SENSE

SENSE

SENSE: Semantic-based Explanation of Cyber-physical Systems

Deliverable 5.1: Technology Stack Implementation

Authors	:	Thomas Frühwirth, Katrin Schreiberhuber, Mohammad Bilal, Tobias Schwarzinger, Konrad Diwold, Fajar Ekaputra
Dissemination Level	:	Public
Due date of deliverable	:	30.09.2024
Actual submission date	:	March 2025
Work Package	:	WP3
Type	:	Report
Version	:	1.0

Abstract

Deliverable D5.1 reports on the implementation of the SENSE technology stack and its integration with the ONLIM chatbot platform. It provides brief descriptions and web links to the GitHub repository of the stack, a template repository for creating new use cases, and a demo use case repository. Furthermore, the deliverable provides a user guideline on configuring the stack for a new use case and describes the interface and the steps necessary for integrating the ONLIM chatbot platform.

History

Version	Date	Reason	Revised by
1.0	28.01.2025	Final draft	MB

Author List

Project Partner	Name (Initial)	Contact Information
TU	Thomas Frühwirth (TF)	thomas.fruehwirth@tuwien.ac.at
WU	Katrin Schreiberhuber (KS)	katrin.schreiberhuber@wu.ac.at
TU	Mohammad Bilal (MB)	mohammad.bilal@tuwien.ac.at
TU	Tobias Schwarzinger (TS)	tobias.schwarzinger@tuwien.ac.at
WU	Fajar J. Ekaputra (FJE)	fajar.ekaputra@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 (CPSs) more transparent and thereby improve the performance and user acceptance of such systems.

This deliverable covers the technology stack implementation and focuses on

- the structure, interconnections, and links of the various GitHub repositories that make up or contribute to the technology stack
- the SENSE User Guideline that guides users through the process of applying SENSE to an existing CPS, a process we call instantiation, the result of which is a SENSE instance

The SENSE Core repository[1] constitutes the main contribution of this deliverable, including implementations of all SENSE modules defined in Deliverable 3.1 [1]. Furthermore, the SENSE Use Case Template repository[2] provides a starting point for instantiating new use cases, an appropriate file structure, templates of the necessary configuration files, and tools to input and convert use-case-specific information into the representation required by the SENSE Core. Finally, the SENSE Demo Use Case repository[3] delivers an exemplary use case that showcases the capabilities of SENSE but can also serve as a starting point for new implementations if preferred over the SENSE Use Case Template repository. More detailed information about the repositories is provided in the README files of the repositories themselves.

This deliverable also covers the SENSE User Guideline, which suggests a process for users to follow when applying SENSE to an existing CPS. The User Guideline consists of six consecutive steps. By following these steps, the user will generate a deep understanding of the goals, system, data, events, states, causes, and explanations within the system. Furthermore, the user will document their progress in a pre-defined Excel template, which is then used to instantiate SENSE for the new use case.

Finally, the deliverable demonstrates how the explanation-interface module of the SENSE Core enables interaction with conversational user interfaces, such as the ONLIM chatbot. It allows non-experts to query explanations and find the root cause of problems in a simple and intuitive manner. In this example, the chatbot backend is hosted on ONLIM's servers. Making the chatbot interface accessible to the users can easily be achieved via a very simple JavaScript snippet, as we exemplified on the project's website[4].

[1] https://github.com/wu-semsys/SENSE-Core
[2] https://github.com/wu-semsys/SENSE-Use-Case-Template
[3] https://github.com/wu-semsys/SENSE-Demo-Use-Case
[4] https://sense-project.net

Table of Content

List of Figures

List of Tables

1 Introduction

1.1 Purpose and Scope of the Document

This deliverable summarizes the results of Task 5.1 (Technology Stack Implementation) and Task 5.2 (Chatbot Integration) of the SENSE conceptual components (cf. Figure 1). Thereby, the results of Task 5.1 form the technical basis for use case implementations, including core functionalities such as data ingestion, event detection, and explanation generation. In addition, Task 5.2 establishes the link between the "Cyber Space" and the "Human Sphere" in Figure 1, enabling users of varying expertise to engage with the SENSE system via a very intuitive and user-friendly conversational interface. The deliverable interlinks with other tasks and work packages as follows.

The technology stack is an implementation of the software modules defined in the Auditable SENSE Architecture developed in Task 3.1 [1]. Details of the Semantic Data Integration & Storage concepts and the Event Detection Algorithms are described in the deliverables accompanying Task 3.2 [2] and Task 3.3 [3], respectively. Major contributions to the technology stack implementation covering the SENSE Semantic Model, Causality Knowledge Acquisitions, Explanation Generation & Ranking, and Personalised & Actionable Explanations also originate from Tasks 4.1 – 4.4 [4]. The development of the technology stack implementation has been driven by and verified with the SENSE Use Cases, primarily based on the PoC Scope and Definition originating from Task 2.3 [5]. The SENSE technology stack implementation, including its integration with the ONLIM chatbot platform, will serve as a basis for the PoC Implementation in Tasks 5.3 and 5.4, and ultimately in their Evaluation in Tasks 6.1 and 6.2.

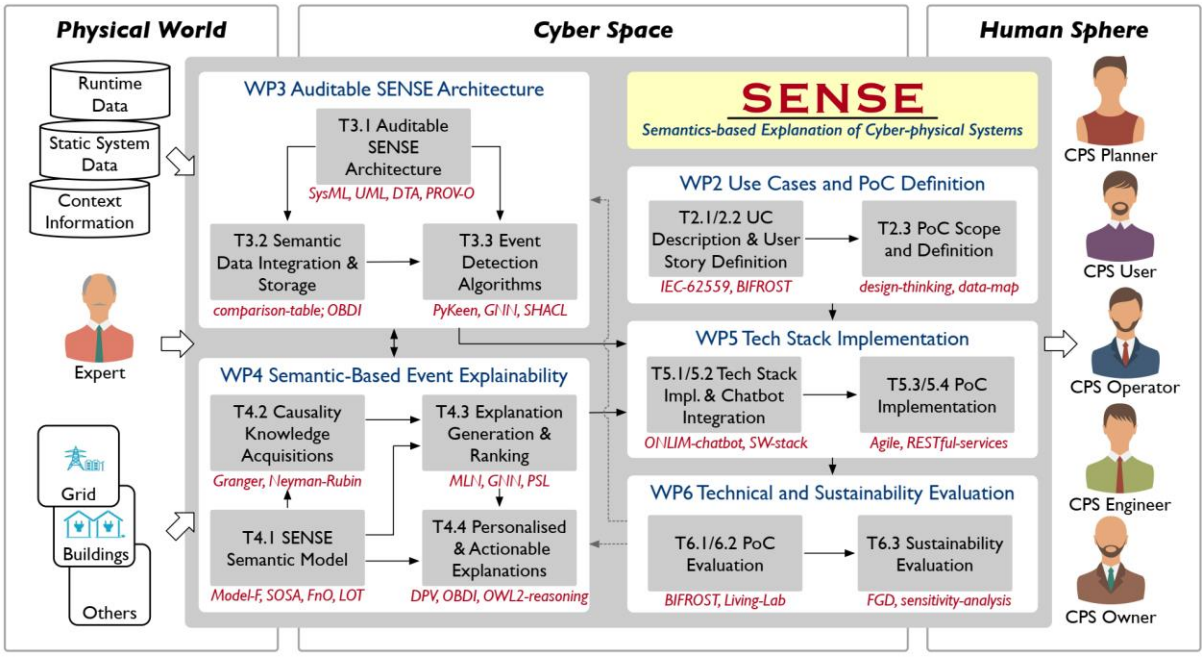

Figure 1: SENSE Conceptual Components, Their Connections, and Relevant WPs

This deliverable incorporates many different requirements from all partners involved in SENSE. 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

The remainder of this document is structured as follows: Section 2 summarizes the contents of the three main repositories contributing to the SENSE technology stack implementation and its usability: SENSE Core, SENSE Use Case Template, and SENSE Demo Use Case. It also provides links to the corresponding GitHub repositories. Section 3 provides a User Guideline that describes how to collect the necessary data to apply SENSE to an existing CPS. Section 4 covers the concepts and steps for the ONLIM chatbot integration. Finally, Section 5 provides a short summary of the contents of this document.

2 Overview of SENSE Repositories

The SENSE system currently comprises three repositories hosted on GitHub:

- SENSE Core: https://github.com/wu-semsys/SENSE-Core
- SENSE Use Case Template: https://github.com/wu-semsys/SENSE-Use-Case-Template
- SENSE Demo Use Case: https://github.com/wu-semsys/SENSE-Demo-Use-Case

An overview of the SENSE repositories is provided in the next three subsections. For additional technical information, refer to the README files of the GitHub repositories.

2.1 SENSE Core

The SENSE Core repository implements the SENSE architecture [1]. It primarily contains the source code of all SENSE Core modules and instructions in the form of Dockerfiles[5] for building images for each of the modules. Furthermore, it provides additional information on detailed concepts and functionalities of each module. The SENSE Core repository typically does not require any changes by the user of SENSE. However, it is still necessary to download/clone the SENSE Core repository to create Docker images of all modules. The SENSE Core structure is illustrated in Figure 2 in the form of a C4 container diagram[6].

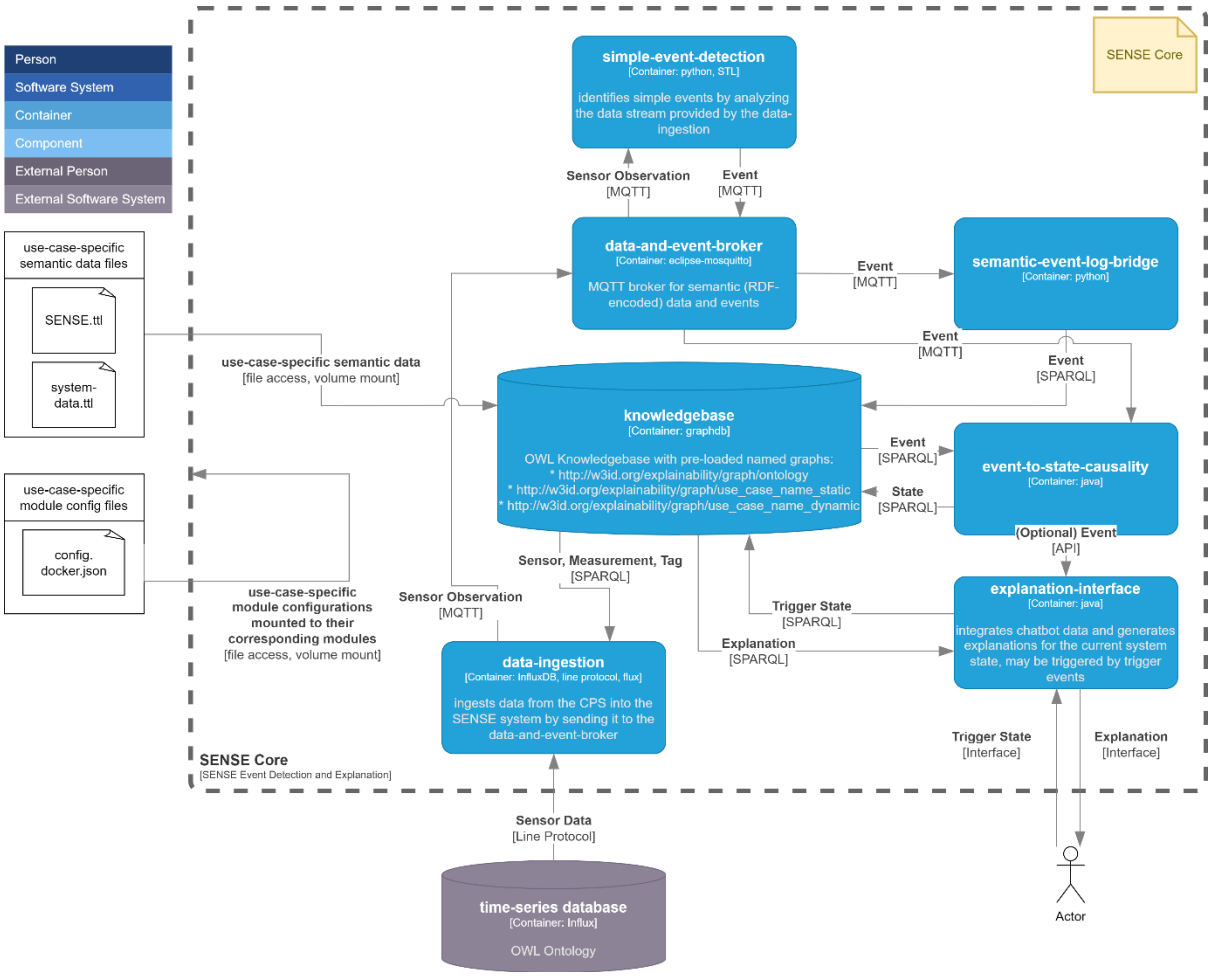

Figure 2: SENSE Core Containers and Interfaces

[5] https://docs.docker.com/build/concepts/dockerfile/
[6] https://c4model.com/diagrams/container

The structure and interdependencies of the SENSE Core modules are as follows.

- **data-ingestion:** This module is responsible for feeding sensor readings into the SENSE system. It currently supports data import from an InfluxDB time series database, as this is often available in existing CPSs. The module can be configured to replay data from the time series database for testing purposes. Alternatively, it can be configured to ingest new data that is added to the time series database into the SENSE system for "live" operation.
- **data-and-event-broker:** This module provides a Mosquitto[7] MQTT broker for message exchange between modules. It uses the official Eclipse-Mosquitto docker image. Hence, no Dockerfile or source code is provided for the data-and-event-broker in this repository.
- **simple-event-detection:** This module is responsible for detecting simple events in the sensor time series data. Events can currently be specified in Signal Temporal Logic (STL) or using customized event monitors implemented in Python.
- **semantic-event-log-bridge:** This module is responsible for listening for detected events and publishing them in the semantic event log, which resides within the knowledgebase. Currently, only a GraphDB is supported as an event log.
- **knowledgebase:** This module is responsible for providing semantic data storage to the SENSE system. It initializes the semantic data storage GraphDB[8] with a repository, named graphs, and TTL files as defined in the configuration.
- **event-to-state-causality:** This module contains a Java-based application that connects to the data-and-event-broker, subscribes to a topic, and processes incoming messages to infer event-to-state causality. The inferred states are then inserted into the knowledgebase.
- **explanation-interface:** This module contains a Java-based application that provides explanations for specific states identified from a semantic model and, if needed, a way to integrate chatbot data. The application connects to a SPARQL endpoint to fetch causal relationships and returns them as JSON responses via a Spring-boot API.

2.2 SENSE Use Case Template

The SENSE Use Case Template repository is provided to simplify the instantiation of the SENSE Core for a new use case. It contains all the files and tools necessary, most importantly, including the Excel template used to collect the necessary input/configuration data. This Excel template must be populated following the SENSE User Guideline documented in Section 3.

Once this process is complete, a Python script that is also included in the SENSE Use Case Template repository has to be executed to export the data collected in Excel into the correct file format to be used by the SENSE Core. The Template repository uses the *use_case_name* as a placeholder, which needs to be replaced with an expressive name for the use case. Additional information can again be found in the repository's README file.

[7] https://mosquitto.org/
[8] https://graphdb.ontotext.com/

2.3 SENSE Demo Use Case

In addition to the SENSE Use Case Template repository, the SENSE Demo Use Case repository provides a relatively simple example instance of the SENSE system. This example fulfills several purposes:

- serve as an alternative to the SENSE Use Case Template if a user wants to start with a running example instead of a template
- illustrate the result of the SENSE User Guideline on a more complex example than used by the User Guideline
- showcase the capabilities of the SENSE system

The example uses case concerns the operation of a regional energy-community (REG) in a distribution network. REGs are a collective of grid entities which trade energy among each other and constitute the BIFROST PoC of SENSE.

A REG is usually operated by an energy community operator, which optimizes the energy exchange to maximize self-consumption among community members. The community must adhere to an operating envelope, to prevent a transformer overload. In the use case the grid operator has full information on the grid (i.e., entities not operated in the community), while the energy community operator has full information on the behavior of its participants, the planned schedule as well as the actual operation of the community.

If a transformer overload occurs, the grid operator is interested in the nature of the overload. Was it due to the energy community violating the operating envelope or did a violation occur despite the community staying within its boundaries (which would suggest that the forecasting method the grid operator uses to calculate the envelope is insufficient).
From the community's point of view, it should be investigated whether a violation was already planned in the schedule (deliberate violation), or whether the forecast of certain elements (e.g. PV production) caused the violation.

The primary focus is to provide event explanations to two key stakeholders: the Energy Community Operator (ECO) and the Distribution System Operator (DSO). Deliverable 2.3 [5] provides comprehensive details on the setup, stakeholder objectives, human-machine interface requirements, and specific scenarios with explanations for both operators.

3 SENSE User Guideline

In this guideline, we outline the steps to be taken to apply SENSE to an existing CPS. We call this process a SENSE use case instantiation, resulting in a new SENSE use case instance. To clarify the requirements, we will illustrate the process with a simplified example briefly introduced in the following. Note that this simplified example only illustrates the steps of this user guide and should not be confused with the more comprehensive SENSE Demo Use Case. General information about the process to be followed is written in black text, whereas everything concerning the example system is added in green text.

As a simplified example, we consider a household with an Electric Vehicle (EV) charger and a battery. If the EV charger is fast charging an EV, this can cause a peak demand at the

household level. However, this peak only happens if it is enabled by an empty battery. A sketch of the example setup is depicted in Figure 3.

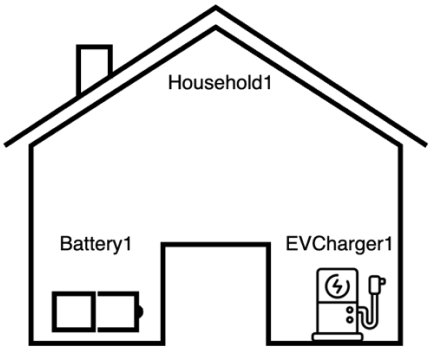

Figure 3: Example System Topology

In six consecutive sections, starting with Section 3.2, the steps that need to be followed for applying SENSE to an existing CPS are described. This connection makes semantics-enabled system explanations possible. While this guideline is a general explanation of the required steps from an organizational perspective, a more technical explanation can be found in our Git Repository[9].

Additionally, the outcome of this process (after all six steps) is a comprehensive understanding of the system and its connections. We provide an Excel template (system data file) to collect this information in a structured format[10], which then serves as an input for parameterizing the SENSE stack according to the use case. We ensure compatibility between this system data file and the SENSE Core stack. The file consists of multiple tabs and columns, which are to be filled in throughout the process. In the following steps, we use purple text when referring to tabs and columns of the Excel template.

3.1 Key Stakeholders for the Process

Integrating the SENSE technology stack into an existing system requires experts from different areas to collaborate. In practice, a person can take over multiple stakeholder roles. The following stakeholders are a minimum requirement to ensure a well-defined system and process:

- **System Engineer:** a person who knows the system as well as the engineering decisions that were made to have a running system and understands the physics and causal relations in the system
- **Data Engineer:** a person who manages the sensor data collected in the system, knows the system data, including how the data is stored, how to access it, and metadata such as measurement units
- **Product Owner/Manager:** a person who understands the goals of the project and defines the users and stakeholders of the system

For the example system, we define the stakeholders as follows:

[9] https://github.com/wu-semsys/SENSE-Core
[10] https://github.com/wu-semsys/SENSE-Use-Case-Template/blob/main/infrastructure/knowledgebase/SystemData.xlsx

- System Engineer: **Electrician** who installed the EV and battery and set the thresholds for the battery management
- Data Engineer: **Electrician** who has experience with the technical components, their interfaces, and the data available through sensors
- Product Owner/Manager: **Homeowner** who has decided to use SENSE in their household and knows what they want to achieve by using SENSE in the future

3.2 Step 1: Common Conceptualization

All participants working on the use case instantiation should be familiar with the following terms before proceeding with the guideline.

Table 2 Vocabulary Definitions

Vocabulary Definitions
Type: In this guideline, the *terms PlatformType, SensorType, EventType,* and *StateType* are used. Type in this context means the concept of a thing, in contrast to an instance of a thing. For example, an ActivePowerSensor as a concept is a SensorType, while ActivePowerSensor1, installed at VenueA, is a Sensor.
Observable Property: a property, which can be measured by a sensor. For example: active power, temperature
Platform: a device, or facility of interest. A platform can host sensors, which measure the properties of this platform. A platform can also host other platforms if they are forming a part of this platform. Finally, a platform does not need to be a physical entity but can also be a logical grouping of platforms if needed. For example, a house is a platform, which can host an ActivePowerSensor measuring ActivePower consumed by the house. A house can also host other platforms, such as a Battery and an EvCharger. A battery is a platform, which can host sensors of its own.
Sensor: measures an observable property. Each sensor is hosted by a platform, which means it is installed on this platform and measures data connected to this platform.
hostedBy: a relation between a platform and its sub-systems. A platform can host sensors and other platforms. This relation defines the hierarchical topology of the system.
Event: happening at a time point, detected by the event detection module. An event causes a transition between states.
State: condition of a system over a time period. A state is started and ended by an event. States are what we aim to explain and what we use within explanations.
Event/State Type: the concept of an event/state. An event/state type is defined by a detection/conversion rule. These types are representations of possible types of occurrences in the system, which are of interest.

State Type Causality: a causal relation between two types of states. Each causal relation has a set of features that define the causal relation in more detail. These features are:

- *topological relation:* how the cause state type has to relate to the effect state type topologically for the causal relation to hold between two instances of these states. There are 3 options:
  - *samePlatform:* cause and effect happen at the same platform
  - *parentPlatform:* the platform of the effect state hosts the platform of the cause state
  - *siblingPlatform:* the states happen at two distinct platforms hosted by the same parent
- *temporal relation:* how the cause state type has to relate to the effect state type temporally for the causal relation to hold. There are 4 options:
  - *before: stateA* starts and ends before *stateB*
  - *overlaps: stateA starts before stateB, but ends after stateB has started*
  - *contains: stateA starts before stateB and ends after stateB has ended*
  - *identity: stateA* starts and ends at the same time as *stateB*
- *causal relation:* the nature of the causal relation between two state types. There are three options:
  - *cause:* if *stateA* occurs, *stateB* most probably occurs as a result
  - *enable:* if *stateA* does not occur, *stateB* most probably does not occur
  - *prevent:* if *stateA* occurs, *stateB* most probably does not occur as a result

3.3 Step 2: Know Your Goals

Key Stakeholder: Product Owner/Manager

As the very first step, you need to define the goals of using the SENSE stack in your system. Which questions should be answered by the system? What are important anomalies you want to detect and explain? The answers to these questions should always be the guideline for any decision in the next steps. Data and sensors that help to reach the defined goals should be identified and included in the SENSE implementation.

In the example system, the SENSE stack should be able to explain why there is a demand peak at the household level.

3.4 Step 3: Know Your System

Key Stakeholders: System Engineer

The following steps are intended to give guidance on how to approach knowledge acquisition for your system. However, if necessary or preferred, a different order of steps is also possible.

3.4.1 Define the Concepts that are Present in the System (PlatformTypes, ObservableProperties)

Before defining the actual topology of the system, create a list of **PlatformTypes** (devices, facilities, and groups of sensors that are of interest) and **ObservableProperties** (properties that can be measured) in the system. The goal is to define the scope of the system, and which

features are covered. Feel free to come back to this step if, at a later stage, you realize that some concept is missing.

List all relevant PlatformTypes in the column PlatformType of the tab PlatformTypes. Sometimes, one PlatformType may be a sub-category of another PlatformType (e.g., Supercharger is a sub-category of EvCharger). In this case, the optional column "subClassOf_PlatformType" can be used to define this relationship between PlatformTypes.

In the example system, we define **Household**, **Battery,** and **EvCharger** as platform types. The resulting Excel sheet is shown in Figure 4.

PlatformType	subClassOf_PlatformType
Household	
Battery	
EvCharger	

Figure 4: SystemData.xlsx – PlatformTypes

List all observable properties relevant to the system in the column ObservableProperty of the tab ObservableProperties. These are usually the properties that are measured by the sensors of the system. However, ObservableProperties might include properties indirectly measured by combining multiple sensor measurements or other means. This list can be extended at any time. It serves as a list of possible values to be used in the following steps.

In the example, ObservableProperties are **ActivePower** and **StateOfCharge.** The resulting Excel sheet is shown in Figure 5.

ObservableProperty
ActivePower
StateOfCharge

Figure 5: SystemData.xlsx – ObservableProperties

3.4.2 Define Instances of Platforms

In this step, the actual platforms and sensors that are present and installed in the system are to be defined. Some general rules apply:

- Each sensor must be hosted by a platform.
- Each sensor observes an observable property.
- A platform can be hosted by another platform (this is the case if a platform is part of the other platform).

List all platforms that are present in the system in the column Platform of the tab Platforms and select the appropriate types via the dropdown in the column PlatformType.

In the example, there is one household platform (**Household1** of type **Household**), one battery (**Battery1** of type **Battery**), and one EV charger (**EvCharger1** of type **EvCharger**). The resulting Excel sheet is shown in Figure 6.

Platform	PlatformType
Household1	Household
Battery1	Battery
EvCharger1	EvCharger

Figure 6: SystemData.xlsx – Platforms

3.4.3 Define Hierarchical Relations Between Platforms

In the next step, the hierarchical relations between platforms are defined via the "hostedBy" relation. As mentioned above, PlatformA is hosted by PlatformB if PlatformA is part of or situated within PlatformB.

In the example, **EvCharger1** and **Battery1** are both hosted by **Household1** as they are logically part of the household. Connections between platforms are defined by the "hostedBy" relation: **(Household1 hosts EvCharger1)** and **(Household1 hosts Battery1).** The resulting Excel sheet is shown in Figure 7.

Platform	PlatformType	hostedBy_Platform
Household1	Household	
Battery1	Battery	Household1
EvCharger1	EvCharger	Household1

Figure 7: SystemData.xlsx – Platforms and Hierarchical Relations

3.4.4 Define Instances of Sensors and Their Hosting Platforms

List all sensors that are installed in the system in the column Sensor of the tab Sensors. For each sensor, define its location by selecting the corresponding platform for column hostedBy_Platform, and specify which data is measured in the column observers_ObservableProperty.

In the example, there is one active power sensor at the household level measuring the total power used in the household (**AP_household1_sensor** hosted by **Household1**, observing **ActivePower**), one power sensor at the EV charger (**AP_evcharger1_sensor** hosted by **Evcharger1**, observing **ActivePower**), one power sensor at the battery (**AP_battery1_sensor** hosted by **Battery1**, observing **ActivePower**) and one state of charge sensor at the battery (**SOC_battery1_sensor** hosted by **Battery1**, observing **StateOfCharge**). The resulting Excel sheet is shown in Figure 8.

Sensor	hostedBy_Platform	observes_ObservableProperty
AP_household1_sensor	Household1	ActivePower
AP_evcharger1_sensor	EvCharger1	ActivePower
AP_battery1_sensor	Battery1	ActivePower
SOC_battery1_sensor	Battery1	StateOfCharge

Figure 8: SystemData.xlsx – Sensors

3.5 Step 4: Know Your Data

Key Stakeholders: Data Engineer

We now have a representation of the system, its devices, and the location of sensors that collect data. In this next step, the data engineer should help in accessing the sensor data. Each sensor in the system should have a corresponding time series data stream. From an architectural viewpoint, a dedicated data source could be defined for each sensor. However, the current implementation expects all measurements to be available in a single InfluxDB time series database. This time series database is not part of the SENSE system but must be operated elsewhere.

Virtual sensors: A virtual sensor may be an appropriate solution if an event can be defined by data that is only implicitly collected (e.g., as a sum of multiple sensors). The value of the virtual sensor must be calculated outside of the SENSE system, e.g., via a dedicated task in the time series database, and fed to the SENSE system as any ordinary sensor value.

Connect the sensors to their corresponding time series by adding information on how to identify the sensor data in the time series database to the column TimeseriesId in the tab Sensors. Additionally, any virtual sensor must be added to the Sensor tab, and the corresponding TimeseriesId must be specified.

In the example, all the data is stored in an InfluxDB database. For example, the data of **AP_household1_sensor** is stored in the InfluxDB database under the **measurement** identified by **AP** and **field name household1.AP**. The resulting Excel sheet is shown in Figure 9.

Sensor	hostedBy_Platform	observes_ObservableProperty	TimeseriesId
AP_household1_sensor	Household1	ActivePower	measurement=AP,field=household1.AP
AP_evcharger1_sensor	EvCharger1	ActivePower	measurement=AP,field=evcharger1.AP
AP_battery1_sensor	Battery1	ActivePower	measurement=AP,field=battery1.AP
SOC_battery1_sensor	Battery1	StateOfCharge	measurement=SOC,field=battery1.SOC

Figure 9: SystemData.xlsx – Sensors and TimeseriesIds

3.6 Step 5: Explore Causal Relations in the System

Key Stakeholders: System Engineer, Product Owner

In this step, the causal relations between occurrences in the system should be explored. To be able to define causal relations, events and states of interest in the system must also be identified. Based on these events and states, causality knowledge between types of states can be explored. In Figure 10, all the components belonging to a causality relation are shown. These causalities should be extracted in a causality workshop with domain experts who know the general causal relations inside the system. As the basis of the workshop, a list of situations is collected that need an explanation in the system. The next sections suggest steps to be followed in a causality workshop.

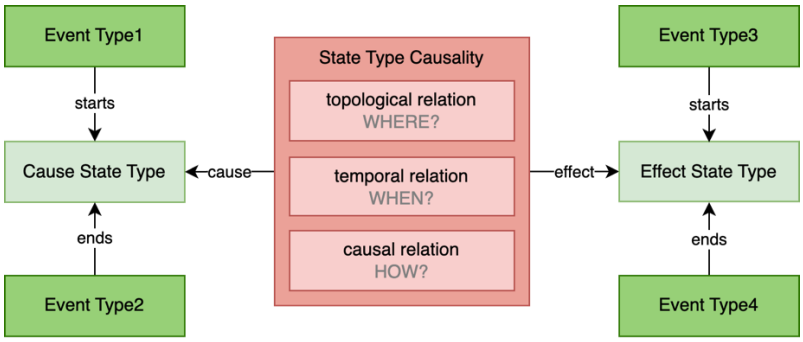

Figure 10: Causality Relation Setup

In the example, we want to explain a demand peak at the household level.

3.6.1 Discuss Scenarios of Potential Causes

Either while coming up with these situations, or in the next step, write up scenarios, which could be a potential explanation of the situations. The more potential explanations added in this step, the more detailed the system will be able to explain situations at runtime.

In the example, a peak in demand can be due to high consumption of the EV charger. However, this is usually not the only reason, as the battery can counteract this high demand. Therefore, a low battery state of charge usually also contributes to the peak demand.

3.6.2 Define Corresponding States

Once there is a list of explanations for each situation which should be explained, connect the explanations to events, states, and causalities in the system. States are happening as time intervals connected to a single sensor in the system. Events define the start and end of a state. Causalities define the relations between the states. Defining states, events, and causalities is an iterative and incremental process. It can very well start with a short list of states, events that activate and deactivate these states, and causalities between the states. During discussions, additional states, events, and casualties are often identified and added.

Define system states in the StateType column of the StateTypes tab. Typically, start with the undesired state that shall be explained and refined from thereon. For each state, select an associatedObservableProperty and an associatedPlatformType, set isTriggerState to TRUE or FALSE depending on whether explanation generation should be triggered automatically upon entering the state, and provide a human-readable Description of the state.

In the example, interesting StateTypes are:

- **PeakDemandState** can be detected by an **ActivePower** sensor on the **Household** platform. As this State triggers an explanation, isTriggerState is set to **TRUE**.
- **EVHighChargingState** can be detected by an **ActivePower** sensor on the **EvCharger** platform.
- **BatterySoCLow** can be detected by a **StateOfCharge** sensor on the **Battery** platform

The resulting Excel sheet is shown in Figure 11.

StateType	associatedObservableProperty	associatedPlatformType	isTriggerState	Description
PeakDemandState	ActivePower	Household	TRUE	Active Power demand over 50kW
EVHighChargingState	ActivePower	Evcharger	FALSE	EV Charger charges at more than 50kW
BatterySoCLowState	StateOfCharge	Battery	FALSE	Battery SoC is below 10%

Figure 11: SystemData.xlsx – StateTypes

3.6.3 Define Events that Start and End States

Next, define the events that activate and deactivate the previously defined states in the EventStateMapping tab of the Excel sheet. Provide an expressive name for the event in the column EventType, define StateType_starts and StateType_ends by selecting from the previously defined states, and specify on which platform (MonitoredPlatform) and on which sensor (MonitoredSignal[11]) the event may occur.

In addition to this information, EventTypes need a machine-readable specification for the SENSE system to be able to detect them. For this reason, we provide predefined signal properties to be used for EventType definitions.

Select an appropriate SignalProperty that best matches the event to be detected. Next, set the SignalPropertyParameters of the signal property. The available signal properties and parameters to select from are explained in Table 3. Some parameters are optional – their default values are indicated within square brackets [] in Table 3. Each of the parameters can then either be set to a fixed value (**LiteralValue**) or be determined by another signal (**SensorValue**).

Table 3: Signal Property and Parameter Definitions

Signal Property	Description	Parameter Definitions
Overshoot	*The signal rises above a threshold*	• *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*

[11] Signals is a more common term than sensors in event detection literature. Note that in the SENSE system, each sensor provides only one signal, making the terms interchangeable.

		• *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*
Undershoot	*The signal falls below a threshold*	• *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	*The signal leaves a predefined band either in the upper or lower direction*	• *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*
WithinBounds	*The signal enters a predefined band*	• *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	*The signal rises by a certain amount within a defined time span*	• *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	*The signal falls by a certain amount within a defined time span*	• *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*
StableTime	*The signal is stable for a defined time span*	• *maximum_delta:* an event will be fired if the signal rises and falls by less than the value defined by *maximum_delta* under the limitations of the remaining parameters • *stable_time:* an event will only be fired if the signal is stable for at least the time interval defined by *stable_time*

In the example, the relationships between states and events (the event-state-mapping) and the event definitions are

- **PeakDemand** is started by a **ThresholdViolatedEvent** (AP sensor > 50kW) and is ended by a **ThresholdNormalEvent** (sensor <= 50kW).
  - The **ThresholdViolatedEvent** is formally expressed via an **Overshoot** signal property with a **threshold** of **50** (**LiteralValue**)
  - The **ThresholdNormalEvent** is formally expressed via an **Undershoot** signal property with a **threshold** of **50** (**Literal Value**)
- **EVHighCharging** is started by an **EVHighChargingEvent** (sensor > 50kW) and is ended by an **EVNormalChargingEvent** (sensor <= 50kW).
  - The **EVHighChargingEvent** is formally expressed via an **Overshoot** signal property with a **threshold** of **50** (**LiteralValue**)
  - The **EVNormalChargingEvent** is formally expressed via **Undershoot** signal property with a **threshold** of **50** (**LiteralValue**)
- **BatterySoCLow** is started by a **BatterySoCDepletedEvent** (SoC < 10%) and is ended by a **BatterySoCNormalEvent** (SoC > **10**%)
  - The **BatterySoCLow** is formally expressed via an **Undershoot** signal property with a **threshold** of **10** (**LiteralValue**)
  - The **BatterySoCNormalEvent** is formally expressed via an **Overshoot** signal property with a **threshold** of **10** (**LiteralValue**)

The resulting Excel sheet is shown in Figure 12.

EventType	StateType_starts	StateType_ends	MonitoredPlatform	MonitoredSignal	SignalProperty	SignalPropertyParameter1		
start defining your events in the next line						Name	Type	Literal OrSensor
ThresholdViolatedEvent	PeakDemandState		Household	HouseholdActivePowerSensor	Overshoot	threshold	LiteralValue	50
ThresholdNormalEvent		PeakDemandState	Household	HouseholdActivePowerSensor	Undershoot	threshold	LiteralValue	50
EVHighChargingEvent	EVHighChargingState		EvCharger	EVActivePowerSensor	Overshoot	threshold	LiteralValue	50
EVNormalChargingEvent		EVHighChargingState	EvCharger	EVActivePowerSensor	Undershoot	threshold	LiteralValue	50
BatterySoCDepletedEvent	BatterySoCLowState		Battery	BatteryStateOfChargeSensor	Undershoot	threshold	LiteralValue	10
BatterySoCNormalEvent		BatterySoCLowState	Battery	BatteryStateOfChargeSensor	Overshoot	threshold	LiteralValue	10

Figure 12: SystemData.xlsx – EventStateMapping

3.6.4 Define Causal Relations Between States

Define causal relations between states in the StateTypeCausality tab. Select one state in the StateType_cause column and another state in the StateType_effect column and then define

their relation via the causalRelation, temporalRelation, and topologicalRelation properties as defined in Table 2.

In the example, causalities between the StateTypes are known as follows:

- **EVHighChargingState causes** a **PeakDemandState**, if the states **overlap** and there is a **parentPlatform** relation.
- **BatterySoCLowState** enables the **PeakDemand** if the states **overlap** and there is a **parentPlatform** relation.

The resulting Excel sheet is shown in Figure 13.

StateType_cause	causalRelatio	temporalRelatior	topologicalRelatior	StateType_effect
EVHighChargingState	causes	overlaps	parentPlatform	PeakDemandState
BatterySoCLowState	enables	overlaps	parentPlatform	PeakDemandState

Figure 13: SystemData.xlsx – StateTypeCausality

3.7 Step 6: Run Your SENSE Instance

Key Stakeholders: Data Engineer (Knowledge Engineer)

Based on the previous steps, you should have all the data necessary to run a SENSE instance for your use case. The person responsible for the implementation should have a look into the repository and the respective README file, where the steps to create a new SENSE instance are described: https://github.com/wu-semsys/SENSE-Use-Case-Template.

4 Chatbot Integration

While the technical core of the SENSE system provides its main functionalities, such as data ingestion, event detection, and explanation generation, interlinking it with a conversational user interface greatly improves its usability by non-domain experts. To this end, we conducted a set of experiments and exploration for integrating the SENSE system with chatbots, focusing on ONLIM Chatbot.

On the technical level, we are working on supporting chatbot integration around three main capabilities: (a) knowledge graph (KG) enrichment, (b) faceted event explanation generation, and (c) dynamic knowledge retrieval. These components are designed to support more intelligent, context-aware, and explainable chatbot interactions.

4.1 KG enrichment

Due to the complex processes involved in chatbot operations, it is often necessary to preprocess and enrich *existing knowledge*—whether in textual or graph form—to align it with the chatbot's internal knowledge model in facilitating seamless conversation between users and the chatbot. In the context of SENSE project, such knowledge is provided as a knowledge graph based on the SENSE ontology, which captures various aspects of cyber-physical systems (CPS), such as topology, observation, causality.

To support integration between Chatbot internal knowledge representation and the SENSE knowledge graph, we developed a dedicated API endpoint (`POST /v1/api/integration`). This endpoint processes and queues new instances of our knowledge graph while allowing for their preprocessing and alignment with chatbot-specific knowledge extensions, such as linking sensor data or contextual information to improve response relevance.

Currently, we have implemented a single instance of this functionality, which focuses on aligning key terminologies between the SENSE ontology and the ONLIM chatbot knowledge model for our proof-of-concept (PoC) scenarios. In the future, we aim to extend this functionality to support more sophisticated integration mechanisms and enable richer chatbot capabilities.

4.2 Faceted Event Explanation

The faceted event explanation provides chatbot with access to context-specific insights of the system's behavior. Through the API endpoint `(GET /v1/api/explanations)`, chatbot can retrieve explanations on specific events based on a given datetime (e.g., through start- and end-time), system topology (e.g., explanation pertaining to specific devices or platforms), and user access control (e.g., through predefined user role definition.

This capability helps the chatbot explain why certain events —such as changes in battery demand—occurred, fostering transparency and user trust. Furthermore, we are currently in the process of pre-selecting a set of SPARQL queries that will be available for both chatbots and users alike, focusing not only on the event explanations, but also context information around the explanation, such as topology information as well as specific observation data leading to the explanation.

4.3 Dynamic Knowledge Retrieval

To handle the complexity of chatbot user queries, it is often insufficient to rely solely on a predetermined set of queries. Therefore, the SENSE framework must support dynamic knowledge retrieval at runtime. To achieve this, the SENSE framework offers two mechanisms for dynamic data access.

The first is *SPARQL Endpoint Access*. SENSE provides a dedicated SPARQL endpoint (`POST /v1/api/explanations/sparql`) that enables users and developers to perform semantic queries over the knowledge graph. This approach supports flexible information retrieval and enables advanced chatbot capabilities such as context-tailored responses and analysis over aggregated data.

The second mechanism is *Snapshot Synchronization*. In this approach, the SENSE Knowledge Graph is periodically archived in a repository, e.g., through GitLab. The chatbot engine can then acquire this data into its internal repository to be used for the conversation. This method

is particularly useful when complex pre-processing is required before the data can be used, making it a necessary complement to real-time querying.

5 Summary

This deliverable summarizes the results of the SENSE Technology Stack Implementation task. The main contributions are provided via the GitHub repositories of the SENSE Core, SENSE Use Case Template, and SENSE Demo Use Case.

Furthermore, the deliverable provides the SENSE User Guideline, a step-by-step process that needs to be followed when applying SENSE to an existing CPS. For the current version of SENSE, this process involves filling in an Excel template to collect information about the sensors, data sources, events, causalities, and explanations. Information in this template is then used to configure the new SENSE instance.

Finally, the deliverable illustrates how the explanation-interface module of the SENSE Core provides an interface for conversational user interfaces, such as the ONLIM chatbot. This is crucial for making the data and explanations accessible to non-expert users in an intuitive and user-friendly way.

List of Abbreviations

Short	Description
API	Application Programming Interface
CPS	Cyber-Physical System
EV	Electric Vehicle
JSON	JavaScript Object Notation
SENSE	Semantic-based Explanation of Cyber-physical Systems
SPARQL	SPARQL Protocol and RDF Query Language
STL	Signal Temporal Logic
TTL	Terse RDF Triple Language; Turtle

References

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

[2] B. Mohammad and T. Frühwirth, "SENSE Deliverable 3.2: Semantic Data Integration Methods," 2024.

[3] T. Frühwirth, T. Schwarzinger and M. Sabou, "SENSE Deliverable 3.3: Anomaly and Event Detection Algorithms," 2024.

[4] K. Schreiberhuber, M. Memedi, F. J. Ekaputra and M. Sabou, "SENSE Deliverable 4.1 (v2): Semantic-Based Explainability Framework," 2024.

[5] R. Poelmans, C. Moser, J. Kainz, D. Hauer, K. Diwold, A. Einfalt, D. Jähnig and T. Frühwirth, "SENSE Deliverable 2.3: Definition of POCs," RESTRICTED ACCESS, 2024.