







TCCS - Data Model_02_Methodology

References

-  [TCCS - Data Model_01_Approach](#)
-  [TCCS - Data Model_02_Schema](#)
-  [TCCS - Data Model_10_INFRA](#)
-  [TCCS - Data Model_11_OI](#)
-  [TCCS - Data Model_11_ENG](#)
-  [TCCS - Data Model_11_TP](#)

1 Table of Contents

| | | |
|-------|--|----|
| 1 | Table of Contents | 1 |
| 2 | Purpose of this document | 2 |
| 3 | Target of specification: What is the "data model" in the ERJU context? | 2 |
| 3.1 | What to specify | 2 |
| 3.2 | How to specify | 3 |
| 4 | Methodology | 5 |
| 4.1 | Modelling language | 7 |
| 4.1.1 | Content of the modelling language | 8 |
| 4.2 | Model design decisions | 11 |
| 4.2.1 | Polymorphism representation | 12 |
| 4.2.2 | Model structure | 14 |
| 4.2.3 | Referencing structs and enums | 15 |
| 4.2.4 | Referencing objects | 16 |
| 4.2.5 | IDs / Keys | 18 |
| 4.2.6 | Object-splitting as object-double-linking | 19 |
| 4.2.7 | Physical values | 22 |

| | | |
|---------|-------------------------------------|----|
| 4.2.8 | Versioning | 23 |
| 4.2.8.1 | Examples: Infrastructure - domain | 23 |
| 4.2.8.2 | Examples: Operational plan - domain | 24 |
| 5 | Model and derived schemata | 25 |
| 5.1 | Complete Data Model | 25 |
| 6 | References | 25 |

2 Purpose of this document

This document serves as a introduction to the definition of basic concepts/classes and associations of the CCS/TMS Data Model, which shall be applicable for interfaces of all domains, i.e. for engineering/configuration/maintenance as well as for operational interfaces. While the TCCS domain of SD1 is located in Task 2 of the System Pillar, SD1 offers the inclusion of other tasks (such as task 3 for TMS/CMS) with the same and consistently applied approach. The general approach of TCCSD SD1 is summarised in [TCCS - Data Model_01_Approach](#). This also contains the reuse of fragments from existing standards based on their applicability to the specific scope and development goals of SD1. The goals of a Data Model within the ERJU System Pillar context are further explained in [3 - Target of specification: What is the "data model" in the ERJU context?](#). The applied methodology for the development of the data model incl. the derivation of different data schemata is defined in [4 - Methodology](#). The Data Model itself in its current development state is documented as separate schemas like [TCCS - Data Model_10_INFRA](#) , [TCCS - Data Model_11_OI](#), [TCCS - Data Model_11_TP](#) , [TCCS - Data Model_11_ENG...](#)

3 Target of specification: What is the "data model" in the ERJU context?

3.1 What to specify

SPT2TS-123881 - The primary objective of the data model is to support **interface specifications** between systems. There are two main ways to establish interfaces:

- Defining the data stream "on the wire" (including airgap interfaces)
- Defining function calls in specific programming languages (e. g. within architectures based on standards like OMG Corba, OMG DDS etc.)

In the context of the CCS/TMS data model, the objective is to specify a **part of the "data stream on the wire for interface specifications"**. The reason is that the programming languages compared to data modelling are too volatile for the railway domain. Also, this fits the goal of the CCS/TMS data model as a

data structure for functional interface specifications within the System Pillar context.

A data exchange protocol based on a continuous data stream (e. g. TCP) normally consists of two parts:

1. A frame defining limits of single messages inside of a continuous data stream
2. An object specification inside of a frame.

As the frame definition depends on the communication technology being used, middleware etc., it remains open for the architecture group, which is responsible for protocol specifications. The objective of this document is only the last point: "definition of objects inside of a frame for the 'on the wire' data stream". That means:

The objective of this document is to define serialisable data structures together with its mapping and representation in a sequence of bytes, which is often called a "data schema".

Consequently, TCCS SD1 does specify not only a data model but also its (automatic) translation into the data schemata for practical usage on the standardised interfaces for engineering/configuration (SMI) or operation (SCI). To achieve this, a formal description is applied to avoid ambiguities and allow automatic transformation for different use cases, as described in the following chapters.

3.2 How to specify

SPT2TS-123880 - The freedom in the definition of the data model is strongly limited by the data schema it should support (e.g. if a schema does not support inheritance, the data model should not use it). So, the question is: which schemata shall be supported by the data model? As the interface specifications are a long-living and innovative process, we can not limit ourselves only to one schema (e.g. XML) but to a schema class containing different schemata depending on the specific interface requirements, e.g. available bandwidth.

In the context of this development, three classes of data schemata should be considered:

- Class 1: **pure data specification** (e.g. OMG Corba, ASN.1, all ETCS telegrams), where each bit on the wire contains object information.
Limitation: Attributes with primitive data types (boolean, int, double) can not be optional. I. e. an optional boolean should be an enumeration with true/false/unknown.
- Class 2: the schemata include **static meta information**, allowing separation of each attribute of the object. This allows backward compatibility, as the outdated client applications would be able to ignore/skip new ("unknown") object attributes. Examples: JSON, XML (subset), Protobuf. Static means that once published, the attribute never changes its type.
Limitation: The data model must avoid polymorphism by inheritance.
- Class 3: the schemata include **dynamic meta information** about the type of an attribute allowing data polymorphism on the wire. Example: XML, ZeroC. Dynamic means that the attribute can be of

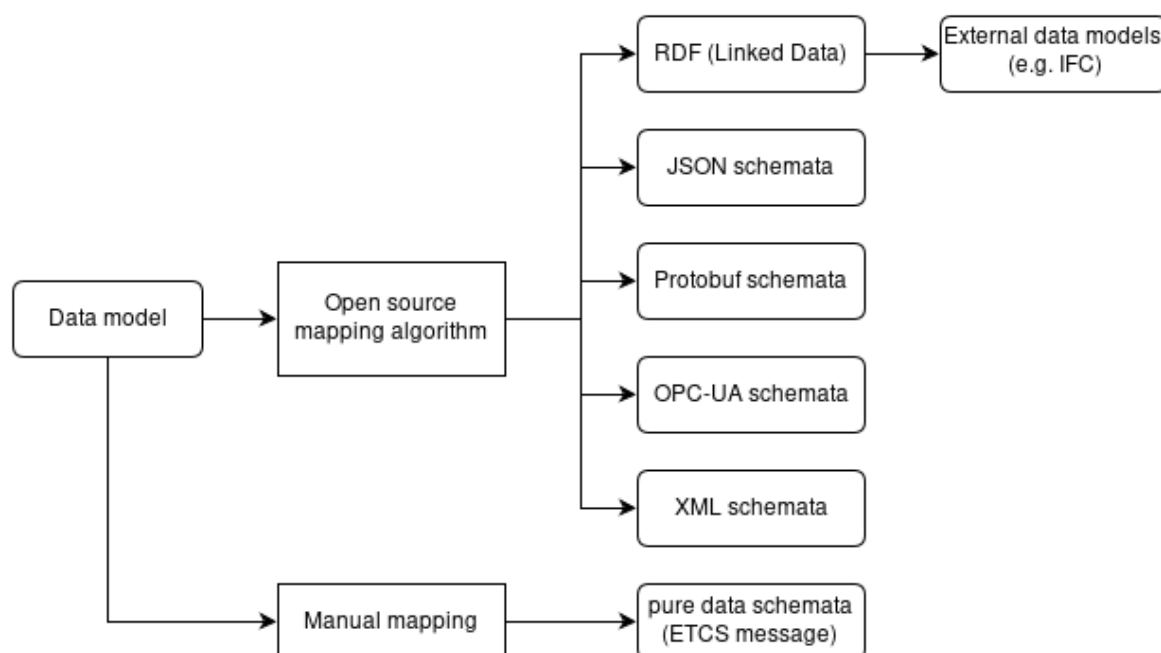
a different type in the next version of the schema, and outdated clients can handle it somehow (either skipping or parsing attributes shared with the "known" parent class).

For this project, it was decided to cover the "schemata with static meta information" (class 2) for the following reasons:

- Main interfaces would have sufficient bandwidth for static meta information, which increases message size by 10-500% depending on selected schema (Protobuf/XML) and object structure (meta information for a boolean attribute increases size much more than for a long string).
- The backwards compatibility allows evolutionary (stepwise) deployment of the next schema version, so the outdated services will also be able to work with updated services. Only a change in the major version would require a full reset of all communication partners using the modified schema.
- The limitation of avoiding polymorphism by inheritance can be solved by the applied modelling methodology defined in 4.2 - Model design decisions.
- Pure data specification (class 1) is insufficient, as schemata must be updated/regenerated with every new version. In some cases, they still will be backwards compatible if new information is appended to the existing simple object.
- Dynamic meta (class 3) data is not well supported, especially by binary schemata (e.g. Protobuf), and even in JSON, it is an "ANY"-object approach, which is not standardized yet. ANY means that the type of the attribute is given by a dedicated string, which can contain any value, being a source of errors.

So, the objective of the data model is to allow the generation of schemata, including static meta information.

Specifically, that means that the data model shall be mappable to JSON, XML (subset), Protobuf, and OPC-UA (subset) schemata. Another use case is the generation of RDF representation with an ability to integrate into semantic ontologies (i.e. ERA vocabulary (<https://data-interop.era.europa.eu/era-vocabulary/>) and leverage the facilities offered by LinkedData, i.e. to transform SD data model to other, external data models such as IFC.



The interface specifications are free to select appropriate middleware and appropriate schema. In some cases, the middleware could support automatic conversion between a set of schemata, e.g. the Integration Layer defined in Shift2rail project X2Rail-4 maps between binary schema Protobuf <-> human-readable schema JSON at runtime without performance impacts.

4 Methodology

This chapter describes how the data model is formally stated in text format and how the translation into machine-usable data schemata works.

SPT2TS-1658 -

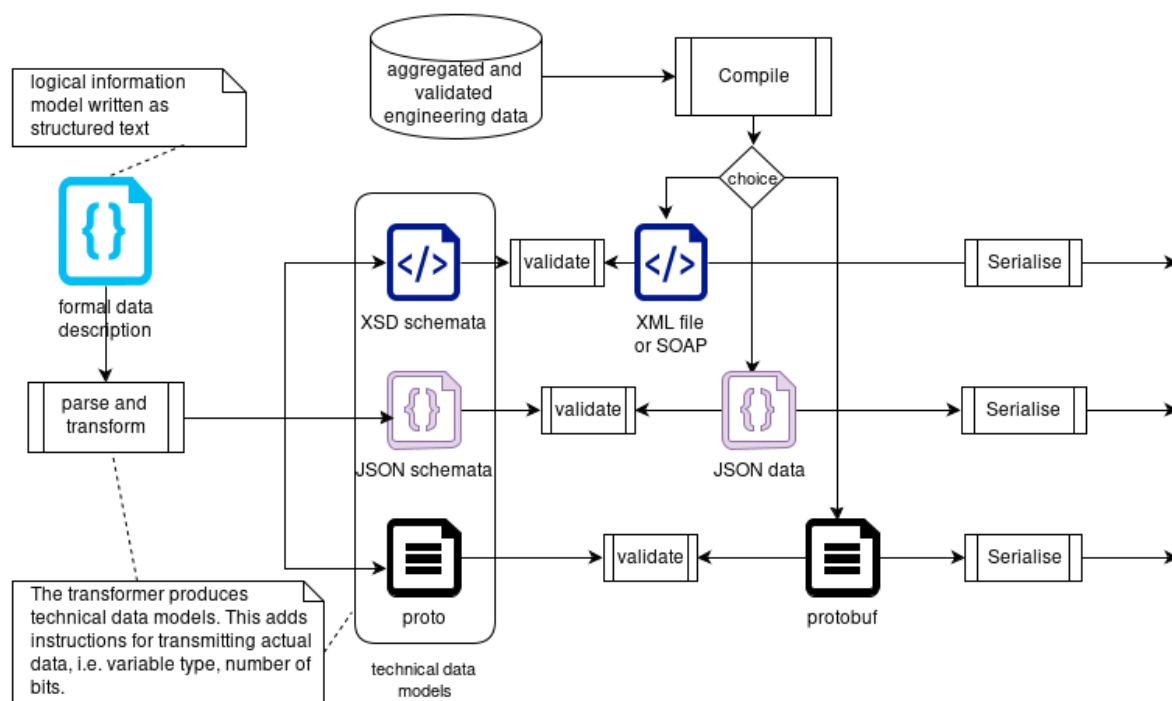


Figure 1: The formal data model is written in a text format which is easily implemented as a schema such as XSD, protobuf or JSON schema

SPT2TS-1158 - The CCS/TMS data model is a logical information model. It is described in a formal language that can be easily parsed by machines and read by humans.

SPT2TS-1655 - The CCS/TMS data model is a logical information model but also is the template from which the selected schemata are derived. This transformation is automated to prevent human error. The use of various schemata implies a compromise because, in particular, real-time binary ones are more restrictive in terms of allowed modelling constructs. Technologies such as OPC-UA and XSD are designed to be rich and expressive, whereas Protobuf is designed for efficient and compact real-time transmission on-the-wire. To allow the seamless transformation of the data model into selected schemata (JSON, Protobuf, XSD, and OPC-UA), the CCS/TMS data model carries additional information i.e., decorators, which ensures unique and unambiguous translation into the target schemata. A set of scripts generate schemata out of the CCS/TMS data model.

SPT2TS-1657 - The model is primarily documented, versioned and stored in Polarion. The model is exported with each release and parsed into the mentioned data schemata. These schemata allow unambiguous standardisation and fast integration into development tools or demonstrators with test data, i.e. Innovation Pillar. This practical usage of the model creates a short feedback loop so that a high level of maturity is quickly attained.

In addition, a translation into UML (e.g. plantUML, XMI) can be used for overall model visualisation, which is automatically created and always consistent with the model itself. The bridge to Capella (if available to

Polarion) will be used to achieve a synchronised model view with the System Pillar architecture and all other domain stakeholders working with Capella.

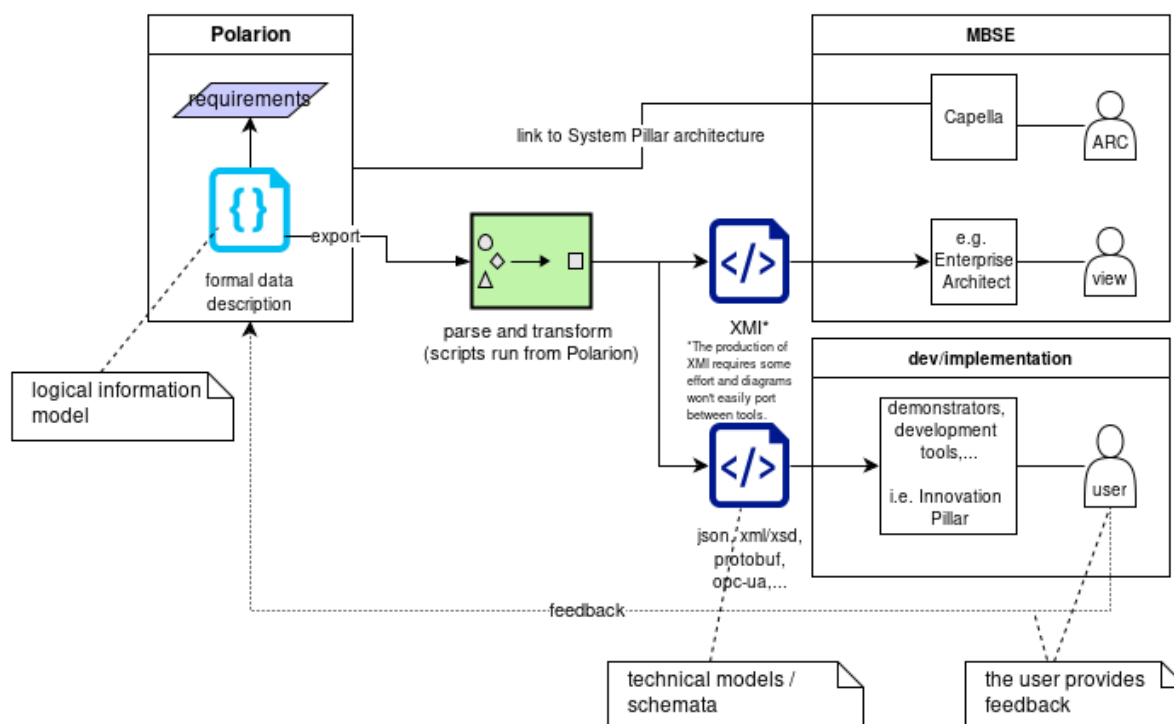


Figure 2: Handling the data model. The data model is a formal description of the object types and their relations. The ensuing technical information models are the schemata for transporting actual data over-the-wire or as files.


4.1 Modelling language

SPT2TS-123990 - This chapter describes the modelling language used to write the model (a "meta-model").

The modelling-language should be a well-known standard and easily transformable into a schema that also supports serialisation on the wire. Its syntax must be enforced by a schema, which prescribes how to define structures, attributes, links and other model-parts. There are several possibilities:

- OMG Interface Definition Language (IDL) is a compact well readable format, but it requires a non-standard parser and non-standard schema.
- UML XMI is an XML representation of UML-diagrams having a schema. This approach has two issues:
 - It is not human-readable
 - It misses additional concepts needed in the data model. Using user-specific tags not enforced by the schema is not formal enough

- Custom XML or JSON. Both formats would equally fulfill the requirements for modelling - human-readable, enforced by a schema. We selected JSON for two reasons:
 - It is more compact and readable inside of Polarion
 - In the use-case of broker/Integration Layer with central data management it is assumed, that the data model will be part of API - the middleware must be provided with the data model via an API to allow automatic data validation and transformation between syntaxes (e.g. binary <-> JSON). It is assumed that a JSON representation of the data will be the most used way for other APIs, so keeping JSON also for the data-model reduces number of libraries in the software (no need of both JSON and XML libraries inclusive validating parsers).

The preliminary meta-model schema is documented here  [TCCS - Data Model_02_Schema](#) and explained in the following sections.

4.1.1 Content of the modelling language

The modelling language, i.e. the meta-model, defines the following aspects of a data model:

SPT2TS-1581 - Package: A package in the data model represents a namespace. A package contains enumerations, structs, and namespace specifications. Names of structs and enumerations are unique within a namespace. The namespace is described by an IRI (Internationalized Resource Identifier) such as `http://example.eu/fascinatingDomain/subdomain/areaOfInterest` and a prefix that acts as a well-known abbreviation for the IRI.

Version is noted as numbers: Major.minor.editorial. Changes in minor and editorial parts keep backwards compatibility for the outdated clients (for backwards compatibility see also section 3.2).

SPT2TS-1216 - Enum/EnumLiteral: An ordered list of named options, also known as fields. The fields are written using camelCase, e.g. `timingPoint`, `colourRed`. The fields have an explicit zero-based index named "intId", so there's a mapping between integers and strings. The enumeration has a name and an annotation. The default value is the first field (with index zero). Binary protocols use intId-representation, JSON and XML use a string-representation of a field.

SPT2TS-1217 - Struct: Represents a named data structure and, as opposed to a class, doesn't support methods

see explanation also in [4.2 - Model design decisions](#)

SPT2TS-1660 - Attribute: Are members of a struct and described by

- name of the attribute
- index, "intId", 1-based integer that can
 - replace the name for efficient data transmission in particular protocols, e.g. protobuf and OPC-UA.
 - to implement compact absolute references between objects, i.e.

container/subcontainer/object[3]

- backward compatibility for binary protocols, the index supports compatibility across versions
- type, which is one of
 - primitive data type that maps to a programming language value type
 - reference to another struct
 - composition of another struct
 - enumeration data type as defined above
- multiplicity, also known as arity, is defined by minimum and maximum integral number of occurrences, e.g. 2, 1..2, 1..*. The default value is 1.
- key, union with "none", "local", and "global" annotates attribute playing a key-role (see section 4.2.5). The default value is "none".
- union annotation of structs with "union" stereotype, defining a union-struct (only one attribute is allowed)

This allows static typing of attributes for protocols such as Protobuf or JSON. The default value is "false".
- sorted, decorator prescribing if true (the default value is "false"), that
 - objects in the sequence must be alphabetically sorted according to the key-attribute,
 - the referencing to a specific object can be done via its key-value,
 - only one object with a specific key-value is allowed per sequence (a map).
- see: defines a reference to the semantic source, either a source model or an ontology node.
- units: defines a physical unit for specific attribute. The list of supported units is defined in the data model schema.
- exponent can be defined for physical values represented via an Integer attribute and defines the exponent in double notation, e.g. 3.1415 = 31415e-4. Use 10^{exponent} as a factor to get a double value in a specific physical unit out of the integer. (see also explanation in [4.2.7 - Physical values](#)). The default value is 0.

References and compositions are directed relations. A relation has a role name that the relation is read as a fact phrase such as "a balise group has 1..8 balises" (subject - predicate - object).

SPT2TS-1219 - Every package, enumeration, struct, struct attributes and enumerator has an **info** field that describes the semantics in natural language.

SPT2TS-1659 - Definition of the infrastructure-Package:

Each package is formally specified by a set of attributes:

- `isDefinedBy` specifies the Internationalized Resource Identifier, where additional information on package can be found.
- `name` gives a long name of the package
- `prefix` is an abbreviation for the IRI, used for referencing classes/enumerations etc. between packages.
- `intId` defines an unique integer representing the package, used for compact referencing of objects between data sets.
- `version` contains a string with major.minor.editorial version-ids. All generated schemata are backwards compatible.
- `info` provides a documentation of the content and purpose of the package
- `containerStruct` specifies the struct inside of the package, which plays the role of root for composition hierarchy, used for referencing and data management.
- `structs` contains an array of classes/structs building the package
- `enums` contains an array of enumerations building the package.

The package schema is specified in a dedicated JSON-Schema in [TCCS - Data Model_02_Schema](#)

Here is an example for specification of the "infrastructure"-package.

```
{
  "$schema": "ERJU meta-model.json",
  "isDefinedBy": "http://ERJU/datamodel/0.3/infrastructure",
  "name": "infrastructure",
  "prefix": "infra",
  "intId": 1,
  "version": "0.3",
  "info": "Railway network topology and topography, position wrt topology,
location on a map, geometry, aka. alignment)",
  "containerStruct": "Infrastructure",
  "structs": [],  "enums": []
}
```

SPT2TS-1700 - UML view "Package":

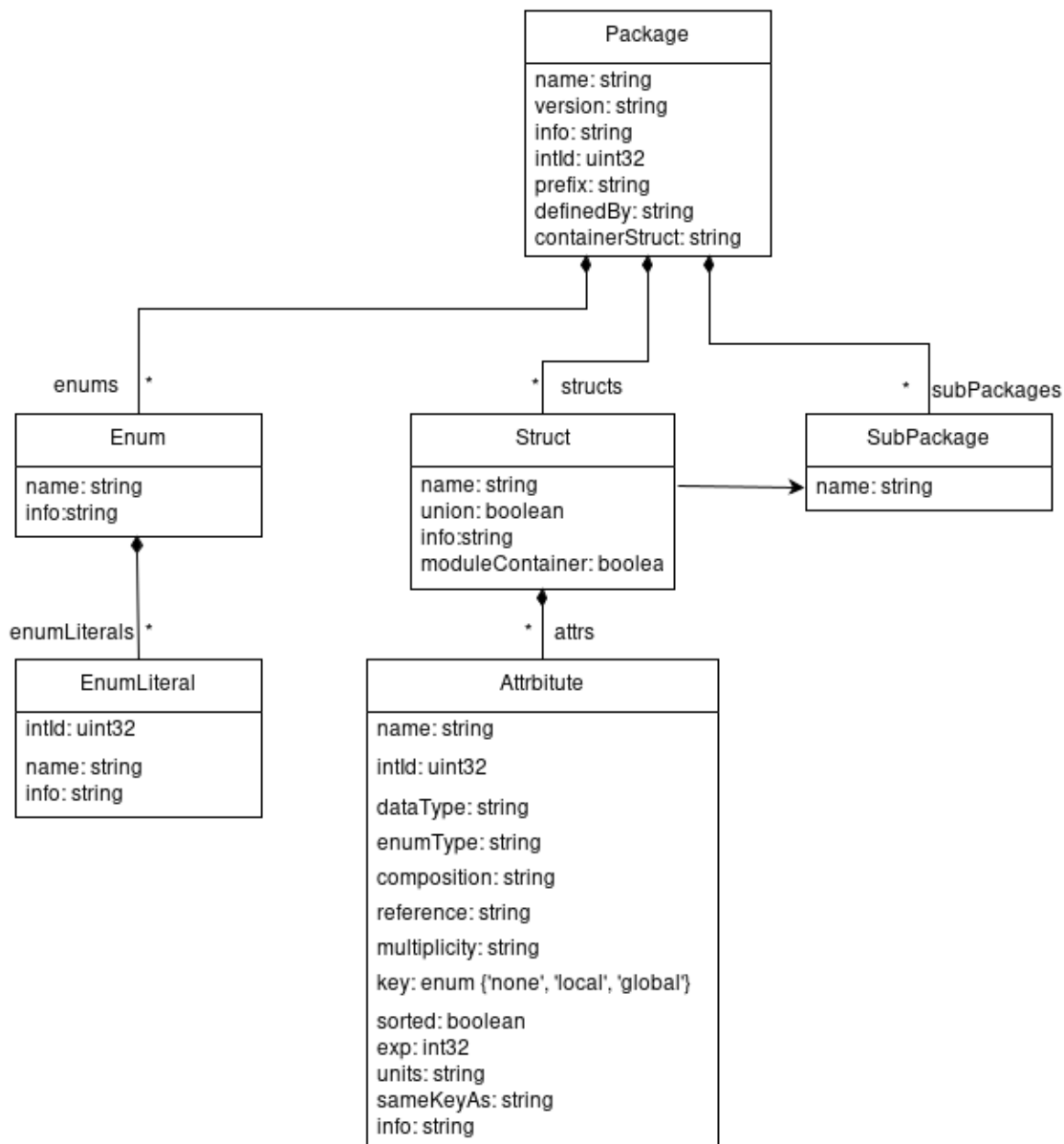


Figure 3. Class diagram representing the meta-model.

4.2 Model design decisions

SPT2TS-127381 - The selected schemata define a set of restrictions which must be considered in the data model, which represents the largest common denominator to the selected schemata. The following aspects must be considered:

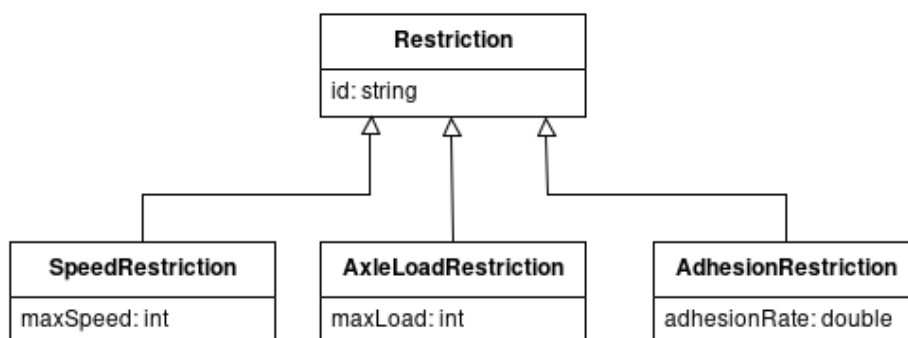
- handling of polymorphism, as not all target schemata support polymorphism and inheritance as such,
- definition of the model-overall structure to define the guideline, where which information to be

inserted,

- handling of physical values.

4.2.1 Polymorphism representation

SPT2TS-1701 - The primary use-case for this data model is serialisation in the context of API calls and the secondary use-case is Model-Driven-Development. Let's look at the inheritance issue in the serialisation context. Assume we have an inheritance hierarchy



If an interface has to transfer objects inherited from Restriction, the encoding would depend on serialisation language:

- XML: `<restriction xsi:type="SpeedRestriction" id="id1" maxSpeed="50"/>`
- JSON: `{"@type": "SpeedRestriction", "id": "id1", "maxSpeed": 50}`
- Protobuf: `{"type": "SpeedRestriction", "value": "a92fd9jhlso=="}`

In these examples, the serialisation followed the approach "use ANY-type to represent inherited objects". This has a disadvantage of dynamic typing.

Another approach is to define a dedicated class representing possible inherited classes statically. In this case, the upper examples change to:

- XML: `<restriction><speedRestriction id="id1" maxSpeed="50"/></restriction>`
- JSON: `{"id": "id1", "specificRestriction": {"speedRestriction": {"maxSpeed": 50}}}`
- Protobuf: `{"id": "id1", "specificRestriction": {"speedRestriction": {"maxSpeed": 50}}}`

SPT2TS-1582 - To provide a schema structure for the "representing" class, a union-construction is used:

```

{ "name": "SpecificRestriction", "union": true,
  "attrs": [
    { "intId": 1, "name": "speedRestriction", "composition": "SpeedRestriction",
    { "intId": 2, "name": "axleLoadRestriction", "composition": "AxleLoadRestriction",
    { "intId": 3, "name": "adhesionRestriction", "composition": "AdhesionRestriction"
  ]
}
  
```

To support this approach an additional attribute "union" is provided for each struct with the default value "false". Also, the API with an inherited reference is possible with the following specification:

```
{ "name": "SpecificRestrictionRef", "union": true,
  "attrs": [
    {"intId": 1, "name": "speedRestriction", "reference": "SpeedRestriction"},
    {"intId": 2, "name": "axleLoadRestriction", "reference": "AxleLoadRestriction"},
    {"intId": 3, "name": "adhesionRestriction", "reference": "AdhesionRestriction"}
  ]
}
```

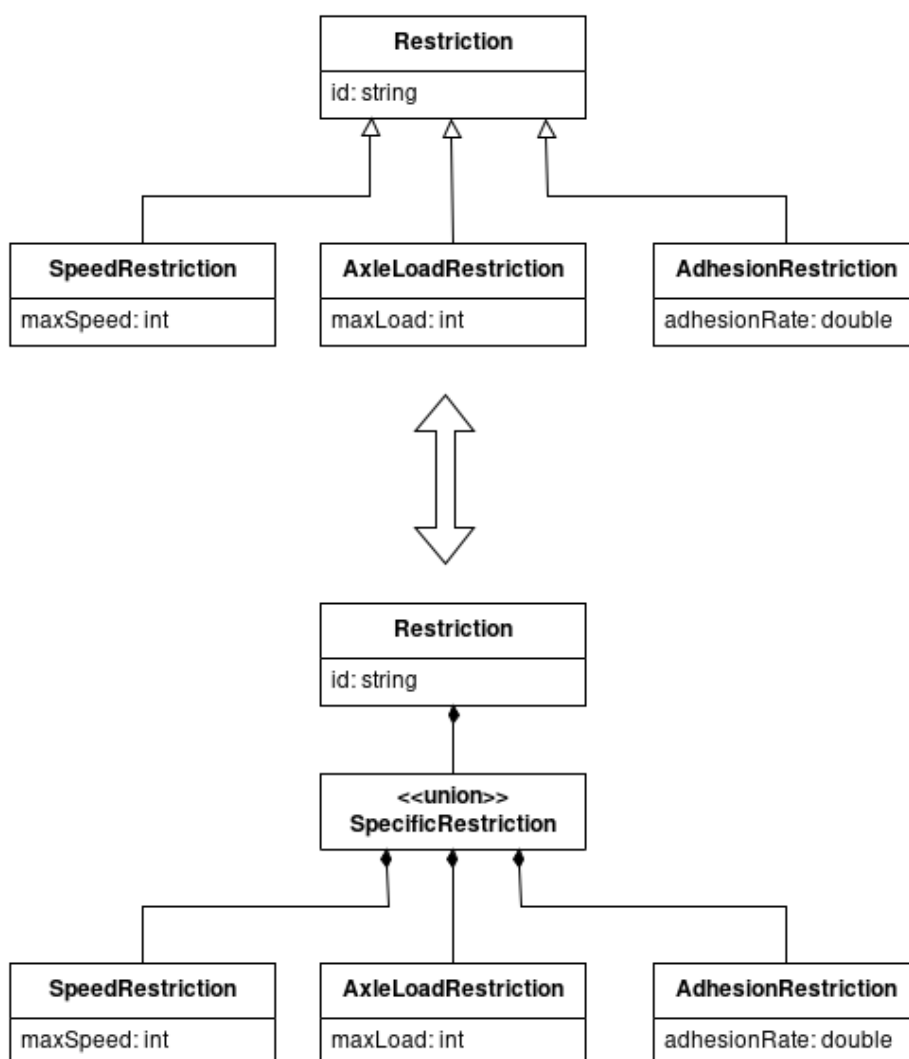


Figure 4: Composition pattern for polymorphism use-case

4.2.2 Model structure

SPT2TS-123593 - Discussion regarding selected model structure:

There are two "opposite" patterns in definition of "composition" hierarchy of the classes:

1. Flat hierarchy: to the very end one big container having all objects. This approach is used in Eulynx-DP. Following SOLID principle the model tries to avoid compositional relations in general and create only references between objects
2. Full-tree hierarchy: with containers creating a hierarchy optimal from data-consumers points of view.

For the current model the second pattern is selected for three main reasons:

- As one of the main objective for the data model is to be the basis for an API (either Point-to-Point or publish/subscribe) the API specification is much more clear in the second case:
 - In the first case: `atoPE.updateInfrastructure(oneBigContainer filled to 15%)`. Here the content of this big container must be specified outside of the API specification, e.g. "don't send Interlocking-configuration part - classA, classB ..."
 - In the second case: `atoPE.updateInfrastructure(topologyArea)`. Here it is absolutely clear, which information will arrive to the application, as the TopoArea-container is fully-self-containing.
- Having a reference to an object, a system S must be able to find its content. If all the objects will be managed by one big database, the system S could ask it for the object and would receive an answer. On the other hand, it is a strong limitation of the model, if it needs a central repository for working. In this sense, the composition hierarchy builds a natural addressability similar to DNS of the internet - having an address inside of hierarchy and using deployment table (DNS nameserver) allows to find the system managing any object inside. This allows to create a consistent linked model of any size and manage it in distributed data management systems.
- Creating compact containers with relatively small number of children simplifies learning of the model. If a developer needs to parse a one-big-container (with partial filling of 10-15%) he must understand all its parts, import all sub-schemata, generate code for all it's children and integrate it into the application

SPT2TS-122470 - The CCS/TMS data model follows a compositional tree structure, wherein the root of the model is a composition of several packages that constitute the CCS/TMS data model (e.g., Infrastructure, Restrictions, Rolling Stock, etc.), as shown in the figure below. Every package can be divided into sub-packages (e.g., TopologyArea, GeometryArea, etc.) depending on its needs. The figure below depicts the CCS/TMS model structure with the current included content:

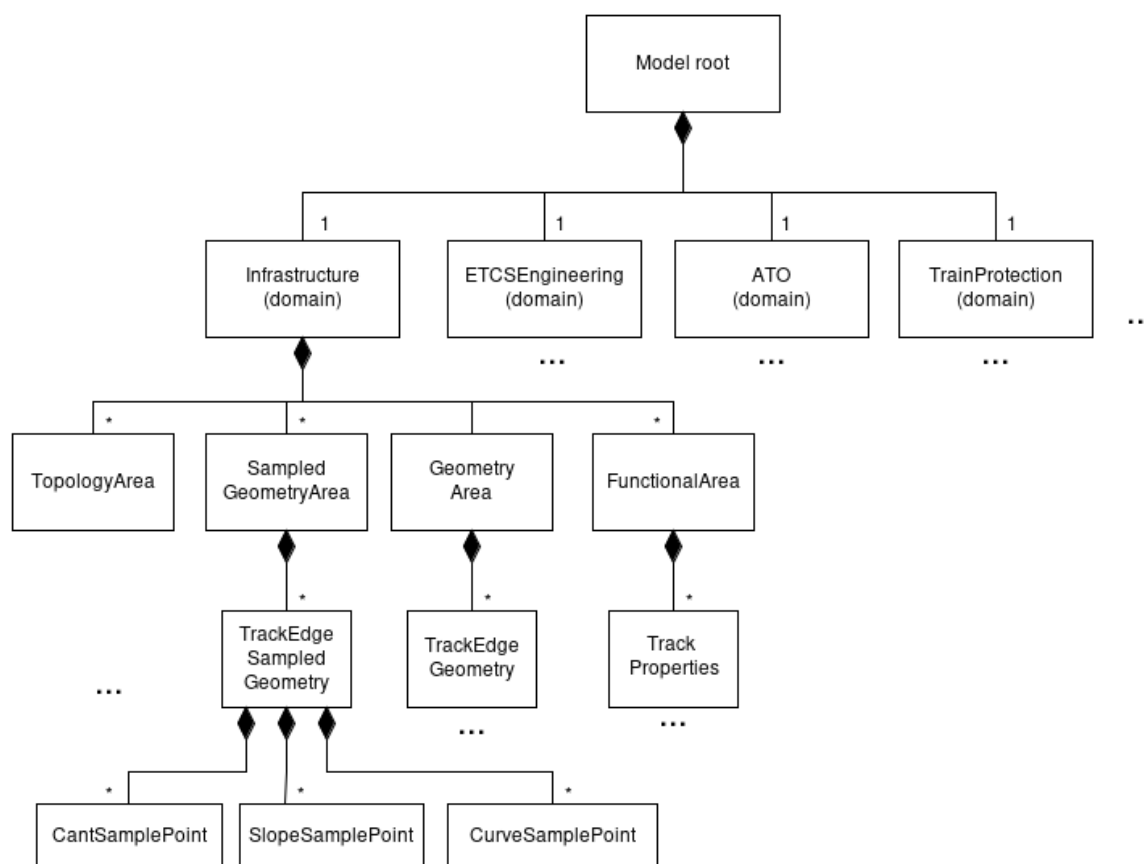


Figure 5 Overview of a compositional tree in data model.

The content of objects within each (sub-) package is defined as per definitions in [4.1.1 - Content of the modelling language](#).

4.2.3 Referencing structs and enums

SPT2TS-122471 - Constructing model packages should avoid cyclic dependencies, so the packages build a layer-hierarchy: upper layer-packages use (refer to) classes and enumerations specified in lower-layers-packages. The referencing is done via the point notation: `PackageId.class` or `PackageId.enumeration`. E.g. a `DangerPoint` in package `Engineering` refers to the `TrackEdgePoint` struct defined in package `Infrastructure`:

```
{
  "name": "DangerPoint",
  "attrs": [
    { "intId": 1, "name": "id", "dataType": "string", "key": "local" },
    { "intId": 2, "name": "name", "dataType": "string" },
    { "intId": 3, "name": "trackEdgePoint", "composition": "infra.TrackEdgePoint" }
  ]
}
```

4.2.4 Referencing objects

Discussion:

SPT2TS-122472 - A reference allows the parsing/reading system to retrieve an object. Example of a reference in the model in class `infra.TrackEdgeSection`:

```
{
  "name": "TrackEdgeSection",
  "attrs": [
    {"intId": 1, "name": "trackEdge", "reference": "TrackEdge", "info": "is positioned on track edge"},
    ...
  ]
}
```

In the message-schemata representing the model a reference is replaced by a string-attribute. The question in this section: what is the content of the string-attribute representing a reference, or e.g. in case of an XML-message:

```
<trackEdgeSection trackEdge="?????" ... />
```

There are several flexible approaches for referencing in different serialisation formats:

- XML/JSON: by Key/Ref pattern: `trackEdge="2774252a-ae71-41de-9f57-b80b11c0c040"`
- XML (XPath/XPointer): [https://dog.com/dogbreeds.xml#/bookstore/book\[1\]](https://dog.com/dogbreeds.xml#/bookstore/book[1]) <-- selects a first book element in the bookstore element
- OPC-UA: namespace/objectId in different variations.

Theoretically, all three approaches are possible, but not optimal as the data storage mechanism remains flexible:

- Key/Ref pattern is assumed as not sufficient: having hundred thousands of referensible objects in a **distributed** data sets would require the reader-system to ask each data set for the searched object, which is not effective.
- XPath/XPointer refers to "document -> address inside of a document", but in our case the document-based data management is not prescribed.
- OPC-UA requires an agreed set of namespaces containing all referenced objects. This is easily possible on a unique OPC-UA-Server managing all nodes. In case of distributed data management doubles the data management overhead.

Recommendation: Instead of referencing objects via data management containers (document in XML, table in SQL database, namespace in a single OPC-UA-Server) the referencing is done via the data model itself by providing the object-position inside of the objects-tree defined by composition relations:

```
trackEdge="/infra/topoArea[area51]/trackEdges[edge21]"
```


Besides an absolute path the object position can be also referred in relative way (e.g. "trackEdge=**edge21**" only) if it is still unambiguous due to pointing within the same area.

All the aspects in the address are parts of the data model itself and are agnostic to the data management technology:

```
{
  "prefix": "infra",
  "intId": 1,
  "containerStruct": "Infrastructure",
  "structs": [
    {
      "name": "Infrastructure",
      "attrs": [{"name": "topoAreas", "composition": "TopoArea", "intId": 2, "multiplicity": "*", ...}]
    },
    {
      "name": "TopoArea",
      "attrs": [{"name": "trackEdges", "composition": "TrackEdge", "intId": 3, "multiplicity": "*", ...}]
    }
  ]
}
```

The reference starts with the prefix of the package. The second element is an attribute of the structure declared as "containerStruct" in the package. As next the path goes through the attributes of type composition.

Having a layered and flexible model structure requires fine-grained objects with MANY references. It is assumed, that references would consume a considerable amount of the bandwidth. To reduce them a compact reference representation is proposed, where the attribute-names are replaced by integer-IDs also specified inside of the data model in "intId"-attributes:

instead of

```
trackEdge=/infra/topoAreas[area51]/trackEdges[edge21]
           use a compact
trackEdge="/1/2[area51]/3[edge21]"
```

The last open question is: "how to select an object from the array-attribute?"

Here we follow the XPath approach - it can be selected by one of two possibilities:

- By a zero-based-index: /1/2[#32]/3[#211]

- By the key-attribute-value: /1/2[area51]/3[edge21]. In this case the class to be selected must have one attribute marked as "key": "local" or "global". Normally it is an "id"-attribute of type string. To allow separation of index-based vs. key-based referencing, the keys must not start with '#'-character.

Both methods are applicable. For clear separation by the parser the '#' should only be used for index based references and is forbidden in key values. In addition, the slash '/' is not allowed in keys as well, to allow separation of absolute paths from relative object positions.

4.2.5 IDs / Keys

SPT2TS-123916 - As shown in previous section, IDs are used in many cases for object-referencing. In most cases the referenced objects have an attribute "id" of type string marked as "key": "global" or "local". From the model point of view this approach provides flexibility to use short IDs in local objects with short life-time and long forevere-unique IDs for objects with a long life-time.

E.g. one can use very short "speaking" ids and get a unique reference by listing the "address" of the object in compositional hierarchy:

`/infrastructure/areas[a51]/trackEdges[e1]`

Special case represents the functional model, which links together operational systems (TMS/Traffic CS) and maintenance systems (incl. diagnosis). The operational systems normally need only the last version of the infrastructure data, so short IDs which are unique inside of the data set would be sufficient. On the other hand, the maintenance system links long-living hardware-objects with functional-objects inside of maintenance history, which requires globally unique IDs from the functional objects, so the involved elements can be found even after restructuring of the element-containers (merging or splitting areas).

E.g. the area a51 will be split into two a51 and a5012, and the trackEdge a51/e1 will be moved to a5012. For real-time systems getting a new version with a new consistent linking between objects this will not be an issue - it is the responsibility of data provider to create a consistent version.

For the maintenance system which used the link to trackEdge a51/e1 to allocate some maintenance activity, to recover the state over twenty years will need to record and reapply all updates, which is not practical. Having a global key instead of "e1" would allow to find the object even after area-restructuring.

If the areas will be stable, as e.g. the ATO-areas, or they will be a subject of modifications, is not yet clear. The meta-model prepares for both cases and provides a mean to require local short key and global long key during the modelling depending on expected modifications.

To annotate this aspect we define the attribute "key" not as a boolean, but as an enumeration with values:
- "none" means not a key, which is the default value. The attribute "key" can be skipped in this case.

- "**local**" means key is unique inside a container. Container is an attribute in a structure which contains (is composition of) the structure with the key and having multiplicity > 1, e.g. the attribute

TopologyArea.trackEdges is a container for structures of type TrackEdge. If TrackEdge would have a local key, they must be different only inside of this one array.

- "**global**" means the key is globally unique.

The systems generating the data is free to select appropriate implementation for the global key, e.g.:

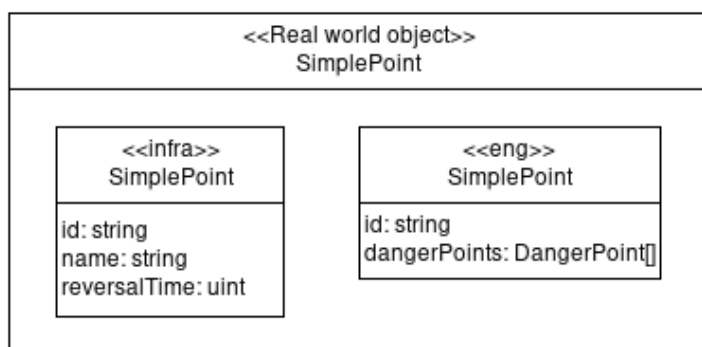
- generate UUID as a 32-bytes string as defined in rfc4122, e.g. "9a9dc630-7b43-4c58-a5c1-a411d238c5ad"
- generate UUID as a 22-bytes string with to base64 encoding, e.g. "mp3GMHtDTFilwQAApBHsOA"
- use centrally managed ID-generator

4.2.6 Object-splitting as object-double-linking

Introduction:

SPT2TS-122469 - The data model is structured in packages/layers covering specific aspects of the reality. In this situation one real-world-object is splitted in several data-objects representing various aspects of it, e. g.

- There is a "real-world-object" SimplePoint,
- Infra-package defines a functional data-object infra.SimplePoint covering name and transversal time important for operations,
- Engineering-package defines also a data-object eng.SimplePoint, adding danger-points to it, important for interlocking programming.



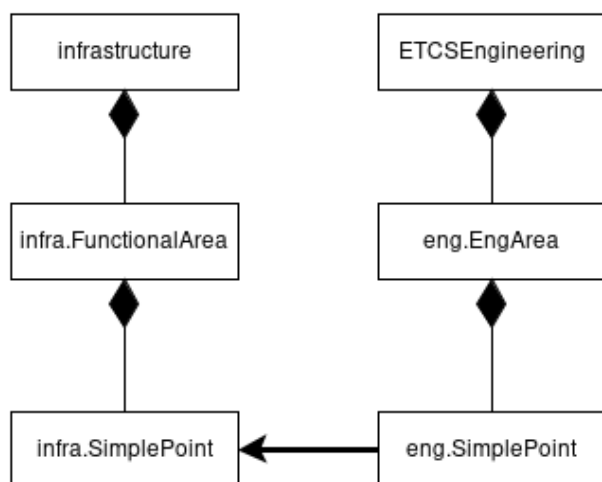
A typical Use-Case for a reading application is to combine several aspects of the physical object in memory. For this purpose it would need to retrieve several data-objects from the data storage and **link** them together.

Discussion:

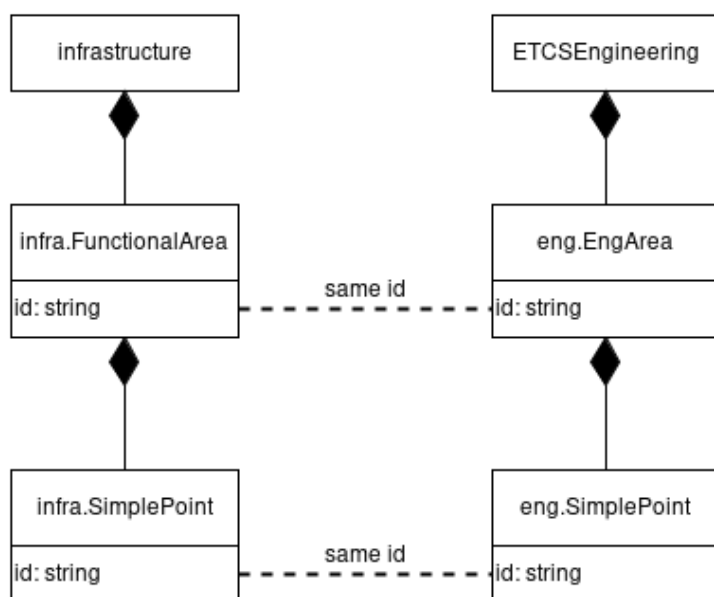
SPT2TS-122475 - The question in this section is: "how to identify data-objects to be linked together?".

There are two possibilities:

1. via **single-link-reference** from "less-used to the more-used" object inside of the data-set:



2. via **double-link-id convention** defined at data-model-level:

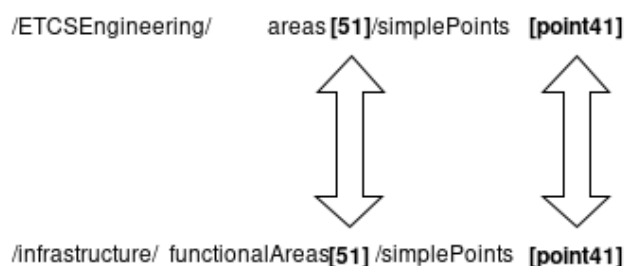


Decision:

SPT2TS-122476 - Splitted objects are linked to each other by **double-link-id-notation**. In case of the double-link-id-convention the real-world-object can be split in many parts and be constructed very effectively, as the reading application can deduce other parts from the current data-object automatically, e.g. knowing the reference to eng.SimplePoint it is easy to construct a reference to infra.SimplePoint: /ETCSEngineering/areas[51]/simplePoints[point41] ->

/infrastructure/functionalAreas[51]/simplePoints[point41]

The same is valid for the opposite direction:



This kind of dependencies are annotated in the model inside of key-attributes (here is a simplified notation from two different packages), e.g.

```
[
  {
    "name": "infra.SimplePoint",
    "attrs": [
      {"name": "id", "dataType": "string", "key": "global", "sameKeyAs": "eng.SimplePoint"}
    ]
  },
  {
    "name": "eng.SimplePoint",
    "attrs": [
      {"name": "id", "dataType": "string", "key": "global", "sameKeyAs": "infra.SimplePoint"}
    ]
  }
]
```

The "sameKeyAs"-dependency is transitive: a developer can follow the dependency-chain to identify all aspects of a real-world-object inside of the data-model.

SPT2TS-123991 - Another example for application of keys, herefor multiple areas Infrastructure-package:

```
{"name": "GeometryArea", ... "sameKeyAs": "TopoArea"}
{"name": "SampledGeometryArea", ... "sameKeyAs": "TopoArea"}
{"name": "PropertiesArea", ... "sameKeyAs": "TopoArea"}
{"name": "FunctionalArea", ... "sameKeyAs": "TopoArea"}
```

For each physical area there are five data objects representing different aspects of this area using the

same value in the attribute "id". Here is a data object of the Infrastructure-type containing these five areas:

```
{
  "topoAreas": [{"id": "area52" ...}],
  "geometryAreas": [{"id": "area52" ...geometry spec...}],
  "sampledGeometryAreas": [{"id": "area52" ...sampled geometry ...}],
  "functionalAreas": [{"id": "area52" ...functional objects...}],
  "propertiesAreas": [{"id": "area52" ...properties...}],
}
```

Use-case 1: Software developer is using "sameKeyAs" for developing a navigation in a part of the model. Normally the developer knows, which aspects of a physical object are required for a certain logic/function. The developer would look in the model specification, if these aspects (objects) use the "sameKeyAs" attributes and point to the same class or to each other.

Use-case 2: Automatic validation of a data set, that all objects representing aspects of a physical entity are present in the model. In this case an algorithm goes through the data model and collects objects created via "sameKeyAs": ObjectA <--> ObjectB. This would produce a set of disjunct graphs. Each such graph contains nodes/objects which use the same key to represent different aspects of a real-world object.

4.2.7 Physical values

SPT2TS-1656 - The modelling of physical values impacts the data model. The CCS/TMS data model could specify value range, precision, the physical unit, and the transformation into the target schema would select the appropriate data type, e.g. double, uint32, string. This could cause dependence on the used technology because data points transmitted in different schemata could subtly differ. Invariant transformation between target schemata is of the essence. Middleware should support several schemata at once, e.g. OPC-UA supports both XML and binary message representation. In this case, the transformation double -> string -> double is specific to the required precision: how many digits behind the point must be represented in string depends on the attribute to be handled. This increases the complexity of the middleware and introduces additional error sources for writing applications. Therefore, we prefer representing physical values as integers instead of floating points. An integer representation of the physical value "length" would be {"name": "length", "dataType": "uint32", "unit": "m", "exponent": -3}, which means physicalValue = uint32*10⁻³ [m], encoding in mm.

Note: It is also assumed that the integer representation fits better for binary transmissions with low bandwidth and safety-related system design in general.

4.2.8 Versioning

SPT2TS-127382 - The data objects modelled using the TMS/CCS-data model will be managed in a distributed way with different lifecycle, e. g. the life-cycle of infra.TopoArea and of opp.Movement are influenced by different systems. The proposed versioning shall be implemented using following rule:

To be consistent, the version of the object-user must be same or higher than the used-object-version.

Assuming an "in-order" distribution of changes (new versions) starting from the bottom of the "initial change" the user-application is able to detect the invalid-state (the rule above is violated) and the consistent state (the rule is valid for all objects *the application is interested in*).

To annotate versions in a way, that they can be compared (one version is same or higher then the other), there are two ways to model:

- Via integer-version starting with 1 as used e.g. ATO SS 126 (SegmentProfile.version as uint32)
- Via timestamp-version representing a microsecond since epoch. It can be encoded as a 64-bits-number in binary protocols and as an ISO-string in human-readable protocols.

The integer-versioning has an advantage to be more compact (4 bytes vs 8 bytes), but it requires implementation of a logical clock [[Lamport timestamp](#)]. Using timestamp-version is more simple, as in railway domain the time-synchronisation below milliseconds is not required.

4.2.8.1 Examples: Infrastructure - domain

SPT2TS-127383 - The various aspects of an infrastructure area are splitted in to several objects (TopoArea, FunctionalArea, GeometryArea ...). All these aspect-objects use the same id-attribute to annotate the describe the same real-world-object. Between these objects a usage-relation is established via the "sameKeyAs"-attribute in the model:

```
{
  {"name": "TopoArea", "attrs": [{"name": "id", "key": "global" ...}, ...
  {"name": "GeometryArea", "attrs": [{"name": "id", "key": "global", "sameKeyAs": "TopoArea" ...}, ...
  {"name": "FunctionalArea", "attrs": [{"name": "id", "key": "global", "sameKeyAs": "TopoArea" ...}, ...
}
```

That means, that for each object of GeometryArea, FunctionalArea ... the attribute "versionTimestamp" shall be >= than TopoArea.versionTimestamp to be valid. Example of data objects:

```
{
  "topoAreas": [
    {"id": "area51", "versionTimestamp": "2023-11-23T19:04:05.298973", ...}
    {"id": "area52", "versionTimestamp": "2023-10-23T19:04:05.298973", ...}
  ],
  "functionalAreas": [
```

```
{
  "id": "area51", "versionTimestamp": "2023-10-23T19:04:06.298973", ...
  "id": "area52", "versionTimestamp": "2023-10-23T19:04:07.298973", ...
}
```

In this example, the functionalAreas[area52] is valid, and functionalAreas[area51] is invalid. The reading-application shall wait until an updated version of functionalArea[area51] arrives before start using it. The functionalAreas[area52] can be used immediately.

4.2.8.2 Examples: Operational plan - domain

SPT2TS-127378 - {

```
"infrastructure": {
  "topoAreas": [
    {"id": "area51", "versionTimestamp": "2023-11-23T19:04:05.298973", ...}
    {"id": "area52", "versionTimestamp": "2023-10-23T19:04:05.298973", ...}
  ],
  ...
},
"operationalPlan": {
  "movements": [
    {"id": "T1", "issuedAt": "2023-11-23T19:04:05.198973", ... "trackEdge": "/1/1[area51]/3[trackEdge_1]
    ...},
    {"id": "T2", "issuedAt": "2023-11-23T19:04:05.298973", ... "trackEdge": "/1/1[area52]/3[trackEdge_6]
    ...}
  ]
}
```

In this example:

- the movement[T1] is invalid as it "uses" the trackEdge from an area (area51) with versionTimestamp > than movement.
- the movement[T2] is valid as it "uses" trackEdges from area (area52) with versionTimestamp < than movement.

5 Model and derived schemata

5.1 Complete Data Model

SPT2TS-127379 - The introduced model parts of this document lead to the following JSON definition (usable for implementation):

see information in  [TCCS - Data Model_00_Release Notes](#) , Table Column "Publish and Review" ( SPT2TS-122465 - [CCS/TMS Data Model Revisions](#))

6 References

SPT2TS-127380 - The CCS/TMS Data model has been defined based on the following models as input/references. The corresponding model references are also provided for relevant data objects in respective domains.

1. RCA Digital Map – MAP Object Catalogue [RCA.Doc.69] v1.0
2. EULYNX Data Model (<https://dataprep.eulynx.eu/2023-03/>) Release 1.2
3. LinX4Rail CDM ->PSM (Platform Specific Data Model)
4. RSM - RailSystemModel
5. Guide on the application of the common specifications of the register of Infrastructure (RINF)
6. TSI 2023 ETCS SS026 v4.0.0
7. TSI 2023 ATO SS126 v 1.0.0.
8. X2R4 ATO GoA 3/4 Specifications Baseline 0.1