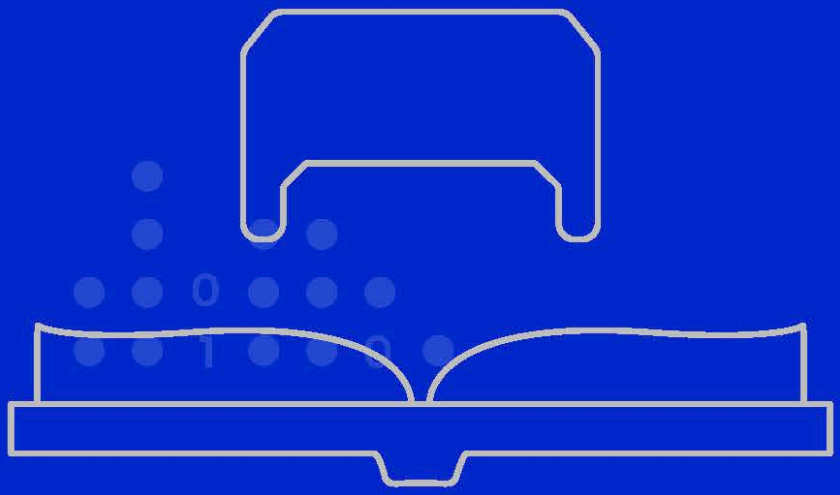

WHITE PAPER

AAS Events

Specification Understanding – Explanation, Modelling Approaches, and Examples

May 2025

Authors and contributors

Alexander Belyaev, ifak e.V.

Raghavendra Ungarala, Hochschule Karlsruhe

Melanie Stolze, ifak e.V.

Prof. Dr.-Ing. Michael Hoffmeister, Hochschule Karlsruhe

Dr.-Ing. Matthias Riedl, ifak e.V.

Contents

List of Figures

List of Abbreviations

AAS	Asset Administration Shell
PCN	Product Change Notification
PRR	Purchase Request Response
POC	Purchase Order Creation
IDTA	Industrial Digital Twin Association
MQTT	Message Queuing Telemetry Transport
HTTP	Hypertext Transfer Protocol
REST	Representational State Transfer
API	Application Programming Interface
SME	SubmodelElement
SM	Submodel
ERP	Enterprise Resource Planning
MES	Manufacturing Execution System
TLS	Transport Layer Security
CAD	Computer-Aided Design
DNS	Domain Name System
IP	Internet Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
JSON	JavaScript Object Notation

1 Introduction and Motivation

According to the AAS Metamodel, it is possible to subscribe and receive updates and changes regarding various elements and concepts within the AAS - as a Digital Twin itself - and its environment. This is achieved through a subscription mechanism, a concept widely used in our everyday lives to receive timely notifications, updates, and alerts about relevant changes or new information, such as social media updates, news alerts, weather forecasts, streaming services, calendar reminders, or software updates.

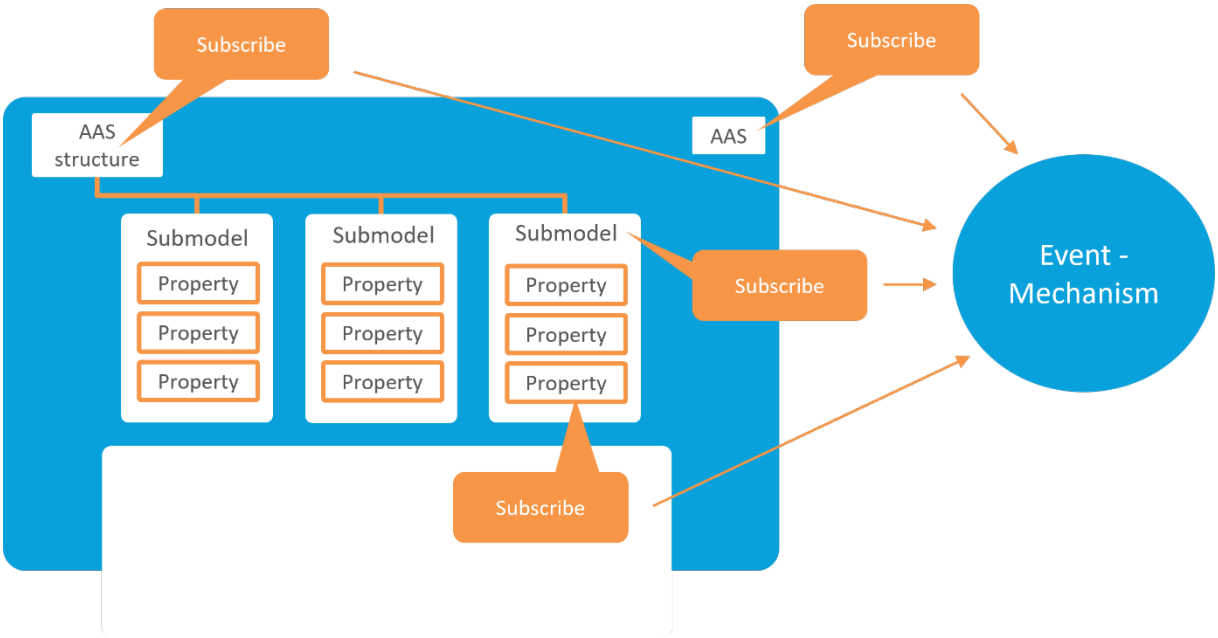

Figure 1: Elements of an AAS can be subscribed to via the AAS event mechanism

In the context of the AAS, the design pattern used for this purpose is the Event Mechanism. The elements of AAS that can be subscribed, or to which the AAS Event Mechanism can be applied, as introduced in Figure 1, include the following:

- Changes to the meta-information of the AAS
- Changes in the structure of the AAS
- Updates to individual Submodels
- Updates to individual Properties

The purpose of this whitepaper is to explain the application of the AAS Event Mechanism, demonstrate how to model the subscription of different components within the AAS using the **BasicEventElement** specified in the AAS Metamodel ((IDTA), April 2023), and illustrate various application scenarios. These include understanding the specification of the AAS **EventElement**, applying and configuring this element to subscribe to changes in the mentioned AAS elements. The concepts defined in the metamodel are quite flexible. They also allow the use of events in the opposite direction — enabling the AAS itself to act as a subscriber and receive updates from another AAS or an external source. The **EventElement** supports the reception of such updates, enabling the AAS to incorporate them into the corresponding elements of its own submodels.

This document serves as a discussion basis intended to inspire and support initial engagement with the AAS event mechanism and to motivate its practical application. The discussion paper does not prescribe specific modelling approaches or implementation technologies. Instead, it encourages the community to further explore and apply the event mechanism concept.

Moreover, this publication is envisioned as part of an ongoing series. Future editions will strongly emphasize the topic of security in the context of AAS events, as well as present concrete application scenarios from Manufacturing-X projects.

To address this objective, the discussion paper is organized as depicted in Figure 2.

Chapter 2 begins by clarifying what constitutes a change that can be subscribed within the context of the AAS, including the types of possible changes and identifying which elements of the AAS can be subscribed for change notifications. Subsequently, the specification of the **EventElement**, a metamodel element of the AAS enabling subscriptions, is explained. Following this, a basic modelling approach is introduced, accompanied by modelling examples.

In **Chapter 3**, an example system architecture is presented, illustrating initial possible configurations and highlighting the flexibility inherent in the AAS event concept. This section details how the **EventElement** introduced in Chapter 2 can be used to model three distinct scenarios: first, subscribing an external application to changes within an AAS; second, the scenario where the AAS itself subscribes to changes from an external data source; and third, the scenario where one AAS subscribes to changes in another AAS.

Chapter 4 addresses various approaches to system architecture and explores different technological implementations of the event mechanism, initially focusing on two technologies, HTTP REST and MQTT.

Chapter 5 illustrates the application of the event concept to subscribing to selected Submodels, including detailed modelling examples. Finally, **Chapter 6** summarizes the current state of discussion regarding the **EventElement** and outlines the next steps.

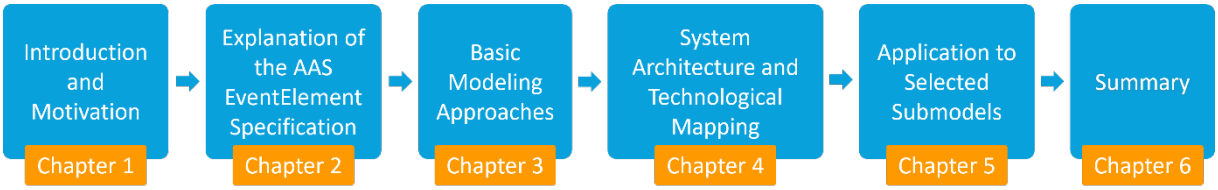

Figure 2: Structure of the discussion paper

2 Explanation of the specification of the AAS BasicEventElement

2.1 General

In this chapter, modelling elements defined by the AAS Metamodel, such as the **BasicEventElement** and **EventPayload**, are explained. Particular attention is given to their specification and significance. Additionally, the actual subscription modelling process is demonstrated, outlining relevant modelling decisions and essential configuration steps. An initial answer is provided to the question, “Where should the **EventElement** be attached?”, presenting two potential approaches: either by attaching it directly to the submodel whose elements are being subscribed, or creating a separate dedicated Submodel to collect and manage all created **EventElements**.

2.2 What exactly can be subscribed?

The **EventElement** is designed to enable subscriptions and notifications regarding changes within the structure of an AAS, including its submodels and the elements they contain. But which specific types of changes can be communicated through event mechanisms? This question is addressed in the following chapter. Relevant changes include:

- Structural changes in AAS

- Updates to SubmodelElements and their attributes

Structural changes encompass the creation of new elements or the deletion of existing ones. Updates typically involve modifying values, such as changing the value of a property. In this context, three types of changes are identified: **create**, **update**, and **delete**.

The change types **create** and **delete** are directly associated with structural modifications, whereas updates to AAS elements (as shown in Figure 3) align primarily with the update type. To enable automated recognition of the meaning of events and identification of the type of change, globally unique semantic IDs for these change types should be defined. This ensures that the specific type of change can be clearly identified in the **EventPayload**. Consequently, the recipient of an event notification can either explicitly recognize the type of change being communicated or, in more advanced applications, easily filter and search for particular types of changes.

Figure 3 demonstrates the change types can be modelled using the **EventElement** and clarifies the meaning of each change type (create, update, delete) in relation to specific AAS elements or the AAS itself. For example, the specified event mechanism allows subscriptions to the creation or deletion of an Asset's AAS. Changes within the AAS can also be subscribed to in a more fine-grained manner. It is possible to subscribe and receive notifications about modifications to specific elements of the AAS, including structural changes or updates within Submodels, relevant concept descriptions, and AAS Meta-information.

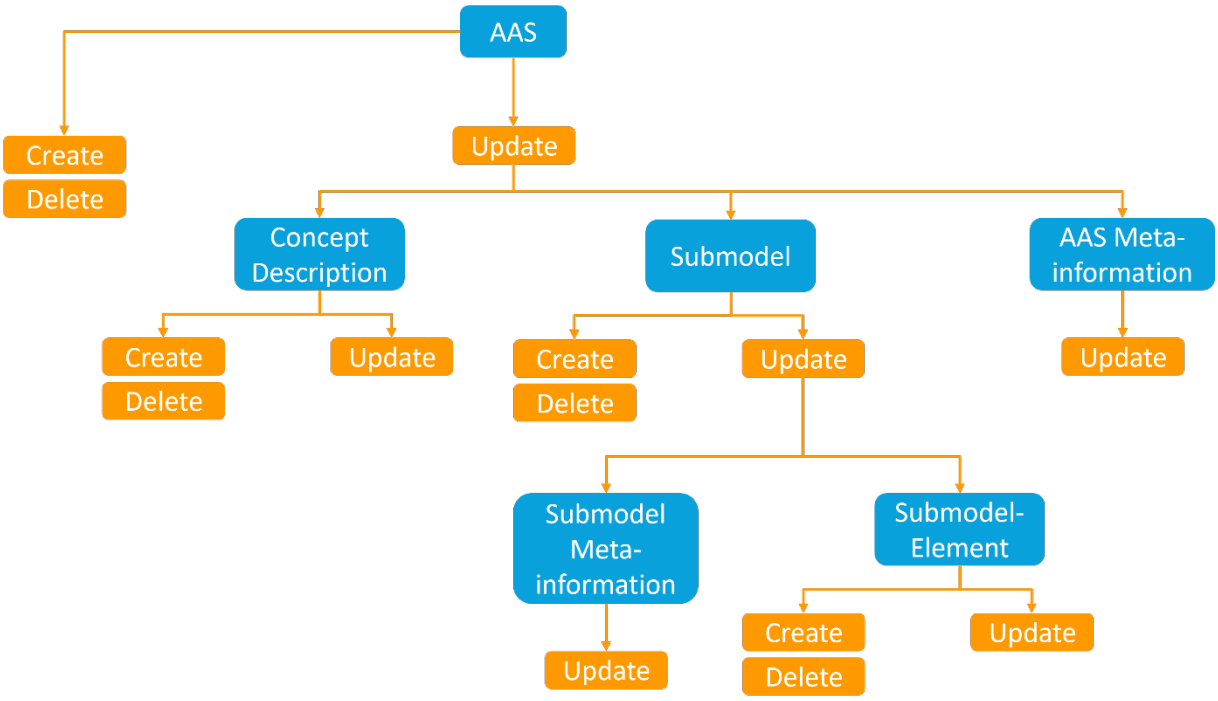

Figure 3: Overview of AAS change types available for event subscription[1]

Updating a Submodel, for instance, can mean structural modifications or updates to underlying **SubmodelElements**, such as creating or deleting a **SubmodelElement** or updating its value.

[1] Created in the Project VWS4LS

2.3 Specification of the BasicEventElement

To enable subscriptions, the AAS Metamodel specifies the **BasicEventElement**, whose specification is illustrated in Figure 4.

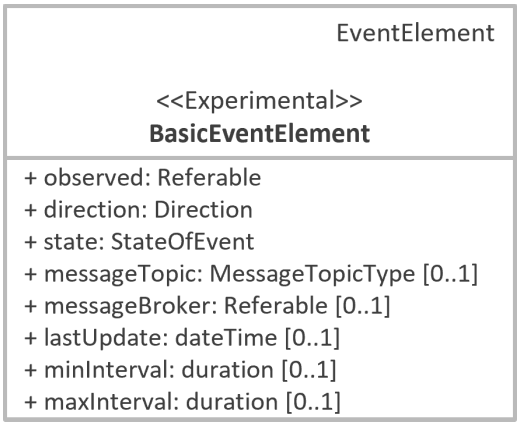

Figure 4: Specification of the BasicEventElement in the AAS Metamodel

The listed attributes of the EventElement have the following meanings ((IDTA), April 2023):

- **observed**: Reference to a referable element.
- **direction:** Defines the direction of the event (input or output).
- **state:** Status of the event (on or off).
- **messageTopic:** Unique identifier of the event for message transmission
- **messageBroker:** Reference to an application responsible for receiving and distributing event messages.
- **lastUpdate:** Timestamp indicating when the most recent event was sent or received.
- **minInterval:** Minimum time interval between sending or receiving events.
- **maxInterval:** Maximum time interval allowed between outgoing event transmissions when no other trigger has occurred. This applies only to outgoing events.

It can be observed that the event description itself is technology neutral. However, the use of typical MQTT terminology in the specification of the **BasicEventElement** may create the impression that the use of MQTT with an MQTT broker and MQTT topics for event transmission is implicitly assumed in the AAS specification. This, however, is not entirely accurate, as such an assumption is not explicitly made in the specification. Therefore, other technologies are not excluded. Nevertheless, alternative technologies frequently adopt concepts like those of MQTT brokers and topics. One example is an HTTP/REST interface, where public servers can analogously act as brokers and endpoints serve as equivalents to the topics to which events are sent.

2.4 Integration of the BasicEventElement in the AAS-Structure

2.4.1 General

The next chapter presents a step-by-step example demonstrating how a **BasicEventElement** can be used to subscribe to a specific element within an AAS. As a guiding example, the subscription to updates of a property value is chosen.

To subscribe to an element of the AAS, the EventElement must first be created and properly integrated into the AAS structure. This means it needs to be assigned to an appropriate location within the model. Two alternative approaches are available:

- **Assignment to an existing Submodel:** The BasicEventElement is assigned directly to the Submodel containing the element to be subscribed. This method is described in Chapter 2.4.2 and corresponds to **Step 1** as illustrated in Figure 5.
- **Creation of a dedicated Submodel:** A dedicated Submodel is created specifically to host and manage all BasicEventElements defined within the AAS. For example, a "Subscriptions" Submodel can be created to collect and centrally manage all event elements. This approach is demonstrated in Chapter 2.4.3.

2.4.2 Assigning the BasicEventElement directly to the existing Submodel

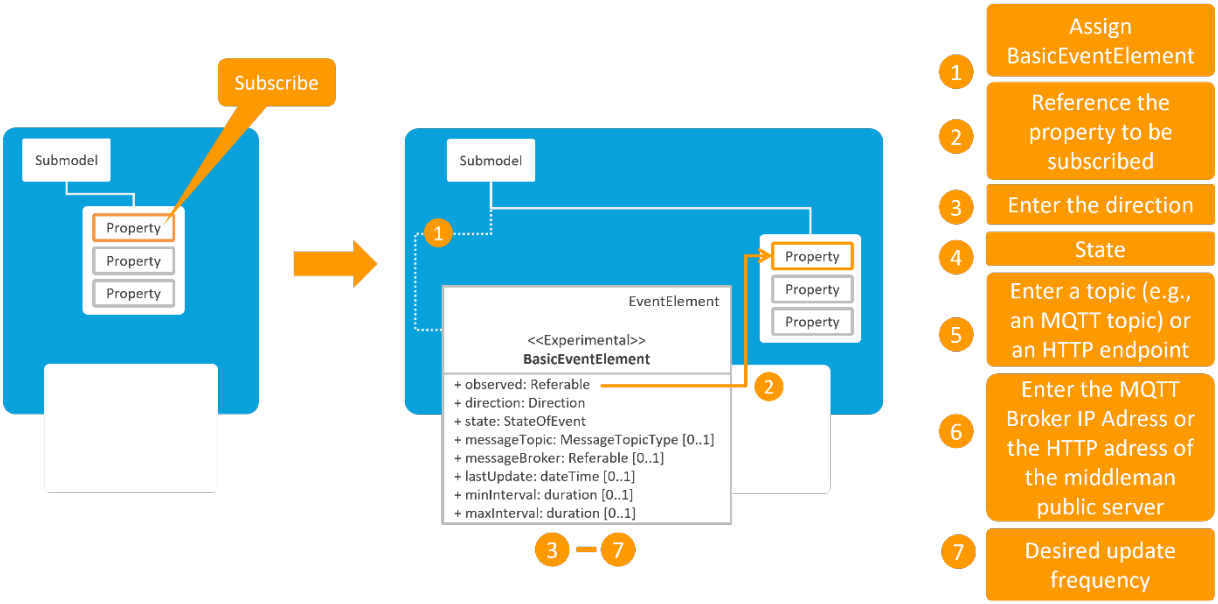

Figure 5: Assignment and parameterization of the BasicEventElement

After creation, the **EventElement** requires parameterization and configuration. In **Step 2**, the property to which the subscription should apply must be referenced. This is done by specifying a reference, including the path to the corresponding AAS element — in this example, a property.

In the next step, the direction of the event must be specified. It is important to note that the event mechanism supports two functional options regarding the direction of information flow:

- **Option1 – EventElement as a source:** In this case, the EventElement serves as the origin of event notifications. A subscribing application can receive updates about the AAS element referenced by this EventElement, such as a property.
- **Option 2 – EventElement as a receiver:** In this case, the EventElement is used to receive events from external sources. These external updates are then applied to the AAS element referenced in the EventElement.

This illustrates the flexibility of the AAS event concept, as it supports bidirectional data flow: external clients can subscribe to changes within the AAS, and the AAS itself can subscribe to updates from external data sources. The direction of the data flow is defined in Step 3 by setting the value to either **OUT** (publishing updates) or **IN** (receiving updates).

In the next step, the **state** attribute can be used to control the intended behavior of the **EventElement**. For example, it can be set to **publish** to activate event publishing, or to **off** to stop publishing.

One possible implementation of the subscription mechanism involves an intermediary component positioned between the data source and the data consumer. This concept is explored further in

Chapter 3. The specification supports the definition of a topic (e.g., an MQTT topic) to which updates are published (**Step 5**). When using the REST API, an HTTP endpoint (URI) should be entered in this parameter so that updates can be received via HTTP methods such as POST or PUT.

In **Step 6**, when connecting to an MQTT broker, the IP address or DNS name along with the appropriate port number must be specified.

Finally, **Step 7** allows for the definition of the desired update frequency, which determines how often updates are sent to the subscriber.

2.4.3 Creation of the separate Submodel for Events

This chapter describes the possibility of the creation of a separate Submodel to serve as a container for all **EventElements**. This concept assumes that for each AAS element to be subscribed, a new **BasicEventElement** is created and assigned to this dedicated Submodel.

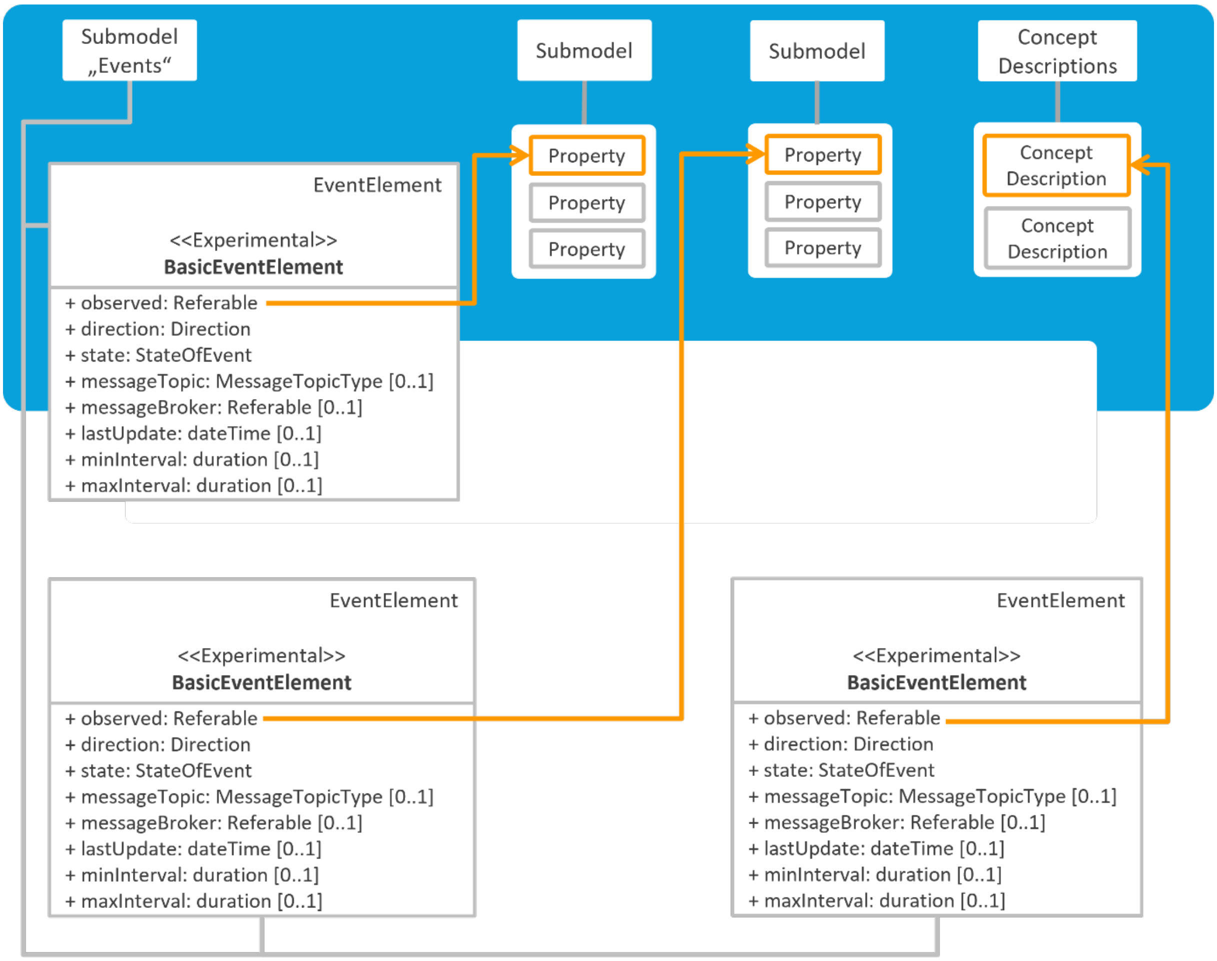

Figure 6: Separate Submodel containing configured BasicEventElements

The individual steps required to configure each **BasicEventElement** are identical to those described in the previous chapter. The assignment of the EventElements to the observed Submodel or AAS element is done by referencing the corresponding element in the observed attribute, as illustrated in Figure 6

2.5 Content of the Event – application of the metamodel for EventPayload

In addition to the **BasicEventElement**, the AAS specification defines the **EventPayload**, the actual content transmitted when an event is triggered. The structure of this payload is specified in the AAS Metamodel, as illustrated in Figure 7 and further explained in the following section.

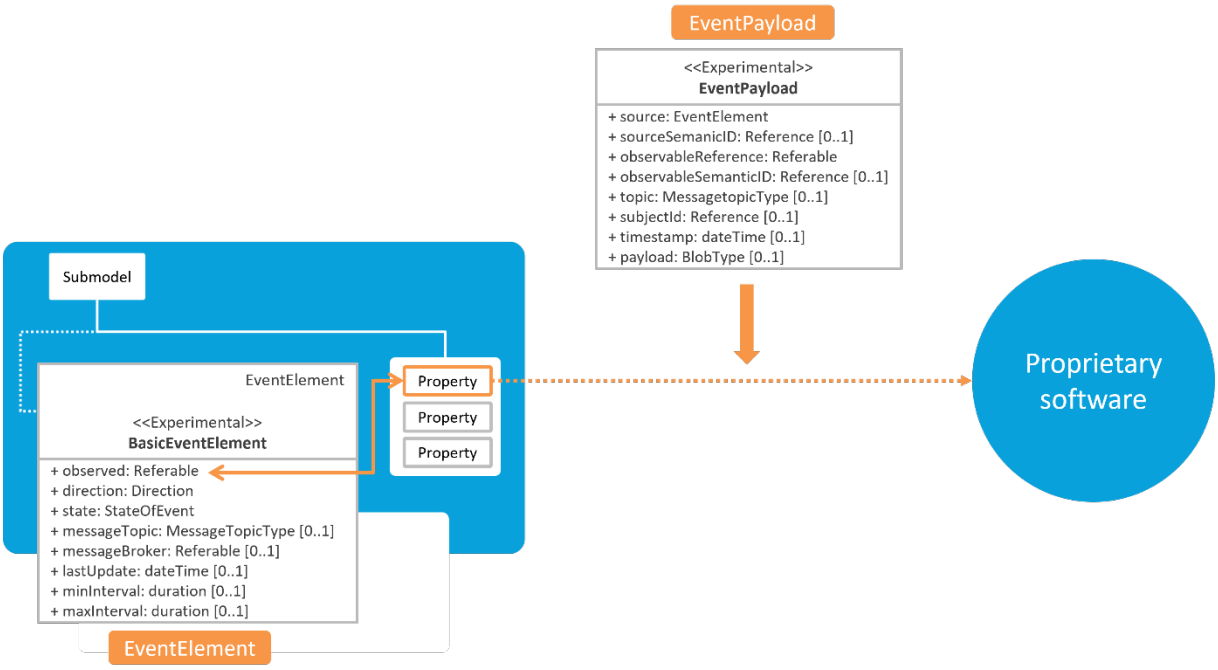

Figure 7: Event content encapsulated in the EventPayload structure

The attributes of the **EventPayload** have the following meanings ((IDTA), April 2023):

- **source**: Reference to the associated **EventElement**.
- **sourceSemanticId**: Semantic ID of the **EventElement**, preferably as an external reference. The concept proposes defining a unique semantic ID for each type of change—namely **create, delete,** and **update**—as identified within the context of this discussion paper. These semantic IDs are specified explicitly, providing the capability to clearly communicate and distinguish the type of change that has occurred. Additionally, User- and Use Case-specific types of changes can be defined.
- **ObservableReference:** Refers to the AAS element that has been updated or created. This is the actual "new" fact that the event intends to communicate to subscribers.
- **ObservableSemanticId:** the semantic ID of the updated or newly created element.

It is important to note that the **observableReference** in the **EventPayload** does not necessarily have to match the element referenced by the **observed** attribute of the **BasicEventElement**. They can be the same, especially in cases where the exact element being observed is also the one that was modified. This is demonstrated in the example in Chapter 2.6, where a specific **SubmodelElement,** a property, is subscribed, and its value is subsequently updated.

However, this is not always the case. In some use cases, the subscription is made to a broader element within the AAS, for example, an entire Submodel, while the actual change affects only a single **SubmodelElement**. Suppose a new property is created and assigned to the Submodel: the intended message to the subscriber would be, “You subscribed to this particular submodel, and it has been updated, specifically, a new property was added.” In such a case, the **observed** attribute of the **EventElement** points to the Submodel, while the **observableReference** in the **EventPayload** refers to the newly created property.

- **topic**: Identical to the **messageTopic** of the **BasicEventElement**.
- **subjectId**: Reference to the subject that generated the event.
- **timestamp**: Timestamp indicating when the event was triggered.

- **payload**: Event-specific content provided as a Blob element.

2.6 Relation between BasicEventElement and EventPayload

Since the two models, **BasicEventElement** and **EventPayload**, are defined independently in the AAS Metamodel and share several similarly named attributes, a natural question arises: *What is the relationship between these two elements, and how should their individual attributes be interpreted and modelled together?* A possible interpretation is illustrated in Figure 8, using the subscription to a property as a guiding example.

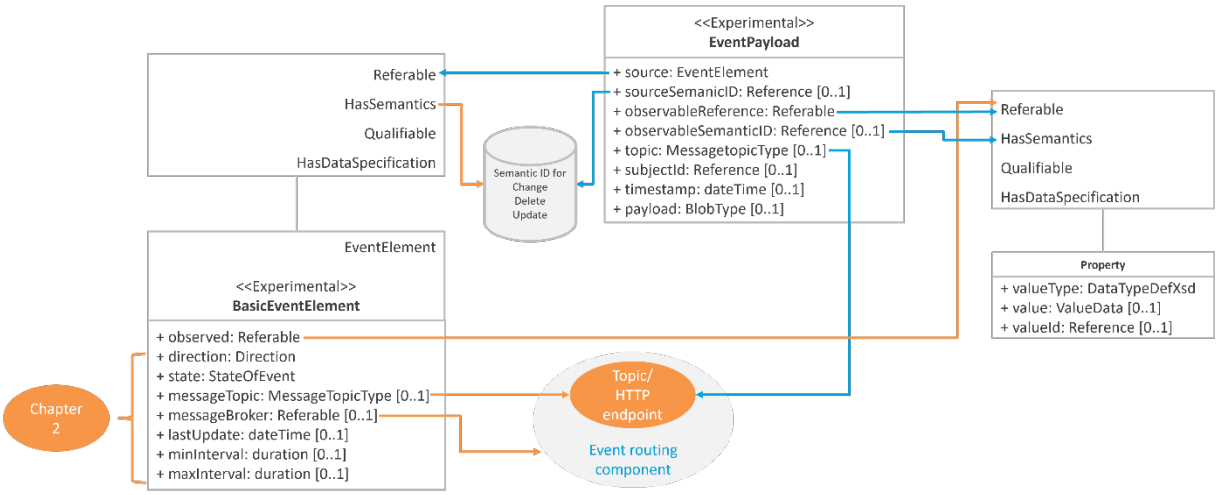

Figure 8: Relationship between BasicEventElement and EventPayload in the Modelling process

It is important to note that this example represents only one specific case, namely, the scenario where the **SubmodelElement** being observed is exactly the same element that was updated. In such a case, the values of the **observed** attribute (in the **BasicEventElement**) and the **observableReference** attribute (in the **EventPayload**) are identical.

However, this one-to-one correspondence is not always present. As described in the previous chapter, there are other event handling patterns, where the observed element is broader, for example, a submodel, while the actual update occurs within one of its contained elements. In such cases, as mentioned above, the **observed** attribute may point to the entire submodel, while the **observableReference** in the **EventPayload** refers to the specific element (e.g., a newly created property) that was changed. Further examples of this subscription pattern are discussed in Chapter 5.

It should be also noted that, according to the metamodel specification, the **BasicEventElement** itself is a **SubmodelElement** and thus inherits descriptive attributes, including **Referable**, **HasSemantics**, **Qualifiable**, and **HasDataSpecification**.

- The **Referable** attribute provides the unique path to identify the **BasicEventElement** within an AAS.
- The **HasSemantics** attribute indicates the type of change the **EventElement** is intended to notify about, e.g., **create**, **update**, **delete**, or custom types, and must be uniquely identified via a semantic ID.
- The **observed** attribute references the **Referable** of the AAS element that is being monitored.

The specification of other attributes follows the descriptions provided in Chapter 2.5.

On the **EventPayload** side, the **source** attribute refers to the **Referable** of the originating **BasicEventElement**. The **sourceSemanticId** reflects the same semantic meaning as the **HasSemantics**

attribute in the **EventElement**. The **observableReference** should reference the AAS element that was actually modified, which may or may not match the observed attribute of the **EventElement**. The **observableSemanticId** holds the semantic ID of that changed element. Lastly, the **topic** attribute carries the same information as the **messageTopic** in the **BasicEventElement**.

3 Generic Use Cases for the Application of Event Mechanism

3.1 System Architecture

Figure 9: Event subscription architecture with a middleman between publisher and subscriber

The use of typical MQTT terminology in the specification of the **EventElement** might create the impression that the use of MQTT with an MQTT broker and MQTT topics is implicitly assumed. However, this is not entirely correct, as it is not explicitly defined in the specification. Other technologies are therefore not excluded. Nevertheless, other technologies often use concepts similar to those of an MQTT broker and topics.

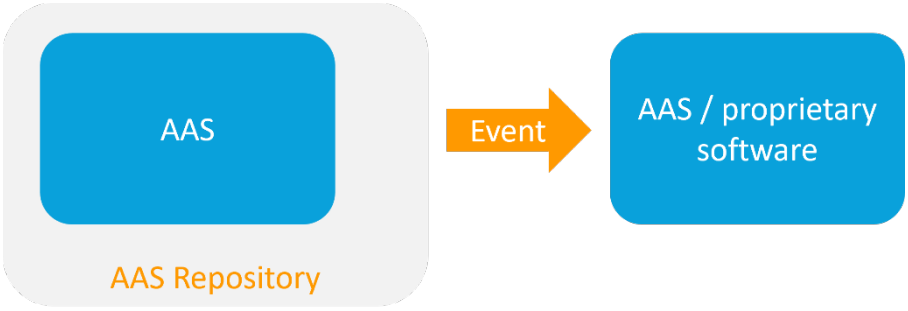

Figure 10: Direct subscription to the event source without intermediary components

Therefore, a technology-agnostic architectural approach shown in Figure 9 can be derived: the use of event routing components that act as intermediaries between event publishers and subscribers. Depending on the chosen implementation technology, this component may take on different forms. For example, in the case of an HTTP/REST interface, public servers act as an analogy to a broker, and endpoints serve as the equivalent of topics to which events are sent.

The implementation concepts made possible by the definition of the **BasicEventElement** are highly flexible, making them relevant even in the absence of a event routing component. This implementation variant is illustrated in *Figure* 10.

In this case, an AAS or another proprietary application can retrieve events directly from another AAS or its repository without requiring event-routing components such as a broker.

3.2 Use Cases

3.2.1 The external software application subscribes the updates of the AAS

The most common Use Case is likely the subscription to updates of an AAS by an external application, as illustrated in Figure 11. In this scenario, the modeler creates a **BasicEventElement** within the AAS and attaches it to the corresponding Submodel. Possible options for this setup are described in the chapter 2.4. Following the outlined modelling process, the modeler configures the subscription. The **observed** attribute references the element being monitored, while the **direction** attribute is set to **OUT**. This indicates that the event is sent externally, meaning the AAS acts as the data source or event source. If an event-routing component is used, the **messageTopic** and **messageBroker** attributes are configured accordingly.

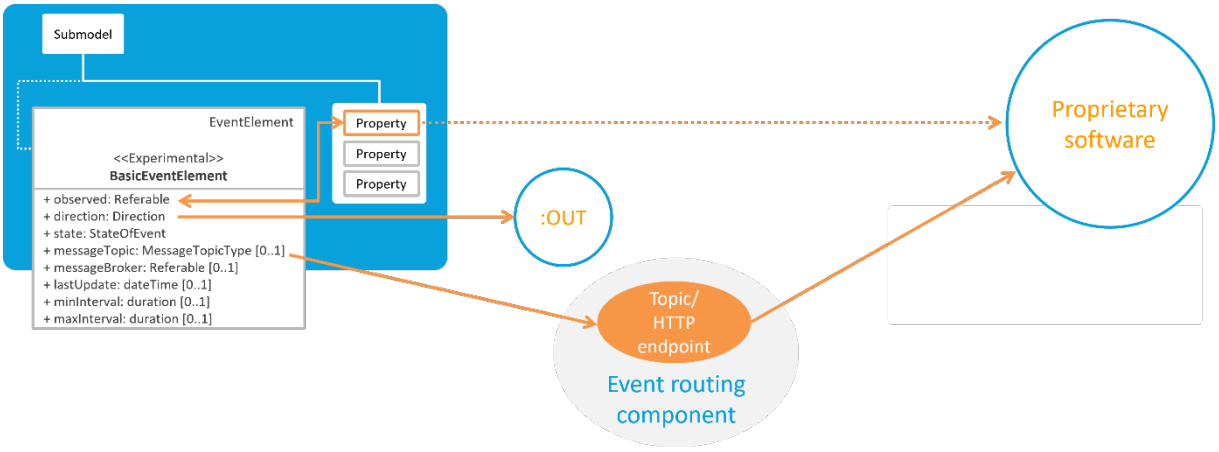

Figure 11: External software application subscribing to updates from the AAS

3.2.2 AAS subscribes the external Data Source

The Event Mechanism not only allows the configuration of subscriptions on the event sender side but also enables an AAS to act as a subscriber to events originating from external applications. To configure the handling of incoming events and the subscription itself, the **BasicEventElement** is also used. This scenario is illustrated in Figure 12.

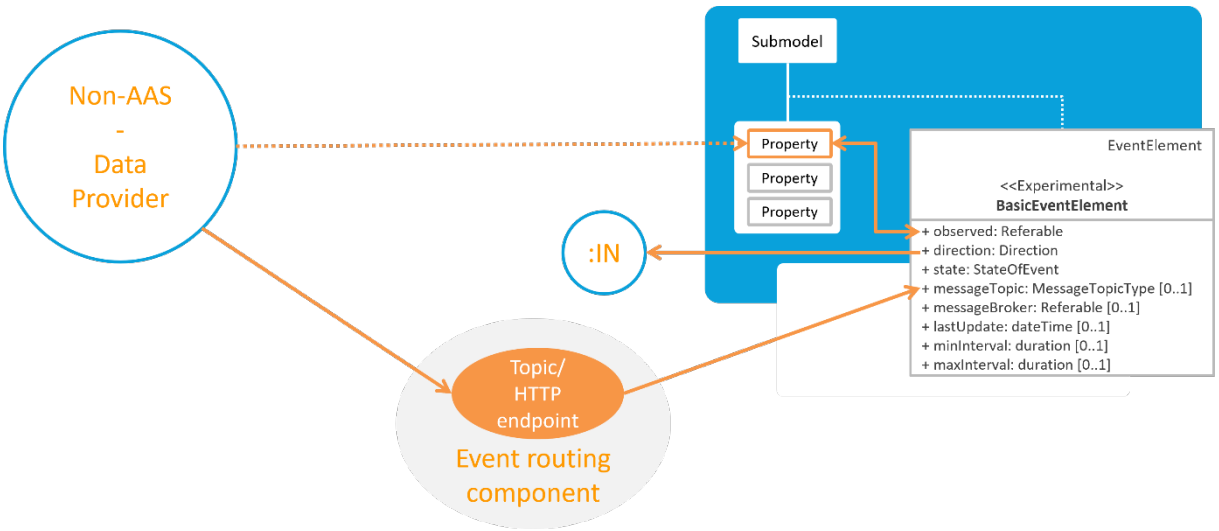

Figure 12: Subscription-based data acquisition from an external source by the AAS

In this setup, the **BasicEventElement** is configured to determine how the AAS processes the information received from an incoming event. Unlike the previous case, the configuration of the **observed** attribute in this scenario does not refer to the element being monitored, but rather to the element within the AAS that is affected or influenced by the observed change.

Additionally, the **direction** attribute is set to **IN** to indicate that the event is being received by the AAS, clarifying its role as an input. The remaining attributes are configured similarly to the previous scenario. In the **messageTopic** and **messageBroker** attributes, the address that the AAS needs to subscribe to is specified.

3.2.3 One AAS subscribes the updates of another AAS

In the possible subscription scenarios for updates from and through an AAS, an external software application does not necessarily have to play a role. In the previously described scenarios, an external application acted as both the event receiver and event source.

As an extension of these scenarios, and thanks to the ability of the **BasicEventElement** to model both incoming and outgoing events, another scenario becomes possible. An AAS could be able to subscribe to updates from another AAS. This scenario is illustrated in Figure 13.

In this case, the AAS that serves as the event source is configured with an **EventElement** where **direction** = **OUT**, while the subscribing AAS contains a corresponding **EventElement** with **direction** = **IN**. If an event routing component (e.g., a broker) is used, it is configured according to the modelling approach described in the previous chapters of this document.

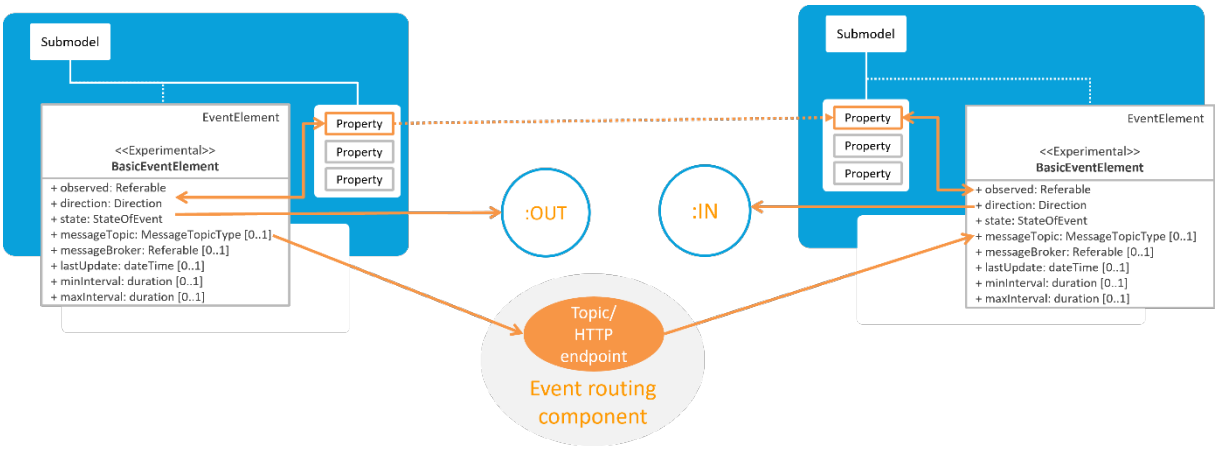

Figure 13: Event-based update mechanism between two AAS

4 Technological implementations

4.1 General

This chapter describes possible technological implementations for transporting events between the event source and event subscriber within the AAS framework. The AAS Metamodel specification does not impose any strict requirements or definitions for event transport. Instead, the event mechanism is designed to be flexible and adaptable to different technologies.

To initiate this discussion and lay a foundation for future exploration, this chapter focuses on two widely used technologies: HTTP REST and MQTT.

HTTP REST is a widely adopted approach for implementing event mechanisms, aligning with traditional web-based architectures. Within the AAS ecosystem, REST can be used in scenarios where a subscriber actively polls for updates rather than receiving push notifications.

Chapter 4.2 outlines how HTTP REST can be utilized for event management, defining key attributes, implementation strategy, and possible use case.

4.2 HTTP REST-Based Event Handling

4.2.1 Key Attributes of AAS Events

Within this conceptual framework, two key attributes might define event implementation using REST-API which are the Event Direction and Event Mode.

Event Direction: As outlined in the AAS Metamodel, the event element includes an attribute called direction, which determines the flow of event communication. This attribute may take one of two values:

- **Inbound (IN):** Indicates that the AAS handles incoming events.
- **Outbound (Out):** Indicates that the AAS generates and transmits the events.

Event Mode: The event mode could be an additional attribute being defined to define the action of the event. Whether it is pulling from or pushing into the AAS. This concept will be particularly relevant in scenarios where different interaction models, such as push-based or pull-based event handling, are considered which is explained in the following sections.

- **Push Mode**: The AAS actively sends event data to a subscriber when an event is triggered.
- **Pull Mode**: The AAS requests event data from a event routing component or another AAS at defined intervals.

While not explicitly defined in the AAS metamodel, the concept of "event mode" is introduced to better understand how events are handled in REST API implementations and system interactions. It refers to the way event data is transmitted or retrieved. This distinction helps in designing interaction models and system architectures, but it does not represent a formal attribute or property in the AAS specification.

4.2.2 HTTP REST as a Transport Layer for AAS Events

A scenario is defined to discuss in detail about the HTTP REST-based Event Mechanism which consists of three main components.

1. AAS hosted in Server A: Detects changes and pushes event data to a server.
2. AAS hosted in Server B: Stores event data and provides an API for event data retrieval.
3. AAS hosted in Server C: Polls the public server for new event updates

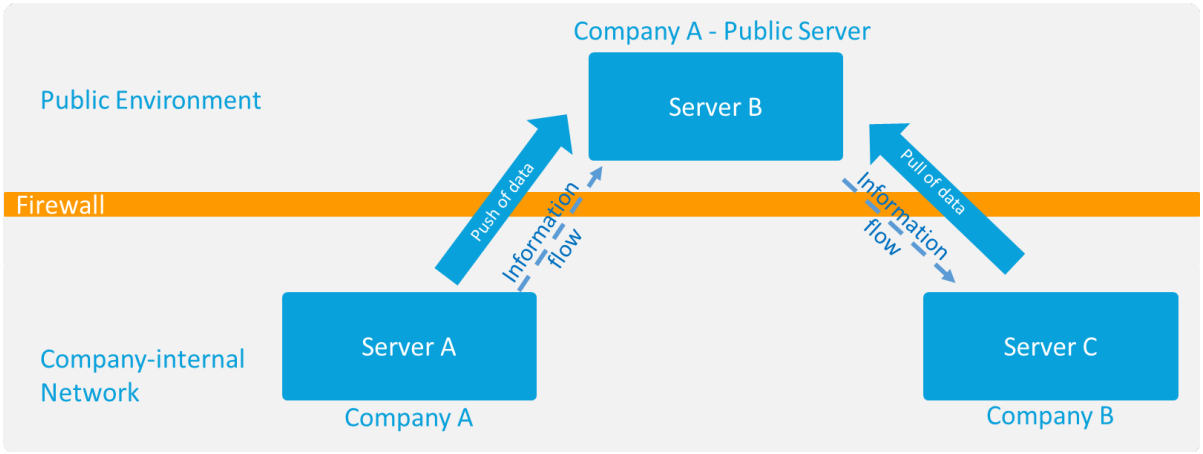

Figure 14: HTTP-based subscription setup involving a middleman service

The three components together illustrate (See *Figure* 14) the PUSH/PULL approach of the REST API, which is further explained below. Also, in this scenario both Server A and Server C are internal company servers, whereas Server B functions as a public server of Company A. A key architectural feature is the implementation of an intermediary layer—such as a firewall—between the three servers. This design enhances security by isolating the internal network from external access, thereby preventing direct exposure of AAS instances to potential external threats

4.2.3 Push scenario in EventHandling using REST API

In the push scenario, an AAS functions as an event source or publisher, actively transmitting data to a consumer (another AAS or a public server) whenever an event element is triggered. Figure 15 illustrates the corresponding Information und request flow.

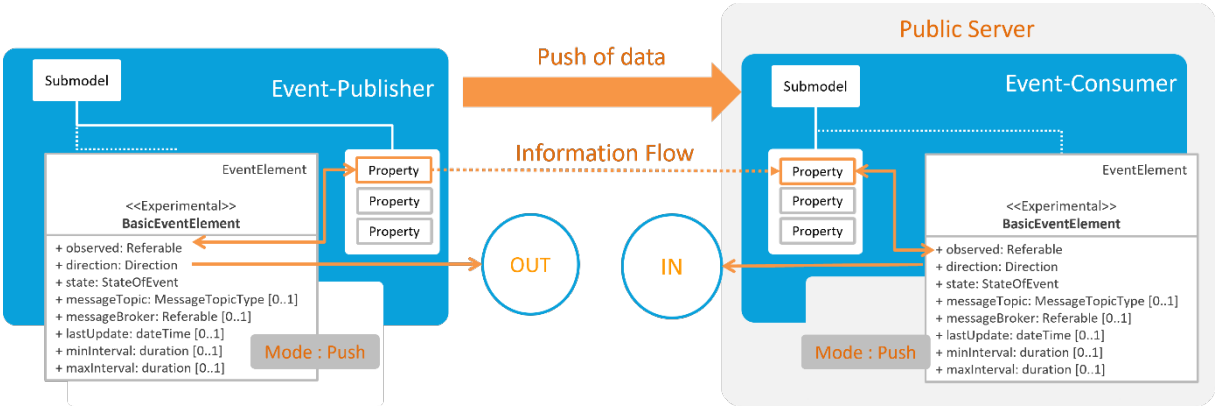

Figure 15: REST API-Based Event Handling: Push Scenario

The **EventElements** in this scenario are modelled in the same Submodel as described in chapter 2.4.2. Upon detecting the change, the event from the publisher AAS pushes the updated data to a designated Submodel hosted on a public server. The public server AAS also contains an event element that monitors the SubmodelElement where the updated information is stored. As a result, there are two interconnected events in the system.

The **EventElement** in the publisher AAS is responsible for monitoring changes and actively pushing updates to the public server. It is modeled as follows:

- **observed:** The Submodel or SubmodelElement in the AAS where the change is detected.
- **direction:** Out (Indicates that the event data is transmitted outward).
- **mode:** Push (The event is modelled for Push scenario).
- **state:** On (The event mechanism is active).
- **messageTopic:** The endpoint of the SubmodelEslement in the publis server where the update is to be pushed.

The **EventElement** in the public server AAS is responsible for observing the update and ensuring that the event is processed. It is modelled as follows:

- **observed:** The Submodel (SM) or SubmodelElement (SME) where the update takes place.
- **direction:** In (Indicates that the data from the events is incoming).
- **mode:** Push (The event is modelled for Push scenario).
- **state:** On (The event mechanism is active).

The payload is delivered via REST API using the PUT method, ensuring that the data is updated efficiently at the designated endpoint.

4.2.4 Pull Scenario in Event Handling using REST API

In a pull-based event handling architecture, the AAS is responsible for actively retrieving event data from a remote AAS server or another AAS instance. Unlike the push model, where updates are automatically sent, the pull model requires the consumer AAS to initiate data retrieval by sending pull requests at predefined intervals or in response to specific triggers.

As shown in Figure 16, the public server maintains an **EventElement** that observes a **Referable** element. When a change is detected in the observed element, the event is triggered and the corresponding EventPayload is updated with the latest data. The Consumer AAS polls the event endpoint where the update is stored by issuing HTTP GET requests. Upon detecting a change often determined by comparing timestamps the Consumer AAS parses the **EventPayload** and uses the contained information to dynamically create or update a **SubmodelElement** within its own AAS instance.

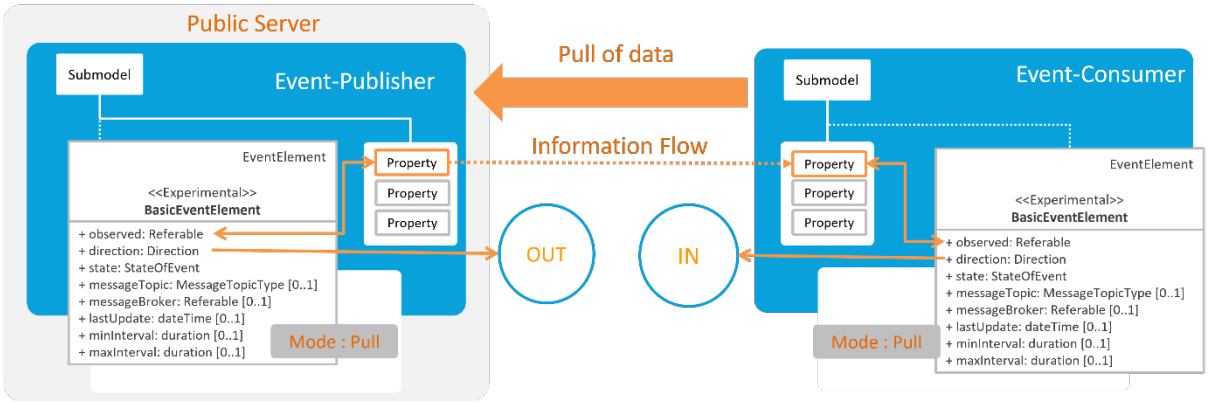

Figure 16: REST API-Based Event Handling: Pull Scenario

The **EventElement** in the AAS of the public server is responsible for monitoring changes and storing recent updates. It is structured as follows:

- **observable:** The Submodel (SM) or SubmodelElement (SME) in the AAS where the change is to be detected.
- **direction:** Out
- **mode:** Pull (The event element operates under the pull model).
- **state:** On (The event mechanism is active).

The **EventElement** within the Consumer AAS is configured to poll the public server for updates. It is modelled as follows:

- **observable:** The Submodel (SM) or SubmodelElement (SME) where the update takes place.
- **direction:** In
- **mode:** Pull (The event element operates under the pull model).
- **state:** On (The event mechanism is active).
- **messageTopic:** The endpoint of the SubmodelElement where the update is stored in the public server

This pull-based configuration optimizes network efficiency by ensuring that data is retrieved only when explicitly requested thus preventing unnecessary data transmission. By employing this structured pull mechanism, the system ensures efficient and controlled event retrieval, making it particularly suitable for scenarios where event frequency varies or where excessive data traffic needs to be avoided.

To ensure secure event processing, the REST API incorporates multiple layers of protection, including authentication, encrypted communication via HTTPS (TLS), and digital signatures. Additionally, rate

limiting is implemented to prevent excessive polling. These security mechanisms will be described in more detail in the subsequent versions.

4.3 MQTT

4.3.1 MQTT as a Transport Layer for AAS Events

MQTT (Message Queuing Telemetry Transport) provides a lightweight, scalable, and efficient mechanism for handling event-based communication. This section describes how MQTT can be used for event-driven data exchange within the AAS ecosystem. As illustrated in the Figure 17 the MQTT can serve as the transport protocol to facilitate the transport of AAS Events, allowing different components, such as AAS instances, industrial applications, or external monitoring systems, to publish and subscribe to event data.

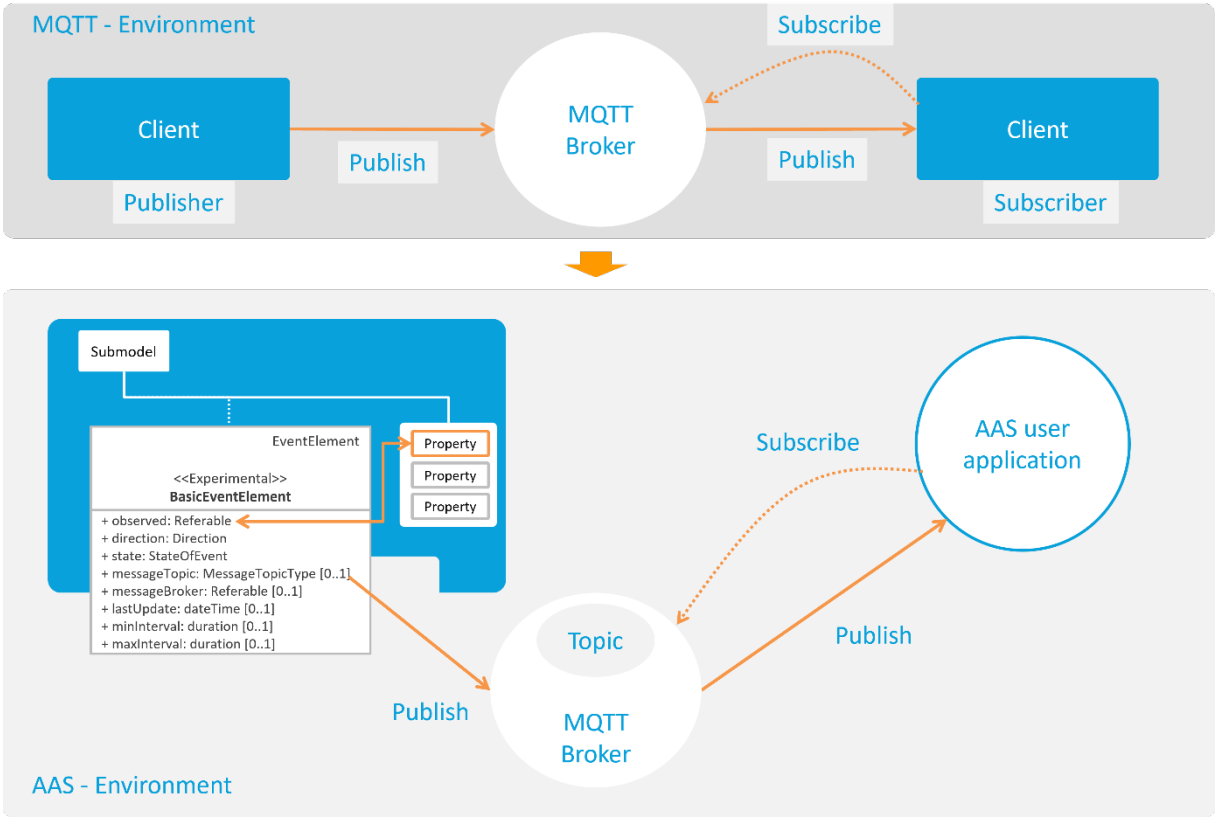

Figure 17: MQTT-based communication between AAS and external systems

In an MQTT-based AAS event architecture, the following roles can be defined:

- **AAS as Event Source:** Publishes change notifications related to its Submodels.
- **AAS user application:** Subscribes to relevant topics to receive change notifications from the subscribed AAS.
- **MQTT Broker:** Routes event messages between event sources and subscribers.

4.3.2 Architecture and Message Flow

Event Publishing (AAS → MQTT Broker)

When an AAS detects a relevant change, it publishes an event message to an MQTT topic.

- The AAS **EventElement** is configured with:
  - **messageTopic**: Defines the MQTT topic for the event.
  - **messageBroker:** Specifies the broker address.

- **direction**: OUT(Indicates an outgoing event).

Event Subscription (Subscriber → MQTT Broker)

A subscriber such as another AAS instance, is an AAS user application, or a monitoring system subscribes to specific MQTT topics.

- The AAS **EventElement** is configured with:
  - **messageTopic:** The same topic as the publisher.
  - **messageBroker:** Address of the MQTT broker.

Event Transmission Workflow

1. AAS publishes an event to an MQTT topic.
2. MQTT broker forwards the message to all subscribers.
3. Subscribers process the event and react accordingly.

5 Generic Event Subscriptions Patterns in the AAS

5.1 Application of the Event Mechanism Across Multiple Use Cases

The event mechanism described here can be applied to a wide range of use cases. The most common scenario, already introduced in Chapter 2.6, involves subscribing to an element within the AAS and receiving a notification when that element is changed.

However, from the authors' perspective, there are additional generic use cases that are likely to occur frequently in practice. One such case arises when a subscription targets a Submodel or another higher-level structure element, and a different, yet associated, AAS element is changed or created. In other words, instead of subscribing to each element individually, a client can subscribe to a parent element and receive notifications about structural changes within its hierarchy. This scenario is further detailed in Chapter 5.2.

Another relevant use case addresses notifications concerning structural changes at the AAS level itself. In this variant, the entire AAS is subscribed to, and the subscriber is notified when, for example, a new Submodel is created. This scenario is described in Chapter 5.3.

5.2 Generic Event Handling for Changes within Submodel Hierarchies

5.2.1 Explanation of the Subscription Pattern

In this chapter, an exemplary modeling scenario is described in which a subscription is made to a specific **SubmodelElement**, and notifications are triggered when another, associated SubmodelElement is created or updated.

To illustrate the concept, the modelling is based on the Product Change Notification (PCN) submodel (IDTA, 2024). A central part of this submodel is a **SubmodelElementList** named Records, which contains multiple entries, each representing a change record as a **SubmodelElementCollection**.

In the described use case, the client application subscribes to changes within a specific entry of the record list (**SubmodelElementList** “Records”). Whenever a new record (i.e., a new instance of a **SubmodelElementCollection** “Record_0001_”) is created and added to the list, the subscribing application .receives a corresponding notification.

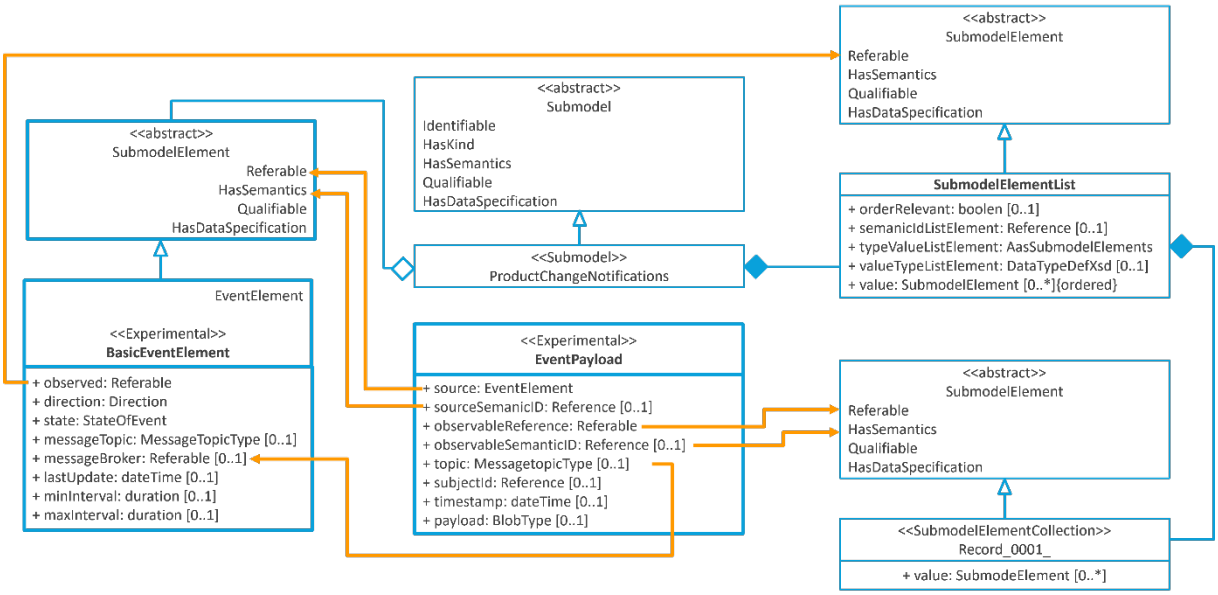

Figure 18: UML class diagram for the use case

Figure 18 illustrates the relationships between the elements of the AAS that are involved in this modelling case. The scenario is represented as a class diagram.

A Submodel called Product Change Notifications contains a **SubmodelElementList** named “Records”. This list is the object of the subscription. To enable this, a **BasicEventElement** is created and attached to the Submodel Product Change Notifications.

When the subscribed **SubmodelElementList** is updated, for example, by creating a new record, the subscriber should be notified accordingly. Upon the occurrence of such an event, the subscriber receives an **EventPayload** as part of the notification.

To properly understand the semantics of the received payload, it is not sufficient merely to indicate that a new **SubmodelElement** was created. The modelling must also allow identifying where this element belongs within the AAS structure, specifically:

- to which parent element it is linked, and
- to which Submodel it belongs.

This relationship must be clearly expressed through the modelling approach. Figure 19 deepens the example by presenting an object diagram derived from the class diagram. The object diagram visualizes the modelling with more concrete details, such as example attribute values that are essential for better understanding the connections shown in Figure 18. For the sake of clarity, only a subset of attributes is shown in the object diagram to exemplify the key aspects without overloading the representation.

The **observed** attribute of the **BasicEventElement** references the Referable of the subscribed **SubmodelElement**, in this example, a **SubmodelElementList**. The **hasSemantics** attribute of the **BasicEventElement** specifies the type of change that should trigger the notification. In this case, the notification is issued when a new instance (e.g., Record_001) is created within the selected **SubmodelElementList**. The semantic identifier used for this creation event has the following value: https://www.AAS.io/Event/create.

Other attributes of the **BasicEventElement** are to be specified in accordance with the definitions provided in Chapter 2.

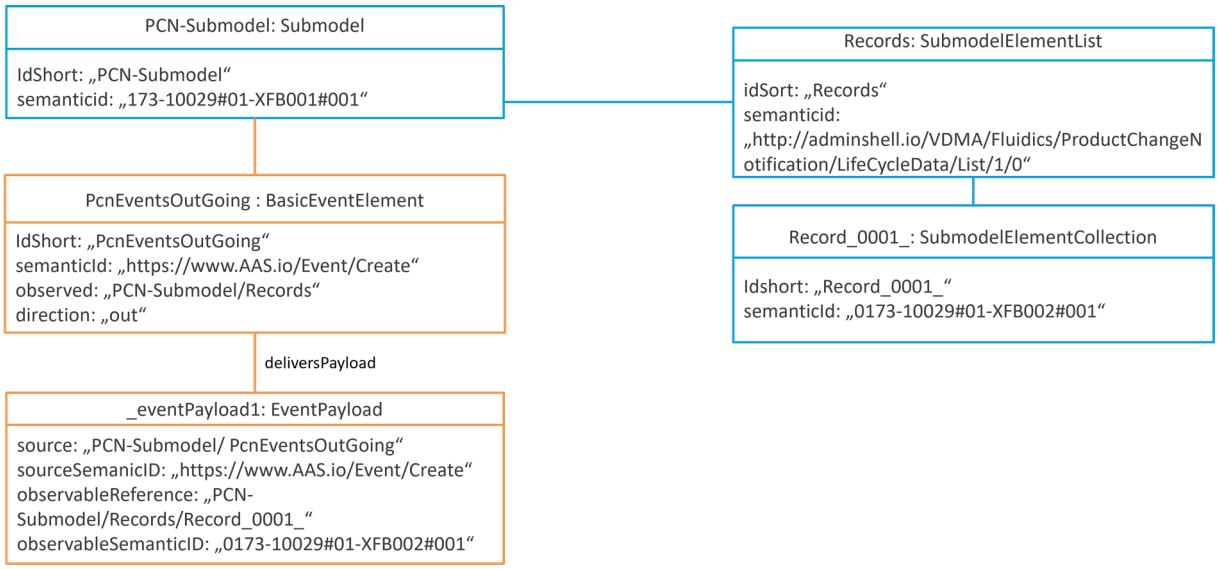

Figure 19: Object diagram for the use case

The **source** attribute in the **EventPayload** contains the **Referable** of the associated **BasicEventElement**, i.e., the path that uniquely identifies this **EventElement** within an AAS. The **sourceSemanticId** attribute of the **EventPayload** corresponds exactly to the **hasSemantics** attribute of the **BasicEventElement**.

In contrast to the observed attribute of the **EventElement**, the **observableReference** attribute of the **EventPayload** refers not to the monitored **SubmodelElementList** itself, but to the **Referable** of the newly created instance, in this case, the **SubmodelElementCollection** Record_0001_.

The **observableSemanticId** contains the semantic ID of the created Record_0001_ element. The topic attribute of the **EventPayload** carries the same value as the **messageTopic** attribute defined in the **BasicEventElement**.

This use case is applicable to a wide range of practical implementations. A subsequent chapter outlines potential application ideas, illustrating how the approach can be utilized with existing submodels and in real-world usage scenarios.

5.2.2 Example Use Case: Automated firmware update notification using PCN submodel

A manufacturer relies on a supplier for a critical electronic component. The supplier releases a firmware update for this component.

- The supplier’s AAS registers a product change by adding a new entry to the list of records within the PCN Submodel.
- The new record contains details about the firmware update of a component, with the changes represented through various SubmodelElements of the record.
- The PCN event element (PcnEventsOutgoing) continuously monitors the Submodel for newly added change records.
- Upon detecting a new entry, the event is automatically triggered within the AAS.
- The event mechanism which is configured in the AAS itself pushes the new record’s information to subscribed clients, ensuring that the manufacturer is promptly informed.
- The manufacturer retrieves the update details from the repository. Subsequently, the production and quality teams adjust assembly and testing procedures to accommodate the new firmware version.

5.2.3 Example Use Case: Real-Time Temperature Monitoring with the Time Series Data Submodel

The Time Series Submodel (IDTA, 2023) is primarily designed for the semantic representation and integration of time-stamped data within the AAS. It organizes and stores time-series data using internal, linked, or external segments depending on the requirements of the specific use case.

In a factory environment, temperature sensors are deployed to continuously monitor critical equipment. Special attention is required when the temperature reaches defined threshold (e.g 90°C or higher), as this may indicate abnormal or potentially hazardous conditions. These sensors are modelled into the factory's AAS and utilize the Time Series Data Submodel to systematically record temperature readings along with their timestamps, enabling precise tracking and analysis over time.

- The temperature sensor monitors a machine and continuously records temperature values. Each reading is stored in the Time Series Data Submodel, with corresponding timestamps to form a structured historical log.
- A **BasicEventElement** is modeled within the same AAS and configured to observe the time series records. Rather than triggering on every new reading, the **EventElement** is set up to activate only when a critical condition is met, specifically, when a temperature value meets or exceeds the defined threshold.
- Upon such a trigger, the **EventElement** generates an **EventPayload** containing the temperature value and timestamp and sends this payload to a subscribed AAS user application using the configured communication mechanism (e.g., HTTP or MQTT).

5.2.4 Example Use Case: Component replacement in the BOM

The Hierarchical Structures Enabling Bills of Materials (IDTA, 2024) provides a standardized and structured approach to representing the composition and relationships of industrial assets. It allows the hierarchical modelling of industrial equipment, covering components, subsystems, and complete systems in an interoperable manner.

The BOM submodel contains the Entity elements EntryNode and Node (IDTA, 2024) represent different levels within this hierarchy, enabling a structured breakdown of complex systems and are used to define the components and subsystems within a hierarchical structure. An **EventElement** in the AAS can be utilized to monitor and notify structural changes in the BoM. Whenever a new component is added, modified, or removed the event mechanism can trigger an update that is pushed to relevant AAS user systems.

In a modern conveyor system, various mechanical and electronic components such as sensors, rollers, motors, pulleys, and idlers work together to enable continuous material movement.

- During routine operations or maintenance, a situation arises where a motor in the conveyor system is replaced. The BoM Submodel is promptly updated to reflect the installation of the new motor, including its serial number, vendor information and technical parameters.
- An **EventElement** within the AAS is configured to monitor the BoM Submodel for changes in its structural composition or metadata. Once the new motor's data is updated, the **EventElement** detects this structural change and triggers an event indicating that a component replacement has occurred.

5.3 Subscription to Structural Changes at AAS Level

5.3.1 Explanation of the Subscription Pattern

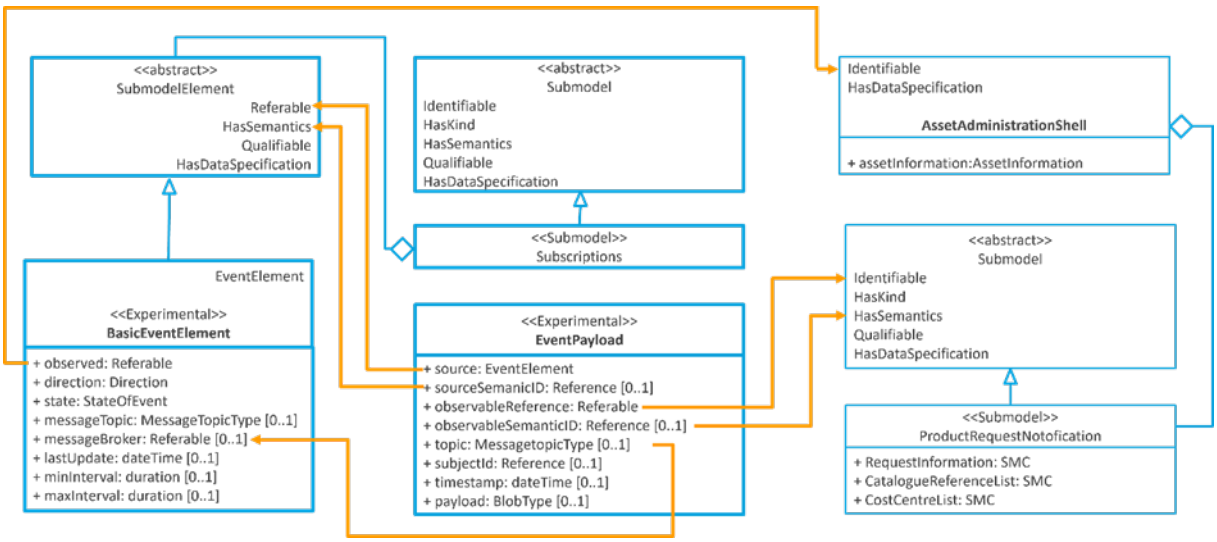

Figure 20: UML class diagram for the use case

Figure 20 illustrates an exemplary modelling of the scenario in which a subscription is configured to monitor the addition of new Submodels to an Asset AAS. In this scenario, the **observed** attribute of the **BasicEventElement** references the AAS ID of the monitored AAS. The **hasSemantics** attribute of the **BasicEventElement** is set to: https://www.example.com/Event/Create.

The associated **EventPayload** is structured in the following way: the **source** attribute points to the **Referable** of the corresponding **BasicEventElement**, providing a unique identification within the AAS. The **sourceSemanticId** exactly matches the **hasSemantics** attribute of the **BasicEventElement**. The **observableReference** attribute refers to the newly added Submodel, while the **observableSemanticId** contains the **semanticId** of this newly created Submodel. The **topic** attribute adopts the same value as the **messageTopic** attribute from the **BasicEventElement**.

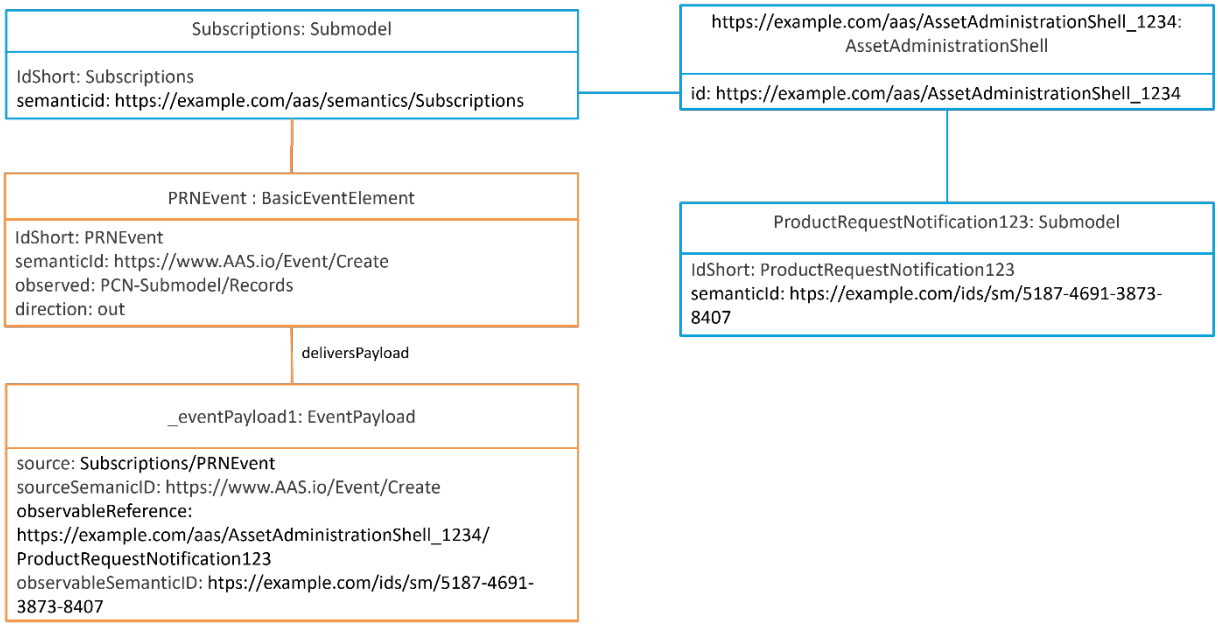

Figure 21: UML object diagram for the use case

The **payload** of the event itself consists either of the completely new Submodel or a link to it, depending on the implementation. To further clarify this modeling approach, Figure 20 first presents a class diagram outlining the relationships at a generic level, followed by an object diagram (See Figure 21) that instantiates selected attributes.

5.3.2 Example Use Case: Procurement Workflow with PRN, PRR, and POC Submodels

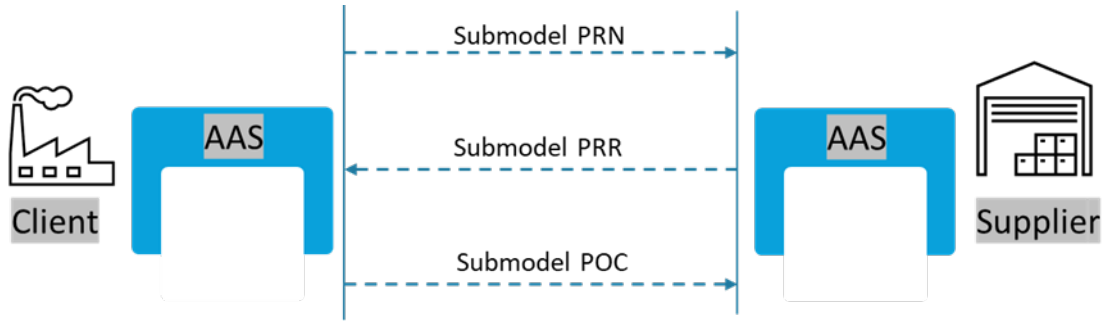

Figure 22: Data Exchange during Procurement using PRN, PRR, and POC Submodels (InterOpera, 2023)

A high-tech electronics manufacturer (client in the Figure 22) requires a custom circuit board from a supplier for a new product. The client's design department provides detailed CAD drawings and specifications through the product's AAS. The client's purchasing department initiates the procurement process by sending a PRN to potential suppliers. This is achieved using the PRN Submodel, transmitted via the ERP system. Upon receiving the PRN, the supplier's production planners assess feasibility, plan resources, and calculate costs. They then prepare an offer using the PRR Submodel, automatically populating a structured offer template. The offer is sent back to the client, who reviews it, negotiates if needed, and awards the contract to the selected supplier. The final purchase order is issued using the POC Submodel, consolidating the relevant data from the previous steps (InterOpera, 2023). The overall sequence of data exchange between the client and supplier AAS, including the use of PRN, PRR, and POC Submodels is illustrated in Figure 22.

While the Submodels (PRN, PRR, POC) provide a structured representation of data and process stages, they are inherently static — they describe *what* the data is, but not *how and when* processes should move forward. This is where the event mechanism plays a role: Events dynamically connect process stages by triggering actions automatically when changes occur in the AAS.

The event-driven automation within this procurement process works as follows:

- **Event Handling in the PRN Process:**

The creation of a PRN Submodel instance triggers an event that captures the relevant information and packages it into an **EventPayload**. This payload is automatically transmitted to the ERP system, which then forwards the PRN to potential suppliers. The event ensures that the necessary data from the Submodel is accurately and promptly communicated without manual intervention.

- **Event Handling in the PRR Process:**

On the supplier side, an event element within the supplier's AAS observes incoming PRN notifications. This event can automatically trigger internal workflows such as:

  - notifying production planners,
  - informing warehouse management,
  - and providing all necessary data for feasibility assessment, resource planning, and cost calculation.

Once the offer is prepared, the supplier generates a PRR Submodel. The creation of the PRR triggers another event, which automatically transmits the PRR back to the client's system, ensuring seamless and timely information flow between business partners.

- **Event Handling in the POC Process:**

After reviewing the PRR and completing any required negotiations, the client selects the supplier and instantiates a POC Submodel. This action triggers a POC event, which automates the dispatch of the finalized purchase order to the selected supplier, thus completing the procurement cycle.

6 Summary and Outloook

This document provides an initial explanation of the event mechanism within the AAS. It outlines possible configurations, architectural variants, implementation technologies, and practical examples using selected SubmodelElements. The goal is to initiate a broader discussion and offer guidance for modelers who are currently evaluating possible applications of AAS eventing. Another secondary goal is to evaluate the current specification of the BasicEventElement and EventPayload through the illustrative and upcoming practical implementations, and to identify potential needs for adjustments to the specification.

The presented concepts aim to support the community in understanding the potential of events in the AAS context and how they can be leveraged to increase interoperability and responsiveness in digital twin systems.

Future publications will address additional aspects in more detail, such as in-depth analyses of relevant security considerations, further modelling examples, and implementation best practices. These will also include contributions toward the evolution of the de facto standard AAS modelling frameworks, enabling a broader and more systematic use of event mechanisms aligned with the specifications and underlying architectural concepts.

If you are interested in contributing to this ongoing discussion and wish to actively participate in shaping the future of event-driven AAS Modelling, we warmly invite you to get in touch with us. Together, we can lay the foundation for a robust and sustainable semantic modelling approach using the AAS.

7 References

(IDTA), I. D. (April 2023). Specification of the Asset Administration Shell - Part 1: Metamodel. Retrieved from https://industrialdigitaltwin.org/wp-content/uploads/2025/03/IDTA-01001-3-0-2_SpecificationAssetAdministrationShell_Part1_Metamodel.pdf

IDTA. (2023). *IDTA 02008-1-1: Time Series Data.* Specification. Retrieved from https://industrialdigitaltwin.org/wp-content/uploads/2023/03/IDTA-02008-1-1_Submodel_TimeSeriesData.pdf

IDTA. (2024). *IDTA 02011-1-1: Hierarchical Structures enabling Bills of Material.* IDTA. Retrieved from https://industrialdigitaltwin.org/wp-content/uploads/2024/06/IDTA-02011-1-1_Submodel_HierarchicalStructuresEnablingBoM.pdf

IDTA. (2024). *IDTA 02036-1-0: Product Change Notifications for industrial product types and items in manufacturing.* Frankfurt am Main: IDTA.

InterOpera. (2023). *Purchase request Notification.* Stuttgart: Steinbeis Innovation gGmbH.

CONTACT:

Industrial Digital Twin Association e. V.

Lyoner Straße 18

60528 Frankfurt am Main

Phone: +49 69 6603 1939

E-Mail: info@idtwin.org