SENSE

SENSE: Semantic-based Explanation of Cyber-physical Systems

Deliverable 3.3: Anomaly and Event Detection Algorithms

Authors	:	Thomas Frühwirth, Tobias Schwarzinger, Marta Sabou
Dissemination Level	:	Public/Restricted
Due date of deliverable	:	30.06.2024
Actual submission date	:	Sept. 2024
Work Package	:	WP3
Type	:	Report
Version	:	1.1

Abstract

Deliverable D3.3 reports on anomaly and event detection algorithms originating from a variety of different research domains, including runtime verification, semantic web, digital signal processing, complex event processing, and data stream processing. The focus lies on simple events, which can be combined to form more complex event patterns if necessary. To make these event detection algorithms useable for domain experts, the deliverable presents a predefined set of signal properties that can be identified by event detection algorithms and organizes these signal properties into a taxonomy. In practical implementations, this approach allows domain experts to specify new event types based on an appropriate signal property and define only very few parameters of these signal properties.

The information in this document reflects only the author's views and neither the FFG nor the Project Team is liable for any use that may be made of the information contained therein. The information in this document is provided "as is" without guarantee or warranty of any kind, express or implied, including but not limited to the fitness of the information for a particular purpose. The user thereof uses the information at his/ her sole risk and liability.

SENSE

History

Version	Date	Reason	Revised by
1.0	05.09.2024	Final draft	
1.1	15.11.2024	Minor language improvements and additional clarifications	Tobias Schwarzinger, Marta Sabou

Author List

Project Partner	Name (Initial)	Contact Information
TU	Thomas Frühwirth (TF)	thomas.fruehwirth@tuwien.ac.at
TU	Tobias Schwarzinger (TS)	tobias.schwarzinger@tuwien.ac.at
WU	Marta Sabou (MS)	marta.sabou@wu.ac.at

Executive Summary

The SENSE project aims to explain events occurring in technical systems from the area of Smart Grid and Smart Buildings. The goal is to contribute to Austria's sustainability goals by making complex systems that underlie key (and often highly polluting) infrastructures more efficient and user-friendly through explanations of (anomalous) events occurring in those systems. The SENSE system to be developed in this project aims to make complex cyber-physical systems (CPS) more transparent and thereby improve the performance and user acceptance of such systems.

This deliverable focuses on simple event types relevant to cover the SENSE use cases as well as use cases in similar domains. These simple event types can be expressed in terms of signal properties. For example, a threshold violation event may be mapped to an "overshoot" signal property with a certain threshold that triggers an event if the threshold exceeded by the signal. Defining a new event type can thus be achieved by selecting an appropriate signal property and specifying the required parameters (e.g., a value for the threshold of the "overshoot" signal property). This approach provides a simple way for domain experts to define event types relevant to their use case, without knowledge of the underlying event detection method or tool.

Numerous approaches from various research domains, including runtime verification, semantic web, digital signal processing, complex event processing, and data stream processing exist that can be applied to detect events on data streams. For example, Signal Temporal Logic (STL) from the domain of runtime verification allows to express signal properties using temporal logic formulas, which then can be supplied to an STL software library alongside the data streams to detect the actual events. Unfortunately, STL is not expressive enough to define all events relevant for the SENSE use cases. As an alternative solution, the concept also supports the implementation of event detection methods, e.g. in the form of Python code, that are specific to certain event types.

Selecting an appropriate event detection method or tool for each event type is subject to numerous factors, including expressiveness, computational complexity, tool support and expert knowledge required for more sophisticated event type definitions than the ones covered by the taxonomy. Furthermore, multiple methods may be used to cover the same event type, offering optimization potential, which is left open as future work.

Table of Content

History 2
Author List 2
Executive Summary 3
Table of Content 4
List of Figures 6
List of Tables 6
1 Introduction 7
1.1 Purpose and Scope of the Document 7
1.2 Structure of the Document 8
2 Methodology 9
3 Event Type Definitions Based on Signal Properties 9
3.1 Taxonomy of Signal Properties 10
3.2 Parameter Definitions for Selected Signal Properties 11
3.3 Example Event Definitions for SENSE Use Cases 12
4 Event Detection Methods and Tools 14
4.1 Runtime Verification 14
4.1.1 Linear Temporal Logic (LTL) 15
4.1.2 Metric Temporal Logic (MTL) 15
4.1.3 Signal Temporal Logic (STL) 15
4.1.4 Signal Temporal Logic* (STL*) 16
4.1.5 Signal First-Order Logic (SFO) 16
4.2 Semantic Web 16
4.2.1 OntoEvent 16
4.2.2 C-SPARQL 17
4.2.3 EP-SPARQL 17
4.2.4 INSTANS 17
4.2.5 CQELS-CEP 17
4.2.6 RSEP-QL 17
4.2.7 STARQL 17
4.3 Digital Signal Processing 17
4.3.1 Cross-correlation 18
4.3.2 Dynamic Time Warping 18
4.4 Complex Event Processing 18
4.4.1 Cayuga 18
4.4.2 SASE+ 18
4.4.3 Siddhi 18
4.4.4 TESLA 19
4.4.5 ETALIS 19
4.5 Data Stream Processing 19
4.5.1 Apache Kafka 19
4.5.2 Apache Flink 19
5 Method Selection 21
6 Summary 23
List of Abbreviations 24
References 25

List of Figures

Figure 1 – SENSE Conceptual Components, Their Connections, and Relevant WPs 7
Figure 2 – SENSE Signal Properties Taxonomy, Adapted from [3] 10

List of Tables

Table 1 Partner's Involvement 7
Table 2 Parameter Definitions for Signal Properties Relevant for SENSE Use Cases 11
Table 3 Events Relevant for SENSE Use Cases and Their Mapping to Signal Properties 13
Table 4 Temporal Operators for Temporal Logics 14
Table 5 Event Detection Rule Templates Relevant for SENSE Use Cases 22
Table 6 Event Detection Rules for Exemplary SENSE Event Types 22

1 Introduction

1.1 Purpose and Scope of the Document

This deliverable summarizes the results of Task 3.3 of the SENSE conceptual components (cf. Figure 1). It interlinks with other tasks and work packages as follows: The deliverable covers concepts and methods for implementing the simple event detection module defined in the Auditable SENSE Architecture [1]. The selected methods cover all event types that have been identified in the definition of the use cases and user stories [2]. It thereby builds upon the semantic and time-series data basis established during Work Package 2. The resulting implementations of these event detection methods will contribute to the technology stack implementation as one of the core modules. Furthermore, the detected events form the basis on which explanations can be generated in Work Package 4.

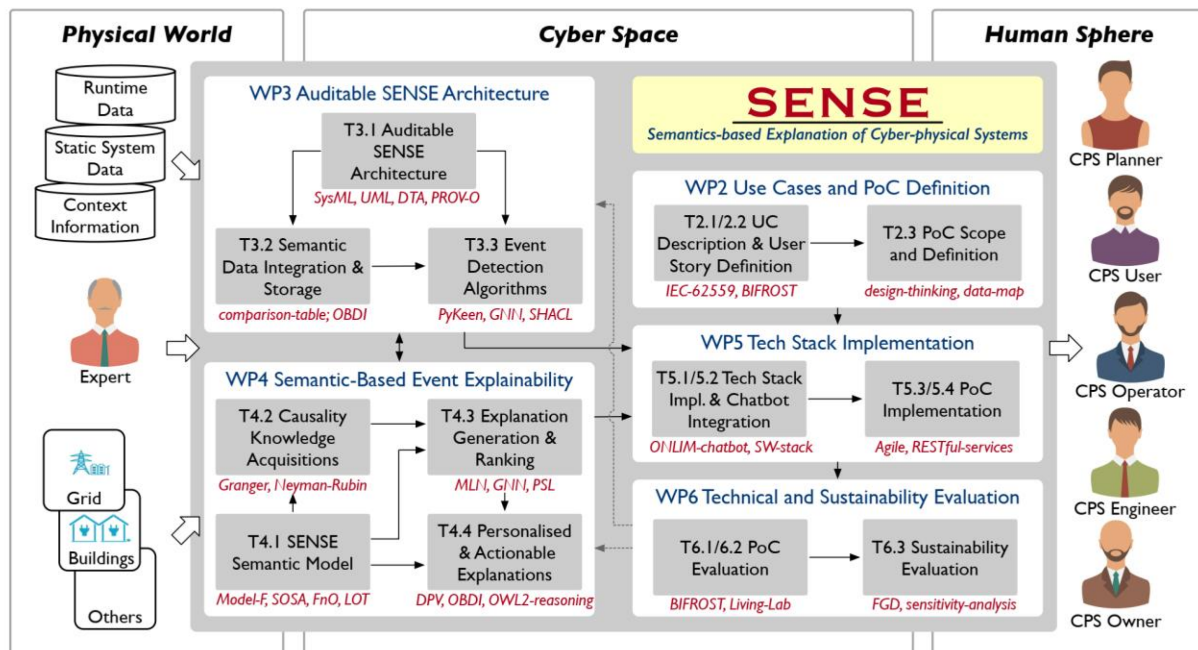

Figure 1 – SENSE Conceptual Components, Their Connections, and Relevant WPs

This deliverable incorporates many different requirements from all partners involved in the SENSE project. An overview of colleagues contributing to this work is summarized in Table 1.

Table 1 Partner's Involvement

Project Partner	Name (Initial)	Role/Tasks
WU	Marta Sabou (MS)	Project Coordination
WU	Katrin Schreiberhuber (KS)	Explainability
WU	Fajar Ekaputra (FE)	Explainability, Auditability
WU	Mevludin Memedi (MM)	Use case elicitation
TU Wien	Wolfgang Kastner (WK)	Project Coordination
TU Wien	Gernot Steindl (GS)	Architecture design
TU Wien	Thomas Frühwirth (TF)	Architecture design
TU Wien	Tobias Schwarzinger (TS)	Rule-based event detection
TU Wien	Mohammad Bilal (MB)	Model-based event detection

Siemens	Konrad Diwold (KD)	Supervisory
Siemens	Alfred Einfalt (AE)	Supervisory, Smart Grid use case expert
Siemens	Daniel Hauer (DH)	Smart Grid use case expert
Siemens	Juliana Kainz (JK)	Smart Grid use case expert
Siemens	Rob Poelmans (RP)	Smart Grid use case expert
Siemens	Gerhard Engelbrecht (GE)	Smart Grid use case expert
Siemens	Simon Steyskal (SS)	Smart Grid use case expert
AEE INTEC	Dagmar Jähnig (DJ)	Smart building use case expert
AEE INTEC	Christoph Moser (CM)	Smart building use case expert
MOOSMOAR Energies	Wolfgang Prüggler (WP)	Use case elicitation, economic considerations
Onlim	Ioan Toma (IT)	Knowledge-driven conversational interface
Onlim	Jürgen Umbrich	Knowledge-driven conversational interface

1.2 Structure of the Document

Section 2 describes the overall methodology applied in this deliverable. Section 3 covers the signal property taxonomy and parameters associated with each signal property. Section 4 then presents an overview of existing event detection methods and tools from various research domains. Section 5 exemplifies how the various signal properties can be mapped to event detection methods to detect events relevant for the SENSE use cases. Section 6 summarizes the main findings of this deliverable.

2 Methodology

The behavior of CPSs can often be verified by analyzing their input and output signals. To detect faults in the system, signal properties may be defined on the signals, which are then monitored by event detection algorithms. For example, systems that employ this technique in the domain of smart buildings are called Automatic Fault Detection and Diagnostic systems (AFDD). While the motivation in the SENSE system is slightly different, as we are not necessarily interested only in faults but in more general events for which an explanation shall be generated, techniques that allow monitoring signals for specific signal properties can also be applied in this context.

The SENSE system distinguishes between two types of events: simple and complex. Thereby, simple events are defined as events that can be identified purely on one or multiple time-series data streams. Complex events are defined on a higher-level as a combination of at least two simple and/or complex events. While there exist tools and concepts that try to cover both aspects, the vast amount of work focuses on either one of the two event types. Furthermore, events that were determined as relevant during the use case analysis are exclusively signal-based. We therefore focus on simple event detection in this deliverable, and only provide a rough overview on complex event detection techniques as a starting point for future use cases that might make use of this technique.

Event definitions for simple events are expressions over the shape of the underlying signals. For example, the system should identify an event if a signal exceeds a predefined threshold. There exist a lot of different techniques that can be applied to solve this task, each with their benefits and drawbacks. However, one commonality they share is that they need to be expressed in a method-specific language, with a trade-off between their expressive power and usability (in terms of easily understanding how to use them).

For this reason, this deliverable demonstrates a way of mapping these event type definitions to a predefined set of signal properties. Event types can then be specified by selecting an appropriate signal property and defining a set of parameters, hiding the underlying event detection method from the user that interacts with the SENSE system, both during the commissioning as well as during the operational phase of the CPS.

The required tasks are as follows: First, we define a taxonomy of signal properties, i.e., basic shapes of time-series data streams that may be used for event definitions (Section 3). Secondly, we investigate and provide an overview of existing event detection methods and tools (Section 4). And lastly, we identify a suitable event detection method for each of the signal properties that are relevant to implement SENSE use cases (Section 5).

3 Event Type Definitions Based on Signal Properties

While properties of signals that lead to events in CPSs might be arbitrarily complex, in practice the vast number of simple events can be mapped to one specific or a combination of relatively simple signal properties. A non-exhaustive list of such properties are threshold violations, peaks, dips, trends, and oscillations. It is therefore useful to provide a list or a taxonomy of predefined properties for a system engineer to select from.

3.1 Taxonomy of Signal Properties

In [3], Boufaied et al. provide such a taxonomy of signal properties of CPSs. This taxonomy served as a basis for the taxonomy illustrated in Figure 2. Only the leaf elements of the taxonomy highlighted in green should be considered properties that can be used to identify events, internal (white) nodes only provide structure to the taxonomy. The taxonomy is relatively simple but already covers a wide range of practically relevant properties. The meanings of the entries under Signal Property are covered in [3].

We adapted the taxonomy in the following ways. Firstly, we decided to exclude the "Data Assertion" type; it is, therefore, greyed out. This is because in [3] "Data Assertion" is defined in terms of a Signal First-Order Logic (SFO) (cf. Section 16) expression. However, currently no implementation of SFO exists. We generalized this idea by introducing the "CustomDefinition", that can be used to define more specific properties than the ones covered by the taxonomy otherwise. Secondly, we added the "OutOfBounds" and "Within Bounds" properties. In general terms, an event that is based on an OutOfBounds property will be fired if the signal overshoots or undershoots a pre-defined value interval. Likewise, an event based on the WithinBounds property will be fired if the signal enters a pre-defined value interval. Finally, we applied some renaming and minor modifications regarding the definitions and the parameters of some signal properties to better match the terminology of SENSE.

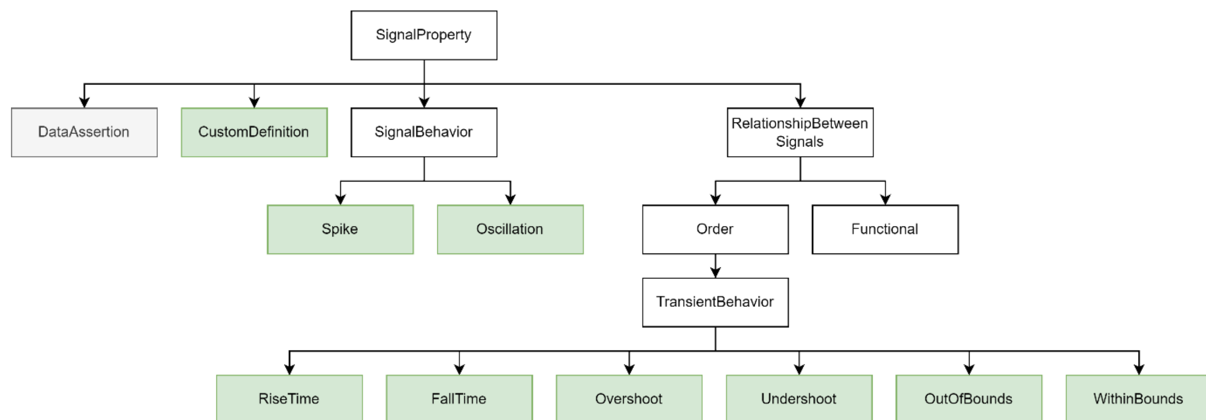

Figure 2 – SENSE Signal Properties Taxonomy, Adapted from [3]

3.2 Parameter Definitions for Selected Signal Properties

In general, creating event type definitions requires defining multiple parameters of the underlying signal property in the time and value domains. These parameters are also covered in [3]. Table 2 summarizes the parameters of signal properties that are relevant for SENSE. Note that we adapted the parameter names to better convey their meaning in expense of increased length of parameter names. We further added default values in square brackets after the parameter name, where appropriate.

Table 2 Parameter Definitions for Signal Properties Relevant for SENSE Use Cases

Signal Property	Parameter Definitions	Notes
Overshoot	• *threshold*: an event will be fired if the signal rises above the *threshold* value under the limitations of the remaining parameters • *overshoot_margin [0]*: an event will only be fired if the signal rises above the *threshold* by at least the margin defined by *overshoot_margin* • *overshoot_interval [0]*: an event will only be fired if the signal rises above the *threshold* for at least the duration specified by *overshoot_interval*	for the most basic overshoot event, which is a simple threshold violation, *overshoot_interval* and *overshoot_margin* are both set to 0
Undershoot	• *threshold*: an event will be fired if the signal falls below the *threshold* value under the limitations of the remaining parameters • *undershoot_margin [0]*: an event will only be fired if the signal falls below the *threshold* by at least the margin defined by *undershoot_margin* • *undershoot_interval [0]*: an event will only be fired if the signal falls below the *threshold* for at least the duration specified by *undershoot_interval*	
OutOfBounds	• *upper_threshold*: an event will be fired if the signal rises above the *upper_threshold* under the limitations of the remaining parameters • *upper_threshold_margin [0]*: an event will only be fired if the signal rises above the *upper_threshold* by at least the margin defined by *upper_threshold_margin* • *upper_threshold_interval [0]*: an event will only be fired if the signal rises above the *upper_threshold* for at least the duration specified by *upper_threshold_interval* • *lower_threshold*: an event will be fired if the signal falls below the lower_*threshold* under the limitations of the remaining parameters • *lower_threshold_margin [0]*: an event will only be fired if the signal falls below the *lower_threshold* by at least the margin defined by lower_threshold_*margin* • *lower_threshold_interval [0]*: an event will only be fired if the signal falls below the *lower_threshold* for at least the duration specified by *lower_threshold_interval*	an OutOfBounds event is simply a combination of Overshoot OR Undershoot
WithinBounds	• *upper_threshold:* an event will be fired if the signal falls below the *upper_threshold* under the limitations of the remaining parameters	

	• *upper_threshold_margin [0]:* an event will only be fired if the signal falls below the *upper_threshold* by at least the margin defined by *upper_threshold_margin* • *upper_threshold_interval [0]:* an event will only be fired if the signal falls below *upper_threshold – upper_threshold_margin* for at least the duration specified by *upper_threshold_interval* • *lower_threshold:* an event will be fired if the signal falls below the *lower_threshold* under the limitations of the remaining parameters • *lower_threshold_margin [0]:* an event will only be fired if the signal rises above the *lower_threshold* by at least the margin defined by *lower_threshold_margin* • *lower_threshold_interval [0]:* an event will only be fired if the signal rises above the *lower_threshold + lower_threshold_margin* for at least the duration specified by *lower_threshold_interval*	
RiseTime	• *delta*: an event will only be fired if the signal rises by at least the value defined by *delta* under the limitations of the remaining parameters • *rise_time*: an event will only be fired if the signal rises by *delta* within the time interval defined by *rise_time*	
FallTime	• *delta*: an event will only be fired if the signal falls by at least the value defined by *delta* under the limitations of the remaining parameters • *fall_time*: an event will only be fired if the signal falls by *delta* within the time interval defined by *fall_time*	
Custom Definition	• *implemented_in:* this parameter refers to an identifier (URI) of a custom implementation of an event detection rule	this event type allows to define a custom monitor for very specific simple events that cannot easily be expressed with events provided by the taxonomy

In addition, each of the parameters can either be set to a fixed value (LiteralValue) or be determined by another signal (SensorValue). For example, the user might want the system to fire an event when the outside temperature exceeds a predefined threshold, e.g., 35°C. Therefore, the user defines a new event type based on the Overshoot signal property, sets the type of the *threshold* parameter to LiteralValue, and its value to 35. Alternatively, the user might want the system to fire an event whenever the power consumption of a device exceeds the operating envelope. Thereby, the operating envelope is defined by a signal that is input to the SENSE system via the data ingestion module [1]. Therefore, the user defines a new event type based on the Overshoot signal property, sets the type of the *threshold* parameter to SensorValue, and its value to the name of the corresponding sensor.

3.3 Example Event Definitions for SENSE Use Cases

To underline the suitability of a signal property taxonomy for covering most practically occurring event types, Table 3 provides a truncated list of event types that have been identified during the definition of SENSE use cases and user stories [2] and their mappings to

elements of the signal property taxonomy. All other event types identified during this phase fall into the same set of event types as the ones defined in Table 3.

Table 3 Events Relevant for SENSE Use Cases and Their Mapping to Signal Properties

Event Name	Description	Underlying Signal Property
DemandEnvelopeViolated	the current active power consumption is above the Operating Envelope (OE)	Overshoot
BatterySocLow	the current State of Charge (SOC) of a peak-shaving battery is below a certain threshold	Undershoot
BatteryNotUsed	the battery is neither actively charging nor discharging, i.e., net power flow, either being fed into the battery (charging) or extracted from the battery (discharging), is minimal and remains within a narrow range around zero	WithinBounds
BatteryApDropped	the active power draw of the battery drops by a specific value within a given time interval, indicating that an EV has been disconnected	FallTime

4 Event Detection Methods and Tools

A large variety of event detection methods/formalisms may be applied to monitor the various signal properties defined in the taxonomy in the previous section. The methods originate from different domains. Some methods are driven by their underlying formalisms and others are more pragmatically driven by the need for implementations, often within the context of an existing framework for a specific task. In the following, we briefly introduce approaches structured by their originating domain, as approaches from the same domain typically share the underlying concepts and only differ in expressiveness, capabilities, or efficiency.

4.1 Runtime Verification

This group employs the most formal underlying theory of the approaches presented in this deliverable. It originates from the formal verification of systems, either during system design or runtime. The underlying idea is the definition of properties a system has to fulfill based on formal logic. Depending on the complexity of the system, it might either be possible to formally proof during design that the system operates within the given specification in all circumstances or monitor the system during simulation and runtime.

Many of the approaches in this category are variants of temporal logic. Temporal logic is a type of modal logic that is used to describe and reason about the behavior of systems over time. It allows for the expression of properties related to the order of events and the evolution of states in a system. Temporal logic formulas are built using a set of logical operators, such as conjunction, disjunction, negation, and temporal operators like 'always', 'eventually', 'until', and 'next'. These formulas can be used to specify and verify the correctness of system properties, such as safety, liveness, and fairness. Table 4 provides examples for typical temporal operators supported by most temporal logics.

Table 4 Temporal Operators for Temporal Logics

Operator	Symbol	Description	Example
Always	G	asserts that a property holds at all future times	G(p) means that p will always be true
Eventually	F	asserts that a property holds at some future time	F(p) means that p will eventually be true at some point in the future
Next	X	asserts that a property holds at the next time step	X(p) means that p will be true in the next time step
Until	U	asserts that a property holds until another property becomes true	Example: p U q means that p is true until q becomes true

The propositional variables, or simply propositions, p and q are statements that can be either true or false. As these operators originate from the domain of formal verification, their description might sound unintuitive for event specification at a first glance. However, their usefulness for this purpose can easily be demonstrated with a simple example: Assume that we want to ensure that a signal, e.g., a temperature value *T*, stays below a certain threshold, e.g., *25*. We can feed the formula defined in Equation 1 into a tool, alongside an array of all temperature readings collected so far. The tool will evaluate the temperature values against the formula and provide, as a result, an array stating if the formula holds for (satisfies) each entry of the temperature value array. Thus, a change in the result from “valid” to “invalid” is equivalent to an Overshoot of the temperature *T* value with threshold 25.

$$(T < 25)$$

Equation 1 Temporal Logic Formula Example

4.1.1 Linear Temporal Logic (LTL)

Linear Temporal Logic (LTL) is a formal system used for describing and reasoning about the behavior of concurrent and reactive systems over time. It allows for the expression of properties related to the evolution of states in a system. As a drawback in the context of simple event detection, LTL only supports logical operators and is thus limited to Boolean input values. Furthermore, LTL is a discrete-time logic, meaning it deals with systems that have a discrete sequence of states over time, but it cannot express timespans, as "holds true for at least two seconds".

- LTL can express: "the Boolean variable p will never be false"
- LTL cannot express: "the Integer variable T will never be lower than 25"
- LTL cannot express: "the Boolean variable p will never be false for more than 2 seconds"

4.1.2 Metric Temporal Logic (MTL)

Metric Temporal Logic (MTL) extends LTL by introducing quantitative timing constraints, allowing for the expression of time-bounded properties. MTL formulas include operators that specify time intervals, such as "within" and "since", which enable the description of temporal relationships between events with specific time bounds. This makes MTL more suitable for applications where precise timing requirements are crucial, such as real-time systems and CPSs. Still, MTL can only reason about Boolean propositions.

- MTL can express: "if the Boolean variable p becomes true, q has to also become true within 2 seconds"
- MTL cannot express: "if the Integer variable T rises above 25, it must return to lower than 25 within 2 seconds"

4.1.3 Signal Temporal Logic (STL)

Signal Temporal Logic (STL) is specifically designed for reasoning about continuous-time signals and their temporal properties. It includes operators for expressing properties related to the behavior of signals over time, such as "eventually", "always", "until", and "since". Additionally, STL supports the use of arithmetic operators and comparison operators, which allows for the description of more complex signal properties. However, like the other temporal logics discussed so far, STL cannot reference the value of a signal at a time instance when a certain property was satisfied. This is often required to compare signals to their previous values.

- STL can express: "if the Integer variable T rises above 25, it must return to lower than 25 within 2 seconds"
- STL cannot express: "if the Integer variable T rises by 2 within 1 second, it must return to its original value within 10 seconds"

4.1.4 Signal Temporal Logic* (STL*)

STL* [4] extends STL with the "freeze" which allows referencing to the value of a signal at a time instance when a certain property was satisfied. Otherwise, it is equivalent to STL. STL* is very expressive and suitable to define all event types identified to be relevant during the SENSE Use Case Definition. Its limitations are minor, for example, STL* cannot reference the value of a local extrema, nor can it use quantifiers in temporal logic operators or value variables.

- STL* can express: "if the Integer variable T rises by 2 within 1 second, it must return to its original value within 10 seconds"
- STL* cannot express: "if the Integer variable T rises above 25, it must return below its lowest value in the last 5 seconds within the next 10 seconds"
- STL* cannot express: "if the Integer variable T rises above 25, there must exist some value r around which T stabilizes within 10 seconds"

4.1.5 Signal First-Order Logic (SFO)

MTL, STL and STL* do not support the use of existential quantifiers for time and value variables, somewhat limiting their capability of expressing very complex signal properties. For example, "the bounded stabilization property requires the signal f to stabilize around some value of r, which can vary during the execution of the system" [5], which cannot be expressed without quantifiers on a value variable. This limitation is addressed with the definition of the Signal First-Order Logic (SFO) [5]. However, while SFO is the most expressive formal framework for specifying signal properties, there is currently no implementation available.

- SFO can express: "if the Integer variable T rises above 25, there must exist some value r around which T stabilizes within 10 seconds"

4.2 Semantic Web

SPARQL, as the default query language in the Semantic Web, is typically executed against a database, whereby the query engine processing the SPARQL query can only provide answers based on the information the database contains at a specific point in time. This raises several problems, e.g., the query has to be evaluated repetitively causing additional delay, duplicated events must be recognized causes additional overhead, and the same data needs to be processed multiple times. Therefore, [6] defines RDF streams, i.e., continuous streams of RDF encoded data. In the following, several approaches that fall into the category of Semantic Web approaches based on SPARQL extensions that operate on RDF streams are discussed.

4.2.1 OntoEvent

Similar to the logics-based approaches presented in Section 4.1., Ontology for Event Description (OntoEvent) [7] defines a language with temporal and logical operators to specify events. Furthermore, it provides a semantic model for expressing these event definitions directly as an OWL ontology. It supports both simple (in OntoEvent terminology called "primitive") events, as well as complex events, which are composition of primitive events.

4.2.2 C-SPARQL

The authors of [6] consider the use of SPARQL for event detection. Furthermore, they introduce Continuous SPARQL (C-SPARQL), which extends SPARQL with stream and timing semantics, as well as keywords to construct and describe such continuous queries. Shortcomings of C-SPARQL are the lack of support from CEP and inferencing.

4.2.3 EP-SPARQL

Event Processing SPARQL (EP-SPARQL) [8] supports the definition of event patterns to detect specific occurrences or combinations of events in the data stream. Furthermore, it adds inferencing capabilities.

4.2.4 INSTANS

Similar to the previous approaches, the authors of [9] also propose the use of SPARQL for event detection. The Incremental eNgine for STANding Sparql (INSTANS) event processing platform implements a continuous query execution engine. Furthermore, it optimizes execution by identifying query structures that are common among multiple running applications and reusing results of such query parts.

4.2.5 CQELS-CEP

Like INSTANTS, Continuous Query Evaluation over Linked Stream (CQELS) [10] implements the idea of continuous execution of queries on semantic/RDF data streams. It achieves this by extending the SPARQL grammar and defining data structures and algorithms for efficient implementation. The authors of [11] further extended CQELS with Complex Event Processing (CEP) capabilities, resulting in CQELS-CEP.

4.2.6 RSEP-QL

RSEP-QL is another RDF Stream Processing Query Language introduced in [12]. It improves upon EP-SPARQL and C-SPARQL and combines Data Stream Management System (DSMS) operators like windows as well as complex event processing features.

4.2.7 STARQL

In [13] Streaming and Temporal ontology Access with a Reasoning-based Query Language (STARQL) is introduced. STARQL is the query language for formulating events. However, it is part of a larger ecosystem of tools, including ExaStream as a backend for query processing and with Ontology-Based Data Access (OBDA) capabilities. Furthermore, OptiqueVQS is a graphical tool that supports end-users in the STARQL query formulation process by accessing and providing information about the ontology that shall be queried.

4.3 Digital Signal Processing

Another domain that heavily applies techniques for analyzing signals is Digital Signal Processing (DSP). While many operations in this domain deal with signal transformations, there exist also concepts that can be utilized for simple event detection, e.g., wavelet analysis, machine learning, filtering, anomaly detection, and pattern recognition. In the following, we briefly introduce cross-correlation and dynamic time warping as two examples of this domain.

4.3.1 Cross-correlation

Cross-correlation is a statistical measure used in signal processing to determine the similarity between two signals as a function of a time-lag applied to one of them. It is a mathematical operation that compares two signals by shifting one of them along the time axis and computing the correlation between the two signals at each position. An event type may be specified by defining the shape of the time series via a “template signal” and firing an event if the monitored signal is sufficiently similar, i.e., the cross-correlation value is sufficiently high at some point in time.

4.3.2 Dynamic Time Warping

Dynamic Time Warping (DTW) is an algorithm that finds an optimal alignment between two signals by warping (non-linearly stretching or compressing) the time dimension of one signal to match the other. It is a non-linear operation that allows for more flexibility in matching sequences with different lengths or speeds than cross-correlation.

4.4 Complex Event Processing

Assuming there exists a method for simple event detection, approaches exist that do not analyze the data directly but aim to analyze streams of events for specific patterns. Related terms in this context are event pattern matching or pattern matching over event streams. As simple event processing is currently powerful enough to cover the SENSE use cases, we only provide a brief overview of existing approaches to complex event processing in the following we found so far, but no comprehensive list.

4.4.1 Cayuga

Cayuga is a system that enables users to create subscriptions for multiple events. It has advanced features like parameterization and aggregation, which make it more expressive than regular pub/sub systems. The subscription language is based on specific operators, ensuring clear meanings and allowing for optimization opportunities. Cayuga is based on a Nondeterministic Finite State Automata (NFA) model [14], [15].

4.4.2 SASE+

SASE+ is a complex event language that allows for repeating patterns in event streams, known as Kleene closure. This language is useful in various applications like finance, inventory management, and healthcare monitoring. While Kleene closure is well-studied in regular expressions, its use in event streams has unique features. This paper proposes a compact language for defining Kleene closure patterns, creates a formal model to describe its semantics and expressiveness, and compares it to other languages in the field [16].

4.4.3 Siddhi

Siddhi is a complex event processing engine that efficiently processes and analyzes event streams in real-time. It is designed to handle high volumes of data and provides a flexible and powerful language for defining event patterns and rules. With its stream processing style architecture, Siddhi offers improved performance and scalability compared to traditional CEP engines. It employs SQL-like programming constructs, similar to Language Integrated Query (LINQ), to define queries on event streams, which are continuously evaluated by the Shiddi event processing engine [17].

4.4.4 TESLA

TESLA is a complex event specification language that allows users to define event patterns and rules for processing incoming data. It uses simple syntax and formal semantics based on first-order, metric temporal logic. TESLA offers high expressiveness and flexibility, providing content and temporal filters, negations, timers, aggregates, and customizable policies for event selection and consumption. The language is supported by an efficient event detection algorithm based on automata, making it suitable for various applications involving real-time event processing [18].

4.4.5 ETALIS

ETALIS [19] is a rule-based language for CEP with clear syntax and semantics. In this respect, it is similar to the logics-based approaches presented in Section 4.1. The paper defines atomic events as instantaneous. Complex events are defined as a combination of atomic and possibly complex events, whereby complex events are not instantaneous but span over a time interval. ETAILS then focuses on defining patterns of complex events, i.e., events may overlap, directly follow each other, or an event has to end before another event starts.

4.5 Data Stream Processing

Finally, there are sophisticated commercial and open-source solutions for transporting, monitoring and analyzing large streams of data. These are often proprietary solutions, and the corresponding event detection algorithms have to be implemented in a high-level programming language rather than focusing on an abstract concept for event definitions. Nevertheless, they can be useful in CPSs for data transport and for implementing highly customized event detection algorithms.

4.5.1 Apache Kafka

Apache Kafka[1] is an open-source distributed event streaming platform developed by the Apache Software Foundation. It is designed to handle real-time data feeds and allows for high-throughput, low-latency streaming of data. Producers send data to Kafka topics, which are divided into partitions, and consumers read data from these topics. Kafka ensures data reliability by replicating data across multiple brokers and allows for parallel processing of data through its partitioning and consumer group features. Kafka is often used as a message broker or a streaming platform for processing and analyzing large volumes of data in real-time.

As such, in particular with its Kafka Streams API, Kafka allows to efficiently process huge amounts of measurements and feed them to the relevant event detection monitors. The algorithm that performs event detection is not covered by Kafka but has to be implemented on a higher level.

4.5.2 Apache Flink

Apache Flink allows to define jobs/functions on data streams, which are then processes either in a local setup or, for high-volume stream processing, in cluster environments such as Kubernetes. It also offers a library for Complex Event Processing to specify patterns of events, which are then evaluated continuously on data streams. Apache Flink provides support for

[1] https://kafka.apache.org/

unbounded (continuous) and bounded (batched) data streams. Apache Flink supports Apache Kafka but also other streaming platforms as data sources and sinks.

The functionality of transformations and event detection has to be specified programmatically. However, Apache Flink provides numerous pre-defined operators to express SQL-like statements on data streams, enriched with operators to combine, group, and filter data streams. Furthermore, it can express certain timing characteristics.

5 Method Selection

Section 3.1 provided a taxonomy of signal properties relevant in many CPSs, including the parameters that need to be defined for each signal property. Section 4 presented numerous methods and tools for detecting events on data streams that match the corresponding signal property definitions. Selecting the most suitable method for detecting events of each type is subject to numerous factors:

- **Expressiveness:** Some of these methods are more capable in terms of expressiveness than others, meaning that there exist event types that cannot be detected by certain methods, as exemplified while discussing the various temporal logics in Section 4.1.
- **Computational Complexity:** Better expressiveness often comes at the cost of higher computational complexity. Optimization potentials as leveraged for example by INSTANS (cf. Section 4.2.4) further complicate the assessment of the overall efficiency.
- **Tool Support:** The maturity of available implementations of each event detection method greatly varies from full enterprise-grade software (e.g., Apache Flink, cf. Section 4.5.2), over very capable open-source implementations (e.g., STL, cf. Section 4.1.3) to "specification only" with no publicly available implementation (e.g., SFO, cf. Section 4.1.5).
- **Expert Knowledge:** As SENSE evolves, we will expand on the simple event detection implementations to support more sophisticated simple event types covering an even larger set of real-world scenarios. However, for certain use cases it may be necessary for the domain expert to define highly specialized events, requiring expert knowledge of the underlying event detection method.

As selecting the best event detection method is not objectively possible on a per-signal-property basis, the SENSE system does not restrict itself to any particular method for event detection. In fact, there might be multiple implementations for the same event type using alternative methods. This enables the possibility to automatically select the appropriate event detection technique based on the above properties. This concept, particularly investigating the optimization potential it offers, is considered future work. However, a preliminary analysis concluded that STL is a suitable method (considering the above factors) for detecting most event types relevant to the currently defined SENSE use [2].

From the relevant signal properties, only the FallTime property cannot be detected with STL. The reason for this shortcoming is that STL is not expressive enough to refer to the past values of a signal but only its current value. For example, it cannot express that a signal falls by a defined value, relative to its original value as there is no operator to refer to the original (past) value.

We have implemented two event detection methods in the SENSE system to demonstrate its flexibility. The SENSE system can evaluate STL specifications over sensor measurements using RTAMT [20], a library for monitoring STL formulas. Furthermore, we have implemented custom Python classes for monitoring signal properties that cannot be expressed in STL, as discussed in Section 4. Future work can extend the repertoire of supported event detection methods within the SENSE system.

Table 5 summarizes the signal properties that need to be detected to implement the SENSE use cases, the currently implemented method for detecting such signal properties, and a template formula that needs to be instantiated for each specific event type. Refer to Table 2

for a definition of the signal properties and their parameters. For example, the Overshoot signal property can be detected using STL with the provided formula. The formula states that an event is present if the signal goes beyond a certain threshold (potentially including an overshoot_margin) for a specified time (overshoot_interval).

Table 5 Event Detection Rule Templates Relevant for SENSE Use Cases

Signal Property	Method	Definition
Overshoot	STL	G[0,*overshoot_interval*](signal > *threshold* + overshoot_*margin*)
Undershoot	STL	G[0,*undershoot_interval*](signal < *threshold* - *undershoot_margin*)
WithinBounds	STL	G[0,*overshoot_interval*](signal < *threshold* + overshoot_*margin*) AND G[0,*undershoot_interval*](signal > *threshold* - *undershoot_margin*)
FallTime	CustomMonitor	FallTimeMonitor[2]

Finally, Table 6 provides example instantiations for each of the event detection rules. The event types have been specified in Table 3. The instantiated rules are used by the simple event detection module to detect events on the data streams. The Demand Envelope Violated event is an instantiation of the Overshoot property and monitors the active power of the entire garage. As the overshoot_interval and overshoot_margin parameters are zero, the previously discussed Overshoot rule template coincides with the simple threshold violation discussed in the natural language definition of the event. In addition, the Battery ApDropped event showcases how a custom monitor class can be instantiated with the parameters of a signal property.

Table 6 Event Detection Rules for Exemplary SENSE Event Types

EventType	Monitored Signal	Signal Property	Parameters	Instantiated Rule
Demand Envelope Violated	AP_Garage_Sensor	Overshoot	*threshold = (SensorValue) OE_Garage_Sensor*	G[0,0](AP_Garage_Sensor > OE_Garage_Sensor)
Battery SocLow	SOC_Battery_Sensor	Undershoot	*threshold = (LiteralValue) 15*	G[0,0](SOC_Battery_Sensor < 15)
Battery NotUsed	AP_Battery_Sensor	Within Bounds	*upper_threshold = (LiteralValue) 0.01* *lower_threshold = (LiteralValue) -0.01*	G[0,0](SOC_Battery_Sensor < 0.01) && G[0,0](SOC_Battery_Sensor < -0.01)
Battery ApDropped	AP_Battery_Sensor	FallTime	*delta* = (LiteralValue) 5 *fall_time* = (LiteralValue) 2	FallTimeMonitor(AP_Battery_Sensor, delta=5, fall_time=2)

[2] FallTimeMonitor is the name of the custom Python class that implements the detection of FallTime signal properties.

6 Summary

This deliverable presented a method for specifying event types without expert knowledge of the underlying event detection algorithm. It defined a taxonomy of signal properties, based on which a domain expert can define new event types by selecting an entry of the taxonomy and specifying the relevant parameters of the signal property. Furthermore, it covered a large variety of event detection methods and tools that can be applied to detect events on data streams based on the event type definitions.

As the event detection method responsible for detecting a specific event type is not predefined but can instead be determined by the SENSE system either a-priory or even during runtime, this opens a wide field for potential optimizations. Furthermore, whether a combination of these relatively simple event types should rather be monitored by a simple or a complex event detection algorithm adds to this optimization potential and will be investigated in future work.

List of Abbreviations

Short	Description
AFDD	Automated Fault Detection and Diagnostics
CEP	Complex Event Processing
CPS	Cyber-Physical System
CQELS-CEP	Continuous Query Evaluation over Linked Stream
C-SPARQL	Continuous SPARQL
DSMS	Data Stream Management System
EP-SPARQL	Event Processing SPARQL
INSTANS	Incremental eNgine for STANding Sparql
LINQ	Language Integrated Query
LTL	Linear Temporal Logic
MTL	Metric Temporal Logic
NFA	Nondeterministic Finite State Automata
OBDA	Ontology-Based Data Access
OE	Operating Envelope
RSEP-QL	RDF Stream Processing Query Language
SFO	Signal First-Order Logic
SOC	State of Charge
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
STARQL	Streaming and Temporal ontology Access with a Reasoning-based Query Language
STL	Signal Temporal Logic
STL*	Signal Temporal Logic Star
UML	Unified Modeling Language

References

[1] T. Frühwirth, G. Steindl, T. Schwarzinger and F. Ekaputra, "SENSE Deliverable 3.1 Auditable SENSE Architecture," 2024.

[2] D. Jähnig, C. Moser, T. Frühwirth, K. Schreiberhuber, J. Kainz, D. Hauer, K. Diwold and M. Sabou, "SENSE Deliverable 2.1: Definition of Use Cases and User Stories," 2023.

[3] C. Boufaied, M. Jukss, D. Bianculli, L. C. Briand and Y. I. Parache, "Signal-based properties of cyber-physical systems: Taxonomy and logic-based characterization," *Journal of Systems and Software,* p. 38, 2021.

[4] L. Brim, P. Dluhoš, D. Šafránek and T. Vejpustek, "STL*: Extending signal temporal logic with signal-value freezing operator," *Information and Computation,* pp. 52-67, 2014.

[5] A. Bakhirkin, T. Ferrère, T. Henzinger and D. Nickovic, "The first-order logic of signals," in *International Conference on Embedded Software (EMSOFT)*, 2018.

[6] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle and M. Grossniklaus, "C-SPARQL: SPARQL for Continuous Querying," in *Proceedings of the 18th international conference on World wide web*, 2009.

[7] M. Ma, L. Liu, Y. Lin, D. Pan and P. Wang, "Event Description and Detection in Cyber-Physical Systems: An Ontology-Based Language and Approach," in *IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, 2017.

[8] D. Anicic, P. Fodor, S. Rudolph and N. Stojanovic, "EP-SPARQL: a unified language for event processing and stream reasoning," in *Proceedings of the 20th international conference on World wide web*, 2011.

[9] M. Rinne, S. Törmä and E. Nuutila, "Data, SPARQL-Based Applications for RDF-Encoded Sensor," *SSN 904,* pp. 81-96, 2012.

[10] D. Le Phuoc, M. Dao-Tran, A. Le Tuan, M. N. Duc and M. Hauswirth, "RDF stream processing with CQELS framework for real-time analysis," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, 2015.

[11] M. Dao-Tran and D. Le Phuoc, "Towards Enriching CQELS with Complex Event Processing and Path Navigation," in *HiDeSt@ KI*, 2015.

[12] D. Dell'Aglio, M. Dao-Tran, J. P. Calbimonte, D. Le Phuoc and E. Della Valle, "A query model to capture event pattern matching in RDF stream processing query languages," in *European Knowledge Acquisition Workshop*, 2016.

[13] E. Kharlamov, T. Mailis, G. Mehdi, C. Neuenstadt, Ö. Özçep, M. Roshchin and A. Waaler, "Semantic access to streaming and static data at Siemens," *Journal of Web Semantics, 44,* pp. 54-74, 2017.

[14] A. Demers, J. Gehrke, M. Hong, M. Riedewald and W. M. White, "Towards expressive publish/subscribe systems," in *Advances in Database Technology-EDBT 2006: 10th International Conference on Extending Database Technology*, Munich, Germany, 2006.

[15] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma and W. M. White, "Cayuga: A General Purpose Event Monitoring System," *Cidr,* pp. 412-422, 2007.

[16] Y. Diao, N. Immerman and D. Gyllstrom, "Sase+: An agile language for kleene closure over event streams," UMass Technical Report, 2007.

[17] S. Suhothayan, K. Gajasinghe, I. Loku Narangoda, S. Chaturanga, S. Perera and V. Nanayakkara, "Siddhi: A second look at complex event processing architectures," in *Proceedings of the 2011 ACM workshop on Gateway computing environments*, 2011.

[18] G. Cugola and A. Margara, "TESLA: a formally defined event specification language," in *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, 2010.

[19] D. Anicic, P. Fodor, S. Rudolph, R. Stühmer, N. Stojanovic and R. Studer, "A rule-based language for complex event processing and reasoning," in *Web Reasoning and Rule Systems: Fourth International Conference*, Bressanone/Brixen, Italy, 2010.

[20] D. Ničković and T. Yamaguchi, "RTAMT: Online Robustness Monitors from STL," in *Automated Technology for Verification and Analysis*, 2020.