Asset Data Modelling

Guideline

Manufacturing-X

Guidance Board

Change Log

Version	1.0
Date	14 April, 2026
Document title	Asset Data Modelling Guideline
Version	1.0
Owner	Manufacturing-X Guidance Board

Contents

1 IMPRINT 5

2 Authors 6

3 Motivation and Aim 7

4 Scope and Objectives 8

5 List of Abbreviations 9

1. Introduction to Asset Administration Shell 10

2. Basic Structure of the AAS 12

3. Concept of the Submodel 13

4. Resources for Asset Administration Shell 15

5. AAS and Submodel Modelling 19

5.1. Purpose and Scope 19

5.2. Shell Modelling 19

5.3. Identification of AAS 20

5.3.1. AssetInformation and assetKind 20

5.3.2. globalAssetId, specificAssetIds, and idShort 20

5.3.3. Use of derivedFrom 20

5.4. How to Model Submodels 21

5.5. Recommended Procedure for Submodel Modelling 21

5.5.1. Define the aspect before defining the fields 21

5.5.2. Reuse standard IDTA Submodel Templates 22

5.6. Selection of SubmodelElements 23

5.6.1. Property and MultiLanguageProperty 23

5.6.2. Range 24

5.6.3. File and Blob 24

5.6.4. ReferenceElement and RelationshipElement 24

5.6.5. SubmodelElementCollection and SubmodelElementList 25

5.6.6. Operation, Capability, Entity, and event-related elements 26

5.7. Modelling Workflow and Quality Assurance 27

5.8. Checklist to Implement a compliant AAS and Submodel 27

6. Modelling Guidelines and Recommendations for the AAS 29

6.1. Asset Concepts and Digital Representation 29

6.2. Identification, Composition, and Versioning 31

6.3. Best Practices for Custom IRI Identifiers 32

6.4. Submodel Structure and Modelling Practices 35

6.5. Units and Multi-Unit Modelling 40

6.6. Multiple AAS and Lifecycle Strategies 41
6.7. Instantiation Modes of Submodels and Versioning Strategy 42
6.8. Semantics and how to use ECLASS and IEC CDD 42
7. Clustering Submodel Templates 47
7.1. Clustering of Submodel Templates (SMTs) Based on Use Cases 47
7.1.1. Digital Product Passport (DPP) 47
7.1.2. Engineering 47
7.1.3. Identification 48
7.1.4. Asset Services 48
7.1.5. Artificial Intelligence (AI) 48
8. Bibliography 49

1 IMPRINT

Publisher: Manufacturing-X Guidance Board

Manufacturing-X Guidance Board

2 Authors

Sandeep Rudra (IDTA, DAVID)

Pooja Gupta (Hochschule Kempten, DAVID)

Sebastian Eicke (HARTING Technology Group)

3 Motivation and Aim

In modern manufacturing and related data ecosystems, it is essential that organizations and applications seamlessly exchange asset information throughout the entire asset lifecycle. Without a common and interoperable data model, inconsistencies and proprietary interfaces can cause integration delays, data errors, and unnecessary engineering effort.

These challenges are addressed by the Asset Administration Shell (AAS), which provides a standardized framework for the digital representation of assets. By embedding asset information within this structure, a single, authoritative source of truth can be established, reducing the need for custom interfaces and enabling consistent data exchange across systems and domains.

4 Scope and Objectives

This guideline is a collaborative outcome of the Manufacturing-X Topic Group for Asset Data Modelling. It provides foundational explanations of the AAS concepts and structure, practical recommendations for implementation, and references to supporting standards, tools, and resources.

The objectives of this guideline are to:

- Offer foundational knowledge about the AAS and its core components to support understanding and interoperability.
- Provide implementation guidance for applying the AAS in diverse Manufacturing-X domains and use cases.
- Supply reference materials and resources, including specifications, templates, and tooling information, to facilitate consistent and compliant AAS adoption across organizations.

Target Audience

- This guideline is intended for:
  - Engineers and architects designing Digital Twin and AAS solutions
  - Software developers implementing AAS-based services and tools
  - Data modelers defining Submodels and semantic identifiers
  - Manufacturing-X projects and consortium partners aligning interoperability

Conventions used

- The key words MUST, MUST NOT, SHOULD, and MAY are used to indicate requirement strength:
  - MUST / MUST NOT: mandatory for compliance
  - SHOULD: strongly recommended, deviations need justification
  - MAY: optional, depends on use case

5 List of Abbreviations

Abbreviation	Full Form
AAS	Asset Administration Shell
AASX	Asset Administration Shell Exchange Package
AI	Artificial Intelligence
API	Application Programming Interface
BOM	Bill of Materials
CDD	Common Data Dictionary
DIN	Deutsches Institut für Normung
DPP	Digital Product Passport
ECLASS	Electrical Classification and Standard
IEC	International Electrotechnical Commission
ID	Identifier
IDTA	Industrial Digital Twin Association
IRDI	International Registration Data Identifier
IRI	Internationalized Resource Identifier
ISO	International Organization for Standardization
IT	Information Technology
JSON	JavaScript Object Notation
OPC UA	Open Platform Communications Unified Architecture
PLC	Programmable Logic Controller
RC	Release Candidate
REST	Representational State Transfer
SDK	Software Development Kit
SI	International System of Units
SMT	Submodel Template
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
XML	Extensible Markup Language

1. Introduction to Asset Administration Shell

The Asset Administration Shell is an interoperable implementation of the digital twin. It is standardized in the IEC 63278 series. As a digital representation of an asset (e.g. component, device, machine, plant) over its entire life cycle, it acts as the "life cycle file" of an asset and is therefore the linchpin in the value creation system. An AAS can already exist before the actual asset has been developed or produced (e.g. as early as the quotation preparation stage). The AAS can also be used for assets that do not have a communication interface (e.g. screws).

The most important features and roles defined for the Asset Administration Shell:

- An AAS has a reference to an asset and represents it digitally,
- an AAS provides one or more interfaces,
- an AAS references one or more Submodels,
- an AAS user application accesses the information of the AAS via data interface(s).
- A Submodel template is used to create a Submodel according to this template,
- a Submodel element can refer to detailed models, dictionaries and ontologies,
- Detailed models, dictionaries and ontologies define the common vocabulary as the basis for interoperability,

Submodels can refer to the asset services that are provided by an asset via an asset integration; reference can be made to other services associated with the asset.

The AAS can be made available in different forms, see Figure 1:

AAS type 1, on the left in the picture: It is used for offline exchange as a so-called AASX package file. The AASX is a serialisation of the AAS for storage and for the semantic modelling of cross-life-cycle Submodels. It makes its entire information content available as the result of a product creation process without initiating its own application actions.

AAS type 2, in the centre of the image: The AAS API (as of April 2023, REST API) can be accessed externally and is used for the online exchange of AAS information and documents, e.g. via a REST server. To ensure that all authorised interested parties have access to it (engineering, sales, service, etc.), the AAS will in all likelihood be physically located at the enterprise level. AAS technology allows the application to be used as a cloud service. However, this is not necessarily required. This enables standardised and granular access to the content of the AAS.

AAS type 3, on the right: *AAS* with the I40 language: AAS could have a direct communication capability in the future.

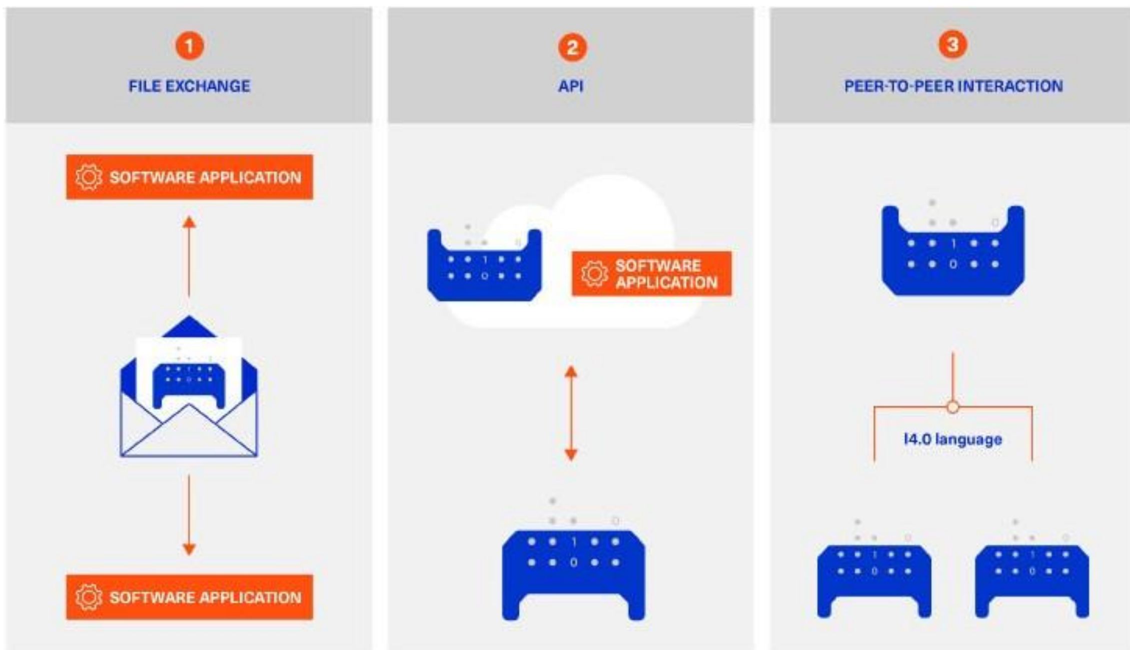

Figure 1: Types of Information Exchange via Asset Administration Shells

2. Basic Structure of the AAS

The Asset Administration Shell represents the standardized digital representation of an asset and constitutes the foundation for interoperability within the Manufacturing-X ecosystem. Its primary function is to provide a uniform structure for describing and exchanging asset-related information across systems, organizations, and domains. The AAS offers a technology-agnostic and semantically consistent framework that supports both physical and logical assets, such as machines, components, processes, software modules, and services.

An AAS is defined as a root element (AssetAdministrationShell) that contains a reference to an AssetInformation entity and one or more Submodels. Both the asset and its corresponding AAS are identified by globally unique identifiers (IRI), ensuring unambiguous referencing and traceability across heterogeneous systems. The optional derivedFrom reference is used to denote relationships between AAS versions or templates, for example to indicate inheritance from a reference model or evolution from a preceding version of the same digital representation.

Each Submodel contains a structured collection of SubmodelElements, for example, Property, Operation, File, or RelationshipElement, that describe specific aspects of the asset, such as technical characteristics, operating conditions, or configuration data. The definitions of SubmodelElements follow standardized data schemas (IEC 61360-1 and ISO 13584-42). Semantic meaning is provided through ConceptDescriptions, which define and reference the concepts used within Submodels. A ConceptDescription may reference external standards through the isCaseOf attribute. This attribute establishes semantic alignment with domain vocabularies such as ECLASS or IEC CDD and should be introduced and maintained by recognized standardization bodies or domain experts to ensure consistency and interoperability.

The AAS metamodel specifies a consistent and implementation-independent structure that supports multiple serialization formats (e.g. XML, JSON, RDF) for data exchange. Data originating from assets, whether physical devices, digital entities, or processes, can be directly mapped to the corresponding SubmodelElements. Applications can access or filter relevant Submodels according to their specific use cases. Requirements concerning security, including confidentiality, integrity, availability, and access management, are integral parts of the AAS architecture and shall be implemented in accordance with the overarching security framework defined within the Manufacturing-X environment.

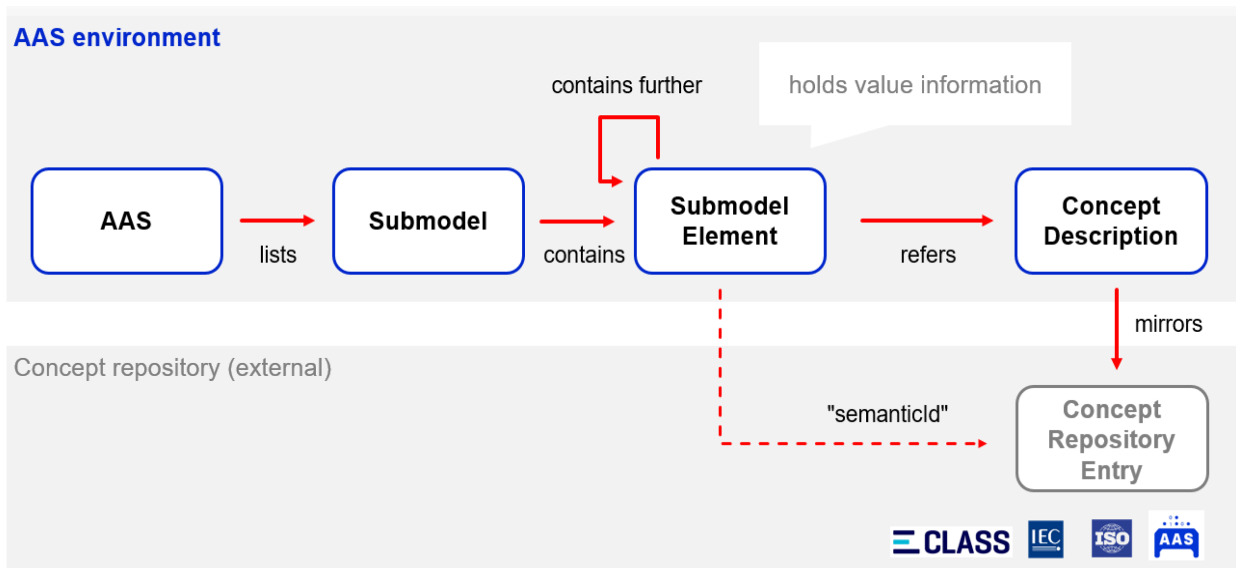

Figure 2: Structure of the AAS

3. Concept of the Submodel

A Submodel is a modular, domain-specific component of an Asset Administration Shell that encapsulates all relevant data and (optionally) services for a clearly defined aspect of an asset's functionality. By structuring parameters, states, constraints, and operating limits for a given domain (e.g. *Drilling*, *Energy Efficiency*), a Submodel provides a well-bounded "aspect view" of the asset and can be developed, versioned, and exchanged independently.

To ensure cross-vendor and cross-domain interoperability, the Industrial Digital Twin Association (IDTA) specifies and publishes standardized Submodel Templates. These templates define a common information model (structure and semantics) for frequently used domains and use cases, and are provided as AASX packages and JSON files together with formal specification documents. Typical examples include:

- Digital Nameplate (IDTA 02006) – structured asset identification and manufacturer information
- Technical Data (IDTA 02003) – technical characteristics and performance data of equipment
- Handover Documentation (IDTA 02004) – structured documentation and deliverables for commissioning and handover

Within a Submodel, asset information is represented by SubmodelElements. In practice, current IDTA Submodel Templates primarily use:

- Property – scalar or structured values (e.g. temperature, voltage rating)
- MultiLanguageProperty – multi-language values (manufacturer product designation)
- File / Blob – binary artefacts such as manuals, CAD models, configuration files or firmware images
- ReferenceElement / RelationshipElement – reference to other SubmodelElements, Submodels, AASs, or external resources

The AAS metamodel additionally supports Operation (callable functions) and Event (notifications on state changes), which can be used by Submodels where executable behaviour or eventing is required.

All SubmodelElements conform to a standardized structure, typically comprising:

- idShort – a local, human-readable identifier for use within the Submodel's namespace
- a semanticId (e.g. IRI or IRDI) referencing a ConceptDescription, which binds the element to a formally defined semantic concept (e.g. IEC CDD, ECLASS)

This consistent metamodel ensures that compliant tools and systems can parse, interpret, and validate Submodel content independently of the original authoring system.

A Submodel is exposed via the AAS interface (e.g. AAS Part 2 REST API), which provides a uniform access layer over heterogeneous data sources. Clients can discover the Submodel structure, read and write Property values, and retrieve referenced File/Blob content in a standardized way. This allows legacy or proprietary back-end systems to be wrapped by an AAS implementation: for example, process data from a legacy PLC using vendor-specific protocols can be mapped into Submodel elements, while higher-level IT/OT applications access the same information via standard AAS APIs alongside data from modern Industrie 4.0 components that natively support AAS. This

abstraction significantly reduces the need for custom integration adapters and promotes interoperable, data-space-ready integration across heterogeneous environments.

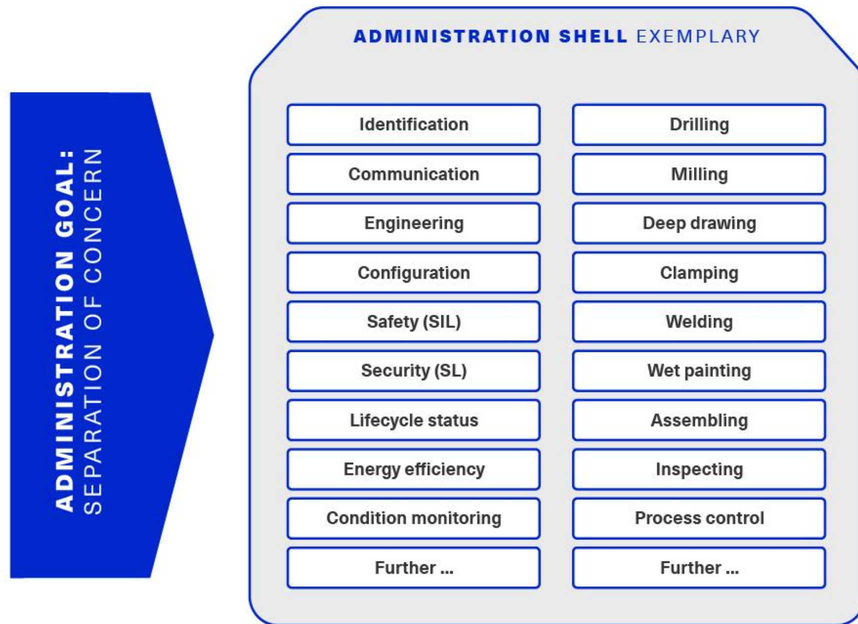

Figure 3: Examples of Different Domains Providing Properties for Submodels of the Administration Shell

4. Resources for Asset Administration Shell

AAS Specifications

The Industrial Digital Twin Association (IDTA) publishes the normative AAS specifications that define the metamodel, interfaces, and communication protocols.
AAS Specifications – IDTA
Includes Parts 1 to 5 of the AAS metamodel and protocol definitions.
Url: https://industrialdigitaltwin.org

Submodel Templates

Submodel Templates (SMTs) provide standardized and reusable data structures that ensure semantic interoperability across domains and applications.

Official Listing of Submodel Templates

Comprehensive overview of published and in-development SMTs.
Url: Submodels - IDTA

IDTA GitHub Repository for Submodel Templates

Repository containing all published templates in AASX and JSON format, including documentation and issue tracking.
Url: https://github.com/admin-shell-io/submodel-templates

IDTA Submodel Template Repository

Web-based, API-enabled repository of official SMTs with AAS interfaces for Shells, Submodels, and ConceptDescriptions.
Url: IDTA Submodel Template Repository

Software Development Kits (SDKs)

Software Development Kits implement the AAS metamodel in various programming languages. They support serialization (XML, JSON, AASX), validation, and integration into application environments.

Java – AAS4J: Implementation of AAS v3.0 (XML, JSON, AASX)

Url: https://github.com/eclipse-aas4j/aas4j

C# – AAS Core C# (v3.0 / v3.0RC02)

Url: https://github.com/aas-core-works/aas-core3.0-csharp

Go – AAS Core Golang (v3.0 / v3.0RC02)

Url: https://github.com/aas-core-works/aas-core3.0-golang

TypeScript – AAS Core TypeScript (v3.0 / v3.0RC02)

Url: https://github.com/aas-core-works/aas-core3.0-typescript

Python – AAS Core Python (v3.0 / v3.0RC02)

Url: https://github.com/aas-core-works/aas-core3.0-python

Python – BaSyx Python SDK (AAS v3.0)

Url: https://github.com/eclipse-basyx/basyx-python-sdk

Rust – BaSyx Rust SDK (AAS v3.0)

Url: https://github.com/eclipse-basyx/basyx-rust-sdk

Client Libraries

Client libraries enable programmatic access to AAS interfaces for integration in custom applications.

AAS API Python Client – Generated from the official OpenAPI specification.

Url: https://github.com/rwth-iat/aas-api-python-client

BaSyx TypeScript SDK – Toolkit for developing AAS-enabled applications.

Url: https://github.com/eclipse-basyx/basyx-typescript-sdk

Server Frameworks and Ready-to-Use Servers

These frameworks and services provide complete implementations of AAS repositories and registries conforming to the AAS REST API (Part 2).

BaSyx Java V2 Server SDK – Java framework for AAS v3 HTTP/REST servers.

Url: https://github.com/eclipse-basyx/basyx-java-server-sdk

BaSyx Go Components – Go framework for AAS v3 HTTP/REST servers.

Url: https://github.com/eclipse-basyx/basyx-go-components

Eclipse BaSyx Python HTTP Server – Repository and Submodel services.

Url: https://github.com/rwth-iat/basyx-python-sdk-http-server-docker

AASX Server – Official AAS server implementation with proprietary endpoints.

Url: https://github.com/admin-shell-io/aasx-server

FAAASt Service – Docker-ready AAS service provided by Fraunhofer IOSB.

Url: https://github.com/FraunhoferIOSB/FAAAST-Service

Editors and Modelling Tools

Editors and modelling tools are essential for creating, inspecting, and maintaining AAS structures and Submodels. These tools are most relevant for AAS and Submodel modelling activities.

AASX Package Explorer – Desktop application for authoring and inspecting AASX packages, Submodels, and ConceptDescriptions.
Url: https://github.com/admin-shell-io/aasx-package-explorer

BaSyx AAS GUI – Web-based interface for visualizing and editing AAS and Submodels.
Url: https://github.com/eclipse-basyx/basyx-aas-web-ui

AAS Manager – Cross-platform editor and viewer (PyQt + BaSyx Python SDK).
Url: https://github.com/rwth-iat/aas_manager

AAS Core React Editor – Browser-based editor for AAS v3.0RC02.
Url: https://github.com/aas-core-works/aas-core3.0rc02-react-editor

AASPortal – Node.js-based portal for managing and visualizing AAS instances.
Url: https://github.com/FraunhoferIOSB/AASPortal

Validation and Test Frameworks

Validation and testing tools ensure compliance of AAS components and Submodels with the AAS specification and related standards.

AAS Core Testgen – Property-based and combinatorial test-data generation (JSON & XML).
Url: https://github.com/aas-core-works/aas-core3.0-testgen

AAS Test Engines – Official compliance engines for validating AAS instances and REST APIs.
Url: https://github.com/admin-shell-io/aas-test-engines

BaSyx Test Orchestrator – Orchestration of automated test suites for AAS-based systems.
Url: https://github.com/eclipse-basyx/basyx-applications/tree/main/test-orchestrator

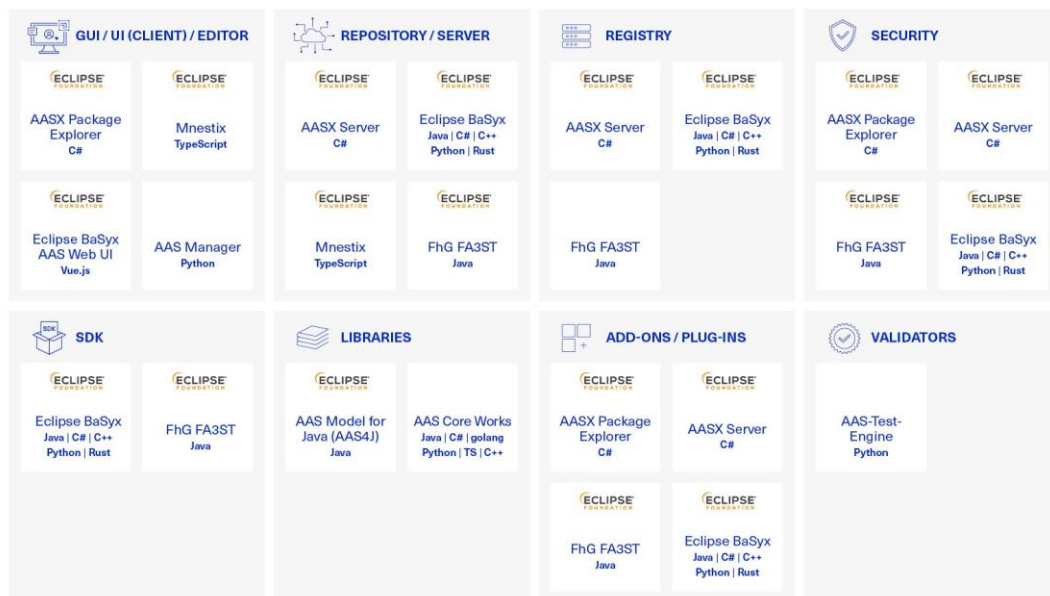

Figure 1:AAS Open Source Software Landscape

Commercial Tools

IDTA Solutions Hub – Directory of commercial AAS platforms, authoring tools, and related services.
Url: https://industrialdigitaltwin.org/solutions-hub

Cluster: Tools and Resources Relevant for Modelling

The following categories summarize the resources most useful for AAS and Submodel modelling activities.

Submodel Authoring: Tools for creating and maintaining Submodels and AASX packages.
Recommended: AASX Package Explorer, AAS Manager, BaSyx AAS GUI.

Template Reference: Access to standardized Submodel Templates (SMTs).
Recommended: IDTA Submodel Templates, IDTA GitHub Repository.

Semantic Definition: Management of ConceptDescriptions and semantic references (ECLASS, IEC CDD).
Recommended: AASX Package Explorer, BaSyx Tools.

Model Validation: Validation of Submodels and AAS against specifications and SMT standards.
Recommended: AAS Test Engines, AAS Core CLI Swiss Knife.

Code and Automation: Generation of Submodel classes and schema validation scripts.
Recommended: AAS Submodel Python Code Generator, AAS Core SDKs.

5. AAS and Submodel Modelling

5.1. Purpose and Scope

This chapter provides detailed guidance on modelling an Asset Administration Shell and its Submodels in a technically correct, semantically consistent, and interoperable manner. It addresses the modelling decisions that typically arise during implementation, including the allocation of content to the AAS level or Submodel level, the structuring of Submodels, the selection of suitable SubmodelElements, and the consistent definition and application of semantics to support reliable exchange across systems and organizations.

5.2. Shell Modelling

An Asset Administration Shell represents an asset. The shell has its own identifier. The represented asset is identified through the AssetInformation. This separation shall be maintained clearly because the shell and the asset are not the same object. The shell is the digital representation. The asset is the physical, digital, or intangible object to which that representation refers.

The AAS should contain the information required to identify the shell, identify the represented asset, and reference the relevant Submodels. Detailed technical content should generally not be placed directly at the root of the AAS. Instead, it should be organised in Submodels that reflect clear aspect views.

Example. The following snippet shows a compact AAS that identifies the shell, identifies the represented asset, and references one Submodel.

```
{
  "assetAdministrationShells": [
    {
      "modelType": "AssetAdministrationShell",
      "idShort": "aas-machine",
      "id": "https://machine.example/aas/aas-machine",
      "assetInformation": {
        "assetKind": "Instance",
        "globalAssetId": "https://machine.example/assets/machine-001"
      },
      "submodels": [
        {
          "type": "ModelReference",
          "keys": [
            {
              "type": "Submodel",
              "value": "https://machine.example/submodels/DigitalNameplate/1/0"
            }
          ]
        }
      ]
    }
  ]
}
```

5.3. Identification of AAS

5.3.1. AssetInformation and assetKind

AssetInformation contains the identifying information of the represented asset. The field assetKind shall be set explicitly because the distinction between type and instance affects how the shell is interpreted. A type AAS is suitable for a generic product definition or blueprint. An instance AAS is suitable for one concrete, individually identifiable asset.

Example. A type shell may represent a motor model. An instance shell may represent one motor with a concrete serial number.

```
{
  "assetInformation": {
    "assetKind": "Instance",
    "globalAssetId": "https://example.com/assets/motor/sn-998877",
    "specificAssetIds": [
      {
        "name": "SerialNumber",
        "value": "SN-998877"
      }
    ]
  }
}
```

5.3.2. globalAssetId, specificAssetIds, and idShort

The globalAssetId identifies the represented asset. specificAssetIds may be used where local, plant-specific, or legacy identifiers must also be retained. The idShort is a local readable identifier within the model namespace. It is useful for navigation, but it does not replace the global identifiers and it does not carry the formal semantic meaning of the element.

Example. A machine may have one global asset identifier for exchange and one internal asset number for local maintenance processes.

```
{
  "idShort": "TechnicalData",
  "assetInformation": {
    "globalAssetId": "https://machine.example/assets/machine-001",
    "specificAssetIds": [
      {
        "name": "InternalAssetNumber",
        "value": "ACM-001"
      }
    ]
  }
}
```

5.3.3. Use of derivedFrom

The derivedFrom reference should be used only where a real derivation relation exists. The typical case is an instance AAS derived from a type AAS. This allows stable type information to remain in the type shell while instance-specific values remain in the instance shell.

Example. The following excerpt shows an instance shell derived from a type shell.

```
{
  "modelType": "AssetAdministrationShell",
  "id": "https://example.com/aas/motor-instance/sn-998877",
  "derivedFrom": {
    "type": "ModelReference",
    "keys": [
      {
        "type": "AssetAdministrationShell",
        "value": "https://example.com/aas/motor-type/m1000"
      }
    ]
  }
}
```

5.4. How to Model Submodels

A Submodel is an identifiable model of the represented asset. It groups the information that belongs to one coherent viewpoint, lifecycle concern, or interoperability use case. The implementer should define the aspect first and only then determine which elements belong to that aspect.

Example. A Nameplate Submodel and a Technical Data Submodel may both describe the same machine, but they serve different purposes. The first identifies the asset and manufacturer. The second describes technical characteristics. These should therefore remain separate.

```
{
  "submodels": [
    {
      "modelType": "Submodel",
      "idShort": "DigitalNameplate",
      "id": "https://machine.example/submodels/DigitalNameplate/1/0"
    },
    {
      "modelType": "Submodel",
      "idShort": "TechnicalData",
      "id": "https://machine.example/submodels/TechnicalData/1/0"
    }
  ]
}
```

5.5. Recommended Procedure for Submodel Modelling

5.5.1. Define the aspect before defining the fields

The modelling process should begin with a clear description of the aspect to be represented. Fields should not be collected first. If the aspect is not defined clearly, the resulting Submodel tends to become a container for unrelated information.

Example. If the aspect is technical data, the model should first define that it represents technical characteristics such as voltage, power, dimensions, and operating range. It should not start by collecting arbitrary fields from a spreadsheet.

5.5.2. Reuse standard IDTA Submodel Templates

Where a published IDTA Submodel Template exists for the intended use case, it should normally be reused. Reuse of an established Submodel Template reduces modelling ambiguity and improves interoperability, since the structure and semantics are already defined in a standardized and commonly interpretable form.

When reusing an existing Submodel Template, the following principles shall be observed:

- The modeller shall first identify the Submodel Template that corresponds to the intended use case and shall then instantiate or integrate that Submodel within the AAS.

- The semantic identifiers defined by the Submodel Template shall be preserved. The identifier of the instantiated Submodel may be adapted where required by the implementation context, but the semanticId of the Submodel and the semanticIds of its contained SubmodelElements shall remain unchanged.

- The cardinality defined for the Submodel Template and for each contained SubmodelElement shall be reviewed before instantiation.

- Elements with cardinality One or OneToMany shall be treated as mandatory. Such elements shall be retained and shall be populated with the corresponding asset-specific information.

- Elements with cardinality ZeroToOne, or ZeroToMany shall be treated as optional. Such elements may be removed where they are not required for the intended use case or where the corresponding asset information is not available.

- All mandatory elements shall remain present in the instantiated Submodel. Optional elements may be included, omitted, or left uninstantiated in accordance with the modelling scope and the available data.

This procedure supports conformance with the selected Submodel Template while maintaining interoperability and implementation flexibility.

Example 1: Reuse of the Digital Nameplate Submodel Template
If the IDTA Digital Nameplate Submodel Template is reused for an industrial motor, the semanticId of the Submodel and the semanticIds of elements such as manufacturerName, productDesignation, and serialNumber shall remain unchanged. The local identifier of the instantiated Submodel may be adapted to the project or system context. However, the semantic references shall be preserved so that external systems can identify the Submodel and its elements according to the standardized template definition.

Example 2: Treatment of mandatory and optional elements
If a reused Submodel Template defines serialNumber with cardinality One and productImage with cardinality ZeroToOne, serialNumber shall be provided for the instantiated asset. productImage may be omitted if no corresponding image is available or if the element is not required for the intended exchange scenario. Omission of the mandatory element would result in a non-conformant instantiation of the template.

Example 3: Reuse of a Technical Data Submodel Template
If a Technical Data Submodel Template contains ratedVoltage with cardinality One, operatingTemperature with cardinality ZeroToOne, and accessoryList with cardinality ZeroToMany, ratedVoltage shall be populated in every instantiated Submodel. operatingTemperature may be included where the information is relevant and available. accessoryList may be omitted where no accessories are to be declared. The decisive criterion is that mandatory elements shall remain present, whereas optional elements may be instantiated according to the actual asset scope.

Example 4: Restriction on semantic modification
The instantiated Submodel may be renamed locally for implementation purposes, for example from TechnicalData to TechnicalData_MotorDrive_A1, where such naming is required by the project environment. However, the semanticId shall not be replaced by a company-internal custom identifier if the Submodel Template already defines a valid semantic reference. Such replacement would impair interoperability, because external systems would no longer be able to identify the Submodel and its elements according to the standardized template semantics.

5.6. Selection of SubmodelElements

5.6.1. Property and MultiLanguageProperty

Property should be used for a single typed scalar value. MultiLanguageProperty should be used when the value is intended for human reading and may need to be provided in more than one language. The two elements have different purposes and should not be used interchangeably.

Example. A serial number is a Property. A product designation shown in English and German is a MultiLanguageProperty.

```
{
  "modelType": "Property",
  "idShort": "SerialNumber",
  "valueType": "xs:string",
  "value": "ACM-2026-0001"
}
```

```
{
  "modelType": "MultiLanguageProperty",
  "idShort": "ManufacturerProductDesignation",
  "value": [
    {
      "language": "en",
      "text": "Academy Packaging Cell"
    },
    {
      "language": "de",
      "text": "Academy Verpackungszelle"
    }
  ]
}
```

5.6.2. Range

Range should be used where the meaning of the information is one interval with a lower and an upper bound. This is preferable to two isolated properties because the interval semantics remain explicit.

Example. A permissible operating temperature is best represented as a Range.

```
{
  "modelType": "Range",
  "idShort": "OperatingTemperature",
  "valueType": "xs:decimal",
  "min": "5",
  "max": "40"
}
```

5.6.3. File and Blob

File should be used where the model points to an external document or binary artefact. Blob should be used where the binary content itself is embedded directly in the model. In most industrial exchange scenarios, File is the more common choice.

Example. A PDF declaration is typically modelled as a File.

```
{
  "modelType": "File",
  "idShort": "CeDeclarationPdf",
  "contentType": "application/pdf",
  "value": "https://example.com/docs/ce_declaration.pdf"
}
```

5.6.4. ReferenceElement and RelationshipElement

ReferenceElement should be used when one element points to another object. RelationshipElement should be used when the relation between two referenced objects is itself part of the intended meaning of the model.

Example. A declaration file may be referenced from another Submodel through ReferenceElement. A component may be related to a machine through RelationshipElement.

```
{
  "modelType": "ReferenceElement",
  "idShort": "CeDeclaration",
  "value": {
    "type": "ModelReference",
    "keys": [
      {
        "type": "Submodel",
        "value": "ExampleSubmodelId"
      },
      {
        "type": "File",
        "value": "CeDeclarationPdf"
      }
    ]
```

```
  }
}
```

```
{
  "modelType": "RelationshipElement",
  "idShort": "MainFrameIsPartOfMachine",
  "first": {
    "type": "ModelReference",
    "keys": [
      {
        "type": "SubmodelElementCollection",
        "value": "MainFrame"
      }
    ]
  },
  "second": {
    "type": "ModelReference",
    "keys": [
      {
        "type": "AssetAdministrationShell",
        "value": "https://machine.example/aas/aas-machine"
      }
    ]
  }
}
```

5.6.5. SubmodelElementCollection and SubmodelElementList

SubmodelElementCollection should be used for one grouped object with named fields. SubmodelElementList should be used for repeated entries of the same pattern. A collection expresses structure. A list expresses repetition.

Example. Dimensions belong to one collection. A repeated list of maintenance actions is suitable for a list.

```
{
  "modelType": "SubmodelElementList",
  "idShort": "MaintenanceActions",
  "orderRelevant": true,
  "typeValueListElement": "Property",
  "valueTypeListElement": "xs:string",
  "value": [
    {
      "modelType": "Property",
      "valueType": "xs:string",
      "value": "Inspect belts"
    },
    {
      "modelType": "Property",
      "valueType": "xs:string",
      "value": "Lubricate linear guide"
    }
  ]
}
```

5.6.6. Operation, Capability, Entity, and event-related elements

Operation should be used for callable behaviour. Capability should be used where the model expresses what an asset is able to do on a semantic level. Entity is suitable where a contained or co-managed unit must be represented as part of a larger structure. Event-related elements should be introduced only when notifications or state-change related interaction is actually required.

Example. A self-test command is an Operation. A packaging ability is a Capability.

```
{
  "modelType": "Operation",
  "idShort": "selfTest",
  "inputVariables": [
    {
      "value": {
        "modelType": "Property",
        "idShort": "testLevel",
        "valueType": "xs:string",
        "value": "standard"
      }
    }
  ],
  "outputVariables": [
    {
      "value": {
        "modelType": "Property",
        "idShort": "resultCode",
        "valueType": "xs:string",
        "value": "0"
      }
    }
  ]
}
```

```
{
  "modelType": "Capability",
  "idShort": "PackagingCapability"
}
```

5.7. Modelling Workflow and Quality Assurance

A practical workflow usually begins with definition of the represented asset, continues with shell creation, then with Submodel delimitation, and only afterwards with the detailed population of SubmodelElements. This preserves the distinction between shell-level identity and aspect-level content.

Semantic references and validation should be introduced early. Waiting until the end often leads to avoidable redesign because datatypes, units, references, and structural decisions then have to be corrected too late.

Example. The following sequence is suitable as a simple modelling procedure.

1. Define the represented asset.
2. Create the AAS and AssetInformation.
3. Define the required Submodels.
4. Select suitable SubmodelElements.
5. Attach semanticId and ConceptDescriptions where needed.
6. Validate datatypes, references, units, and template constraints.

5.8. Checklist to Implement a compliant AAS and Submodel

Shell Modelling

- The AAS id and the asset id are clearly separated
- AssetInformation is present
- assetKind is defined correctly
- derivedFrom is used only where a real derivation exists

Asset Identification

- globalAssetId uniquely identifies the represented asset
- specificAssetIds are used only where additional local identifiers are needed
- idShort is used only as a local readable identifier
- AAS id, globalAssetId, and idShort are not mixed

Submodel Definition

- Each Submodel represents one clear aspect of the asset
- Unrelated information is not placed in the same Submodel
- Submodel boundaries follow the intended use case
- Different lifecycle or stakeholder views are separated where needed

Reuse of Submodel Templates

- A suitable IDTA Submodel Template is checked before a custom Submodel is created
- semanticId of the reused Submodel remains unchanged
- semanticIds of template-defined elements remain unchanged
- Cardinality is checked before instantiation
- Mandatory template elements are kept and populated
- Optional template elements are included only where needed

Selection of SubmodelElements

- Property is used for single typed values
- MultiLanguageProperty is used for multilingual text
- Range is used for lower and upper limit values
- File is used for external documents
- Blob is used only for embedded binary content
- ReferenceElement is used for references
- RelationshipElement is used where the relation itself is important
- SubmodelElementCollection is used for one grouped object
- SubmodelElementList is used for repeated entries
- Operation, Capability, Entity, and event-related elements are used only where required

Use of ECLASS and IEC CDD

- Existing ECLASS or IEC CDD concepts are checked before defining custom semantics
- Reused concepts match the intended meaning of the element
- Datatype and unit are consistent with the reused concept

6. Modelling Guidelines and Recommendations for the AAS

This section provides structured guidance and explanations on essential aspects of modelling within the Asset Administration Shell. The content is organized into thematic clusters to support a clear understanding of modelling concepts, identification mechanisms, and implementation practices. All recommendations and examples are derived from the Asset Administration Shell specifications and related best practices.

6.1. Asset Concepts and Digital Representation

a) Definition of an Asset in the Context of the AAS

An asset is any physical, digital, or intangible entity that holds value for an individual, an organization, or a system.

Examples include:

- Physical assets: machines, production equipment, sensors, components, or entire factories
- Digital assets: process definitions, simulation models, or real-time data streams
- Intangible assets: software licenses, configurations, or business rules

Each asset in the AAS is assigned a globally unique identifier (e.g., a UUID, URN, or Identification Link according to the IEC 61406 series), the so-called globalAssetId, so that it can be unambiguously referenced throughout its entire lifecycle from engineering and operation to maintenance and eventual decommissioning.

b) Digital Twin as the Model / Representation of an Asset

The model, often called the *digital twin*, is the semantic, machine-readable digital representation of that asset.

The model is contained within the AAS and organized into Submodels, each addressing a specific aspect of the asset.

A digital representation comprises:

- Information about the asset
  (properties, parameters, events, schematics, visualizations)
- Services that act on or provide data for the asset
  (history queries, real-time updates, simulations)
- Viewpoints that define which aspect of the asset is in focus
  (mechanical, electrical, commercial, etc.)

Examples of information:

- Properties: maximum operating temperature, rated power
- Actual parameters: current speed, oil pressure
- Events: status changes, alarm notifications

Examples of services:

- Retrieving the configuration history of a device
- Subscribing to real-time velocity or temperature updates
- Running “what-if” simulations for load changes

Examples of viewpoints:

- Mechanical: geometry, bearing loads
- Electrical: wiring, voltage ratings
- Commercial: purchase order, warranty status

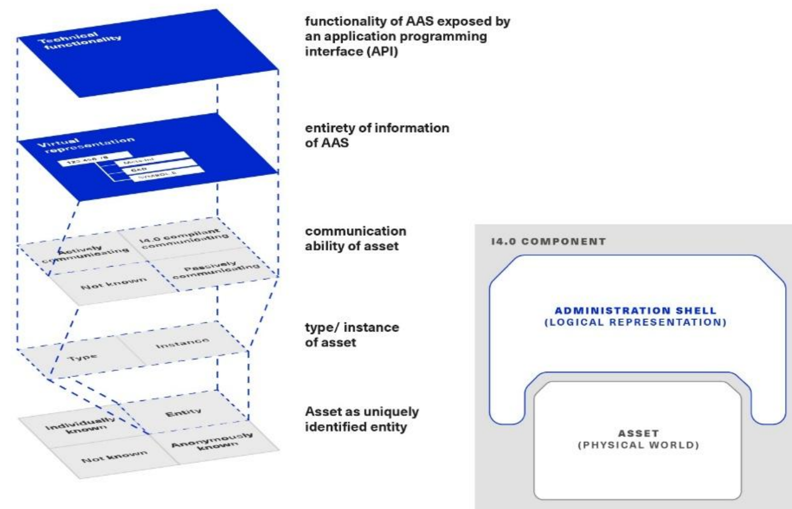

Figure 5: Important Concepts of Industry 4.0 attached to the Asset

c) Existence of a Digital Representation Independent of a Physical Asset

The AAS specification supports modelling a type asset (a product blueprint) before any physical instance exists.

During development, an organization can define an AAS template for a planned product, including all required Submodels, properties, and services. Once physical units are produced, each unit’s AAS is instantiated from and linked to this predefined type asset model.

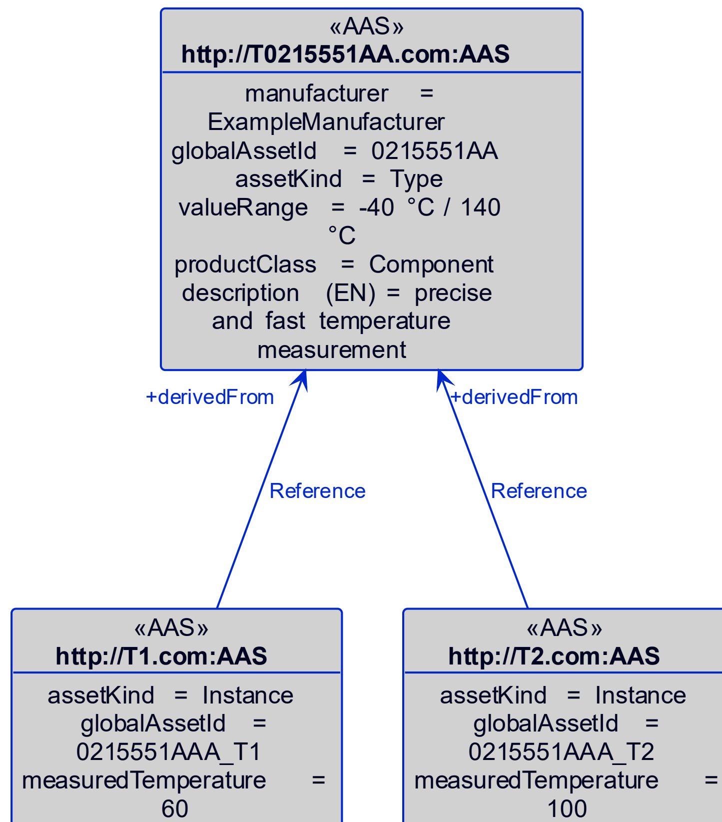

Figure 6: Example for Type and Instance

6.2. Identification, Composition, and Versioning

d) Identification Mechanisms for Asset, Model, and Stakeholders

Unique identifiers are mandatory for every AAS element. The following mechanisms ensure clear distinction between the physical world and the digital representation:

- globalAssetId: This is the Global Asset Identifier, typically expressed as an IRI or URL. It identifies the real-world physical or logical asset. For example, this could be the Identification Link encoded in a QR code on a product label according to the IEC 61406 series, allowing for an automatic look-up of that specific asset.
    - Example: https://manufacturerexample.com/id/3f2504e0-4f89-11d3-9a0c-0305e82c3301
- AAS id: This is the AAS unique identifier for the digital shell instance itself. It is also expressed as an IRI or URL. It tells systems exactly which digital representation they are accessing, separate from the physical asset.
    - *Example*: urn:idta:sg2:aas:1:1:demo11232322
- idShort: This is a local, human-readable short identifier used for Submodels and Submodel elements. Unlike the IDs above, it does not need to be globally unique; it only needs to be unique within its immediate context (e.g., within a specific Submodel).
    - *Examples*: PackagingLineMotor01, Nameplate, or SerialNumber

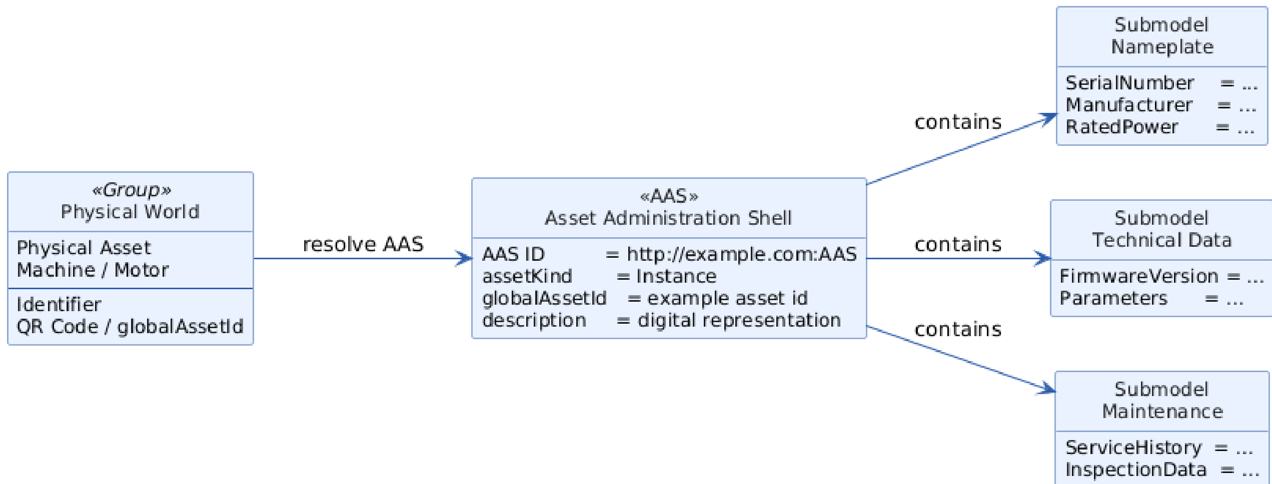

Figure 7: Identification Mechanism

6.3. Best Practices for Custom IRI Identifiers

To ensure global interoperability and semantic consistency within the Asset Administration Shell ecosystem, developers and modellers should follow these standardized best practices for Internationalized Resource Identifiers (IRIs):

- Standardized Prefix: Always use the official prefix, e.g. https://admin-shell.io/, for generic or reusable concept IRIs to avoid naming conflicts and ensure cross-vendor compatibility.
- Protocol and Subdomain: Mandate the use of the https protocol and explicitly omit the www. subdomain to keep identifiers clean and standardized.
- IRI Normalization: During implementation, systems must normalize IRIs by stripping the protocol and subdomain. This ensures that technically different strings are treated as the same unique semantic concept.

Equivalence Examples

Through the process of normalization, the following four IRIs are treated as functionally equivalent within an AAS environment:

1. https://admin-shell.io/some_id_example
2. http://admin-shell.io/some_id_example
3. https://www.admin-shell.io/some_id_example
4. http://www.admin-shell.io/some_id_example

e) Composite Assets and Vertical Supply Chains

In vertical supply chains a composite asset like a production machine is made up of many smaller parts motors, sensors, controllers etc. and each of these should be treated as an independent asset with its own unique assetId. Instead of linking AAS instances directly to one another, the “Hierarchical Structures enabling Bill of Material” (BOM) Submodel template inside the composite asset’s AAS is used to:

- list all component assetIds
- define semantic relations such as *isPartOf* and *isIdentical* for those components

Components can be managed in two ways: they can be self-managed with their own AAS that is discoverable via a registry (useful when parts are reused across systems), or co-managed with their AAS information embedded entirely inside the composite asset's AAS (useful when the part is specific to this machine). By encapsulating all component data and relations in a single BOM model, you get a complete blueprint of the assembly and streamlined lifecycle management and version control for the entire machine.

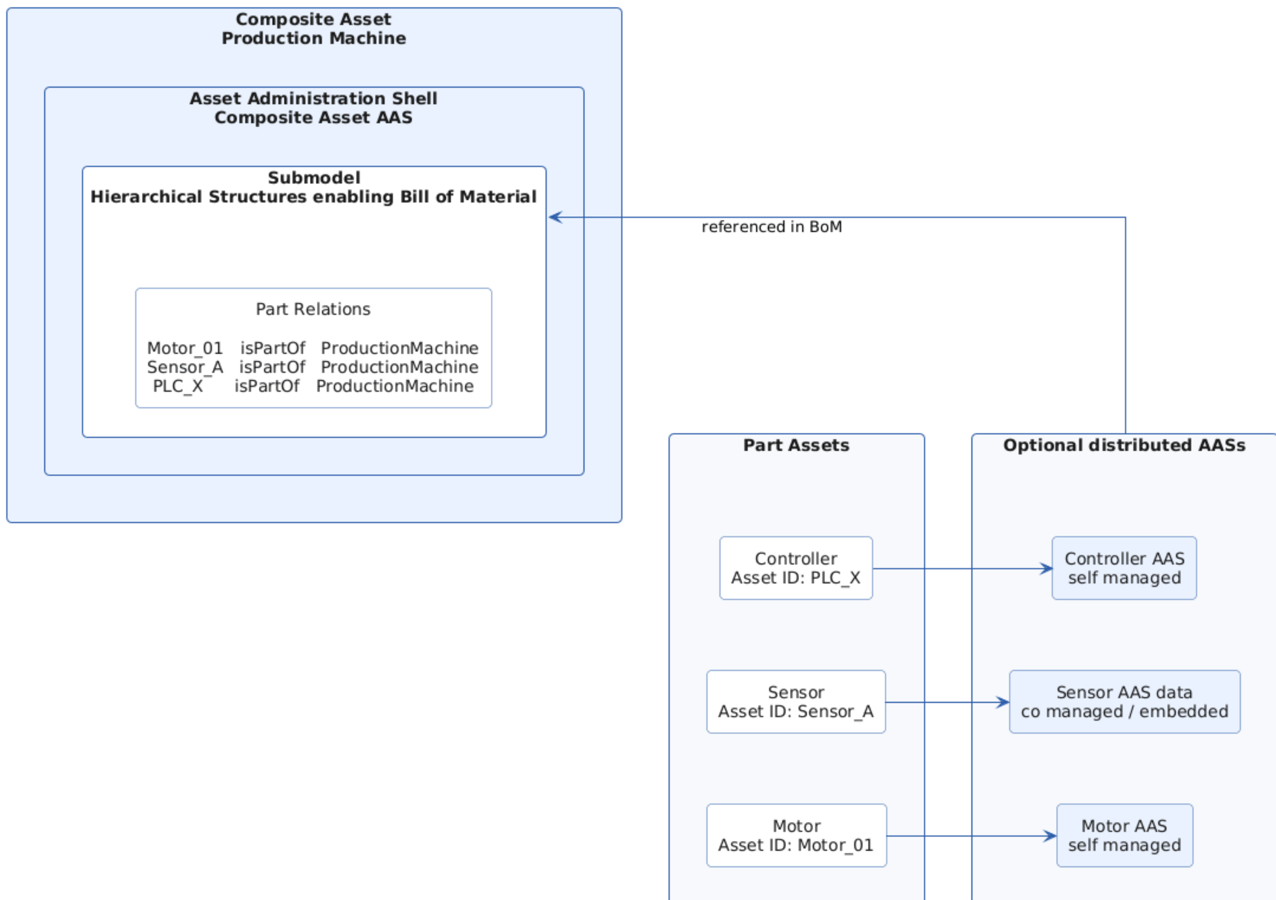

Figure 8: Composite Assets

f) Instance–Type Relationship in AAS

Establish a link between an Instance AAS (serialized physical asset) and its Type AAS (general blueprint) to enable dynamic inheritance of specifications. This ensures a "single source of truth" and avoids data redundancy across multiple instances.

Implementation Rules (Instance AAS)

To model this relationship correctly within the Instance AAS, apply these three mandatory settings:

1. Define Kind: In the AssetInformation section, set assetKind to Instance.
2. Link Physical Definition: In AssetInformation, set assetType to the globalAssetId of the Type Asset.

3. Link Digital Shell: In the AAS Header, configure the derivedFrom reference to point to the AAS id of the Type AAS.

Behavior and Benefits

- Dynamic Inheritance: Tools resolving the Instance AAS follow the derivedFrom link to view specifications defined only in the Type AAS.
- Maintenance Consistency: Updates to general specifications (e.g., an accuracy rating improved in the Type blueprint) are automatically reflected in all instances referencing it, without needing to copy data into individual shells.

Example: A specific sensor instance (SN-998877) stores its unique calibration date locally. However, it inherits its shared measurement range (-20 to +100°C) by using the derivedFrom property to link to the general "Sensor Model T100" Type AAS.

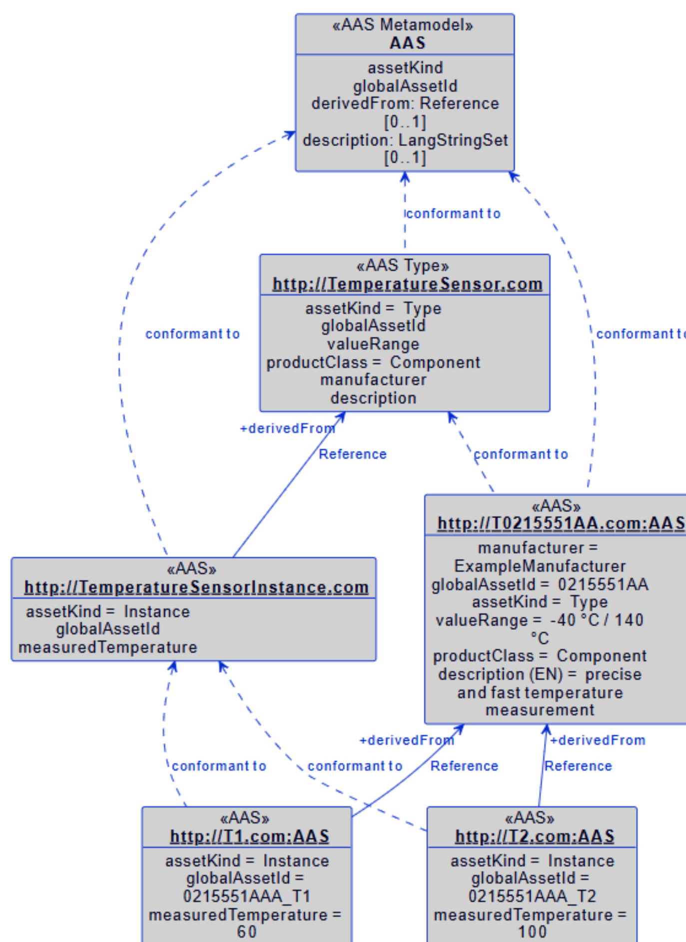

Figure 9: Example: Asset Administration Shell, Asset Administration Shell Types and Instances

g) Versioning Impact on Submodel References

The Asset Administration Shell employs two primary mechanisms to manage the evolution of digital representations. For robust interoperability, implementation must distinguish between structural and content-related updates.

Versioning Within the Semantic ID (Structural)

This method embeds versioning directly into the semanticId (IRI or IRDI) to define the formal meaning and data schema of an element.

- Identifier Examples: IRDIs like 0173-1#01-AHF578...#version or IRIs such as https://admin-shell.io/idta/nameplate/3/0/Nameplate.
- Significance: A version change indicates either breaking semantic changes (altering fundamental meaning) or backward-compatible extensions.
- Standardization: "Major" and "Minor" segments currently align with the release version of the corresponding Submodel Template (SMT) specification

Administrative Versioning (Content)

Administrative versioning is recorded within the AAS metadata and is independent of the element's identity.

- Focus: Relates strictly to internal content updates and does not modify external interface structures.
- Standardization: The specific logic for incrementing Major and Minor versions in this administrative context is not yet globally standardized.

Implementation Requirements

To maintain reliable connectivity within decentralized systems, the following rules apply:

- Identifier Embedding: Because many current AAS tools and references ignore separate versioning metadata, any version-critical information must be embedded directly into the element's identifier.
- SMT Alignment: Each instantiated Submodel should be mapped to the specific SMT version and dictionary release (e.g., ECLASS) it consumes to ensure semantic clarity.

6.4. Submodel Structure and Modelling Practices

h) Submodel Element Collection vs Submodel Element List

Use a collection when it represents one structured object. A typical use case is a block with named fields that belong together. SubmodelElements of an SMC are addressed by their idShort path.

Example, taken from IDTA 02003 – Technical Data:

```
{
  "modelType": "SubmodelElementCollection",
  "idShort": "GeneralInformation",
  ...
  "value": [
    {
      "modelType": "Property",
      "idShort": "ManufacturerName",
      "valueType": "xs:string",
      "value": "Example Company",
      ...
    },
    {
      "modelType": "Property",
      "idShort": "ManufacturerArticleNumber",
```

```
      "valueType": "xs:string",
      "value": "1234567890",
      ...
    },
    ...
  ]
}
```

Use a list when there are many similar entries. Typical use cases include document lists, measurement point lists, sensor calibration lists, and event lists, where SubmodelElements come and go and/or the order of the SubmodelElements is of interest. SubmodelElements of an SML are addressed not by idShort, but by index.

Example, taken from IDTA 02023 – Carbon Footprint:

```
{
  "modelType": "SubmodelElementList",
  "idShort": "PcfCalculationMethods",
  "orderRelevant": false,
  "semanticIdListElement": {
    ...
  },
  "typeValueListElement": "Property",
  ...
  "value": [
    {
      "modelType": "Property",
      "valueType": "xs:string",
      "value": "EN 15804",
      "valueId": "0173-1#07-ABU223#003"
      ...
    },
    {
      "modelType": "Property",
      "valueType": "xs:string",
      "value": "ISO 14067",
      "valueId": "0173-1#07-ABU218#003"
      ...
    },
    ...
  ]
}
```

i) Using qualifiers in a machine-readable way

Qualifiers are structured pairs consisting of a type and a value, which provide additional information about an element. This allows tools to validate and interpret the element accurately. For example, IDTA uses qualifiers such as ExpressionSemantic and PredicateRelation from DIN SPEC 92000 to express statements like threshold rules.

Guideline

Use a controlled vocabulary for qualifier types and allowed values. Ensure values are machine-friendly, such as enums, numbers, or IDs. Attach qualifiers directly to the element they constrain, not to unrelated parent nodes.

Example:

```json
{
  "modelType": "Property",
  "idShort": "maxRotationSpeed",
  "valueType": "xs:integer",
  "value": "1500",
  "qualifiers": [
    {
      "type": "PredicateRelation",
      "valueType": "xs:string",
      "value": "GREATER_THAN_0"
    },
    {
      "type": "ExpressionSemantic",
      "valueType": "xs:string",
      "value": "ASSURANCE_1"
    }
  ]
}
```

j) Modelling constraint ranges and tolerances

Keep the measured or specified value separate from the rules for ranges. Use the SubmodelElement Range, which is defined with minimum and maximum values.

Example, for allowed operating range:

```json
{
  "modelType": "Range",
  "idShort": "operatingTemperature",
  "valueType": "xs:decimal",
  "min": "-10",
  "max": "60"
}
```

For tolerances, use a consistent pattern agreed upon by your ecosystem. Common options include:

- A collection containing nominals, tolerancePlus, and toleranceMinus values.
- Qualifiers for minimum, maximum, and predicate rules when machine validation is required.

Apply universally valid constraints in the concept description data specification when they are relevant everywhere. Use qualifiers for context-specific constraints when they depend on the customer, project, or lifecycle phase.

Example, tolerance as SubmodelElementCollection:

```json
{
  "modelType": "SubmodelElementCollection",
  "idShort": "shaftDiameter",
  "value": [
    {
      "modelType": "Property",
      "idShort": "nominal",
      "valueType": "xs:decimal",
      "value": "20.0"
    },
```

```
    {
      "modelType": "Property",
      "idShort": "tolerancePlus",
      "valueType": "xs:decimal",
      "value": "0.02"
    },
    {
      "modelType": "Property",
      "idShort": "toleranceMinus",
      "valueType": "xs:decimal",
      "value": "0.01"
    }
  ]
}
```

k) Referencing documentation without copying files

Store the file once in a SubmodelElement File. Then, reuse it via SubmodelElement ReferenceElement or RelationshipElement, allowing multiple parts to point to the same file object. This ensures a single source of truth and simplifies version control.

Example, SubmodelElement File; stored once in a Submodel:

```
{
  ...
  "submodels": [
     {
      "idShort": "ExampleSubmodel",
      "id": "ExampleSubmodelId",
      "submodelElements": [
        {
          "modelType": "File",
          "idShort": "CeDeclarationPdf",
          "contentType": "application/pdf",
          "value": "https://example.com/docs/ce_declaration.pdf",
          ...
        },
        …
      ],
      ...
     },
     ...
  ],
  ...
}
```

Example reference, e.g. from within another Submodel:

```
{
  "modelType": "ReferenceElement",
  "idShort": "CeDeclaration",
  "value": {
    "type": "ModelReference",
    "keys": [
      {
        "type": "Submodel",
        "value": "ExampleSubmodelId"
```

```
      },
      {
        "type": "File",
        "value": "CeDeclarationPdf"
      }
    ]
  }
}
```

l) Storing Certificates and Conformance Declarations in Submodels

A dedicated Submodel is proposed with

- idShort = "CertificatesAndDeclarations" and
- id = https://admin-shell.io/submodels/CertificatesAndDeclarations.

This Submodel shall:

- Contain boolean or text properties indicating conformance to certificates.
- Store actual certificate documents (e.g., scanned TÜV reports) in the “Documentation” Submodel, which is then referenced from elements in “CertificatesAndDeclarations.”

Examples:

- Boolean property 0173-1#02-BAF053#008 set to true if a CE qualification is present.
- Text property 0173-1#02-AAE327#001 holding the name of the fulfilled conformance.

m) Best Practices for idShort of Elements

For selecting an idShort, apply the following recommendations (rules and order may be used for automatic generation):

- If an English shortName is available in the related ConceptDescription, use it (in camelCase).
- If an English preferredName is present, convert it to an idShort (e.g., “Maximum rotation speed” → maximumRotationSpeed).
- Observe constraint [AASd-002 of the IDTA 01001 Part 1: Metamodel specification]()
- If using a Submodel Template specification as a base, use the idShort defined there.

Note: The idShort does not carry semantic meaning; descriptive, user-facing names should use displayName.

n) Use of Qualifiers in Submodels

Qualifiers are meant to provide example values and temporary information within the Submodel and its SubmodelElements.

For example:

```
"qualifiers": [
            {
```

```
              "semanticId": {
                "keys": [
                  {
                    "type": "GlobalReference",
                    "value": "https://admin-
shell.io/SubmodelTemplates/AllowedValue/1/0"
                  }
                ],
                "type": "ExternalReference"
              },
              "kind": "ValueQualifier",
              "type": "SMT/AllowedValue",
              "value": "good | bad | uncertain | others",
              "valueId": {
                "keys": [
                  {
                    "type": "GlobalReference",
                    "value": "0112/2///61360_7#CTA052"
                  }
                ],
                "type": "ExternalReference"
              },
              "valueType": "xs:string"
            },
```


6.5. Units and Multi-Unit Modelling

o) Use of Physical Units for Quantifiable Properties

IDTA 01003a – Specification of the Asset Administration Shell Part 3a: Data Specification – IEC 61360 defines unit and unitId for concept descriptions.

Recommendations:

- Use unitIDs that refer to predefined ECLASS (e.g., 0173-1#05-AAA480#002) or IEC CDD (e.g., 0112/2///62720#UAA862#001) IRDIs
  Example, ECLASS IRDI for millimeters:
  `{ ... "unitId": "0173-1#05-AAA480#002", "unit": "mm" ... }`

- If no global IRDI exists for a unit, follow the guidance for custom semantic IDs to define a custom unitID.

p) Defining Custom and Non-Physical Units

The concept of units in AAS is independent of whether they are physical or non-physical.

For properties:

- Define the numeric value (e.g., OrderQuantity.value = 5).
- Accompany this with a DataSpecificationIec61360 entry that provides:
    - a unit attribute (e.g., "pallet")
    - a unitID (e.g., a catalogue reference from UNECE, IEC CDD, or ECLASS)

These catalogues can include non-SI units (like “dozen piece” or “pallet”). An updated Units of Measure DataSpecification will provide richer metadata.

Defining a separate string property for the unit (e.g., OrderQuantityUnit) is considered an anti-pattern, because it reduces semantic fidelity and forces tools to rely on implicit associations.

r) Modelling Multiple Units for a Single Concept Description

To preserve unique semantics, a single value property should not carry multiple unit definitions.

Two modelling options are available:

1. Generic value plus explicit unit property
    - Introduce a generic value property and a separate unit property to allow runtime unit variation.
2. Discrete properties per unit
    - Define distinct properties for each unit (e.g., massInKg, massInG).

The selection depends on the balance between flexibility and schema clarity.

6.6. Multiple AAS and Lifecycle Strategies

q) Coexistence of Multiple AAS for a Single Asset

Multiple AAS instances may exist for one asset, as long as they reference the same globalAssetId.

These AAS instances can:

- Differ in Submodel composition, data scope, and ownership – even within the same company or across organizations.
- Be copied or shared during ownership transfer or lifecycle transitions.

The original manufacturer often retains an AAS imprint, enabling:

- master data synchronization, and
- secure cross-enterprise data exchange via data-space infrastructures.

6.7. Instantiation Modes of Submodels and Versioning Strategy

Both on-demand instantiation and repository pre-staging of Submodels are valid, since semantic definitions remain immutable at runtime.

Version management must cover all artifact classes:

- Submodel Templates (SMTs):
  Multiple SMT versions may coexist; instantiation logic selects the appropriate version via semantic tags or content hashes.
- ECLASS Dictionaries:
  Each release identifier is mapped to the SMTs that consume it.
- Aspect Models & Constraints:
  Version metadata (e.g., timestamps, semantic versions) is embedded in constraint schemas, and the generator or repository uses it to bind the correct validation rules.

6.8. Semantics and how to use ECLASS and IEC CDD

r) What is Semantic ID?

In the Asset Administration Shell, many elements implement the interface HasSemantics (e.g. Submodel, SubmodelElement, ConceptDescription).
For these elements, the field semanticId is:

A *Reference* to a semantic concept that defines the meaning of that element.

Formally:

- semanticId is a Reference (usually of type GlobalReference).
- It points to a ConceptDescription, which carries the formal semantics (name, definition, datatype, unit, value list, etc.) – typically via DataSpecificationIEC61360.
- In practice, the semanticId often contains:
  - an IRDI (from ECLASS / IEC CDD) or
  - an IRI under a controlled namespace.

Key points:

- idShort = local, technical name (for humans, code, navigation).
- semanticId = semantic key (for interoperability, mappings, validation).
- Tools should *never* treat idShort as the primary semantic identifier; they should always use semanticId.

s) What are ECLASS and IEC CDD?

ECLASS is an industrial reference data standard and classification system. Very short:

- It defines product classes and properties as standardized identifiers.
- Each property, class, and unit has an IRDI (International Registration Data Identifier).

- It is based on IEC 61360 / ISO 29002 and widely used in product master data, catalogues, and now AAS.

In the AAS context:

- ECLASS is used as a semantic dictionary.
- IDTA Submodel Templates often use ECLASS IRDIs as semanticId for properties and sometimes as unitId.

Example IRDI (property):

- 0173-1#02-AAO677#002 → "Manufacturer name"
- 0173-1#02-BAF053#008 → "CE qualification present"

IEC CDD (*IEC Common Data Dictionary*) is a reference data library maintained by IEC:

- It contains standardized concepts for electrical / industrial domains: characteristics, units, codes, etc.
- Each concept is identified by an IRDI (e.g. 0112/2///61360_4#AAF131#001 for "number of phases").
- It is the "canonical" implementation of IEC 61360.

In the AAS context:

- IEC CDD can also be used as a semantic dictionary.
- IRDIs from IEC CDD are used as semanticId / unitId where appropriate, especially in more "electrical" or generic measurement contexts.

t) How to use semanticId, ECLASS, and IEC CDD together

Think of it as a 3-layer chain:

AAS Property → semanticId → ConceptDescription → ECLASS / IEC CDD concept.

Each SubmodelElement should have a semanticId and optionally one or more supplementalSemanticIds.

Example, taken from IDTA 02006 – Digital Nameplate:

```
{
  "idShort": "manufacturerName",
  "modelType": "MultiLanguageProperty",
  "semanticId": {
    "keys": [
      {
        "type": "GlobalReference",
        "value": "0112/2///61987#ABA565#009"
      }
    ],
    "type": "ExternalReference"
  },
  "supplementalSemanticIds": [
    {
```

```
      "keys": [
        {
          "type": "GlobalReference",
          "value": "0173-1#02-AAO677#004"
        }
      ],
      "type": "ExternalReference"
    }
  ],
  "value": [
    {
      "language": "en",
       "text": " Example Company "
    }
  ],
}
```

Define a ConceptDescription for the semanticId, and optional as well as for the supplementalSemanticIds:

Example, ConceptDescription for a SubmodelElement ManufacturerName with ECLASS IRDI 0173-1#02-AAO677#004

```
{
  "modelType": "ConceptDescription",
  "idShort": "ManufacturerName",
  "id": "0173-1#02-AAO677#004",
  "description": [
    ...
    {
      "language": "de",
      "text": "Bezeichnung für eine natürliche oder juristische Person, die …"
    },
    {
      "language": "en",
      "text": "Legally valid designation of the natural or judicial person which …"
    },
    ...
  ],
  "displayName": [
    ...
    {
      "language": "de",
      "text": "Herstellername"
    },
    {
      "language": "en",
      "text": "Manufacturer name"
    },
    ...
  ],
  "embeddedDataSpecifications": [ {
    "dataSpecification": {
```

```
      "keys": [ {
        "type": "GlobalReference",
        "value": "http://admin-shell.io/DataSpecificationTemplates/…"
      } ],
      "type": "ExternalReference"
    },
    "dataSpecificationContent": {
      "modelType": "DataSpecificationIec61360",
      "dataType": "STRING",
      "definition": [
        ...
        {
          "language": "de",
          "text": "Bezeichnung für eine natürliche oder juristische Person, die …"
        },
        {
          "language": "en",
          "text": "Legally valid designation of the natural or judicial person which …"
        },
        ...
      ],
      "preferredName : [
        ...
        {
          "language": "de",
          "text": "Herstellername"
        },
        {
          "language": "en",
          "text": "Manufacturer name"
        },
        ...
      ]
    }
  } ],
}
```

The semanticId of the Property points to that ConceptDescription, and the ConceptDescription points to the dictionary semantics (ECLASS / IEC CDD).

Result: Any tool that understands AAS can:

- read the Property,
- resolve its semanticId,
- and know *exactly* what “ManufacturerName” means, independent of the idShort.

u) How to look up an IRDI (ECLASS / IEC CDD) in practice

Manual (for engineers):

- For IEC CDD:
  - Go to the [IEC CDD web site] and open the search in the corresponding domain
  - Paste the IRDI (e.g. 0112/2///61360_4#AAF131#001) in the search box.
  - Read the concept: preferred name, definition, datatype, unit, value list.
- For ECLASS:
  - Use [ECLASS portal web site] / CMT / export (requires licence).
  - Search for the IRDI (e.g. 0173-1#02-BAF053#008).
  - Read preferred name, definition, datatype, unit, values, etc.

Then transfer this information into your ConceptDescription in the AAS.

Note: Observe the relevant license terms regarding use.

v) How to use them in software tools

- In tools (validators, editors, backends):
- Maintain a semantic registry / dictionary module:
  - Internal DB or config that maps:
    - IRDI → ConceptMetadata
      e.g. { preferredName, definition, dataType, unit, dictionarySource, version }
  - Data loaded from:
    - ECLASS export (for properties/units you use), and
    - IEC CDD export or curated subset.
- On AAS load / validation:
  - For each semanticId:
    - Resolve IRDI in the registry.
    - If missing → warning or error.
  - Optionally auto-generate missing ConceptDescriptions from the registry.
- In your UI / integration logic:
  - Use the resolved semantics to:
    - Display human-readable labels and definitions,
    - Check datatypes and units,
    - Drive mappings to OPC UA, DB schemas, messages, etc., based on the *semantics*, not on idShort.

7. Clustering Submodel Templates

7.1. Clustering of Submodel Templates (SMTs) Based on Use Cases

The IDTA provides a variety of Submodel Templates (SMTs) that serve different purposes within industrial ecosystems. To support efficient adoption and implementation, SMTs can be clustered according to their application domains. This guideline proposes a clustering based on five major use cases: Digital Product Passport (DPP), Engineering, Identification, Asset Services, and Artificial Intelligence (AI).

This categorization enables stakeholders to identify relevant SMTs more effectively and ensures that the appropriate templates are applied in specific industrial scenarios, such as product transparency, engineering workflows, product identification, asset-level services, or AI lifecycle management.

7.1.1. Digital Product Passport (DPP)

The Digital Product Passport cluster addresses product transparency, sustainability, and lifecycle traceability. It ensures that relevant technical, structural, and environmental data is made available in a standardized and interoperable form.

Included SMTs:

- IDTA 02002 – Contact Information
- IDTA 02006 – Digital Nameplate
- IDTA 02004 – Handover Documentation
- IDTA 02003 – Technical Data
- IDTA 02011 – Hierarchical Structures (BoM)
- IDTA 02023 – Carbon Footprint

7.1.2. Engineering

The Engineering cluster supports design, simulation, verification, and operational analysis. SMTs in this category provide structured data exchange mechanisms for functional safety, product sizing, configuration, and reliability.

Included SMTs:

- IDTA 02003 – Technical Data
- IDTA 02013 – Reliability
- IDTA 02014 – Functional Safety
- IDTA 02021 – Power Drive Train Sizing
- IDTA 02045 – Asset Location
- IDTA 02026 – Provision of 3D Models
- IDTA 02053 – Control Config (NC/CNC)
- IDTA 02016 – Control Component Instance
- IDTA 02017 – Control Component Type

7.1.3. Identification

The Identification cluster standardizes the digital identification of assets and software components. It ensures compliance with interoperability requirements and enables reliable, machine-readable product identification within industrial environments.

Included SMTs:

- IDTA 02002 – Contact Information (v1.0)
- IDTA 02006 – Digital Nameplate

7.1.4. Asset Services

The Asset Services cluster focuses on service-related connectivity and communication. SMTs in this category define models for asset interfaces, wireless communication, and interface mapping to enable seamless service integration.

Included SMTs:

- IDTA 02017 – Asset Interfaces Description (AID)
- IDTA 02022 – Wireless Communication
- IDTA 02027 – Asset Interface Mapping Configuration (AIMC)

7.1.5. Artificial Intelligence (AI)

The Artificial Intelligence cluster addresses the management and deployment of AI within the AAS. These SMTs enable structured handling of datasets, deployment workflows, and AI model identification in order to ensure trustworthy and interoperable AI integration.

Included SMTs:

- IDTA 02058 – AI Dataset
- IDTA 02059 – AI Deployment
- IDTA 02060 – AI Model Nameplate

This clustering of SMTs provides a structured framework for developers to select and apply the appropriate templates according to their specific use cases. It contributes to consistent implementation practices, facilitates interoperability, and supports the strategic goals of transparency, efficiency, and innovation in industrial applications.

8. Bibliography

1. IEC 63278-1:2023, *Asset Administration Shell for industrial applications – Part 1: Asset Administration Shell structure,* International Electrotechnical Commission, 2023. Available: https://webstore.iec.ch/en/publication/65628
2. Industrial Digital Twin Association, *Specification of the Asset Administration Shell Part 1: Metamodel (IDTA-01001),* Industrial Digital Twin Association, 2024. Available: https://industrialdigitaltwin.org/wp-content/uploads/2024/06/IDTA-01001-3-0-1_SpecificationAssetAdministrationShell_Part1_Metamodel.pdf
3. Industrial Digital Twin Association, *Specification of the Asset Administration Shell Part 2: Application Programming Interfaces (IDTA-01002),* Industrial Digital Twin Association. Available: https://industrialdigitaltwin.org/en/content-hub/aasspecifications
4. Industrial Digital Twin Association, *Specification of the Asset Administration Shell Part 3a: Data Specification – IEC 61360 (IDTA-01003-a),* Industrial Digital Twin Association, 2023. Available: https://industrialdigitaltwin.org/en/wp-content/uploads/sites/2/2023/04/IDTA-01003-a-3-0_SpecificationAssetAdministrationShell_Part3a_DataSpecification_IEC61360.pdf
5. Industrial Digital Twin Association, *Specification of the Asset Administration Shell Part 4: Security (IDTA-01004),* Industrial Digital Twin Association. Available: https://industrialdigitaltwin.org/en/content-hub/aasspecifications
6. Industrial Digital Twin Association, *Specification of the Asset Administration Shell Part 5: Package File Format (AASX) (IDTA-01005),* Industrial Digital Twin Association. Available: https://industrialdigitaltwin.org/en/content-hub/aasspecifications
7. Details of the Asset Administration Shell – Part 1 (V3.0RC02), Industrial Digital Twin Association, 2022. Available: https://industrialdigitaltwin.org/wp-content/uploads/2022/06/DetailsOfTheAssetAdministrationShell_Part1_V3.0RC02_Final1.pdf
8. *Asset Administration Shell (AAS)*, Wikipedia. Available: https://de.wikipedia.org/wiki/Asset_Administration_Shell
9. Arena2036, *Asset Administration Shell – standardized digital twin,* Arena2036. Available: https://arena2036.de/en/forschung/verwaltungsschale-der-standardisierte-digitale-zwilling/
10. A. Metović, N. Maisch, S. Ajdinović, A. Lechler, A. Wortmann, and O. Riedel, "Industrial Semantics-Aware Digital Twins: A Hybrid Graph Matching Approach for Asset Administration Shells," *arXiv preprint arXiv:2601.06613,* 2026. Available: https://arxiv.org/abs/2601.06613
11. H. Eichelberger and A. Weber, "Model-driven realization of IDTA submodel specifications: The good, the bad, the incompatible?," *arXiv preprint arXiv:2406.14470,* 2024. Available: https://arxiv.org/abs/2406.14470
12. aas-core-works, *awesome-aas: List awesome tools, platforms, libraries and documentation about Asset Administration Shell,* GitHub. Available: https://github.com/aas-core-works/awesome-aas.