriate specification language, based on the type of requirements they are going to define and the expressiveness of each language. To tackle these challenges, in this chapter we have presented a taxonomy of the most common types of Signal-based Temporal Properties, accompanied by a comprehensive and detailed description of signal-based behaviors and their precise characterization in terms of a temporal logic (*SFO*). Engineers can rely on such characterization to derive—from informal requirements specifications—formal specifications to be used in various V&V activities. Furthermore, we have reviewed the expressiveness of state-of-the-art signal-based temporal logics (i.e., *STL*, *STL**, *SFO*) in terms of the property types identified in the taxonomy, while also taking into account the complexity of monitoring algorithms and the availability of the corresponding tools. Our analysis indicates that *SFO is the most expressive language for the property types of our taxonomy*; however, the application of *SFO* in V&V activities is still challenging given the computational complexity of the corresponding monitoring algorithm and the lack of tools. We have also applied our taxonomy to classify the requirement specifications of an industrial case study in the aerospace domain. The case study has shown the feasibility of expressing requirements specifications of a real-world CPS using the property types included in our taxonomy, and has provided evidence of the completeness of our taxonomy.

Table 4.7: Coverage of property types (from our taxonomy, see figure 4.1 for acronyms) in example specifications from the literature.

Reference	Formalism	DA	SPK	RT (FT)	OSH (USH)	OSC	RSH
[MN04]	STL	+	-	-	-	-	+
[MN13]	STL/PSL	+	-	+	-	-	+
[KJD+16]	PSTL	+	+	+	+	-	+
[BDŠV14]	STL*	+	-	-	-	+	-
[AF10]	MTL	+	-	-	-	-	+
[AFS+13]	MTL	+	-	-	-	-	+
[ARB+17]	CTMTL	+	-	-	-	-	-
[ARB+17]	XCTL	+	-	-	-	-	+
[ARB+17]	CLTL	+	-	-	-	-	+
[BGK+13]	STL	+	+	-	-	-	+
[AH15]	STL	+	-	-	-	-	+
[AH15]	AVSTL	+	-	-	-	-	+
[BBN13]	STL	+	-	-	-	-	+
[BBNS15]	STL	+	+	-	-	+	+
[BCMT09]	KSL	+	-	-	-	-	+
[BMS15]	MITL	+	-	-	-	-	-
[BBS+14]	MITL	+	-	-	-	-	-
[DDG+17]	STL	+	-	-	-	+	+
[DDG+15]	STL	+	-	-	-	+	+
[DZS+15]	MTL	+	-	-	-	-	-
[DM10]	STL	+	-	-	-	-	+
[DDD+15]	STL	+	-	-	-	-	-
[Fer16]	TRE	+	+	-	-	-	+
[HMF15]	STL	+	-	-	-	-	+
[JBGN16]	STL	+	-	-	-	-	+
[JDJS14]	STL	+	-	-	-	-	+
[JDJS14]	PSTL	+	-	-	-	-	-
[CFMS15]	MTL	+	-	-	-	-	+
[NSF+10]	MTL	+	-	-	-	-	+
[DHF14]	MTL	+	-	-	-	-	+
[DHF15]	MITL	+	-	-	-	+	+
[DHF15]	STL	+	-	-	-	-	+
[NN16]	STL	+	+	-	-	-	+
[JDDS15]	STL	+	+	-	-	-	+
[JDDS15]	PSTL	+	+	-	+	-	+

Table 4.7: (continued)

Reference	Formalism	DA	SPK	RT (FT)	OSH (USH)	OSC	RSH
[Don10]	MITL	+	-	-	-	-	+
[DFG+11]	STL	+	-	-	-	+	+
[EF07]	PSL	+	-	-	-	-	+
[FP06]	MTL	+	-	-	-	-	+
[FP06]	MITL	+	-	-	-	-	-
[FP06]	MTL	+	-	-	-	-	+
[FSUY12]	MTL	+	-	-	-	-	+
[FMNU15]	TRE	+	+	-	-	-	-
[HBA+14]	PMTL	+	-	-	-	-	-
[HBA+14]	MTL	+	-	-	-	-	+
[HDF18]	MTL	+	-	-	-	-	+
[HDF18]	PMTL	+	-	-	-	-	+
[JBG+15]	STL	+	-	-	-	+	+
[Kan15]	BMTL	+	-	-	-	-	+
[MNP08]	STL	+	-	-	-	-	+
[Nic08]	MITL	+	-	-	-	-	+
[Nic08]	STL	+	-	-	-	-	+
[Nic08]	STL/PSL	+	-	+	-	-	+
[Nic08]	MTL-B	+	-	-	-	-	+
[NM07]	STL/PSL	+	-	-	-	-	+
[PMS+14]	CTL	+	-	-	-	-	+
[RBFS08]	LTL(R)	+	-	-	-	+	-
[RBFS08]	QFLTL(R)	+	-	-	-	+	-
[SF12]	MTL	+	-	-	-	-	+
[SJN+17]	STL	+	+	+	-	-	-
[SJN+17]	TRE	+	+	+	-	-	-
[SDB+13]	STL	+	-	-	-	-	-
[UFAM14]	TRE	+	-	-	-	+	+
[YHF12]	PMTL	+	-	-	-	-	-
Total		64	10	5	2	10	48

Chapter 5

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

5.1 Overview

In this chapter 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 [FKRT18]:

- 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 [LVR19], QTL [HPU17]) 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 [SACO02], Structured English Grammar for real-time specifications [KC05], Temporal OCL [KT13], OCLR [DBB14], VISPEC - graphical formalism [HMF15], TemPsy [DBB17a], TemPsy-AG [BBB19], ProMoboBox - property language [MVDS20], FRETISH [GPMS20]) 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 [AGL+15]. 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 [CANZ19, CZss] 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 chapter, we consider the problem of trace-checking SBTPs (see 4.2 for more details). 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 [MN04], STL* [BDŠV14], SFO [BFHN18], RFOL [MNGB19]) and domain-specific [BOV+19] languages that have been proposed in the literature to express SBTPs, such languages either do not support the specification of important types of properties [BJB+20], or are not supported by (efficient) trace-checking procedures [BDŠV14, BFHN18].

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 [BJB+20]. 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 [DBB17a] for the verification of temporal properties based on [DAC99]'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 [OMG12] and, as a result, is supported by a mature constraint checking technology, such as the constraint checker included in Eclipse OCL [Ecl20]. Based on these observations and the encouraging efficiency results reported in the literature for model-driven trace checking approaches [DBB17a, BBB19], 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 chapter is organized as follows. Section 5.2 presents our case study in the satellite domain and discusses the motivations for this work. Section 5.3 provides a detailed introduction of the traces and the corresponding notations used in the rest of the chapter. Section 5.4 provides an overview of SB-TemPsy, further detailed in section 5.5 (which presents SB-TemPsy-DSL) and in section 5.6 (which presents our model-driven trace checking procedure implemented in SB-TemPsy-Check). Section 5.7 reports on the empirical investigation of our contributions. Section 5.8 discusses related work.

5.2 Case Study and Motivations

LuxSpace, our industrial partner, has developed, in collaboration with ESA [ESA20b] and ExactEarth [exa20], 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 [ESA20a])[1], which includes several activities, such as the definition of the system requirements and inter-

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

faces, 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 tele-command* (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 [MNGB19, MNBYI20].

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 [PPM+19] 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 4.2.

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.

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

[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.

and oscillations).

5.3 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 $f : T \rightarrow 2^{\mathcal{A}}$. 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 this chapter. 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.

5.4 The SB-TemPsy Approach

Model-driven trace checking [DBB17a] 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 chapter, 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 [OMG12] and, as a result, is supported by a mature constraint checking technology, such as the constraint checker included in Eclipse OCL [Ecl20]. Furthermore, existing approaches for model-driven trace checking (for checking linear temporal logic properties [DBB17a] and properties with temporal aggregations [BBB19]) 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 5.7.

Our SB-TemPsy approach is illustrated in figure 5.1; 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 5.3. The property ϕ represents a requirement of a CPS, expressed using one of the types of SBTPs, already defined in the previous chapter (see section 4.2).

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 [DBB17a]—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 [BJB+20] and presented in section 4.2.

SB-TemPsy-DSL differs from existing pattern-based specification languages for temporal properties (such as PROPEL - DNL [SACO02], Structured English Grammar for real-time specifications [KC05], Temporal OCL [KT13], OCLR [DBB14], VISPEC - graphical formalism [HMF15], TemPsy [DBB17a], TemPsy-AG [BBB19], ProMoboBox - property language [MVDS20]), since it is tailored to express SBTPs. Furthermore, differently from existing logic-based specification languages for SBTPs (e.g., STL [MN04], STL* [BDŠV14], RFOL [MNGB19], SFO [BFHN18]), 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 [BJB+20]), 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 5.3, 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 5.6.1.

Model-driven trace checking. This step (detailed in section 5.6.3) 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 $\bar{\pi}$.

5.5 The SB-TemPsy-DSL Language

5.5.1 Syntax

The syntax of SB-TemPsy-DSL is shown in figure 5.2; optional items are enclosed in square brackets; the symbol | separates alternatives. A *property* (non-terminal ϕ) is defined using a

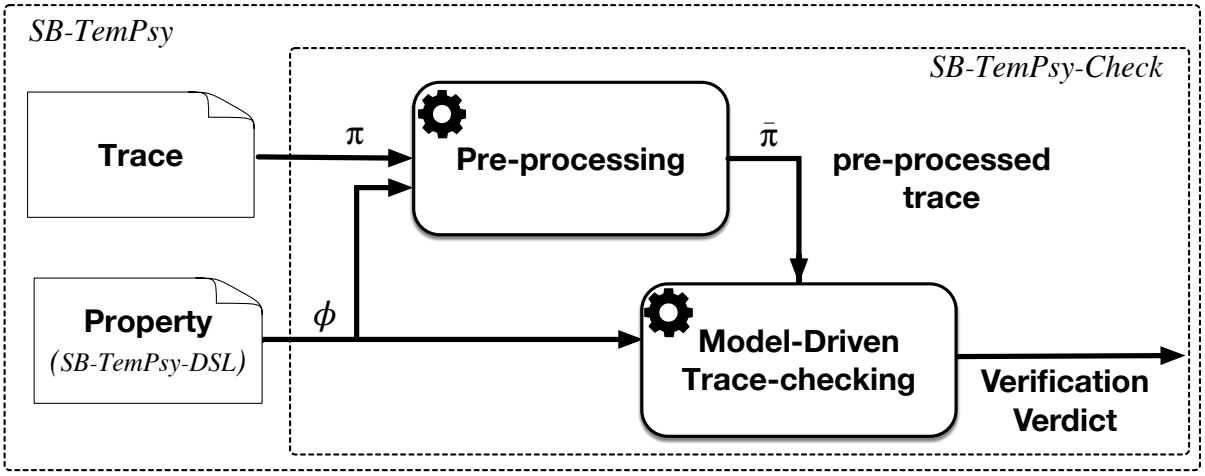


Figure 5.1: Overview of SB-TemPsy

scope (non-terminal $\langle sc \rangle$) or as a Boolean expression over other properties.

A scope operator constrains a *pattern* (non-terminal $\langle p \rangle$) to hold within a given time interval delimited either by absolute time instants (denoted by, $\langle t \rangle$, $\langle t_1 \rangle$, and $\langle t \rangle_2$) or by events, i.e., occurrences of a pattern, (denoted by $\langle p \rangle$, $\langle p \rangle_1$, and $\langle p \rangle_2$). For example, the "**between** $\langle t \rangle_1$ **and** $\langle t \rangle_2$ $\langle p \rangle$" scope operator specifies that pattern $\langle p \rangle$ holds between the time instants $\langle t \rangle_1$ and $\langle t \rangle_2$. The scope operators supported by SB-TemPsy-DSL are inspired by those proposed by [DAC99] (**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** $\langle s \rangle$ **with p2pAmp** $\langle v \rangle_1$ **period** $\langle v \rangle_2$" specifies that the value of signal $\langle s \rangle$ shows an oscillatory behavior with a peak-to-peak amplitude (denoted by the keyword **p2pAmp**) equal to value $\langle v \rangle_1$ and a period (denoted by the keyword **period**) that is equal to value $\langle v \rangle_2$; it corresponds to the property type "oscillation" discussed in section 4.2.

A *condition* (non-terminal $\langle c \rangle$) is a logical predicate on a signal $\langle s \rangle$ (of the form "$\langle s \rangle \sim \langle v \rangle$", with $\sim \in \{<, >, =, \neq, \leq, \geq\}$) or a logical expression over predicates.

Below we show how the example properties from our case study, presented in English in section 4.2, can be expressed in SB-TemPsy-DSL (for simplicity, we omitted the corresponding scopes):

Property	$\phi ::=$	ϕ_1 **and** ϕ_2 | ϕ_2 **or** ϕ_1 | **not** ϕ | $\langle sc \rangle$
Scope	$\langle sc \rangle ::=$	**globally** $\langle p \rangle$ | **before** $\langle t \rangle$ $\langle p \rangle$ | **after** $\langle t \rangle$ $\langle p \rangle$ | **at** $\langle t \rangle$ $\langle p \rangle$ | **before** $\langle p \rangle_1$ $\langle p \rangle$ **after** $\langle p \rangle_1$ $\langle p \rangle$ | **between** $\langle t \rangle_1$ **and** $\langle t \rangle_2$ $\langle p \rangle$ | **between** $\langle p \rangle_1$ **and** $\langle p \rangle_2$ $\langle p \rangle$
Pattern	$\langle p \rangle ::=$	**assert** $\langle c \rangle$ | $\langle s \rangle$ **becomes** $\sim \langle v \rangle$ | **if** $\langle p \rangle_1$ **then** [**within** $\bowtie \langle t \rangle$] $\langle p \rangle_2$ | **exists spike in** $\langle s \rangle$ [**with** [**width** $\sim \langle v \rangle_1$][**amplitude** $\sim \langle v \rangle_2$]] | **exist oscillations in** $\langle s \rangle$ [**with** [**p2pAmp** $\sim \langle v \rangle_1$][**period** $\sim \langle v \rangle_2$]] | $\langle s \rangle$ **rises** [**monotonically**] **reaching** $\langle v \rangle$ | $\langle s \rangle$ **falls** [**monotonically**] **reaching** $\langle v \rangle$ | $\langle s \rangle$ **overshoots** [**monotonically**] $\langle v \rangle_1$ **by** $\langle v \rangle_2$ | $\langle s \rangle$ **undershoots** [**monotonically**] $\langle v \rangle_1$ **by** $\langle v \rangle_2$
	$\bowtie ::=$	**exactly** | **at most** | **at least**
Condition	$\langle c \rangle ::=$	$\langle c \rangle_1$ **and** $\langle c \rangle_2$ | $\langle c \rangle_1$ **or** $\langle c \rangle_2$ | **not** $\langle c \rangle$ | $\langle s \rangle \sim \langle v \rangle$

$\langle t \rangle, \langle t \rangle_1, \langle t \rangle_2 \in \mathbb{R}$; $\langle v \rangle, \langle v \rangle_1, \langle v \rangle_2 \in \mathbb{R}$; $\sim \in \{<, >, =, \neq, \leq, \geq\}$;
$\langle s \rangle$ is a signal in S or a mathematical expression over the signals in S

Figure 5.2: SB-TemPsy-DSL syntax

```
pDA  assert (s1 >= -90 and s1 <= 90)
pSK  exists spike in s2 with amplitude < 90
pOS  exist oscillations in s3 with p2pAmp <= 8000
     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
```

5.5.2 Formal Semantics

Figure 5.3 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 $\bar{\pi}, t \models \langle c \rangle$ to indicate that condition $\langle c \rangle$ holds in the pre-processed trace $\bar{\pi}$ at time t. For example, the predicate $\langle s \rangle \sim \langle v \rangle$ holds at time t if the value $f_p(t, s)$ assigned to signal $\langle s \rangle$ at time t satisfies the predicate $f_p(t, s) \sim \langle v \rangle$.

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 $\bar{\pi}, [t_l, t_u] \models \langle p \rangle$ to indicate that pattern $\langle p \rangle$ 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 5.1. These predicates evaluate to true if, within a given time interval, the signal has an extremum

Property	$\bar{\pi} \models \langle sc \rangle \Leftrightarrow \bar{\pi}, [t_1, t_m] \models \langle sc \rangle$ $\bar{\pi} \models \phi_1$ **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**	$\bar{\pi}, t \models \langle s \rangle \sim \langle v \rangle \Leftrightarrow f_p(t, s) \sim \langle v \rangle$ $\bar{\pi}, t \models \langle c \rangle_1$ **and** $\langle c \rangle_2 \Leftrightarrow (\bar{\pi}, t \models \langle c \rangle_1)$ and $(\bar{\pi}, t \models \langle c \rangle_2)$ $\bar{\pi}, t \models \langle c \rangle_1$ **or** $\langle c \rangle_2 \Leftrightarrow (\bar{\pi}, t \models \langle c \rangle_1)$ or $(\bar{\pi}, t \models \langle c \rangle_2)$ $\bar{\pi}, t \models$ **not** $\langle c \rangle \Leftrightarrow (\bar{\pi}, t \not\models \langle c \rangle)$
Absolute Scope	$\bar{\pi}, [t_l, t_u] \models$ **globally** $\langle p \rangle \Leftrightarrow$ $\bar{\pi}, [t_l, t_u] \models \langle p \rangle$ $\bar{\pi}, [t_l, t_u] \models$ **before** $\langle t \rangle$ $\langle p \rangle \Leftrightarrow$ $t_l \leq \langle t \rangle \leq t_u$ and $\bar{\pi}, [t_l, \langle t \rangle] \models \langle p \rangle$ $\bar{\pi}, [t_l, t_u] \models$ **after** $\langle t \rangle$ $\langle p \rangle \Leftrightarrow$ $t_l \leq \langle t \rangle \leq t_u$ and $\bar{\pi}, [\langle t \rangle, t_u] \models \langle p \rangle$ $\bar{\pi}, [t_l, t_u] \models$ **between** $\langle n \rangle$ **and** $\langle m \rangle$ $\langle p \rangle \Leftrightarrow$ $t_l \leq \langle n \rangle < \langle m \rangle \leq t_u$ and $\bar{\pi}, [\langle n \rangle, \langle m \rangle] \models \langle p \rangle$ $\bar{\pi}, [t_l, t_u] \models$ **at** $\langle t \rangle$ $\langle p \rangle \Leftrightarrow$ $\exists t, t_l \leq \langle t \rangle \leq t_u$ and $\bar{\pi}, [\langle t \rangle, \langle t \rangle] \models \langle p \rangle$	**Event Scope**	$\bar{\pi}, [t_l, t_u] \models$ **before** $\langle p \rangle_1$ $\langle p \rangle \Leftrightarrow$ $\forall t_1, t_2, t_l < t_1 < t_2 \leq t_u, \bar{\pi}, [t_1, t_2] \models \langle p \rangle_1$ and $\exists t_3, t_4, t_l < t_3 < t_4 < t_1, \bar{\pi}, [t_3, t_4] \models \langle p \rangle$ $\bar{\pi}, [t_l, t_u] \models$ **after** $\langle p \rangle_1$ $\langle p \rangle \Leftrightarrow$ $\forall t_1, t_2, t_l < t_1 < t_2 \leq t_u, \bar{\pi}, [t_1, t_2] \models \langle p \rangle_1$ and $\exists t_3, t_4, t_2 < t_3 < t_4 < t_u, \bar{\pi}, [t_3, t_4] \models \langle p \rangle$ $\bar{\pi}, [t_l, t_u] \models$ **between** $\langle p \rangle_1$ **and** $\langle p \rangle_2$ $\langle p \rangle \Leftrightarrow$ $\forall t_1, t_2, t_3, t_4, t_l \leq t_1 < t_2 < t_3 < t_4 \leq t_u,$ $(\bar{\pi}, [t_1, t_2] \models \langle p \rangle_1$ and $[t_3, t_4] \models \langle p \rangle_2) \Rightarrow$ $\bar{\pi}, [t_2, t_3] \models \langle p \rangle$
Data Assertion	$\bar{\pi}, [t_l, t_u] \models$ **assert** $\langle c \rangle \Leftrightarrow \forall t \in [t_l, t_u], (\bar{\pi}, t \models \langle c \rangle)$ $\bar{\pi}, [t_l, t_u] \models \langle s \rangle$ **becomes** $\sim \langle v \rangle \Leftrightarrow \exists t \in (t_l, t_u], (f_p(t, \langle s \rangle) \sim \langle v \rangle$ and $\forall t_1 \in (t_l, t), (f_p(t_1, \langle s \rangle) \not\sim \langle v \rangle))$		
Spike	$\bar{\pi}, [t_l, t_u] \models$ **exists spike in** $\langle s \rangle$ $\left[$**with** $[$**width** $\sim \langle v \rangle_1]_\beta [$**amplitude** $\sim \langle v \rangle_2]_\gamma \right]_\alpha \Leftrightarrow$ $\exists t_1, t_2, t_3 \in [t_l, t_u], t_1 < t_2 < t_3,$ $\Big((uni\_m\_min\,(\bar{\pi}, s, t_1, [t_l, t_2])$ 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 $uni\_m\_max\,(\bar{\pi}, s, t_3, [t_2, t_u]))$ $\Big[[\text{and } \lvert t_3 - t_1 \rvert \sim \langle v \rangle_1]_\beta [\text{and } \max(\lvert f_p(t_1, \langle s \rangle) - f_p(t_2, \langle s \rangle) \rvert, \lvert f_p(t_2, \langle s \rangle) - f_p(t_3, \langle s \rangle) \rvert) \sim \langle v \rangle_2]_\gamma \Big]_\alpha \Big)$		
Oscillation	$\bar{\pi}, [t_l, t_u] \models$ **exist oscillations in** $\langle s \rangle$ $\Big[$**with** $[$**p2pAmp** $\sim \langle v \rangle_1]_\beta$ $[$**period** $\sim \langle v \rangle_2]_\gamma \Big]_\alpha \Leftrightarrow$ $\exists t_1, t_2, t_3, t_4, t_5 \in [t_l, t_u], t_1 < t_2 < t_2 < t_3 < t_4 < t_5,$ $\Big((uni\_sm\_min\,(\bar{\pi}, s, t_2, [t_1, t_3])$ 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 $uni\_sm\_max\,(\bar{\pi}, s, t_4, [t_3, t_5]))$ $\Big[[\text{and } \lvert f_p(t_2, \langle s \rangle) - f_p(t_3, \langle s \rangle) \rvert \sim \langle v \rangle_1$ and $\lvert f_p(t_3, \langle s \rangle) - f_p(t_4, \langle s \rangle) \rvert \sim \langle v \rangle_1]_\beta [\text{and } (t_4 - t_2) \sim \langle v \rangle_2]_\gamma \Big]_\alpha \Big)$		
Rise Time	$\bar{\pi}, [t_l, t_u] \models \langle s \rangle$ **rises** $[$**monotonically**$]_\alpha$ **reaching** $\langle v \rangle \Leftrightarrow \exists t \in (t_l, t_u],$ $\Big(f_p(t, \langle s \rangle) \geq \langle v \rangle$ and $\forall t_1 \in (t_l, t), (f_p(t_1, \langle s \rangle) < \langle v \rangle) [\text{and } \forall t_2 \in (t_l, t), \forall t_3 \in (t_2, t], (f_p(t_2, \langle s \rangle) < f_p(t_3, \langle s \rangle))]_\alpha \Big)$		
overshoot	$\bar{\pi}, [t_l, t_u] \models \langle s \rangle$ **overshoots** $[$**monotonically**$]_\alpha \langle v \rangle_1$ **by** $\langle v \rangle_2 \Leftrightarrow \exists t \in (t_l, t_u],$ $\Big(f_p(t, \langle s \rangle) \geq \langle v \rangle_1$ and $\forall t_1 \in (t, t_u], (f_p(t_1, \langle s \rangle) \leq \langle v \rangle_1 + \langle v \rangle_2)$ $[\text{and } \forall t_2 \in (t_l, t), \forall t_3 \in (t_2, t], (f_p(t_2, \langle s \rangle) < f_p(t_3, \langle s \rangle))]_\alpha \Big)$		
Order Relationship	$\bar{\pi}, [t_l, t_u] \models$ **if** $\langle p \rangle_1$ **then** $[$**within** $(\bowtie)$ $\langle t \rangle]_\alpha$ $\langle p \rangle_2 \Leftrightarrow \forall t_1, t_2 \in [t_l, t_u], t_1 < t_2,$ $\Big(\bar{\pi}, [t_1, t_2] \models \langle p \rangle_1 \Rightarrow \exists t_3, t_4 \in [t_2, t_u], t_3 < t_4, (\bar{\pi}, [t_3, t_4] \models \langle p \rangle_2 [\text{and } (t_3 - t_2) [\![\bowtie]\!] \langle t \rangle]_\alpha) \Big)$ where $\bowtie \in \{$**exactly**, **at most**, **at least**$\}$ and $[\![\bowtie]\!]$ is defined such that $[\![$**exactly**$]\!] \equiv$ '=', $[\![$**at most**$]\!] \equiv$ '<=', $[\![$**at least**$]\!] \equiv$ '>='		

$\langle t \rangle, \langle t \rangle_1, \langle t \rangle_2 \in \mathbb{R}$; $\langle v \rangle, \langle v \rangle_1, \langle v \rangle_2 \in \mathbb{R}$; $\sim \in \{<, >, =, \neq, \leq, \geq\}$; $\langle s \rangle$ is a signal in S or a mathematical expression over the signals in S; $\langle sc \rangle$ is a scope; $\langle p \rangle$ is a pattern.

Figure 5.3: SB-TemPsy-DSL formal semantics

Table 5.1: Definition of predicates uni_m_max, uni_sm_max (predicates uni_m_min, uni_sm_min are the dual)

Predicate	Mathematical Formulation	Description
$uni\_m\_max(\bar{\pi}, s, t, [t_l, t_u])$	$f_p(t, \langle s \rangle) = x$ and $\forall t_1 \in [t_l, t_u]$, $f_p(t_1, \langle s \rangle) < x$ and $\forall t_1, t_2 \in [t_l, t]$, if $t_1 < t_2$ then $f_p(t_1, \langle s \rangle) \leq f_p(t_2, \langle s \rangle)$ and $\forall t_1, t_2 \in [t_l, t]$, if $t_1 < t_2$ then $f_p(t_1, \langle s \rangle) \geq f_p(t_2, \langle s \rangle)$	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.
$uni\_sm\_max(\bar{\pi}, s, t, [t_l, t_u])$	$f_p(t, \langle s \rangle) = x$ and $\forall t_1 \in [t_l, t_u]$, $f_p(t_1, \langle s \rangle) < x$ and $\forall t_1, t_2 \in [t_l, t]$, if $t_1 < t_2$ then $f_p(t_1, \langle s \rangle) < f_p(t_2, \langle s \rangle)$ and $\forall t_1, t_2 \in [t_l, t]$, if $t_1 < t_2$ then $f_p(t_1, \langle s \rangle) > f_p(t_2, \langle s \rangle)$	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.

(minimum or maximum) and its value changes in a certain way (see column "Description" in Table 5.1 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 4.2). 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** $\langle t \rangle_1$ **and** $\langle t \rangle_2$ $\langle p \rangle$" scope operator evaluates pattern $\langle p \rangle$ in the time interval $[\langle t \rangle_1, \langle t \rangle_2]$.

The semantics of a *property* is defined by evaluating the satisfaction of a scope $\langle sc \rangle$ 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 $\bar{\pi} \models \phi$.

5.6 SB-TemPsy-Check

In this section, we present the main steps of and the artifacts used within SB-TemPsy-Check. Section 5.6.1 describes the pre-processing step, Section 5.6.2 illustrates the meta-model of the

pre-processed trace, and Section 5.6.3 explains how the OCL constraint solver is used to perform trace checking.

5.6.1 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 $\eta \colon \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ 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 5.3; 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 $\eta(v, v) = v$. 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 5.7.

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.

5.6.2 Trace Meta-model

Our trace meta-model of the pre-processed trace is an extension of the one proposed by [DBB17a], 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. 5.4 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.

5.6.3 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

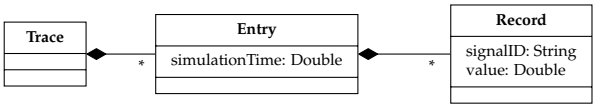


Figure 5.4: UML Class Diagram of the Trace Meta-model

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

Figure 5.5: OCL function for the *oscillation* pattern of SB-TemPsy-DSL

(i.e., an instance of the trace meta-model described in section 5.6.2) 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 5.2. As an example, below we illustrate the OCL function (shown in figure 5.5) 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 5.3. The function first saves the value of the pattern parameter $\langle s \rangle$ (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 5.3. 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 5.3. Functions `isLMin` and `isLMax` implement functions uni_sm_min and uni_sm_max described in section 5.5.2. 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 $\text{EVAL}(\bar{\pi}_{obj}, \phi_{OCL})$. 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 $\text{EVAL}(\bar{\pi}_{obj}, \phi_{OCL})$ yields *true*.

Correctness. The correctness of our procedure (intuitively) follows from these observations. The semantics of SB-TemPsy-DSL presented in section 5.5.2 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 5.6.1 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 5.3, our procedure is correct, i.e., the verdict returned by our trace-checking procedure is consistent with the semantics presented in Section 5.5.2.

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 5.5 contains five nested existential operators, leading to a procedure with a time complexity of $\mathcal{O}(|\bar{\pi}|^5)$.

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 5.7).

5.7 Evaluation

In this section, we report on the evaluation of our contributions. First, we evaluate our specification language SB-TemPsy-DSL in 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 5.2). 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 5.7.1)

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 5.7.2)

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

5.7.1 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 [MN04] and SFO [BFHN18], 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 [Rap16]; 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 5.2 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 [BJB+20], 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

Table 5.2: 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		

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 [BJB+20], 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 [BFHN18]. On the other hand, as discussed in section 5.6, 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.

5.7.2 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 [DFM13], 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 [BDD+18] (i.e., AMT [NLM+18] and S-TaLiRo [ALFS11]), because AMT 2.0, in contrast to Breach, is not publicly available, and also because Breach has been shown [DFM13] 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 $18\,\mathrm{h}$. Their size (in number of entries)

ranges from 41844 to 1202241 entries ($avg = 389771$, $StdDev = 393718$); the corresponding file size ranges from $\approx$1.7 MB to $\approx$58.9 MB (avg $\approx$17.6 MB, $StdDev$ $\approx$19.4 MB).

Methodology and settings. Our dataset contains traces recorded while a satellite system was tested in different environmental conditions (see section 5.2); 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 $\approx$255 B to $\approx$4.7 MB (avg $\approx$0.4 MB, $StdDev$ $\approx$0.8 MB).

We configured SB-TemPsy-Check to use *linear interpolation* as interpolation function for the pre-processing step (see section 5.6.1). 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 5.7.1). 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 [VBCG14]. 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 5.2). The total wall-clock time to run all the experiments was $\approx$ 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 $\approx$ 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 $\approx$ 99% of the completed runs (1870 out of 1884).

In the remaining 286 runs ($\approx$ 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" pat-

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

tern or an "event" scope. As discussed in section 5.6.3, 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 $0.01\,\mathrm{s}$ ($min = 0.006\,\mathrm{s}$, $max = 0.15\,\mathrm{s}$); SB-TemPsy-Check finished within the timeout in $\approx 97\%$ of the runs (1064 out of 1100). For these runs, SB-TemPsy-Check took on average $85.28\,\mathrm{s}$ ($min = 0.18\,\mathrm{s}$, $max = 7076.8\,\mathrm{s}$). For the remaining $\approx 3\%$ 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 $\approx 97\%$ 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 $10\,\mathrm{s}$ 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, $\approx 99\%$). 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 $48.7\,\mathrm{s}$. We deem this time to be reasonable for practical applications, since it is orders of magnitude lower than the time needed for simulation and trace generation (as discussed in section 5.2). 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 $10\,\mathrm{s}$ in $\approx 99\%$ of the cases, despite supporting the verification of a much larger set of property types.

5.7.3 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 $\approx 87\%$ 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 $\approx 97\%$ of the runs in which SB-TemPsy-Check terminated, it was slower than Breach but it was able to yield a verdict within $10\,\mathrm{s}$ for $\approx 99\%$ 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 5.6.3, 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.

5.8 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 [MN04] has been one of the first logic-based languages proposed for specifying SBTPs. STL* [BDŠV14] is an extension of STL with the signal-value freezing operator, which binds the value of a signal at a precise time instant. SFO [BFHN18] is a first-order temporal logic for signals. [BJB+20] compared the expressiveness of STL, STL*, and SFO analyti-

cally, based on different formulations of the main types of SBTPs (see section 4.2); in contrast, we have empirically compared the expressiveness of SB-TemPsy-DSL, STL and SFO as part of our evaluation (section 5.7.1), using properties from a representative industrial case study. [BB19] 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. [MNGB19] 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 [SACO02], Structured English Grammar for real-time specifications [KC05], Temporal OCL [KT13], OCLR [DBB14], VISPEC - graphical formalism [HMF15], TemPsy [DBB17a], SpeAR [FWH+17], TemPsy-AG [BBB19], ProMoboBox - property language [MVDS20], FRETISH [GPMS20]. The majority of them is based on some property specification patterns [AGL+15, DAC99]; none of them support SBTPs. The contract language proposed by [BOV+19] 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 [BJB+20]'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 5.1), 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 [NLM+18], Breach [Don10], and S-Taliro [ALFS11]) for offline trace checking; a tool is also available for the STL extension proposed in [BB19]. No tools are available for STL* and SFO. RFOL is supported by an online trace-checking procedure (SOCRaTEs [MNGB19]). The contract language proposed by [BOV+19] 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 [DBB17a] for the verification of simple temporal properties. The work in [DBB17a] has been extended by [BBB19] to support service provisioning specification patterns [BGPS12, BGS13] using aggregate operators. In this work we have applied the idea of model-driven trace checking from [DBB17a] in the context of SBTP through the development of the SB-TemPsy approach. The main differences with [DBB17a] 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 [BJB+20]; also, the scope operators,

though inspired by [DAC99]'s work, have been tailored to the domain of SBTP (e.g., to support absolute time instants). On the other hand, the TemPsy language [DBB17a] (and its predecessor OCLR [DBB14]) are only based on [DAC99]'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 5.5.2.

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

5.9 Summary

In this chapter, 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.

Chapter 6

Trace Diagnostics for Signal-based Temporal Properties

6.1 Overview

Trace checkers typically rely on qualitative semantics, returning a Boolean verdict: *true*, if the requirement is satisfied by the execution trace, and *false*, if it is violated. A requirement violation indicates that the system did not behave as expected. In this case, engineers need to understand the cause of the violation and address it. In most industrial CPS applications, this still largely remains a manual activity supported by ad-hoc solutions: engineers manually inspect the input signals through plotting their values over time, trying to understand the cause(s) of the violation. As traces typically come with a large number of trace records (each trace record represents a signal value at a specific time instant), manually determining the cause of a requirement violation is extremely difficult.

Pattern-based trace-checking tools like SB-TemPsy evaluate temporal properties based on a specific catalogue of property specification patterns; each of these patterns encodes a recurrent requirement that is checked by system developers. For example, a requirement (*pSpk*) that constrains the existence of a *spike* behavior in an input signal (e.g., signal spk_4 in figure 6.1) can be expressed in English as follow: *"the signal spk_4 shall show a spike with an amplitude greater than 1.5 and with a width less than 2"*. The corresponding property specification in SB-TemPsy-DSL is: `exists spike in spk4 with amplitude >1.5 with width <2`.

When using a pattern-based trace-checking tool like SB-TemPsy, a property violation can be represented as one or more distinct signal shapes. Engineers can systematically analyze and classify these signal shapes under a specific class of signal behaviors. We propose to represent

each of these classes as a *diagnostics pattern*. The intuition is that, since the property of interest is already defined based on a specific specification pattern, each of its possible violations also follows a specific *diagnostics pattern*. In other words, the latter captures a significant class of signal behaviors that leads to the violation of the property of interest. For example, when a *spike*-based property is violated, the violation might be caused by:

1. the existence of a spike with a violation of one or more feature-based predicate(s) (e.g., width-based predicate), as in spk_1 / spk_2 in figure 6.1. Note that a feature violation is based on the feature-based predicate defined in the property.

2. The absence of a strict local extremum (maximum or minimum) value from the signal records. Based on figure 6.1, this is the case for a flat signal (spk_5), a decreasing signal (spk_4) or an increasing signal (spk_5).

These *diagnostics patterns* can be captured using logical formulae, which map each pattern to its semantics. They can be defined based on domain knowledge provided by engineers, who can identify — for each property pattern — the most critical and meaningful violations to capture.

Each *diagnostics pattern* can be associated with a *violation type*, through which we precisely classify and distinguish between the possible signal behaviors causing the property violation. For example, let us consider signal spk_3 depicted in figure 6.1 as the execution trace for the evaluation of the following *spike* requirement: *in signal* spk_3*, there shall exist a spike with a maximum amplitude of 1.* As depicted in the figure, signal spk_3 is flat. In this case, we can characterize this faulty behavior as a *NSExt violation type*, which stands for *Non Shown Extremum(Extrema)*.

From a practical point of view, an effective *trace diagnostics* approach should provide informative verdicts, i.e., it should provide some *diagnostics information* that explains the cause of a property violation. For this reason, we propose to associate each *diagnostics pattern* with the corresponding *diagnostics information*. For example, identifying a decreasing signal (spk_4 in figure 6.1) does not reveal any information about the range of values over which that signal decreases. Nevertheless, an informative verdict containing some information about the signal records, showing the maximum and the minimum signal values, could provide more details about how the signal decreases, from the highest to the lowest value.

Based on the needs for trace diagnostics identified above, in this chapter we introduce TD-SB-TemPsy, a model-driven trace diagnostics approach for SBTPs expressed in SB-TemPsy-DSL and previously checked with SB-TemPsy-Check.

The rest of this chapter is organized as follows. Section 6.2 presents an overview of TD-SB-TemPsy; our pattern-based trace diagnostics approach. Section 6.3 defines the formal semantics of the *diagnostics patterns* and the corresponding *diagnostics information*. Section 6.4 illustrates and details the different *diagnostics patterns* and the corresponding *diagnostics information*, for each pattern type. Section 6.5 discusses the preliminary evaluation we conducted to evaluate the implementation of TD-SB-TemPsy. Section 6.6 discussed related work.

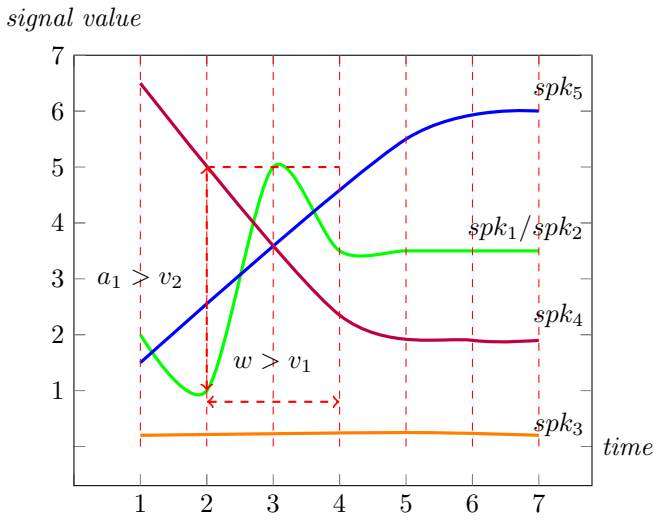


Figure 6.1: Examples of signal shapes that violate a *spike* property pattern.

6.2 Overview of TD-SB-TemPsy

An overview of TD-SB-TemPsy is presented in figure 6.2. TD-SB-TemPsy takes as input a property ϕ and a pre-processed trace $\bar{\pi}$ (see details about trace pre-processing in section 5.6.1). The property is obtained by instantiating a pattern[1] p with a list v of parameter values. For example, the *spike* pattern used to define property $pSpk$ specifies that signal spk_4 shows a spike behavior with constraints over the width (which shall be lower than 2) and the amplitude (which shall be greater than 1.5) features. Note that the threshold values used in the features represent some of the parameters of the *spike* pattern. For example, let us consider a *spike* pattern ($p = spike$) defined in 5.3 as follows:

`exists spike in` $\langle s \rangle$ `with [width` $\sim_1$ $\langle v\_1 \rangle$`]_`β `[amplitude` $\sim_2$ $\langle v\_2 \rangle$`]_`γ`]_`α

The list of parameters v defined by that pattern are arguments representing the input values that shall be supplied by the users. These arguments are the following: 1) the signal name ($\langle s \rangle$), 2) the relational operator ($\sim_1$), 3) the width value $\langle v\_1 \rangle$, 4) the relational operator ($\sim_2$) and 5) the amplitude value $\langle v\_2 \rangle$. The parameter values ($s \mapsto spk_4$, $\sim_1 \mapsto$ ' $<$', $v_1 \mapsto 2$, $\sim_2 \mapsto$ ' $>$' and $v_2 \mapsto 1.5$) then lead to the *spike* property ϕ (see section 6.1 for the SB-TemPsy-DSL specification of property $pSpk$). More formally, given a property pattern p, and a set of parameters v used to define the pattern, we use the notation $\phi \leftarrow p_v$, to indicate that the pattern p, parametrized with the parameter values v, leads to the property ϕ. Based on a specific property ϕ, we determine a set DP of *diagnostics patterns*. Each *diagnostics pattern* dp instance is obtained by instantiating the corresponding parameters values $dp_{v1}, dp_{v2}, ..., dp_{vn}$. As a result, each *diagnostics pattern* dp

[1] In this chapter, we assume, without loss of generality, that a property is always bounded by a *globally* scope (see section 5.5), which is then omitted from the definition to keep the notation light.

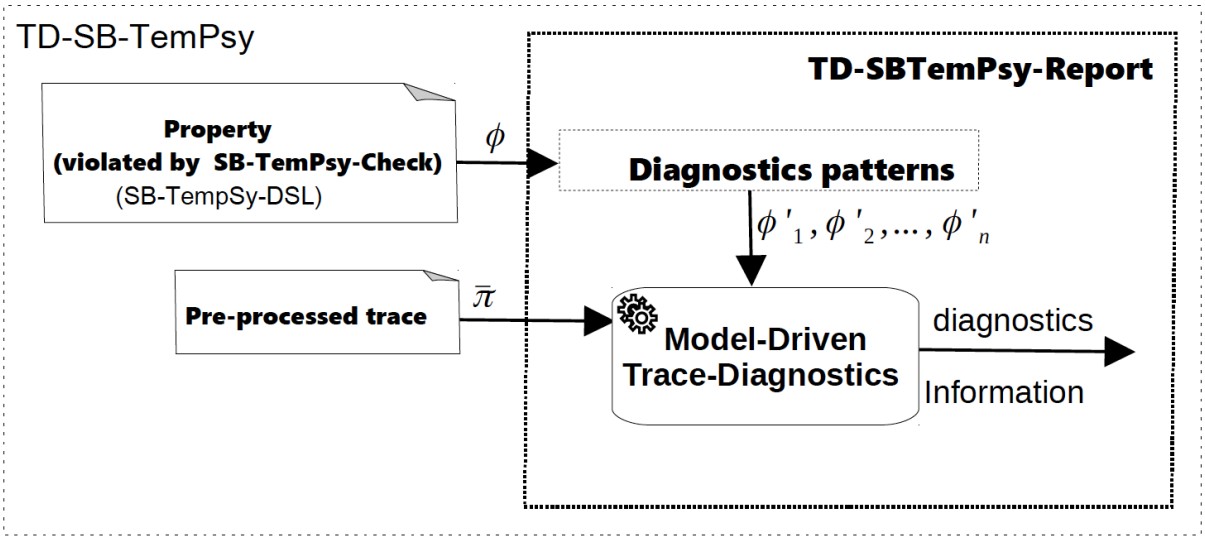


Figure 6.2: Overview of TD-SB-TemPsy

generates an instance formula ϕ', i.e., $\phi' \leftarrow dp_v$. More generally, based on property ϕ, we get the set $DP = \{\phi'_1, \phi'_2, ..., \phi'_n\}$, where each ϕ'_i is obtained by the instantiation of the corresponding dp_{vi}. Based on the set of *diagnostics patterns* DP and the pre-processed trace ϕ', TD-SB-TemPsy returns the following *diagnostics information*:

1. the signal name that is responsible of the property violation;
2. the *violation details*, which consist of information on one or more trace records (a timestamp, a signal value, or both) that represent relevant details about an unexpected behavior, characterized as a *diagnostics pattern*. These *violation details* result from the signal shape exhibiting a behavior compliant with *diagnostics patterns* for the corresponding property type (see details in § 6.3.2);
3. the *violation type* that describes the signal behavior identified by the *diagnostics pattern*.

6.3 Diagnostics patterns and Diagnostics Information: formal definition

A *diagnostics pattern* is a logical formula that formally defines and characterizes a specific shape of a signal that corresponds to a violation of a certain property type. Since each *diagnostics pattern* should characterize a set of signals that lead to the violation of the property of interest, we should ensure that, whenever the pattern holds on an execution trace, the property of interest shall be violated by that trace.

Table 6.1: Spike *Diagnostics Patterns*

'spike'	A signal $\langle s \rangle$ has a spike within the interval $[t_l, t_u]$ if there exists a strict maximum at time instant t_2 (the maximum of the spike) surrounded by two (non strict) minima respectively at time instant t_1 and t_3. The width $width(\langle s \rangle, t_1, t_3)$ of the spike constraints the size of the time interval $[t_1, t_3]$. The amplitude $amp(\langle s \rangle, t_2, t_3, t_4)$ constraints the maximum amplitude of the spike.
$spk_1{}^*$	None of the spikes satisfies the amplitude constraint (**NSF_a**): $\forall t_1, t_2, t_3 \in [t_l, t_u]((t_1 < t_2 < t_3 \wedge ((uni\_m\_min(s, t_1, [t_l, t_2]) \wedge uni\_sm\_max(s, t_2, [t_1, t_3]) \wedge uni\_m\_min(s, t_3, [t_2, t_u])) \vee (uni\_m\_max(s, t_1, [t_l, t_2]) \wedge uni\_sm\_min(s, t_2, [t_1, t_3]) \wedge uni\_m\_max(s, t_3, [t_2, t_u])))) \rightarrow \neg amp(\langle s \rangle, t_1, t_2, t_3))$
$spk_2{}^*$	None of the spikes satisfies the width constraint (**NSF_p**): $\forall t_1, t_2, t_3 \in [t_l, t_u]((t_1 < t_2 < t_3 \wedge ((uni\_m\_min(s, t_1, [t_l, t_2]) \wedge uni\_sm\_max(s, t_2, [t_1, t_3]) \wedge uni\_m\_min(s, t_3, [t_2, t_u])) \vee (uni\_m\_max(s, t_1, [t_l, t_2]) \wedge uni\_sm\_min(s, t_2, [t_1, t_3]) \wedge uni\_m\_max(s, t_3, [t_2, t_u])))) \rightarrow \neg width(\langle s \rangle, t_1, t_3))$
spk_3	The signal $\langle s \rangle$ is constant (**NSExt**): $\forall t \in [t_l, t_u](f_p(t, \langle s \rangle) = f_p(t_l, \langle s \rangle))$
spk_4	The signal $\langle s \rangle$ always decreases (**NSExt**): $\forall t_1 \in [t_l, t_u)(\forall t_2 \in (t_1, t_u](f_p(t_1, \langle s \rangle) \geq f_p(t_2, \langle s \rangle)))$
spk_5	The signal $\langle s \rangle$ always increases (**NSExt**): $\forall t_1 \in [t_l, t_u)(\forall t_2 \in (t_1, t_u](f_p(t_1, \langle s \rangle) \leq f_p(t_2, \langle s \rangle)))$

$amp(\langle s \rangle, t_1, t_2, t_3) \equiv \max(|f_p(t_2, \langle s \rangle) - f_p(t_1, \langle s \rangle)|, |f_p(t_2, \langle s \rangle) - f_p(t_3, \langle s \rangle)|) \sim_2 \langle v\_2 \rangle$
$width(\langle s \rangle, t_1, t_2) \equiv (t_2 - t_1) \sim_1 \langle v\_1 \rangle$

More in details, let p be a property pattern, v be a set of parameter values and ϕ be a property specified using an instantiation of p based on the corresponding parameter values, i.e., $\phi \leftarrow p_v$; a *diagnostics pattern* dp for p, is a pattern such that, when parameterized[2] with the values in v, it generates the instance formula ϕ' (formally, $\phi' \leftarrow dp_v$) that satisfies the following *diagnostics relation* (on any execution trace $\bar{\pi}$)

$$\text{if } \bar{\pi} \models \phi' \text{ then } \bar{\pi} \not\models \phi$$

This diagnostics relation defines a *diagnostics pattern*. It ensures that, if the instance formula of the *diagnostics pattern* dp parametrized with v holds, then the instance formula of the property pattern p parametrized with v does not hold.

6.3.1 Diagnostics Patterns Definition: Methodology

To define our *diagnostics patterns* and to ensure that they satisfy the diagnostics relation, we performed three steps: *behavior analysis*, *semantics definition*, and *verification of the diagnostics relation*.

Behavior Analysis. We identified classes of undesired signal behaviors that lead to property violations and described each of these behaviors. Each class of signal behaviors is associated with a pattern. For example, Figure 6.1 reports five possible signal behaviors, belonging to five different classes of undesired signal behaviors we identified, that lead to the violation of the *spike* property pattern defined above. The signal shapes we capture are the following:

- spk_1: in a similar signal shape, all spike instances violate the amplitude constraint, showing a violation of type *Non-Satisfied amplitude Feature (NSF_a)*;

[2]Each pattern p has a different set of parameters.

- spk_2: all spikes in the signal violate the width constraint, showing a violation of type *Non-Satisfied Feature width (NSF_w)*;
- spk_3: the signal is flat with a violation type *Non-Shown Extremum (Extrema) (NSExt)*;
- spk_4: the signal does not show any spike; instead, it decreases showing a violation of type $NSExt$;
- spk_5: the signal does not show any spike; instead, it increases showing a violation of type $NSExt$.

Semantics Definition. We then characterized each behavior through a diagnostic pattern dp. For each class, we used a logical formula that is satisfied only when a signal belongs to that class, i.e., satisfies the semantics of the pattern. The semantics of each *diagnostics pattern* defines specific constraints on an execution trace $\bar{\pi}$, taking into account its pattern dp and the corresponding parameter values from the set v. For example, the following semantics (which correspond to the 5-th row in table 6.1) of the diagnostics pattern spk_4 (related to the decreasing signal spk_4 in figure 6.1)

$$\forall t_1 \in [t_l, t_u)\ (\forall t_2 \in (t_1, t_u]\ (f_p(t_1, \langle s \rangle) \geq f_p(t_2, \langle s \rangle))) \tag{6.1}$$

specifies that for any time instant t_1 from the left-closed interval delimited by the lower and the upper boundaries (t_l and t_u, respectively), the value ($f_p(t_2, \langle s \rangle)$) of the signal $\langle s \rangle$ at any time instant t_2 that comes after t_1 shall be lower than the value $f_p(t_1, \langle s \rangle)$. By associating the signal name parameter of the *diagnostics pattern* spk_4 with the corresponding value in v, the obtained instance formula from this *diagnostics pattern* instantiation (dp) is then (ϕ') (e.g., signal spk_4). Let us consider the *diagnostics pattern* spk_1 that comes with a more complex semantics than spk_4 defined and explained above. The semantics definition of spk_1 is: (see also the 2-d row in table 6.1)

$$\begin{aligned}
&\forall t_1, t_2, t_3 \in [t_l, t_u]((t_1 < t_2 < t_3 \wedge \\
&((uni_m_min(s, t_1, [t_l, t_2]) \wedge uni_sm_max(s, t_2, [t_1, t_3]) \wedge uni_m_min(s, t_3, [t_2, t_u])) \vee \\
&(uni_m_max(s, t_1, [t_l, t_2]) \wedge uni_sm_min(s, t_2, [t_1, t_3]) \wedge uni_m_max(s, t_3, [t_2, t_u])))) \rightarrow \\
&\neg amp(\langle s \rangle, t_1, t_2, t_3))
\end{aligned} \tag{6.2}$$

specifies that for all the detected spike instances (line 1 of the formula), each showing either a strict local maximum surrounded by two local minima (line 2 of the formula) or a strict local minimum surrounded by two local maxima (line 3 of the formula), the predicate over the amplitude spike feature shall be violated (line 4 of the formula). The semantics comes with 3 parameters, instantiated through associating the corresponding parameter values as follows:

1) the signal name spk_1;

2) the relational operator $\sim_2$ related to the amplitude predicate (see the corresponding semantics (semantics of $amp(\langle s \rangle, t_1, t_2, t_3)$) right below table 6.1) and

3) the amplitude value v_2 defined in the property definition

We denote by ζ a Boolean function that takes as input a pre-processed trace $\bar{\pi}$, a specification pattern p or a diagnostic pattern dp, and the corresponding set of parameters v.

Function $\zeta(\bar{\pi}, p, v)$ returns *true* when the property ϕ resulting from instantiating pattern p with the corresponding parameter values v holds over the execution trace $\bar{\pi}$. More formally, $\zeta(\bar{\pi}, p, v)$ is *true* if (and only if) $\phi \leftarrow p_v$ and $\bar{\pi} \models \phi$.

Similarly, function $\zeta(\bar{\pi}, dp, v)$, taking in the pre-processed trace $\bar{\pi}$, the *diagnostics pattern* dp and the corresponding set of parameters v, return *true* when the property ϕ' resulting from instantiating the *diagnostics pattern* dp with the corresponding parameter values v holds over the execution trace $\bar{\pi}$. More formally, $\zeta(\bar{\pi}, dp, v)$ is *true* if (and only if) $\phi' \leftarrow dp_v$ and $\bar{\pi} \models dp$.

Verification of the Diagnostics Relation. To check for the satisfaction of the diagnostics relation, as a way to ensure that the the latter holds for each of the *diagnostics patterns* we defined, we checked whether

$$\zeta(\bar{\pi}, dp, v) \rightarrow \neg\zeta(\bar{\pi}, p, v) \tag{6.3}$$

holds for each *diagnostics pattern*.
To automatically check whether $\zeta(\bar{\pi}, dp, v) \rightarrow \neg\zeta(\bar{\pi}, p, v)$, we consider the formula

$$\Psi \equiv \neg(\zeta(\bar{\pi}, dp, v) \rightarrow \neg\zeta(\bar{\pi}, p, v)) \tag{6.4}$$

and checked whether Ψ is satisfiable. If Ψ is unsatisfiable, then $\zeta(\bar{\pi}, dp, v) \rightarrow \neg\zeta(\bar{\pi}, p, v)$ holds and the *diagnostics pattern* guarantees the satisfaction of the *diagnostics relation*.

We use the Microsoft Z3 [3] SMT solver to check for the satisfiability of Ψ. For example, Z3 returns an `unsat` result when the formula Ψ, obtained by considering the semantics of the spk_4 *diagnostics pattern* and the original *spike* pattern, is evaluated. This proves that the *diagnostics pattern* spk_4 satisfies the diagnostics relation for the *spike* property. In other words, a signal shape that always decreases (signal spk_4 in figure 6.1) indicates one possible violation of the *spike* property ϕ, which requires the signal to show a spike shape conforming to the features-based predicates used in the property definition. This is formalized by formula spk_4 in table 6.1.

A property can be violated for several reasons. This implies that more than one *diagnostics pattern* can be satisfied by TD-SB-TemPsy. For this reason, we design our tool in such a way that we prioritize over a list of defined *diagnostics patterns*, that characterize the possible violations. The priority list is defined based on the user's needs.

[3] `https://github.com/Z3Prover/z3`

6.3.2 Diagnostics Information

The *diagnostics information* concisely describes why a *diagnostics pattern* is matched. For example, signal spk_4 in figure 6.1 shows a violation of type *NSExt*. The *violation details* consist of the information about two records of the signal (see semantics of $spk_4 - NSExt$ *violation details* in table 6.8), which correspond to its last-seen maximum and first-seen minimum values (6.5 and 2, at timestamps 1 and 5, respectively). These two extrema illustrate the range of values over which the signal changes.

Figure 6.3 shows an example of the OCL implementation, based on the spk_1 *diagnostics pattern* and the corresponding *diagnostics information*. Function `reportPatternSpike` takes

```
1 def: reportPatternSpike(trace:OrderedSet(trace::TraceElement), pattern::Spike, tl:
      Real,tu:Real):
2
3         <vSignal:String,violationDetails:Sequence{Tuple},vType:String>: Tuple =
4 let s : String = pattern.s in
5 if trace->forAll(t1, t2, t3| tl<= t1 and t1.simulationTime < t2.simulationTime
6                 and t2.simulationTime < t3.simulationTime and t3.simulationTime <=
                      tu
7                 and ( isLMax(trace,t1,tl,t2,s) and isLMin(trace,t2,t1,t3,s) and
                      isLMax(trace,t3,t2,tu,s)
8                  or (isLMin(trace,t1,tl,t2,s) and isLMax(trace,t2,t1,t3,s) and
                       isLMin(trace,t3,t2,tu,s)))
9                 implies Not(p2p(s,t1,t2,t3)))
10 then
11         if (getValue(s,t1)-getValue(s,t2).abs()>getValue(s,t2)-getValue(s,t3).abs
               ()) then
12                  <vSignal=s,
13                  violationDetails=Sequence{
14                         Tuple{timestamp=t1.simulationTime, signalValue=getValue(s,
                               t1)},
15                         Tuple{timestamp=t2.simulationTime, signalValue=getValue(s,
                               t2)}},
16                  vType=NSF_p2p>
17         else
18                  <vSignal=s,
19                  violationDetails=Sequence{
20                         Tuple{timestamp=t2.simulationTime, signalValue=getValue(s,
                               t2)},
21                         Tuple{timestamp=t3.simulationTime, signalValue=getValue(s,
                               t3)}},
22                  vType=NSF_p2p>
23         endif
24 endif
```

Figure 6.3: OCL function for the *spike* pattern based on $spk - NSF_a$ *diagnostics pattern*

as input a trace, a *spike* pattern and two boundaries (tl and tu, standing for the left and right boundaries that delimit the trace to a sub-trace over which we are interested in reporting *diagnostics information* in case of the *spike* property violation). Function `reportPatternSpike` selects the first satisfied *diagnostics pattern* formula (lines 5–9) from the priority list of *diagnostics patterns*. The *diagnostics pattern* illustrated in the figure matches the spk_1 *diagnostics pattern* in table 6.1. The satisfied *diagnostics pattern* comes with *diagnostics information* (lines 11–23); the function returns a tuple with three elements (line 3): 1) the signal name from the property definition ($vSignal$), 2) a sequence of tuples of *violation details* ($violationDetails$), returning the two extrema from the spike instance, that determine its amplitude value that is closest to the satisfaction of the amplitude-based predicate and 3) a violation type $vType$ set to NSF_a.

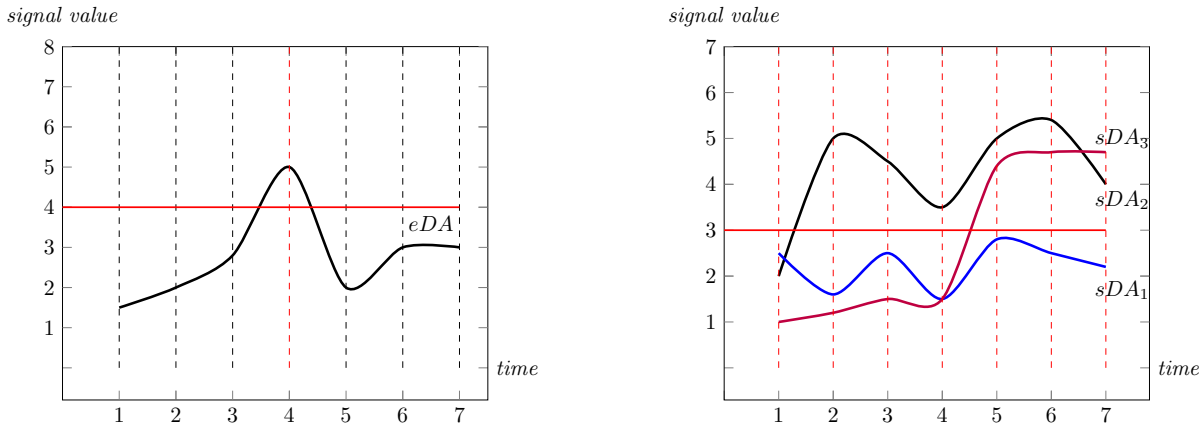
6.4 Defining Diagnostic Patterns and Diagnostic Information for SB-TemPsy-DSL property types

In this section, we describe the *diagnostics patterns* and the corresponding *diagnostics information* for each property (pattern) type supported by SB-TemPsy-DSL.

6.4.1 Data Assertion

A data assertion property might come in two distinct variants: an *event-based* and a *state-based* data assertion. For the *event-based* data assertion, based on the original property definition (see semantics in the first line of the third row of table 5.3), the existence of at least one record that violates the predicate is sufficient to trigger the violation of the original property. For example, let us consider the following property: *"Signal eDA shall be always less than the value of 4"*. As depicted in figure 6.4a, the signal violates the property at timestamp 4 showing a value equal to 5. This type of property violation is: $Non-SatisfiedEvent(NSE)$. For the *state-based* data assertion, based on the original property definition (see semantics in the second line of the third row of table 5.3), three possible signal shapes might cause the property violation. For example, the violation of property $pSDA$: *"the signal shall become less than 3"* might be caused by one of the following signal shapes (depicted in figure 6.4b):

- sDA_1: the signal is always below the value of 3, showing a violation of type *Non-Shown State (NSS)*;
- sDA_2: the signal is above the value of 3 over a left-open boundary interval, showing a violation of the same type as for the previous signal shape (*NSS*);
- sDA_3: the signal shows a dual behavior (a behavior different from the one expected from the property definition $pSDA$). The signal becomes greater than 3, instead of going below that value). This type of property violation is *Shown-Dual State (SDS)*.



(a) event-based data assertion

(b) state-based data assertion

Figure 6.4: Examples of signal shapes that violate a '`data assertion`' pattern

Table 6.2: Data assertion *Diagnostics Patterns*

'`assert`'	For every time instant the condition $\langle c \rangle$ holds
eDA	There exists a time instant where the condition $\langle c \rangle$ does not hold (**NSE**): $\exists t \in [t_l, t_u]\,(\bar{\pi}, t \not\models \langle c \rangle)$
'`becomes`'	There exists a time instant t where $f_p(t, \langle s \rangle) \sim \langle v \rangle$ is true, and for any time instant t_1 that precedes t, $f_p(t_1, \langle s \rangle) \not\sim \langle v \rangle$
sDA_1	The value $f_p(t, \langle s \rangle)$ of the signal $\langle s \rangle$ at any time instant t does never satisfy the relation $f_p(t, \langle s \rangle) \sim \langle v \rangle$ (**NSS**): $\forall t \in (t_l, t_u]\big(f_p(t, \langle s \rangle) \not\sim \langle v \rangle\big)$
sDA_2	Every time instant t is preceded by a time instant t_1 where $f_p(t_1, \langle s \rangle) \sim \langle v \rangle$ is satisfied (**NSS**): $\forall t \in (t_l, t_u]\big(\exists t_1 \in (t_l, t)(f_p(t_1, \langle s \rangle) \sim \langle v \rangle)\big)$
sDA_3	The value of the signal $\langle s \rangle$ satisfies the predicate $\sim \langle v \rangle$, until time instant t. Starting from t onwards, the value of $\langle s \rangle$ satisfies $\not\sim \langle v \rangle$ (**SDS**): $\exists t \in (t_l, t_u)(\forall t_1 \in [t_l, t)(f_p(t_1, \langle s \rangle) \sim \langle v \rangle) \wedge \forall t_2 \in [t, t_u](f_p(t_2, \langle s \rangle) \not\sim \langle v \rangle))$

We provide *diagnostics patterns* (see details in table 6.2) as formalizations of the signal shapes identified earlier. For each *diagnostics pattern*, we provide a short description in English, the corresponding violation type (text in bold) and the formal semantics. Note that the first row in the table presents our *diagnostics pattern* (annotated as eDA) for an event-based data assertion while the second row defines and formalizes the possible *diagnostics patterns* of a state-based data assertion violation (sDA_1,sDA_2 and sDA_3).

The *violation details* for properties of type data assertion are the following:

- *eDA-NSE*; when an event-based data assertion property is violated, we return the first-seen record by which the signal violated the predicate;

- $sDA_1 - NSS / sDA_2 - NSS$; when a state-based data assertion is violated due to always satisfying the predicate or always violating it, we return two records from the signal such

Table 6.3: *Violation Details* associated with Data Assertion *Diagnostics Patterns* defined in Table 6.2.

eDA	One time instant t and signal value $f_p(t, \langle s \rangle)$ that do not satisfy the condition $\langle c \rangle$: $\langle t, f_p(t,s) \rangle \mid t_l \leq t \leq t_u \wedge f_p(t,s) \not\models \langle c \rangle \wedge \forall t' \in [t_l, t)(f_p(t', s) \models \langle c \rangle)$
sDA1/sDA2	The time instants t_1 and t_2 and the values $f_p(t_1, \langle s \rangle)$ and $f_p(t_2, \langle s \rangle)$ of the first maximum and the minimum of the signal $\langle s \rangle$: $\langle \langle t_1, f_p(t_1,s) \rangle, \langle t_2, f_p(t_2,s) \rangle \rangle \mid t_l \leq t_1 \leq t_u \wedge t_l \leq t_2 \leq t_u \wedge \exists t_3, t_4 \in [t_l, t_u]((uni\_m\_max(s, t_1, [t_l, t_3]) \wedge uni\_m\_min(s, t_2, [t_3, t_4])) \vee (uni\_m\_min(s, t_1, [t_l, t_3]) \wedge uni\_m\_max(s, t_2, [t_3, t_4])))$
sDA3	The last timestamp t_1 at which the signal $\langle s \rangle$ satisfies the predicate $\sim \langle v \rangle$, and the next timestamp t_2 at which the signal violates it: $\langle \langle t_1, f_p(t_1,s) \rangle, \langle t_2, f_p(t_2,s) \rangle \rangle \mid tl \leq t_1 < t_2 \leq t_u \wedge f_p(t_1, \langle s \rangle) \sim \langle v \rangle \wedge f_p(t_2, \langle s \rangle) \not\sim \langle v \rangle \wedge \neg \exists t_3 \in [t_l, t_u](t_1 < t_3 < t_2)$

that they show the minimum and maximum values;

- $sDA_3 - SDS$; when a state-based property is violated due to a dual behavior (the signal goes above the value of 3, instead of becoming less than this value in property $pSDA$), we return the last-seen record by which the signal was satisfying the predicate followed by the very first-seen record by which the signal violates it.

Overall, the full set of reported *diagnostics information* (considering the sDA_3 *diagnostics pattern*) consists of:

1. the signal name (sDA_3), retrieved from the property definition,
2. the following *violation details*: the last seen record that satisfies the pattern predicate (timestamp t_1 with the value $f_p(t_1, s)$), and the first seen record (occurs at timestamp t_2, right after t_1) where the pattern predicate is satisfied, and
3. the violation type (already associated to the corresponding *diagnostics pattern*) which is set to SDS.

6.4.2 Rise Time (and Fall Time)

We distinguish between four possible signal shapes for a *rise time* property violation. Let us consider property *"the signal shall rise (possibly monotonically) reaching the value of 3"*. A possible violation might be due to one of the following signal shapes (see also figure 6.5):

- rt_1: the signal is always below the value of 3, with a violation type NSS;
- rt_2: the signal is always above the value of 3, with a violation type NSS;
- rt_3: the signal rises reaching the target value with a violation of the monotonicity constraint, showing a violation of type *Non-Satisfied Monotonicity (NSM)*;

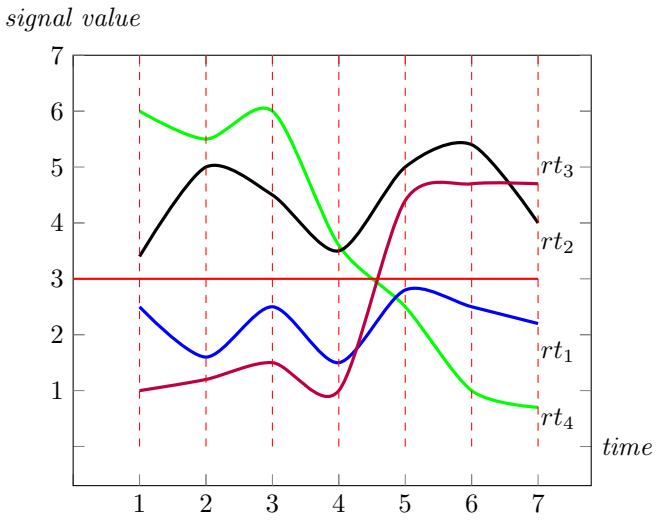


Figure 6.5: Examples of signal shapes that violate a '`rise time`' pattern.

Table 6.4: Rise Time *diagnostics patterns*

'`rises`'	There exists a time instant t where $f_p(t, \langle s \rangle) \geq \langle v \rangle$ is true, and for any time instant t_1 that precedes t, $f_p(t_1, \langle s \rangle) < \langle v \rangle$. An optional construct can be used to force the signal to rise monotonically.		
rt_1	The signal value is always below $\langle v \rangle$ (**NSS**): $\forall t \in [t_l, t_u](f(t, \langle s \rangle) < \langle v \rangle)$	*rt_2*	The signal value is always above $\langle v \rangle$ (**NSS**): $\forall t \in [t_l, t_u](f(t, \langle s \rangle) \geq \langle v \rangle)$
rt_3	The signal rises, but not monotonically (**NoMONOT**): $\exists t \in (t_l, t_u](f_p(t, \langle s \rangle) \geq \langle v \rangle \wedge \forall t_1 \in [t_l, t)(f_p(t1) < v) \wedge \neg(\forall t_2 \in [t_l, t)(\forall t_3 \in (t_2, t](f_p(t_2) < f_p(t_3))))))$		
rt_4	The value of the signal is initially above the threshold $\langle v \rangle$ and remains above that threshold for a certain amount of time. Then, it drops and remain below the threshold $\langle v \rangle$ (**SDS**): $\exists t \in (t_l, t_u)(\forall t_1 \in [t_l, t)(f_p(t_1, \langle s \rangle) \geq \langle v \rangle) \wedge \forall t_2 \in [t, t_u](f_p(t_2, \langle s \rangle) < \langle v \rangle))$		

- rt_4: the signal falls instead of rising, showing a dual behavior with a violation type SDS.

We define *diagnostics patterns* (see table 6.4) of a *rise time* behavior w.r.t the signal shapes identified above.

Table 6.5 shows the *violation details* we report for a rise time(and fall time) signal behavior. These details are the following:

- $rt_1 - NSS / rt_2 - NSS$; when a signal does not rise reaching the target value (it remains always above or below that value). Similarly to sDA_1 and sDA_2 *diagnostics patterns* of data assertion, we report information about its maximum and minimum values through returning the two corresponding records;

Table 6.5: *Violation Details* associated with Rise Time *Diagnostics Patterns* defined in Table 6.4.

	'rises'
rt_1-NSS1	The values $f_p(t_a, \langle s \rangle)$ and $f_p(t_i, \langle s \rangle))$ of the first maximum and the minimum of the signal $\langle s \rangle$ and their time instants t_a and t_i.
rt_2-NSS2	$\langle (t_i, f_p(t_i, s)), (t_a, f_p(t_a, s)) \rangle \mid t_l \leq t_i \leq t_u \wedge t_l \leq t_a \leq t_u \ \wedge \forall t \in [t_l, t_u)((f_p(t, s) \geq f_p(t_i, s)) \wedge (f_p(t, s) \leq f_p(t_a, s))) \ \wedge \forall t_1 \in [t_l, t_i)(f_p(t_1, s) > f_p(t_i, s)) \wedge \forall t_2 \in [t_l, t_a)(f_p(t_2, s) < f_p(t_a, s))$
rt_3-NSM	Two consecutive signal values and their timestamps t_1 and t_2 that do not satisfy the monotonicity constraint. $\langle \langle t_1, f_p(t_1, s) \rangle, \langle t_2, f_p(t_2, s) \rangle \rangle \mid tl \leq t_1 < t_2 < t_s \ \wedge \ \exists t_s \in (t_l, t_u](f_p(t_s, \langle s \rangle) \geq v) \ \wedge f_p(t_1, \langle s \rangle) < v \wedge f_p(t_2, \langle s \rangle) < v \ \wedge \ f_p(t_1, \langle s \rangle) \geq f_p(t_2, \langle s \rangle) \ \wedge \nexists t \in (t_1, t_2)((f_p(t, s) < v)))$
rt_4-SDS	The last timestamp t_s at which the signal $\langle s \rangle$ is greater than or equal to the target value $\langle v \rangle$, and the next timestamp t_v at which the signal falls, going below $\langle v \rangle$. $\langle \langle t_s, f_p(t_s, s) \rangle, \langle t_v, f_p(t_v, s) \rangle \rangle \mid \wedge tl \leq t_s < t_v \leq t_u \wedge f_p(t_s, \langle s \rangle) \geq \langle v \rangle \wedge f_p(t_v, \langle s \rangle) < \langle v \rangle \wedge \neg \exists t \in [t_l, t_u](t_s < t < t_v)$

- $rt_3 - NSM$; when the signal rises reaching the target value, but not monotonically when the monotonicity constraint is required in the property definition, we report information about two consecutive records that violate this constraint while the signal already rises to reach the target value;
- $rt_4 - SDS$; when the signal falls instead of rising, we need to capture informative details about this violation, through returning the last seen record by which the signal satisfied the predicate and the first seen record by which it violates it.

For example, $rt_3 - NSM$ represents the *violation details* associated with the rt_3 *diagnostics pattern* from table 6.4. The reported *diagnostics information* then consist of:

- the signal name (s) from the property definition;
- the following *violation details*: two consecutive records whose values are less than the target value, set to the 3 in signal rt_3 of figure 6.5 and that do not satisfy the monotonicity constraint in *diagnostics pattern* rt_3 (at timestamps 3 and 4) such that the signal value of the first record is less than or equal to the value of the second one ($f_p 3, s \leq f_p(4, s)$); where the condition ($f_p 3, s > f_p(4, s)$) shall be satisfied instead;
- the violation type which is set to NSM.

6.4.3 Overshoot

We distinguish between four possible signal shapes for an *overshoot* property violation. Let us consider the property *"the signal shall overshoot (possibly monotonically) reaching the value of 3 by*

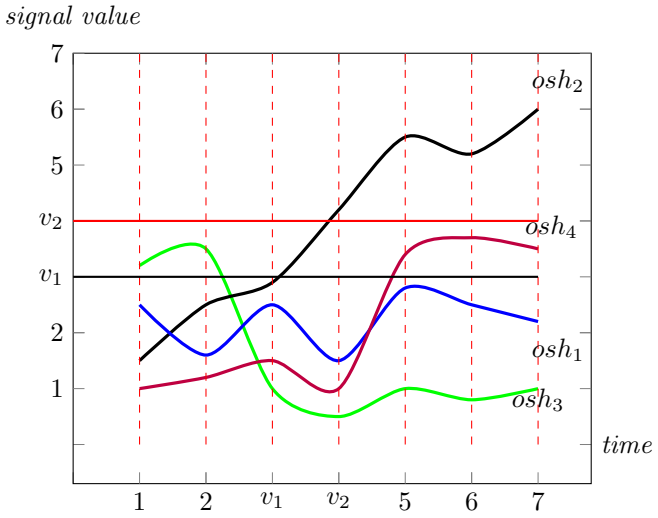


Figure 6.6: Examples of signal shapes that violate an '`overshoot`' pattern.

a maximum of 1". A possible violation might be due to one of the following signal shapes (see figure 6.6:

- osh_1: the signal is always below the value of 3, with a violation type NSS;
- osh_2: the signal goes above the sum of the target value and the maximum value ($v_1 + v_2$) and remains above this threshold, coming with a violation type NSS;
- osh_3: the signal undershoots below the target value, showing a violation of *SDS*;
- osh_4: the signal overshoots the target value without exceeding the possible threshold ($v_1 + v_2$), but it violates the monotonicity constraint before it reaches the target value v_1, showing a violation of type NSM.

We define four *diagnostics patterns* (details are in table 6.6), each characterizing a signal shape from the ones we identified above. Table 6.7 shows the *violation details* corresponding to each *diagnostics pattern*, which are the following:

- $osh_1 - NSS$ / $osh_2 - NSS$: the corresponding *violation details* and the violation type are as same as the ones of $rt_1 - NSS$ and $rt_2 - NSS$; we return the records showing the minimum and the maximum values the signal takes;
- $osh_3 - SDS$: same *violation details* as $rt_4 - SDS$ (see table 6.5) apply here;
- $osh_4 - NSM$; same *violation details* as $rt_3 - NSM$.

For example, the full reported *diagnostics information* $osh_3 - SDS$, corresponding to signal osh_3 consist of:

Table 6.6: Overshoot *Diagnostics Patterns*

`'overshoots'`	The value of a signal $\langle s \rangle$ exceeds its target value $\langle v \rangle_1$. After overshooting, the value of $\langle s \rangle$ remains below $\langle v \rangle_1 + \langle v \rangle_2$. An optional construct can be used to force the signal to rise monotonically before exceeding the target value $\langle v \rangle_1$.
osh_1	The signal $\langle s \rangle$ does not exceed $\langle v \rangle_1$ (**NSS1**): $\forall t \in [t_l, t_u](f_p(t, \langle s \rangle) < \langle v \rangle_1)$
osh_2	The value of $\langle s \rangle$ exceeds $\langle v \rangle_1 + \langle v \rangle_2$ and remains above this threshold (**NSS1**): $\exists t \in [t_l, t_u](f_p(t, \langle s \rangle) > v1 + v2 \wedge \forall t_1(t, t_u](f_p(t_1, \langle s \rangle) > v1 + v2))$
osh_3	The value of $\langle s \rangle$ stays between $\langle v1 \rangle$ and $\langle v1 \rangle + \langle v2 \rangle$ for a certain time interval and then goes below the threshold $\langle v1 \rangle$ (**SDS**): $\exists t \in (t_l, t_u](\forall t_1 \in [t_l, t](f_p(t_1, \langle s \rangle) \geq \langle v1 \rangle \wedge f_p(t_1, \langle s \rangle) \leq \langle v1 \rangle + \langle v2 \rangle) \wedge \forall t_2(t, t_u](f_p(t_2) < \langle v1 \rangle))$
osh_4	The signal overshoots, but not monotonically (**NoMONOT**): $\exists t \in (t_l, t_u](f(t, \langle s \rangle) \geq v1 \wedge f(t, \langle s \rangle) \leq v1 + v2 \wedge \forall t_1 \in (t, t_u](f_p(t_1, \langle s \rangle) \leq v1 + v2 \wedge f_p(t_1, \langle s \rangle) \geq v1) \wedge \neg(\forall t_2 \in [t_l, t)(\forall t_3 \in (t_2, t](f_p(t_2, \langle s \rangle) < f_p(t_3, \langle s \rangle))))$

Table 6.7: *Violation Details* associated with Overshoot *Diagnostics Patterns* defined in Table 6.6.

	`'overshoots'`
osh_1-NSS	The values $f_p(t_a, \langle s \rangle)$ and $f_p(t_i, \langle s \rangle))$ of the first maximum and the minimum of the signal $\langle s \rangle$ and their time instants t_a and t_i. $\langle (t_i, f_p(t_i, s)), (t_a, f_p(t_a, s)) \rangle \mid t_l \leq t_i \leq t_u \ \wedge \ t_l \leq t_a \leq t_u \ \wedge \ \forall t \in [t_l, t_u)((f_p(t, s) \geq f_p(t_i, s)) \wedge (f_p(t, s) \leq f_p(t_a, s))) \ \wedge \ \forall t_1 \in [t_l, t_i)(f_p(t_1, s) > f_p(t_i, s)) \wedge \forall t_2 \in [t_l, t_a)(f_p(t_2, s) < f_p(t_a, s))$
osh_2-NSS	
osh_4-NSM	Two consecutive signal values and their timestamps t_1 and t_2 that do not satisfy the monotonicity constraint. $\langle \langle t_1, f_p(t_1, s) \rangle, \langle t_2, f_p(t_2, s) \rangle \rangle \mid tl \leq t_1 < t_2 < t_s \ \wedge \ \exists t_s \in (t_l, t_u](f_p(t_s, \langle s \rangle \geq v_1 \wedge f_p(t_s, \langle s \rangle \leq v_1 + v_2)) \ \wedge \ f_p(t_1, \langle s \rangle) < v_1 \wedge f_p(t_2, \langle s \rangle) < v_1 \wedge f_p(t_1, \langle s \rangle) \geq f_p(t_2, \langle s \rangle) \ \wedge \ \nexists t \in (t_1, t_2)((f_p(t, s) < v_1)))$
osh_3-SDS	The last timestamp t_s at which the signal $\langle s \rangle$ is bounded by v_1 and $v_1 + v_2$ and the next timestamp t_v at which the signal falls, going below $\langle v_1 \rangle$. $\langle \langle t_s, f_p(t_s, s) \rangle, \langle t_v, f_p(t_v, s) \rangle \rangle \mid \wedge tl \leq t_s < t_v \leq t_u \wedge f_p(t_s, \langle s \rangle) \geq \langle v_1 \rangle \wedge f_p(t_s, \langle s \rangle) \leq \langle v_1 + v_2 \rangle \wedge f_p(t_v, \langle s \rangle) < \langle v_1 \rangle \wedge \neg\exists t \in [t_l, t_u](t_s < t < t_v)$

- the signal name (osh_3) from the property definition;

- the following *violation details*: two consecutive records that show a violation of the monotonicity constraint: the record at timestamp 2 with value 3.5 and the record at timestamp 3 showing value 1;

- the violation type which is set to NSM.

6.4.4 Spike

As illustrated and detailed in section 6.3, we distinguish between five possible signal shapes for a spike property violation (see figure 6.1).

For each *diagnostics pattern* of a *spike* behavior, we define *violation details* as shown in table 6.8 and described below:

- $spk_1 - NSF_a$: returns the time interval of the spike that is closer to satisfying the amplitude (a)-based predicate among all spike instances in the signal. It also returns the amplitude of that spike instance;

- $spk_2 - NSF_w$: returns the time interval of the spike that is closer to satisfying the width(w)-based predicate among all spike instances in the signal. It also returns the width of that spike instance;

- $spk_3 - NSExt$: returns the lower and the upper boundary throughout which the signal is flat, and the signal value;

- $spk_4 - NSExt / spk_5 - NSExt$; return two records from the decreasing/ increasing signal with the minimum and the maximum values.

For example, as shown in table 6.8, $spk_3 - NSExt$ represents the *violation details* associated with the spk_3 *diagnostics pattern* from table 6.1. The reported *diagnostics information* consists of:

- the signal name (s) from the property definition;

- the following *violation details*: the timestamps of the first and the third local extrema from the detected spike shape (2 and 4, respectively), the width of the detected spike ($4-2=2$) such that that width is the closest spike width to satisfying the width-based predicate from the property definition;

- the violation type, which is set to NSF_w.

Table 6.8: *Violation Details* associated with Spike *Diagnostics Patterns* defined in Table 6.1.

$spk_1 - NSF_a{}^*$	Returns the amplitude a and the time interval $[t_1, t_2]$ of the spike that is closest to satisfying the amplitude constraint: $\langle [t_1, t_2], p2p \rangle | (\exists t_3, t_4, t_5 \in [t_l, t_u] : (spk(\langle s \rangle, t_3, t_1, t_4, t_2, t_5) \wedge \neg p2p(\langle s \rangle, t_1, t_4, t_2) \wedge \forall t_6, t_7, t_8, t_9, t_{10} \in [t_l, t_u] : ((t_7 \neq t_1 \wedge t_8 \neq t_4 \wedge t_9 \neq t_2 \wedge spk(\langle s \rangle, t_6, t_7, t_8, t_9, t_{10})) \rightarrow ampv(\langle s \rangle, t_1, t_4, t_2) < ampv(\langle s \rangle, t_7, t_8, t_9))))$.
$spk_2 - NSF_w{}^*$	Returns the width w and the time interval $[t_1, t_2]$ of the spike that is closest to satisfying the width constraint: $\langle [t_1, t_2], w \rangle | (\exists t_3, t_4, t_5 \in [t_l, t_u] : (spk(\langle s \rangle, t_3, t_1, t_4, t_2, t_5) \wedge \neg width(\langle s \rangle, t_1, t_2) \wedge w = widthv(\langle s \rangle, t_1, t_2) \wedge \forall t_6, t_7, t_8, t_9, t_{10} \in [t_l, t_u] : ((t_7 \neq t_1 \wedge t_8 \neq t_4 \wedge t_9 \neq t_2 \wedge spk(\langle s \rangle, t_6, t_7, t_8, t_9, t_{10})) \rightarrow widthv(\langle s \rangle, t_1, t_2)\ ?\ widthv(\langle s \rangle, t_7, t_9))))$. The symbol ? is equal to $>$ or $<$ depending on whether the property requires the width to be greater than or lower than a certain value.
$spk_3 - NSExt$	The first and the last time stamps (t_1 and t_2) delimiting the exact interval throughout which the signal is flat and the signal value: $\langle [t_1, t_2], v \rangle | t_1 \leq t_l \wedge t_u \leq t_2 \wedge \forall t_3 \in [t_1, t_2] : (f_p(t_3, \langle s \rangle) = v \wedge \neg(\exists t_4 \in [t_f, t_1) : (\forall t_5 \in (t_4, t_1) : (f_p(t_5, \langle s \rangle) = v))) \wedge \neg(\exists t_6 \in (t_2, t_e] : (\forall t_7 \in (t_6, t_e] : (f_p(t_7, \langle s \rangle) = v))))$
$spk_4 - NSExt/$ $spk_5 - NSExt$	Returns the last seen maximum ($f_p(t_1, s)$) and the first seen minimum ($f_p(t_2, s)$) values of the signal and respectively the first (t_1) and the last (t_2) timestamps at which they are reached: $\langle \langle t_1, f_p(t_1, s) \rangle, \langle t_2, f_p(t_2, s) \rangle \rangle | tl \leq t_1 \leq t_2 \leq tu \wedge \forall t_3 \in [tl, t_1)(f_p(t_3, s) = f_p(t_1, s)) \wedge \forall t_4 \in (t_1, t_u](f_p(t_4, s) < f_p(t_1, s)) \wedge \forall t_5 \in [t2, t_u)(f_p(t_5, s) = f_p(t_2, s)) \wedge \forall t_6 \in [t_l, t_2)(f_p(t_6, s) > f_p(t_2, s))$

* $spk(\langle s \rangle, t_1, t_2, t_3, t_4, t_5) = (uni\_m\_min(s, t_2, [t_1, t_3]) \wedge uni\_sm\_max(s, t_3, [t_2, t_4]) \wedge uni\_m\_min(s, t_4, [t_3, t_5])) \vee (uni\_sm\_max(s, t_2, [t_1, t_3]) \wedge uni\_m\_min(s, t_3, [t_2, t_4]) \wedge uni\_sm\_max(s, t_4, [t_3, t_5]))$
$widthv(\langle s \rangle, t_1, t_2) = t_2 - t_1$
$ampv(\langle s \rangle, t_1, t_2, t_3) = |\max(|f_p(t_2, \langle s \rangle) - f_p(t_1, \langle s \rangle)|, |f_p(t_2, \langle s \rangle) - f_p(t_3, \langle s \rangle)|) - \langle v_2 \rangle|$

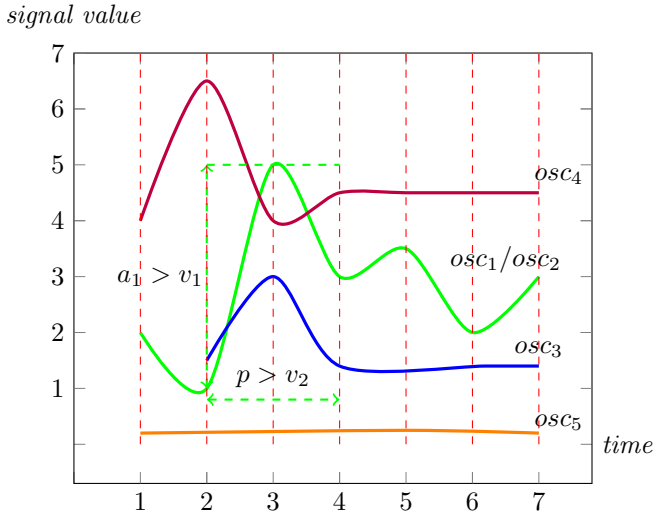


Figure 6.7: Examples of signal shapes that violate an '`oscillations`' pattern.

6.4.5 Oscillations

We distinguish between seven possible signal shapes for an oscillatory behavior-based property violation. (see figure 6.7):

- osc_1: all oscillatory-behaviors in the signal violate the peak-to-peak ($p2p$) amplitude constraint, showing a violation of type NSF_{p2p};
- osc_2: all oscillatory-behaviors in the signal violate the period (p) constraint, showing a violation of type NSF_p;
- osc_3: the signal does not oscillate. It only shows one strict extremum (minimum or maximum), the violation here is of type *NSExt*;
- osc_4: the signal does not oscillate. It rather shows two strict extrema (a minimum followed by a maximum, or vise versa), the violation here is of type $NSExt$;
- osc_5: the signal is flat, with a violation type $NSExt$;

Moreover, we also capture two more signal shapes, that we do not show in figure 6.1 as they are similar to the two *spike* signal shapes spk_6 and spk_7. These additional signals are, respectively:

- osc_6: the signal does not oscillate. It decreases instead coming with a violation of type $NSExt$;
- osc_7: the signal does not oscillate. It increases instead, showing the same violation type, $NSExt$.

Table 6.9 presents our *diagnostics patterns* for the *oscillations* pattern type, w.r.t the signal shapes considered in figure 6.7.

Below, we illustrate the *violation details* that match each oscillations *diagnostics pattern* as shown in table 6.10.

- $osc_1 - NSF_{p2p}$: returns the time interval of the oscillations that are closer to satisfying the predicate (based on the peak-to-peak amplitude) among all oscillations instances in the signal. It also returns the peak-to-peak amplitude of that oscillatory behavior;
- $osc_2 - NSF_p$: returns the time interval of the oscillation that is closer to satisfying the width-based predicate among all oscillations instances in the signal. It also returns the corresponding period;
- $osc_3 - NSExt$: returns the record that shows to be a strict extremum (maximum or minimum);
- $osc_4 - NSExt$: returns two records that represent two different strict extrema (maximum followed by a minimum, or vise versa);

Table 6.9: Oscillations *Diagnostics Patterns*

'oscillation'	A signal $\langle s \rangle$ shows an oscillation within the interval $[t_l, t_u]$ if there exists a strict maximum at time instant t_2 surrounded by two strict minima respectively at time instant t_1 and t_3. The period $period(\langle s \rangle, t_1, t_3)$ of the oscillation constraints the size of the time interval $[t_1, t_3]$. The peak-to-peak-amplitude construct $p2p(\langle s \rangle, t_1, t_2, t_3)$ constraints the maximum peak-to-peak amplitude of the oscillation.
$osc_1{}^*$	None of the oscillations satisfies the peak-to-peak amplitude constraint (**NSFeat_p2p**): $\forall t_1, t_2, t_3, t_4, t_5 \in [t_l, t_u](t_1 < t_2 < t_3 < t_4 < t_5 \wedge ((uni\_sm\_min(s, t_2, [t_1, t_3]) \wedge uni\_sm\_max(s, t_3, [t_2, t_4]) \wedge uni\_sm\_min(s, t_4, [t_3, t_5])) \vee (uni\_sm\_max(s, t_2, [t_1, t_3]) \wedge uni\_sm\_min(s, t_3, [t_2, t_4]) \wedge uni\_sm\_max(s, t_4, [t_3, t_5]))) \rightarrow \neg p2p(\langle s \rangle, t_1, t_2, t_3))$
$osc_2{}^*$	None of the oscillations satisfies the period constraint (**NSFeat_p**): $\forall t_1, t_2, t_3, t_4, t_5 \in [t_l, t_u](t_1 < t_2 < t_3 < t_4 < t_5 \wedge ((uni\_sm\_min(s, t_2, [t_1, t_3]) \wedge uni\_sm\_max(s, t_3, [t_2, t_4]) \wedge uni\_sm\_min(s, t_4, [t_3, t_5])) \vee (uni\_sm\_max(s, t_2, [t_1, t_3]) \wedge uni\_sm\_min(s, t_3, [t_2, t_4]) \wedge uni\_sm\_max(s, t_4, [t_3, t_5]))) \rightarrow \neg period(\bar{\pi}, \langle s \rangle, t_1, t_3))$
osc_3	The signal $\langle s \rangle$ contains only one local minimum/maximum (**NSExt**): $\exists t_1 \in [t_l, t_u)(\exists t_2, t_3(t_l, t_u](t_1 < t_2 < t_3 \wedge uni\_sm\_max(s, t_2, [t_1, t_3]) \vee uni\_sm\_min(s, t_2, [t_1, t_3]))) \wedge \forall t_4, t_5, t_6 \in [t_l, t_u]((t_5 \neq t_2 \wedge t_4 < t_5 < t_6) \rightarrow \neg(uni\_sm\_max(s, t_5, [t_4, t_6]) \vee uni\_sm\_min(s, t_5, [t_4, t_6])))$
osc_4	The signal $\langle s \rangle$ shows only two local extrema (maxima/ minima) (**NSExt**): $\exists t_1 \in [t_l, t_u)(\exists t_2, t_3(t_l, t_u](t_1 < t_2 < t_3 \wedge uni\_sm\_max(s, t_2, [t_1, t_3]) \vee uni\_sm\_min(s, t_2, [t_1, t_3]))) \wedge \exists t_4, t_5, t_6 \in [t_l, t_u]((t_5 \neq t_2 \wedge t_4 < t_5 < t_6) \wedge uni\_sm\_max(s, t_5, [t_4, t_6]) \vee uni\_sm\_min(s, t_5, [t_4, t_6]) \wedge \forall t_7, t_8, t_9 \in [t_l, t_u]((t_8 \neq t_2 \wedge t_8 \neq t_5 \wedge t_7 < t_8 < t_9) \wedge \neg(uni\_sm\_max(s, t_8, [t_7, t_9]) \vee uni\_sm\_min(s, t_8, [t_7, t_9])))$
osc_5	The signal $\langle s \rangle$ is constant (**NSExt**): $\forall t \in [t_l, t_u](f_p(t, \langle s \rangle) = f_p(t_l, \langle s \rangle))$
osc_6	The signal $\langle s \rangle$ always decreases (**NSExt**): $(\forall t_1 \in [t_l, t_u)\ (\forall t_2 \in (t_1, t_u](f_p(t_1, \langle s \rangle) \geq f_p(t_2, \langle s \rangle))$
osc_7	The signal $\langle s \rangle$ always increases (**NSExt**): $(\forall t_1 \in [t_l, t_u)\ (\forall t_2 \in (t_1, t_u](f_p(t_1, \langle s \rangle) \leq f_p(t_2, \langle s \rangle))$

Given the oscillations syntax (see table 5.2):

'exist' 'oscillations' 'in' $\langle s \rangle$ $\left['\texttt{with}' \left['\texttt{p2pAmp}' \sim \langle v \rangle_1\right]_\beta \left['\texttt{period}' \sim \langle v \rangle_2\right]_\gamma\right]_\alpha$

we define $p2p(\langle s \rangle, t_1, t_2, t_3)$ and $period(\langle s \rangle, t_1, t_2)$ functions such that:

$p2p(\langle s \rangle, t_1, t_2, t_3) \equiv \max\left(\left|f_p(t_2, \langle s \rangle) - f_p(t_1, \langle s \rangle)\right|, \left|f_p(t_2, \langle s \rangle) - f_p(t_3, \langle s \rangle)\right|\right) \sim_1 \langle v\_1 \rangle$

$period(\langle s \rangle, t_1, t_2) \equiv (t_2 - t_1) \sim_2 \langle v\_2 \rangle$

- $osc_5 - NSExt$: returns the left and the right boundary of the signal, throughout which the signal is flat;
- $osc_6 - NSExt$ / $osc_7 - NSExt$; returns two records from the decreasing/ increasing signal with the minimum and the maximum values.

For example, as shows table 6.10, $osc_4 - NSExt$ represents the *violation details* associated with *diagnostics pattern* osc_4. The full reported *diagnostics information* consists of:

- the signal name (osc_4) from the property definition;
- the corresponding *violation details*: the first local extremum (occurring at timestamp 2 showing a value of 6.5) and the second distinct local extremum (occurring at timestamp 3 showing a value of 4);
- the violation type, which is set to $NSExt$.

Table 6.10: *Violation Details* associated with Oscillations *diagnostics patterns* defined in Table 6.9.

$osc_1 - NSF_{p2p}{}^*$	Returns the amplitude $p2p$ and the time interval $[t_1, t_2]$ of the closest oscillations to satisfying the peak-to-peak amplitude constraint: $\langle [t_1, t_5], p2p\rangle \mid (\exists t_2, t_3, t_4 \in [t_l, t_u] : (osc(\langle s\rangle, t_1, t_2, t_3, t_4, t_5) \wedge \neg p2p(\langle s\rangle, t_2, t_3, t_4) \wedge \forall t_6, t_7, t_8, t_9, t_{10} \in [t_l, t_u] : ((t_8 \neq t_2 \wedge t_9 \neq t_3 \wedge t_{10} \neq t_4 \wedge osc(\langle s\rangle, t_6, t_8, t_9, t_{10}, t_7)) \rightarrow p2pv(\langle s\rangle, t_2, t_3, t_4) < p2pv(\langle s\rangle, t_8, t_9, t_{10}))))$.
$osc_2 - NSF_p{}^*$	Returns the period and the time interval $[t_1, t_2]$ of the closest oscillations to satisfying the period constraint: $\langle [t_1, t_5], p2p\rangle \mid (\exists t_2, t_3, t_4 \in [t_l, t_u] : (osc(\langle s\rangle, t_1, t_2, t_3, t_4, t_5) \wedge \neg p2p(\langle s\rangle, t_2, t_3, t_4) \wedge \forall t_6, t_7, t_8, t_9, t_{10} \in [t_l, t_u] : ((t_8 \neq t_2 \wedge t_9 \neq t_3 \wedge t_{10} \neq t_4 \wedge osc(\langle s\rangle, t_6, t_8, t_9, t_{10}, t_7)) \rightarrow periodv(\langle s\rangle, t_2, t_3, t_4) ? periodv(\langle s\rangle, t_8, t_9, t_{10}))))$. Where the symbol ? is equal to $>$ or $<$ depending on whether the property requires the period to be greater than or lower than the threshold period. .
$osc_3 - NSExt^*$	Returns the timestamp and the signal value of the record at which the only strict extremum occurs in the signal: $\langle t_1, f_p(t_1, s)\rangle \mid \exists t_2, t_3 \in [t_l, t_u] : t_l \leq t_2 < t_1 \wedge t_1 < t_3 \leq t_u \wedge (smin(s, t_1, [t_2, t_3]) \vee smax(s, t_1, [t_2, t_3]))$.
$osc_4 - NSExt^*$	Returns the timestamps and the signal values of the records at which the only two strict extrema occur in the signal: $\langle t_1, f_p(t_1, s), t_4, f_p(t_4, s)\rangle \mid \exists t_2, t_3 \in [t_l, t_u] : t_l \leq t_2 < t_1 \wedge t_1 < t_4 \wedge t_4 < t_3 \leq t_u \wedge (smin(s, t_1, [t_2, t_4]) \vee smax(s, t_1, [t_2, t_4])) \wedge (smin(s, t_4, [t_1, t_3]) \vee smax(s, t_4, [t_1, t_3])) \wedge t_4 \neq t_1$.
$osc_5 - NSExt$	The first and the last timestamps (t_1 and t_2) delimiting the exact interval throughout which the signal is flat: $\langle [t_1, t_2], v\rangle \mid t_1 \leq t_l \wedge t_u \leq t_2 \wedge \forall t_3 \in [t_1, t_2] : (f_p(t_3, \langle s\rangle) = v \wedge \neg(\exists t_4 \in [t_f, t_1) : (\forall t_5 \in (t_4, t_1) : (f_p(t_5, \langle s\rangle) = v))) \wedge \neg(\exists t_6 \in (t_2, t_e] : (\forall t_7 \in (t_6, t_e] : (f_p(t_7, \langle s\rangle) = v))))$
$osc_6 - NSExt/$ $osc_7 - NSExt$	Returns the last seen maximum ($f_p(t_1, s)$) and the first seen minimum ($f_p(t_2, s)$) values of the signal and respectively the first (t_1) and the last (t_2) timestamps at which they are reached: $\langle \langle t_1, f_p(t_1, s)\rangle, \langle t_2, f_p(t_2, s)\rangle\rangle \mid tl \leq t_1 < t_2 \leq tu \wedge \forall t_3 \in [tl, t_1)(f_p(t_3, s) = f_p(t_1, s)) \wedge \forall t_4 \in (t_1, t_u](f_p(t_4, s) < f_p(t_1, s)) \wedge \forall t_5 \in [t2, t_u)(f_p(t_5, s) = f_p(t_2, s)) \wedge \forall t_6 \in [t_l, t_2)(f_p(t_6, s) > f_p(t_2, s))$

* $osc(\langle s\rangle, t_1, t_2, t_3, t_4, t_5) = (uni\_sm\_min(s, t_2, [t_1, t_3]) \wedge uni\_sm\_max(s, t_3, [t_2, t_4]) \wedge uni\_sm\_min(s, t_4, [t_3, t_5])) \vee (uni\_sm\_max(s, t_2, [t_1, t_3]) \wedge uni\_sm\_min(s, t_3, [t_2, t_4]) \wedge uni\_sm\_max(s, t_4, [t_3, t_5]))$
$p2pv(\langle s\rangle, t_1, t_2, t_3) = |\max(|f_p(t_2, \langle s\rangle) - f_p(t_1, \langle s\rangle)|, |f_p(t_2, \langle s\rangle) - f_p(t_3, \langle s\rangle)|) - \langle v_1\rangle|$
$periodv(\langle s\rangle, t_1, t_2) = t_2 - t_1$

Note that we consider two more *diagnostics patterns* for oscillations (osc_6 and osc_7), which are not in figure 6.10, because they are the same as the $spk_4 - NSExt$ and $spk_5 - NSExt$ *violation details* of the spike behavior in figure 6.8.

6.5 Implementation and Preliminary Evaluation

To implement our TD-SB-TemPsy, we adopted a one-to-one mapping from the formal semantics of the *diagnostics patterns* (e.g., *diagnostics pattern* spk_1 in table 6.1) and the corresponding *diagnostics information* to OCL code (see example in figure 6.3).

We conducted a preliminary evaluation to assess the applicability of TD-SB-TemPsy. Specifically, to evaluate the performance of the implementation of TD-SB-TemPsy, we used a subset of the pre-processed traces used to evaluate our trace-checking approach SB-TemPsy-Check, and

a subset of properties for which SB-TemPsy-Check reported a violation (see chapter 5 for more details),

Methodology. For each pattern type discussed in this thesis, we selected one property and checked it on traces of different length (defined in terms of the number of entries). Each of these traces exhibits an unexpected behavior (signal shape) characterized as a *diagnostics pattern*. We run TD-SB-TemPsy on each pair of ⟨ property, trace ⟩ and measured its execution time. We set a timeout of 10 minutes.

Results. The preliminary results are show in table 6.11; each row indicates the property id, the pattern type used in the property, the number of entries in the trace, the violation type (column **V-type)**, and the execution time. These results show that:

- the evaluation of *event-based data assertion* properties ($p4$ and $p15$) identified the same violation type NSE in a short execution time: less than one second for $p4$ on a trace with 4042 entries, and $1.57\,\mathrm{s}$ for $p15$ on a trace of 8714 entries;
- the execution time to identify a violation type NSS for *state-based data assertion* ($np37$), *rise time* ($pRt1$) and *overshoot* properties ($pOsh1$) ranged from $4.19\,\mathrm{s}$ for $np37$ over a trace with 208 entries to $98.12\,\mathrm{s}$ over a trace with 2137 entries for $pRt1$. However, the tool timed out for ($np37$) over a larger trace with 8714 entries, with the same violation type (NSS)
- the execution time to identify a violation type SDS for *state-based data assertion* ($np37$), *rise time* ($pRt1$) and *overshoot* properties ($pOsh1$) ranged from $13.4\,\mathrm{s}$ for $np37$ over a trace with 208 entries to $59.21\,\mathrm{s}$ over a trace with 230 entries for $pOsh1$.
- the execution time to identify a violation type NSM for *rise time* ($pRt2$) and *overshoot* ($pOsh2$) properties ranged from $50.63\,\mathrm{s}$ for $pRt2$ over a trace with 36 entries to $62.22\,\mathrm{s}$ over a trace with 50 entries for $pOsh2$.
- the execution time to identify a violation type $NSExt$ for *spike* and *oscillations* properties ($pSpk1$, $pSpk2$, and $pOsc1$) ranged from $0.30\,\mathrm{s}$ over a trace with 10 entries to $124.31\,\mathrm{s}$ over a trace with 70 entries. The difference in execution time is due to the complexity of the different *spike* and *oscillations diagnostics patterns* formulations (and the corresponding *diagnostics information*).
- identifying a violation type NSF_a for a *spike* property ($pSpk1$) over a trace with only 4 entries took $75.50\,\mathrm{s}$. However, identifying the same violation type for the same property on a slightly longer trace (with 10 entries) timed out. Similar considerations can be made for identifying a violation type NSF_w for a *spike* property ($pSpk2$).
- identifying violation types NSF_{p2p} and NSF_p for an *oscillations* property ($pOsc1$) over an execution trace with 5 entries timed out.

Overall, the preliminary evaluation of TD-SB-TemPsy shows that our approach can identified *diagnostics information* for several violation types, except for the *spike* (NSF_a, NSF_w) and

Table 6.11: Preliminary Evaluation Results

property id	(pattern- violation)-Type	#Entries	V-type	execution time (s)
p4	DA-NSE	4042	NSE	0.44
p15	DA-NSE	8714	NSE	1.57
np37	DA-NSS	208	NSS	4.19
np37	DA-NSS	8641	-	-
np37	DA-NSS	208	SDS	13.4
pRt1	RT-NSS	2137	NSS	98.12
pRt1	RT-SDS	56	SDS	2.06
pRt1	RT-SDS	230	SDS	52.73
pRt2	RT-NSM	36	NSM	50.63
pOsh1	OSH-NSS	2137	NSS	39.66
pOsh1	OSH-SDS	230	SDS	59.21
pOsh2	OSH-NSM	50	NSM	62.22
pSpk1	SPK-NSExt	37	NSExt	1.89
pSpk1	SPK-NSExt	51	NSExt	3.94
pSpk1	SPK-NSExt	311	NSExt	0.47
pSpk1	SPK-NSFp2p	4	NSFa	75.50
pSpk1	SPK-NSFp2p	10	-	-
pSpk2	SPK-NSFp	4	NSFw	7.19
pSpk2	SPK-NSFp	10	-	-
pOsc1	OSC-NSExt	37	NSExt	1.69
pOsc1	OSC-NSExt	51	NSExt	3.79
pOsc1	OSC-NSExt	311	NSExt	1.21
pOsc1	OSC-NSExt	10	NSExt	0.30
pOsc1	OSC-NSExt	30	NSExt	5.84
pOsc1	OSC-NSExt	70	NSExt	124.31
pOsc1	OSC-NSExt	100	-	-
pOsc1	OSC-NSExt	10	NSExt	18.7
pOsc1	OSC-NSExt	30	NSExt	113.12
pOsc1	OSC-NSFp2p	5	-	-
pOsc1	OSC-NSFp	5	-	-

the *oscillations* (NSF_{p2p}, NSF_p) violations types. We plan to address this limitation in the future by optimizing the current OCL encoding for these four violation types to increase scalability, similarly to what we did for SB-TemPsy-Check (see chapter 5 for more details).

6.6 Related Work

The seminal work by Dou et al. on model-driven trace diagnostics [DBB18] is the closest one to TD-SB-TemPsy. The main difference is in terms of the supported language, since TD-SB-TemPsy supports SB-TemPsy-DSL, which is more expressive than *TemPsy* and tailored to the CPS domain. To the best of our knowledge, TD-SB-TemPsy is the first approach to provide a model-driven trace diagnostics for Signal-based Temporal Properties.

An error diagnostics algorithm for trace diagnostics of STL formulae is proposed in [FMN15]. The algorithm relies on identifying implicants, which characterize small sub-signals (as explanatory sub-models) that are sufficient to imply a property violation. By analogy, this is quite similar to the *diagnostics patterns* we defined for TD-SB-TemPsy, in the sense that we rely on the satisfaction of these patterns to conclude the violation of the whole property. However, in TD-SB-TemPsy we characterize undesired behaviors that imply the property violation, rather than looking for a minimal subformula (i.e., an implicant) that implies that violation.

The algorithm in [FMN15] has been adopted by many approaches that solve the trace diagnostics problem for Signal-based Temporal Properties in the CPS domain, and integrated into several tools such as AMT2.0 [NLM+18], CPSDebug [BMM+19] and the Breach extension using IA-STL formalism [FND+19]. For example, Bartocci et al. [BFMN18] propose *epoch diagnostic*, an automated procedure that provides additional context, reported as segments (also called *epoch*) that contribute to violation/ satisfaction of all the signals that are defined in the property. The work proposed in [FND+19] enhances the algorithm defined in [BFMN18] by supporting a robustness analysis that computes worst-case diagnostics. Such an enhancement relies on a robustness degree as a precise measurement of the property violation (e.g., capturing the distance between the actual signal value and the target threshold).

In the context of model checking, counterexamples of a property are widely adopted as a way to determine the reason for the violation of that property and to help with debugging. Some of these approaches automate the process by dynamically constructing counterexamples exploring possible (and best) witnesses that lead to the satisfaction of these counterexamples.(e.g., [CG07], [SQL04]). Counterexamples are similar to *diagnostics patterns*, in the sense that they characterize an undesired behavior that leads to the property violation. However, in TD-SB-TemPsy we do not generate our *diagnostics patterns*; we define them based on possible signal shapes that lead to their satisfaction.

6.7 Summary

Many trace checking tools yield Boolean verdicts, which do not provide enough information to understand the cause of a property violation. This makes the trace diagnostics activity tedious and ineffective, especially when a property violation can be due to several, distinct causes.

In this chapter, we proposed TD-SB-TemPsy-Report, a model driven trace diagnostics approach for properties expressed in SB-TemPsy-DSL, which complements a trace checking tool like SB-TemPsy-Check and provide informative verdicts about the cause of a property violation. At the basis of TD-SB-TemPsy-Report there is a characterization of diagnostic patterns, which capture a significant class of signal behaviors that leads to the violation of the property of interest. We formalized these diagnostic patterns for each property type supported by SB-TemPsy-DSL; each diagnostic pattern is associated with the corresponding diagnostic information, which explains the cause of a property violation.

Chapter 7

Conclusions & Future Work

7.1 Conclusions

The goal of this thesis is to develop methods and tools for the specification and trace checking of two types of complex quantitative temporal properties:

- properties defined using aggregation operators;
- signal-based temporal properties from the Cyber Physical System (CPS) domain.

In this thesis we have made the following contributions towards the achievement of this goal:

(i) *TemPsy-AG*, an extension of the *TemPsy* language, which supports the most used service provisioning patterns identified in the study in [BGPS12];

(ii) TEMPSY-CHECK-AG, an extension of the model-driven approach TEMPSY-CHECK, which relies on an optimized mapping of temporal requirements written in *TemPsy-AG* into OCL constraints on a conceptual model of execution traces.

(iii) A comprehensive taxonomy of SBTPs describing signal behaviors in the CPS domain.

(iv) SB-TemPsy-DSL, an expressive specification language for SBTPs.

(v) SB-TemPsy-Check, an efficient model-driven trace-checking procedure for trace checking SBTPs, expressed in SB-TemPsy-DSL; the tool has been released[1] under an open-source

[1] `https://github.com/SNTSVV/SB-TemPsy`

license. We assessed the scalability of SB-TemPsy by verifying real properties derived from a case study of our satellite integrator partner on real execution-traces.

(vi) TD-SB-TemPsy-Report, a model-driven approach for trace diagnostics of SBTPs, to complement SB-TemPsy-Check.

Contributions (i) and (ii) provides a framework for specifying and verifying temporal properties with aggregation. Contributions (iii)–(vi) provides a comprehensive specification framework for defining the most common types of SBTPs in the CPS domain as well as a scalable trace checking procedure for such properties, complemented by informative verdicts.

Our technological contributions rely on *model-driven* approaches for trace checking and trace diagnostics. Such approaches consist in *reducing* the problem of checking (respectively, determining the diagnostics information of) a property ρ over an execution trace λ to the problem of evaluating an OCL (Object Constraint Language) constraint (semantically equivalent to ρ) on an instance (equivalent to λ) of a meta-model of the trace. The results — in terms of efficiency or our model-driven tools — presented in this thesis are in line with those presented in previous work [DBB17a, DBB18], and confirm that model-driven technologies can lead to the development of tools that exhibit good performance from a practical standpoint, also when applied in industrial contexts.

7.2 Future Research Directions

This dissertation sets the basis to follow different research directions in the future:

Extension of the taxonomy on signal-based temporal properties. We plan to extend the taxonomy proposed in chapter 4 by assessing the expressiveness of other temporal logics (such as SCL - Signal Convolution Logic [SNBB18], the extension of STL proposed in [BB19], and the *shape expressions* formalism [NQF+19]) in terms of the property types identified in the taxonomy.

Expressiveness of SB-TemPsy-DSL. We plan to extend SB-TemPsy-DSL with additional constructs, based on the expressiveness results of our evaluation.

Effectiveness of SB-TemPsy-Check. We are also going to develop alternative OCL definitions in SB-TemPsy-Check, optimized to minimize the number of timeouts when checking specific types of properties (e.g., properties with *order relationship between signals* pattern, properties with an *event-based* scope). We also plan to investigate how different implementations of SB-TemPsy-Check (e.g., using an SMT-based encoding as done in the Theodore [MVBB21], or another type of logic-based encoding relying on tools like R2U2 [MRS17]) fare with respect to the one based on OCL.

Distributed, model-driven trace checking. We plan to investigate the use of Big Data technologies to parallelize [SSA+19] our trace checking approaches implemented in TEMPSY-CHECK-AG and SB-TemPsy-Check, inspired by the existing work on distributed trace checking [BBG+16, BGK14, BCE+14, BKSB+12].

User studies. We plan to conduct several user studies to assess the usefulness of our languages (*TemPsy-AG* and SB-TemPsy-DSL) and tools (TEMPSY-CHECK-AG, SB-TemPsy-Check, and TD-SB-TemPsy-Report) for the application of trace checking in industrial contexts.

Bibliography

[Acı05] Nurettin Acır. Automated system for detection of epileptiform patterns in EEG by using a modified RBFN classifier. *Expert Systems with Applications*, 29(2):455–462, 2005.

[ACM02] Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *Journal of the ACM*, 49(2):172–206, March 2002.

[AF10] Y. S. R. Annapureddy and G. E. Fainekos. Ant colonies for temporal logic falsification of hybrid systems. In *Proc. 36th Annual Conference on IEEE Industrial Electronics Society (IECON2010)*, pages 91–96, Nov 2010.

[AFS+13] Houssam Abbas, Georgios Fainekos, Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta. Probabilistic temporal logic falsification of cyber-physical systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s):95, 2013.

[AG04] Nurettin Acır and Cüneyt Güzeliş. Automatic spike detection in EEG by a two-stage procedure based on support vector machines. *Computers in Biology and Medicine*, 34(7):561–575, 2004.

[AGL+15] Marco Autili, Lars Grunske, Markus Lumpe, Patrizio Pelliccione, and Antony Tang. Aligning qualitative, real-time, and probabilistic property specification patterns using a structured English grammar. *IEEE Trans. Softw. Eng.*, 41(7):620–638, 2015.

[AH15] Takumi Akazaki and Ichiro Hasuo. Time robustness in MTL and expressivity in hybrid system falsification. In *Proc. International Conference on Computer Aided Verification (CAV2015)*, pages 356–374. Springer, 2015.

[ALFS11] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. S-taliro: A tool for temporal logic falsification for hybrid systems. In *Proc. TACAS 2011*, pages 254–257, Berlin, Heidelberg, 2011. Springer.

[AMM+14] A. Adam, N. Mokhtar, M. Mubin, Z. Ibrahim, M. Z. M. Tumari, and M. I. Shapiai. Feature selection and classifier parameter estimation for EEG signal peak detection using gravitational search algorithm. In *Proc. 4th International Conference on Artificial Intelligence with Applications in Engineering and Technology (AIFU2014)*, pages 103–108, 2014.

[AOK+05] Nurettin Acir, Ibrahim Oztura, Mehmet Kuntalp, Baris Baklan, and Cuneyt Guzelis. Automatic detection of epileptiform events in EEG by a three-stage procedure based on artificial neural networks. *IEEE Transactions on Biomedical Engineering*, 52(1):30–40, 2005.

[ARB+17] Houssam Abbas, Alena Rodionova, Ezio Bartocci, Scott A Smolka, and Radu Grosu. Quantitative regular expressions for arrhythmia detection algorithms. In *Proc. International Conference on Computational Methods in Systems Biology (CMSB2017)*, pages 23–39. Springer, 2017.

[BB19] Alexey Bakhirkin and Nicolas Basset. Specification and efficient monitoring beyond STL. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS2019)*, pages 79–97. Springer, 2019.

[BBB19] Chaima Boufaied, Domenico Bianculli, and Lionel C. Briand. A model-driven approach to trace checking of temporal properties with aggregations. *Journal of Object Technology*, 18(2):15:1–15:21, 2019. doi:10.5381/jot.2019.18.2.a15.

[BBG+14] Marcello Maria Bersani, Domenico Bianculli, Carlo Ghezzi, Srđan Krstić, and Pierluigi San Pietro. SMT-based checking of SOLOIST over sparse traces. In *Proc. of FASE 2014*, volume 8411 of *LNCS*, pages 276–290. Springer, April 2014.

[BBG+16] Marcello Maria Bersani, Domenico Bianculli, Carlo Ghezzi, Srđan Krstić, and Pierluigi San Pietro. Efficient large-scale trace checking using mapreduce. In *Proc. ICSE2016*, pages 888–898. ACM, 2016.

[BBN13] Ezio Bartocci, Luca Bortolussi, and Laura Nenzi. A temporal logic approach to modular design of synthetic biological circuits. In *Proc. International Conference on Computational Methods in Systems Biology (CMSB2013)*, pages 164–177. Springer, 2013.

[BBNS15] Ezio Bartocci, Luca Bortolussi, Laura Nenzi, and Guido Sanguinetti. System design of stochastic models using robustness of temporal properties. *Theoretical Computer Science*, 587:3–25, 2015.

[BBS+14] Sara Bufo, Ezio Bartocci, Guido Sanguinetti, Massimo Borelli, Umberto Lucangelo, and Luca Bortolussi. Temporal logic based monitoring of assisted ventilation in intensive care patients. In *Proc. International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA2014)*, pages 391–403. Springer Berlin Heidelberg, 2014.

[BCE+14] David Basin, Germano Caronni, Sarah Ereth, Matúš Harvan, Felix Klaedtke, and Heiko Mantel. Scalable offline monitoring. In *Proc. RV2014*, volume 8734 of *LNCS*, pages 31–47. Springer, 2014.

[BCMT09] Ezio Bartocci, Flavio Corradini, Emanuela Merelli, and Luca Tesei. Model checking biological oscillators. *Electronic Notes in Theoretical Computer Science*, 229(1):41–58, 2009.

[BCW17] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice: Second Edition*. Morgan & Claypool Publishers, 2nd edition, 2017.

[BDD+18] Ezio Bartocci, Jyotirmoy Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Ničković, and Sriram Sankaranarayanan. Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In *Lectures on Runtime Verification*, pages 135–175. Springer, 2018.

[BDŠV14] Lubos Brim, P Dluhoš, D Šafránek, and Tomas Vejpustek. STL*: Extending signal temporal logic with signal-value freezing operator. *Information and Computation*, 236:52–67, 2014.

[BFFR18] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 1–33. Springer, 2018.

[BFHN18] Alexey Bakhirkin, Thomas Ferrère, Thomas A. Henzinger, and Dejan Ničković. The first-order logic of signals: Keynote. In *Proc. International Conference on Embedded Software (EMSOFT2018)*, EMSOFT '18, pages 1:1–1:10. IEEE Press, 2018.

[BFMN18] Ezio Bartocci, Thomas Ferrère, Niveditha Manjunath, and Dejan Ničković. Localizing faults in simulink/stateflow models with stl. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week)*, pages 197–206, 2018.

[BGK+13] Ezio Bartocci, Radu Grosu, Atul Karmarkar, Scott A. Smolka, Scott D. Stoller, Erez Zadok, and Justin Seyster. Adaptive runtime verification. In *Proc. International Conference on Runtime Verification (RV2013)*, pages 168–182. Springer Berlin Heidelberg, 2013.

[BGK14] Domenico Bianculli, Carlo Ghezzi, and Srđan Krstić. Trace checking of metric temporal logic with aggregating modalities using MapReduce. In *Proc. SEFM2014*, volume 8702 of *LNCS*, pages 144–158. Springer, 2014.

[BGKSP14] Domenico Bianculli, Carlo Ghezzi, Srdan Krstic, and Pierluigi San Pietro. Offline trace checking of quantitative properties of service-based applications. In *Proc. SOCA2014*, pages 9–16. IEEE, 2014.

[BGPS12] Domenico Bianculli, Carlo Ghezzi, Cesare Pautasso, and Patrick Senti. Specification patterns from research to industry: a case study in service-based applications. In *Proc. ICSE2012*, pages 968–976. IEEE, 2012.

[BGS13] Domenico Bianculli, Carlo Ghezzi, and Pierluigi San Pietro. The tale of SOLOIST: a specification language for service compositions interactions. In *Proc. FACS2012*, volume 7684 of *LNCS*, pages 55–72. Springer, 2013.

[BJB+20] Chaima Boufaied, Maris Jukss, Domenico Bianculli, Lionel Claude Briand, and Yago Isasi Parache. Signal-based properties: Taxonomy and logic-based characterization. *Journal of Systems and Software*, page 110881, 2020.

[BKMZ15] David Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zălinescu. Monitoring of temporal first-order properties with aggregations. *Formal methods in system design*, 46(3):262–285, 2015.

[BKSB+12] Benjamin Barre, Mathieu Klein, Maxime Soucy-Boivin, Pierre-Antoine Ollivier, and Sylvain Hallé. MapReduce for parallel trace validation of LTL properties. In *Proc. RV2012*, volume 7687 of *LNCS*, pages 184–198. Springer, 2012.

[BMB+20] Chaima Boufaied, Claudio Menghi, Domenico Bianculli, Lionel Briand, and Yago Isasi-Parache. Trace-checking Signal-based Temporal Properties: A model-driven approach. In *Proc. International Conference on Automated Software Engineering (ASE2020)*. IEEE, September 2020.

[BMM+19] Ezio Bartocci, Niveditha Manjunath, Leonardo Mariani, Cristinel Mateis, and Dejan Ničković. Automatic failure explanation in cps models. In *International Conference on Software Engineering and Formal Methods*, pages 69–86. Springer, 2019.

[BMS15] Luca Bortolussi, Dimitrios Milios, and Guido Sanguinetti. U-check: Model checking and parameter synthesis under uncertainty. In *Proc. Quantitative Evaluation of Systems (QEST2015)*, pages 89–104. Springer International Publishing, 2015.

[BOV+19] M. Bernaerts, B. Oakes, K. Vanherpen, B. Aelvoet, H. Vangheluwe, and J. Denil. Validating industrial requirements with a contract-based approach. In *Proc. MODELS 2019 (Companion))*, pages 18–27, Los Alamitos, CA, USA, 2019. IEEE.

[BVvF13] L. Brim, T. Vejpustek, D. Šafránek, and J. Fabriková. Robustness analysis for value-freezing signal temporal logic. In *Proc. Second International Workshop on Hybrid Systems and Biology (HSB2013)*, volume 125 of *Electronic Proceedings in Theoretical Computer Science*, pages 20–36. Open Publishing Association, 2013.

[CANZ19] Christoph Czepa, Amirali Amiri, Evangelos Ntentos, and Uwe Zdun. 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):3331–3371, 2019. doi:10.1007/s10270-019-00721-4.

[CFMS15] Fraser Cameron, Georgios Fainekos, David M Maahs, and Sriram Sankaranarayanan. Towards a verified artificial pancreas: Challenges and solutions for runtime verification. In *Proc. International Conference on Runtime Verification (RV2015)*, pages 3–17. Springer, 2015.

[CG07] Marsha Chechik and Arie Gurfinkel. A framework for counterexample generation and exploration. *International Journal on Software Tools for Technology Transfer*, 9(5-6):429–445, 2007.

[CGP10] Christian Colombo, Andrew Gauci, and Gordon Pace. Larvastat: Monitoring of statistical properties. In *Proc. RV2010*, volume 6418 of *LNCS*, pages 480–484. Springer, 2010.

[CP99] Marsha Chechik and Dimitrie O. Paun. Events in property patterns. In *Proc. 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking (SPIN1999)*, pages 154–167. Springer-Verlag, 1999.

[CP17] Christian Colombo and Gordon Pace. Runtime verification using larva. In *Proc. RV-CuBES2017*, volume 3 of *Kalpa Publications in Computing*, pages 55–63. EasyChair, 2017.

[CZss] C. Czepa and U. Zdun. On the understandability of temporal properties formalized in linear temporal logic, property specification patterns and event processing language. *IEEE Transactions on Software Engineering*, in press. doi: 10.1109/TSE.2018.2859926.

[DAC99] Matthew B Dwyer, George S Avrunin, and James C Corbett. Patterns in property specifications for finite-state verification. In *Proc. ICSE1999*, pages 411–420. ACM, 1999.

[DBB14] Wei Dou, Domenico Bianculli, and Lionel Briand. OCLR: a more expressive, pattern-based temporal extension of OCL. In *Proc. ECMFA 2014*, volume 8569 of *LNCS*, pages 51–66, Heidelberg, Germany, July 2014. Springer.

[DBB17a] Wei Dou, Domenico Bianculli, and Lionel Briand. A model-driven approach to trace checking of pattern-based temporal properties. In *Proc. MODELS2017*, pages 323–333. IEEE Computer Society, 2017.

[DBB17b] Wei Dou, Domenico Bianculli, and Lionel Briand. TemPsy-Check: a tool for model-driven trace checking of pattern-based temporal properties. In *Proc. RV-CuBES2017*, volume 3 of *Kalpa Publications in Computing*, pages 64–70. EasyChair, September 2017.

[DBB18] Wei Dou, Domenico Bianculli, and Lionel Briand. Model-driven trace diagnostics for pattern-based temporal specifications. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 278–288, 2018.

[DDD+15] Tommaso Dreossi, Thao Dang, Alexandre Donzé, James Kapinski, Xiaoqing Jin, and Jyotirmoy V. Deshmukh. Efficient guiding strategies for testing of temporal properties of hybrid systems. In *Proc. NASA Formal Methods (NFM2015)*, pages 127–142. Springer International Publishing, 2015.

[DDG+15] Jyotirmoy V. Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A. Seshia. Robust online monitoring of signal temporal logic. In *Proc. International Conference on Runtime Verification (RV2015)*, pages 55–70. Springer International Publishing, 2015.

[DDG+17] Jyotirmoy V. Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A. Seshia. Robust online monitoring of signal temporal logic. *Formal Methods in System Design*, 51(1):5–30, Aug 2017.

[DFG+11] Alexandre Donzé, Eric Fanchon, Lucie Martine Gattepaille, Oded Maler, and Philippe Tracqui. Robustness analysis and behavior discrimination in enzymatic reaction networks. *PloS one*, 6(9):e24246, 2011.

[DFM13] Alexandre Donzé, Thomas Ferrère, and Oded Maler. Efficient robust monitoring for STL. In *Proc. CAV 2013*, pages 264–279, Berlin, Heidelberg, 2013. Springer.

[DHF14] Adel Dokhanchi, Bardh Hoxha, and Georgios Fainekos. On-line monitoring for temporal logic robustness. In *Proc. International Conference on Runtime Verification (RV2014)*, pages 231–246. Springer, 2014.

[DHF15] Adel Dokhanchi, Bardh Hoxha, and Georgios Fainekos. Metric interval temporal logic specification elicitation and debugging. In *Proc. International Conference on Formal Methods and Models for Codesign (MEMOCODE2015)*, pages 70–79. IEEE, 2015.

[DJCF93] Alison A Dingle, Richard D Jones, Grant J Carroll, and W Richard Fright. A multistage system to detect epileptiform activity in the EEG. *IEEE Transactions on Biomedical Engineering*, 40(12):1260–1268, 1993.

[DM10] Alexandre Donzé and Oded Maler. Robust satisfaction of temporal logic over real-valued signals. In *Proc. International Conference on Formal Modeling and Analysis of Timed Systems (Formats2010)*, pages 92–106. Springer Berlin Heidelberg, 2010.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[DMB+12] Alexandre Donzé, Oded Maler, Ezio Bartocci, Dejan Nickovic, Radu Grosu, and Scott Smolka. On temporal logic and signal processing. In *Proc. International Symposium on Automated Technology for Verification and Analysis (ATVA2012)*, pages 92–106. Springer, 2012.

[Don10] Alexandre Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *Proc. International Conference on Computer Aided Verification (CAV2010)*, pages 167–170. Springer, 2010.

[Dou16] Wei Dou. *A Model-Driven Approach to Offline Trace Checking of Temporal Properties*. PhD thesis, University of Luxembourg, 2016. URL: `http://hdl.handle.net/10993/29184`.

[DRS82] Surya R Dumpala, S Narasimha Reddy, and Sushil K Sarna. An algorithm for the detection of peaks in biological signals. *Computer Programs in Biomedicine*, 14(3):249–256, 1982.

[DSI11] DSI consortium. DSI3 bus standard, February 2011.

[DSS+05] B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H.B. Sipma, S. Mehrotra, and Z. Manna. Lola: runtime monitoring of synchronous systems. In *Proc. TIME 2005*, pages 166–174. IEEE, june 2005.

[DZS+15] A. Dokhanchi, A. Zutshi, R. T. Sriniva, S. Sankaranarayanan, and G. Fainekos. Requirements driven falsification with coverage metrics. In *Proc. International Conference on Embedded Software (EMSOFT2015)*, pages 31–40, Oct 2015.

[Ecl20] Eclipse. Eclipse OCL tools. `https://projects.eclipse.org/projects/modeling.mdt.ocl`, 2020.

[EF07] Cindy Eisner and Dana Fisman. *A practical introduction to PSL*. Springer Science & Business Media, 2007.

[ESA20a] ESA. Building and testing spacecraft, 2020. URL: https://www.esa.int/Science_Exploration/Space_Science/Building_and_testing_spacecraft.

[ESA20b] The european space agency (esa), 2020. URL: https://www.esa.int/.

[exa20] exactearth, 2020. URL: https://www.exactearth.com/.

[Fer16] Thomas Ferrere. *Assertions and measurements for mixed-signal simulation*. PhD thesis, University of Grenoble, 2016.

[FHR13] Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. In *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 141–175. IOS Press, 2013.

[FKRT18] Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. A taxonomy for classifying runtime verification tools. In *Proc. RV 2018*, volume 11237 of *Lecture Notes in Computer Science*, pages 241–262, Cham, 2018. Springer.

[FMN15] Thomas Ferrère, Oded Maler, and Dejan Ničković. Trace diagnostics using temporal implicants. In *Proc. ATVA2015)*, volume 9364 of *LNCS*, pages 241–258. Springer, 2015.

[FMNU15] Thomas Ferrere, Oded Maler, Dejan Ničković, and Dogan Ulus. Measuring with timed patterns. In *Proc. International Conference on Computer Aided Verification (CAV2015)*, pages 322–337. Springer, 2015.

[FND+19] Thomas Ferrère, Dejan Nickovic, Alexandre Donzé, Hisahiro Ito, and James Kapinski. Interface-aware signal temporal logic. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pages 57–66, 2019.

[FP06] Georgios E Fainekos and George J Pappas. Robustness of temporal logic specifications. In *Formal Approaches to Software Testing and Runtime Verification*, pages 178–192. Springer, 2006.

[FSS05] Bernd Finkbeiner, Sriram Sankaranarayanan, and Henny Sipma. Collecting statistics over runtime executions. *Formal Methods in System Design*, 27:253–274, 2005.

[FSUY12] Georgios E Fainekos, Sriram Sankaranarayanan, Koichi Ueda, and Hakan Yazarel. Verification of automotive control applications using s-taliro. In *Proc. American Control Conference (ACC2012)*, pages 3567–3572. Citeseer, 2012.

[FWH+17] Aaron W. Fifarek, Lucas G. Wagner, Jonathan A. Hoffman, Benjamin D. Rodes, M. Anthony Aiello, and Jennifer A. Davis. Spear v2.0: Formalized past ltl specification and analysis of requirements. In *Proc. NFM 2017*, pages 420–426, Cham, 2017. Springer International Publishing.

[GL06] Volker Gruhn and Ralf Laue. Patterns for timed property specifications. *Electron. Notes Theor. Comput. Sci.*, 153(2):117–133, 2006.

[GPMS20] Dimitra Giannakopoulou, Thomas Pressburger, Anastasia Mavridou, and Johann Schumann. Generation of formal requirements from structured natural language. In *Requirements Engineering: Foundation for Software Quality (REFSQ 2020)*, pages 19–35, Cham, 2020. Springer International Publishing.

[GPVN+18] Carlos Alberto Gonzalez Perez, Mojtaba Varmazyar, Shiva Nejati, Lionel Briand, et al. Enabling model testing of cyber-physical systems. In *Proc. 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS2018)*, pages 176–186, 2018.

[Häg95] Tore Hägglund. A control-loop performance monitor. *Control Engineering Practice*, 3(11):1543–1551, 1995.

[Hal16] Sylvain Hallé. When RV meets CEP. In *Proc. RV2016*, volume 10012 of *LNCS*, pages 68–91. Springer, 2016.

[HBA+14] Bardh Hoxha, Hoang Bach, Houssam Abbas, Adel Dokhanchi, Yoshihiro Kobayashi, and Georgios Fainekos. Towards formal specification visualization for testing and monitoring of cyber-physical systems. In *Proc. Int. Workshop on Design and Implementation of Formal Tools and Systems (DIFTS2014)*, pages 1–9, 2014.

[HDF18] Bardh Hoxha, Adel Dokhanchi, and Georgios Fainekos. Mining parametric temporal logic properties in model-based design for cyber-physical systems. *International Journal on Software Tools for Technology Transfer*, 20(1):79–93, 2018.

[HMF15] Bardh Hoxha, Nikolaos Mavridis, and Georgios Fainekos. Vispec: A graphical tool for elicitation of mtl requirements. In *Proc. IROS2015*, pages 3486–3492, Los Alamitos, CA, USA, 2015. IEEE.

[HPU17] K. Havelund, D. Peled, and D. Ulus. First order temporal logic monitoring with BDDs. In *Proc. FMCAD 2017*, pages 116–123, Los Alamitos, CA, USA, 2017. IEEE.

[JBG+15] Stefan Jakšić, Ezio Bartocci, Radu Grosu, Reinhard Kloibhofer, Thang Nguyen, and Dejan Ničković. From signal temporal logic to FPGA monitors. In *Proc. Formal Methods and Models for Codesign (MEMOCODE2015)*, pages 218–227. IEEE, 2015.

[JBGN16] Stefan Jakšić, Ezio Bartocci, Radu Grosu, and Dejan Ničković. Quantitative monitoring of STL with edit distance. In *Proc. International Conference on Runtime Verification (RV2016)*, pages 201–218. Springer International Publishing, 2016.

[JDDS15] Xiaoqing Jin, Alexandre Donzé, Jyotirmoy V Deshmukh, and Sanjit A Seshia. Mining requirements from closed-loop control models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(11):1704–1717, 2015.

[JDJS14] G. Juniwal, A. Donzé, J. C. Jensen, and S. A. Seshia. Cpsgrader: Synthesizing temporal logic testers for auto-grading an embedded systems laboratory. In *Proc. International Conference on Embedded Software (EMSOFT2014)*, pages 1–10, Oct 2014.

[Kan15] Aaron Kane. *Runtime monitoring for safety-critical embedded systems*. PhD thesis, Carnegie Mellon University, 2015.

[KC05] Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In *Proc. ICSE2005*, pages 372–381. ACM, 2005.

[KDJ+16] James Kapinski, Jyotirmoy Deshmukh, Xiaoqing Jin, Hisahiro Ito, and Ken Butts. Simulation-based approaches for verification of embedded control systems: An overview of traditional and advanced modeling, testing, and verification techniques. *IEEE Control Systems Magazine*, 36:45–64, 12 2016.

[KJD+16] James Kapinski, Xiaoqing Jin, Jyotirmoy Deshmukh, Alexandre Donze, Tomoya Yamaguchi, Hisahiro Ito, Tomoyuki Kaga, Shunsuke Kobuna, and Sanjit Seshia. St-lib: A library for specifying and classifying model behaviors. Technical report, SAE Technical Paper, 2016.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299, 1990.

[KT13] Bilal Kanso and Safouan Taha. Temporal constraint support for OCL. In *Proc. SLE 2012*, volume 7745 of *LNCS*, pages 83–103, Berlin, Heidelberg, 2013. Springer.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, May/June 2009.

[LS16] Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. The MIT Press, 2nd edition, 2016.

[Luc01] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[LVR19] Jianwen Li, Moshe Y Vardi, and Kristin Y Rozier. Satisfiability checking for mission-time ltl. In *Proc. CAV2019*, pages 3–22, Cham, 2019. Springer.

[LZY02] He Sheng Liu, Tong Zhang, and Fu Sheng Yang. A multistage, multimethod approach for automatic detection and classification of epileptiform EEG. *IEEE Transactions on biomedical engineering*, 49(12):1557–1566, 2002.

[MN04] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *Proc. FTRTFT2004*, pages 152–166. Springer, 2004.

[MN13] Oded Maler and Dejan Ničković. Monitoring properties of analog and mixed-signal circuits. *International Journal on Software Tools for Technology Transfer*, 15(3):247–268, 2013.

[MNBB18] R. Matinnejad, S. Nejati, L. Briand, and T. Bruckmann. Test generation and test prioritization for Simulink models with dynamic behavior. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[MNBYI20] Claudio Menghi, Shiva Nejati, Lionel C. Briand, and Parache Yago Isasi. Approximation-refinement testing of compute-intensive cyber-physical models: An approach based on system identification. In *Proc. ICSE 2020*, New York, NY, USA, 2020. ACM.

[MNGB19] Claudio Menghi, Shiva Nejati, Khouloud Gaaloul, and Lionel C. Briand. Generating automated and online test oracles for simulink models with continuous and uncertain behaviors. In *Proc. ESEC/FSE 2019*, pages 27–38, New York, NY, USA, 2019. ACM.

[MNP08] Oded Maler, Dejan Nickovic, and Amir Pnueli. Checking temporal properties of discrete, timed and continuous behaviors. In *Pillars of computer science2008*, pages 475–505. Springer, 2008.

[MRS17] Patrick Moosbrugger, Kristin Y. Rozier, and Johann Schumann. R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems. *Formal Methods in System Design*, 51:31–61, April 2017. doi:10.1007/s10703-017-0275-x.

[MVBB21] Claudio Menghi, Enrico Viganò, Domenico Bianculli, and Lionel C Briand. Trace-checking cps properties: Bridging the cyber-physical gap. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 847–859. IEEE, 2021.

[MVDS20] B. Meyers, H. Vangheluwe, J. Denil, and R. Salay. A framework for temporal verification support in domain-specific modelling. *IEEE Transactions on Software Engineering*, 46(4):362–404, 2020.

[Nic08] Dejan Nickovic. *Checking timed and hybrid properties: Theory and applications*. PhD thesis, Université Joseph-Fourier-Grenoble I, 2008.

[Nič15] Dejan Ničković. Monitoring and measuring hybrid behaviors. In *Proc. International Conference on Runtime Verification (RV2015)*, pages 378–402. Springer International Publishing, 2015.

[NKJ+17] Luan Viet Nguyen, James Kapinski, Xiaoqing Jin, Jyotirmoy V Deshmukh, Ken Butts, and Taylor T Johnson. Abnormal data classification using time-frequency temporal logic. In *Proc. 20th international conference on hybrid systems: Computation and control (HSCC2017)*, pages 237–242. ACM, 2017.

[NLM+18] Dejan Ničković, Olivier Lebeltel, Oded Maler, Thomas Ferrère, and Dogan Ulus. AMT 2.0: Qualitative and quantitative trace analysis with extended signal temporal logic. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS2018)*, pages 303–319. Springer, 2018.

[NM07] Dejan Nickovic and Oded Maler. AMT: A property-based monitoring tool for analog systems. In *Proc. International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS2007)*, pages 304–319. Springer Berlin Heidelberg, 2007.

[NN16] Thang Nguyen and Dejan Nikovi. Assertion-based monitoring in practice checking correctness of an automotive sensor interface. *Science of Computer Programming*, 118(C):40–59, March 2016.

[NQF+19] Dejan Ničković, Xin Qin, Thomas Ferrère, Cristinel Mateis, and Jyotirmoy Deshmukh. Shape expressions for specifying and extracting signal features. In *Proc. International Conference on Runtime Verification (RV2019)*, pages 292–309. Springer, 2019.

[NSF+10] Truong Nghiem, Sriram Sankaranarayanan, Georgios Fainekos, Franjo Ivancić, Aarti Gupta, and George J. Pappas. Monte-carlo techniques for falsification of temporal properties of non-linear hybrid systems. In *Proc. 13th ACM international conference on Hybrid systems: computation and control (HSCC2010)*, HSCC '10, pages 211–220. ACM, 2010.

[NY20] Dejan Ničković and Tomoya Yamaguchi. RTAMT: Online robustness monitors from STL. In *Proc. International Symposium on Automated Technology for Verification and Analysis (ATVA 2020)*. Springer, October 2020.

[OMG12] OMG. ISO/IEC 19507 (OCL v2.3.1). `http://www.omg.org/spec/OCL/ISO/19507/PDF`, April 2012.

[PMS+14] Miroslav Pajic, Rahul Mangharam, Oleg Sokolsky, David Arney, Julian Goldman, and Insup Lee. Model-driven safety analysis of closed-loop medical systems. *IEEE Transactions on Industrial Informatics*, 10(1):3–16, 2014.

[PPM+19] Yago Isasi Parache, Aleix Pinardell, Antonio Márquez, Christophe Molon-Noblot, Alexander Wagner, Marc Gales, and Miroslav Brada. The esail multipurpose simulator. Poster at the Workshop on Simulation and EGSE for Space Programmes (SESP 2019), 2019.

[Rap16] Nicolas Rapin. Reactive property monitoring of hybrid systems with aggregation. In *Proc. RV2016*, volume 10012 of *LNCS*, pages 447–453. Springer, 2016.

[RBFS08] Aurélien Rizk, Grégory Batt, François Fages, and Sylvain Soliman. On a continuous degree of satisfaction of temporal logic formulae with applications to systems biology. In *Proc. International Conference on Computational Methods in Systems Biology (CMSB2008)*, pages 251–268. Springer, 2008.

[SACO02] R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. Propel: an approach supporting property elucidation. In *Pro. ICSE 2002*, pages 11–21, Los Alamitos, CA, USA, 2002. IEEE.

[SDB+13] Szymon Stoma, Alexandre Donzé, François Bertaux, Oded Maler, and Gregory Batt. STL-based analysis of trail-induced apoptosis challenges the notion of type I/type II cell line classification. *PLoS computational biology*, 9(5):e1003056, 2013.

[SF12] Sriram Sankaranarayanan and Georgios Fainekos. Falsification of temporal properties of hybrid systems using the cross-entropy method. In *Proc. 15th ACM international conference on Hybrid Systems: Computation and Control (HSCC2012)*, pages 125–134. ACM, 2012.

[SJN+17] Konstantin Selyunin, Stefan Jaksic, Thang Nguyen, Christian Reidl, Udo Hafner, Ezio Bartocci, Dejan Nickovic, and Radu Grosu. Runtime monitoring with recovery of the SENT communication protocol. In *Proc. International Conference on Computer Aided Verification (CAV2017)*, pages 336–355. Springer, 2017.

[SNBB18] Simone Silvetti, Laura Nenzi, Ezio Bartocci, and Luca Bortolussi. Signal convolution logic. In *Proc. International Symposium on Automated Technology for Verification and Analysis (ATVA2018)*, pages 267–283. Springer International Publishing, 2018.

[SQL04] ShengYu Shen, Ying Qin, and Sikun Li. Localizing errors in counterexample with iteratively witness searching. In *International Symposium on Automated Technology for Verification and Analysis*, pages 456–469. Springer, 2004.

[SSA+19] 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. A survey of challenges for runtime verification from advanced application domains (beyond software). *Formal Methods Syst. Des.*, 54(3):279–335, 2019. doi:10.1007/s10703-019-00337-w.

[SW95] A Prasad Sistla and Ouri Wolfson. Temporal conditions and integrity constraints in active database systems. In *ACM SIGMOD Record*, volume 24, pages 269–280. ACM, 1995.

[UFAM14] Dogan Ulus, Thomas Ferrère, Eugene Asarin, and Oded Maler. Timed pattern matching. In *Proc. International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS2014)*, pages 222–236. Springer International Publishing, 2014.

[VBCG14] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos. Management of an academic hpc cluster: The ul experience. In *Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014)*, pages 959–967, Los Alamitos, CA, USA, July 2014. IEEE.

[YHF12] Hengyi Yang, Bardh Hoxha, and Georgios Fainekos. Querying parametric temporal logic properties on embedded systems. In *Proc. International Conference on Testing Software and Systems (IFIP2012)*, pages 136–151. Springer, 2012.