HYBRIS – Efficient Specification and Analysis of Hybrid Systems - Part III: RCSD – a UML 2.0 Profile for the Railway Control System Domain

Kirsten Berkenkötter Ulrich Hannemann Jan Peleska
University of Bremen
P.O.B. 330 440
28334 Bremen, Germany
{kirsten, ulrichh, jp}@informatik.uni-bremen.de

Draft version

December 18, 2006

Contents

Chapter 1

Introduction

In this report we develop a domain specific language based on the UML 2.0 to support the *model driven development* process of railway control systems.

1.1 A Domain Specific Formalism for Railway Control Systems

With emphasis on a modeling language and its formal semantics, we support the foundation of the widely automated generation of controller components in the railway domain. As we provide a means to capture the requirements of these control components thoroughly and unambiguously, the focus within the development process shifts towards the modeling phase, i.e. the formalization of the application users' view onto the system.

We demonstrate our approach of utilizing a UML 2.0 profile as a *domain specific language* for a problem in the railway control system domain. The *domain of control* – also called *physical model* – consists of a railway network composed of track segments, points, signals, and sensors. Trains enter the domain of control at distinguished entry segments and request to take pre-defined routes through the network. Detection of trains is possible only via sensor observations. A *controller* monitors state changes within the network, derives train locations, and governs signals and points to enable the correct passage of trains through the network. With all activities, the controller must ensure that no hazardous situation arises, formulated by requiring compliance with a specific set of *safety conditions*.

The railway control domain is a perfect candidate to apply a domain specific language as it contains a rather limited amount of different entities. The specialized objects involved may exhibit only a limited variation of behavior, and the high safety requirements already established in the railway domain have resulted in a decent formalization of component descriptions. Part of the challenge of formulating a domain theory of railways [rai] lies in the long history of the domain where domain experts gathered a respectable amount of knowledge which is hard to contain in a computing science formalism. Thus, an approach to deal with critical railway control applications has to carefully connect the expertise in railway engineering with the development techniques of safety critical software.

Among the various proposed solutions, we observe a number of characteristics that we deem desirable:

1. The UniSpec language within the EURIS method [FKvV98] provides a domain specific language with *graphical elements* to reflect the topology of a railway network.

2. In order to support the development process with *standard tools* the wide-spectrum Unified Modeling Language UML [RJB04] is used in the SafeUML project [Hun06] which specifically aims at generating code conforming to safety standards. The use of UML is restricted here by guidelines to ensure maintenance of safety requirements which still allow sufficiently expressiveness for the modeling process.

3. In [PBH00, HP02, HP03] the domain analysis concentrates on the relevant issues for formal treatment of the control problem using a presentation form of tables and lists as foundation for

a *formal model*.

Based on these experiences, we propose to use the *profile* mechanism for UML 2.0 [OMG05b, OMG05a] to create a *domain-specific* description formalism for requirements modeling in the railway control systems domain (RCSD). This approach allows us to use a graphical representation of the domain elements with domain specific icons in order to facilitate the communication between domain experts and specialists for embedded control systems development. As the profile mechanism is part of the UML standard, the wide-spread variety of existing tools can be adapted within the very spirit of the UML using UML-inherent concepts. Since a profile allows to introduce new semantics for the elements of the profile we can attach a rigorous mathematical model to the descriptions of the domain model. Timed state transition system semantics form the base for formal transformations towards code generation for the controller as well as for the verification task that guarantees conformance to the safety requirements. Consequently, the RCSD profile constitutes the first and founding step in a development process for the automatic generation and verification of controllers derived from a domain model as outlined in [HP03, JPD04, PHK+06].

The next section gives a brief introduction to the railway control domain terminology as background for the development of a profile. Section 1.3 explains the basic concepts and techniques for the construction of a UML 2.0 profile. The main part, Chapter 2, contains the full formal notation of the RCSD profile. An example in Section 3.2 demonstrates the successful connection between the typical domain notation and the conceptual view of the profile.

1.2 Elements of the Railway Domain

Creating a domain specific profile requires identifying the elements of this domain and their properties as e.g. described in [Pac02]. We focus on the modeling of main tracks. All elements that are not allowed on main tracks as e.g. track locks are discarded. The further elements are divided into track elements, sensors, signals, automatic train runnings, and routes. Elements in the domain that come in different but similar shapes like signals are modeled as one element with different characteristics. In this way, we can abstract the railway domain to eight main modeling elements. These are described in the following:

Figure 1.1: Segment

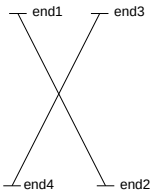

Figure 1.2: Crossing

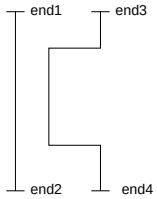

Figure 1.3: Interlaced segments

1.2.1 Track Elements

The track network consists of segments, crossings, and points. Segments are rails with two ends (see Fig. 1.1), while crossings consist of either two crossing segments or two interlaced segments (see Fig. 1.2 and Fig. 1.3). In general, the number of trains on a crossing is restricted to one to ensure safety. Points allow a changeover from one segment to another one. We use single points with a stem and a branch (see Fig. 1.4). There is no explicit model element for double points, as these are seldom used in praxis. If needed, they can be modeled by two single points. Single slip points and double slip points are crossings with one, respectively two, changeover possibilities from one of the crossing segments to the other one (see Fig. 1.5 and Fig. 1.6). All points have in common that the number of trains at each point in time is restricted to one.

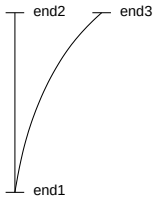

Figure 1.4: Single point

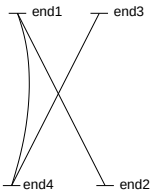

Figure 1.5: Single slip point

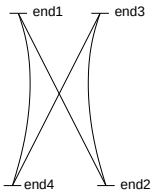

Figure 1.6: Double slip point

1.2.2 Sensors

Sensors are used to identify the position of trains on the track network, i.e. the current track element. To achieve this goal, track elements have entry and exit sensors located at each end. The number of sensors depends on the allowed driving directions, i.e. the uni- or bidirectional usage of the track element. Each sensor is the exit sensor of one track element and the entry sensor of the following one. If the track elements can be used bidirectionally, another sensor is needed that works vice versa.

1.2.3 Signals

Signals come in various ways. In general, they indicate if a train may go or if it has to stop. The permission to go may be constrained, e.g. by speed limits or by obligatory directions in case of points. As it is significant to know if a train moves according to signaling, signals are always located at sensors.

1.2.4 Automatic Train Running

Automatic train running systems are used to enforce braking of trains, usually in safety-critical situations. The brake enforcement may be permanent or controlled, i.e. it can be switched on and off. Automatic train running systems are also located at sensors.

1.2.5 Route Definition

As sensors are used as connection between track elements, routes of a track network are defined by sequences of sensors. They can be entered if the required signal setting of the first signal of the route is set. This can only be done if all points are in the correct position needed for this route. Conflicting routes cannot be released at the same time. Some conflicts occur as the required point positions or signal settings are incompatible. Another problem are routes that cross and are potentially safety-critical.

1.3 The UML 2.0 RCSD Profile

The next step is tailoring the UML 2.0 to the railway domain to provide the previously identified elements of the domain. There are two approaches to achieve this goal. The first one is using the UML 2.0 profile mechanism described in [OMG05b] and [OMG05a] that allows for:

- introducing new terminology,
- introducing new syntax/notation,
- introducing new constraints,
- introducing new semantics, and
- introducing further information like transformation rules.

Changing the existing metamodel itself e.g. by introducing semantics contrary to the existing ones or removing elements is not allowed. Consequently, each model that uses profiles is a valid UML model. The second approach is adapting the UML 2.0 metamodel to the needs of the railway domain by using MOF 2.0 (see [OMG06]). This approach offers more possibilities as elements can be added to or removed from the metamodel, syntax can be changed, etc. In fact, a new metamodel is created that is based on UML but is not UML anymore.

We have chosen the first approach - defining a UML 2.0 profile - as this supports exactly the features we need: the elements of the railway domain are new terminology that we want to use as modeling elements.

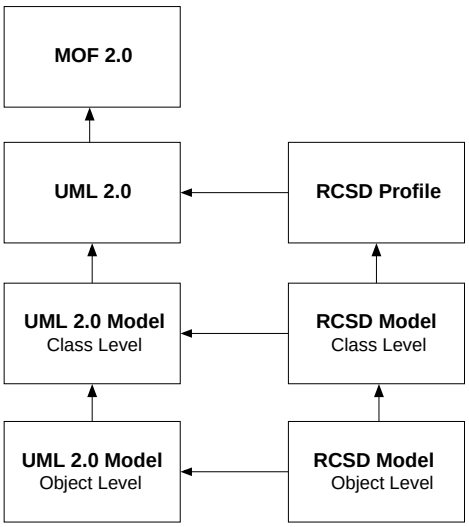

Figure 1.7: RCSD Profile in the UML metalevel context

To simplify communication between domain specialists and system developers, the usual notation of the railway domain should be used in a defined way. Therefore, constraints are needed to determine the meaning of the new elements. Track networks described with the new profile are transfered to transition systems. This is done by transformation rules. Also, we have valid UML models and therefore various tool support.

A UML 2.0 profile mainly consists of stereotypes, i.e. extensions of already existing UML modeling elements. You have to choose which element should be extended and define the add-ons. The RCSD profile uses either *Class*, *Association*, or *InstanceSpecification* as basis of stereotypes. In addition, new primitive datatypes and enumerations can be defined as necessary.

Unfortunately, defining eight stereotypes as suggested by the domain analysis in Sec. 1.2 is not sufficient. New primitive datatypes and enumerations are needed to model attributes adequately. Special kinds of association are needed to model interrelationships between stereotypes. Furthermore, UML supports two modeling layers, i.e. the model layer itself (class diagrams) and the instances layer (e.g. object diagrams). In the RCSD profile, both layers are needed: class diagrams are used to model specific parts of the railway domain, e.g. tramways or railroad models. They consist of the same components but with different characteristics. Second, object diagrams show explicit track layouts for such a model. Here, the symbols of the railway domain have to be used. We need stereotypes on the object level to define these features. For these reasons, the RCSD profile is structured in five parts: the definition of primitive datatypes and literals, network elements on class level, associations between these elements, network elements and associations on object level, and routes.

Defining new primitive types is the easiest part. New datatypes must be identified and their range of values specified. In our case, these are identifiers for all controllable elements, identifiers for routes (e.g. to specify conflicting ones), time instants and durations. For each primitive datatype, a corresponding literal has been described that defines the appearance of valid literal values.

The next part of the profile defines all track network elements, i.e. segments, crossing, points, signals, sensors, and automatic train runnings. *Segment*, *Crossing*, and *Point* have in common that

they form the track network itself, therefore they are all subclasses of the abstract *TrackElement*. Similarly, *SinglePoint* and *SlipPoint* are specializations of *Point*, while *DoubleSlipPoint* is a specialization of *SlipPoint*. All elements are equipped with a set of constraints that define which properties each element must support and how it is related to other elements.

Associations are used to connect track network elements. *SensorAssociations* connect track elements and sensors, *SignalAssociations* are used to associate signals to sensors, and *AutoRunAssociations* connect automatic train runnings and sensors. Here, constraints are needed to determine the kind of stereotype at the ends of each association. Most important, several constraints of *SensorAssociation* describe that each sensor is the exit sensor of one track element and the entry sensor of the following one. In that way, routes can be defined as sequences of sensors.

For each non-abstract modeling element and each association, there exists a corresponding instance stereotype. Domain-specific notation is defined here. Of course, usual UML notation can be used but is infeasible as we can see in the direct comparison in Section 3.2. Constraints are heavily used to describe correct values of instances. Hence, it is possible to determine well-formed object diagrams, i.e. well-formed track layouts.

Furthermore, the profile defines routes and their instances. Each *Route* is defined by an ordered sequence of sensors. Also, the signal setting for entering the route is given. Other properties are ordered sets of required point positions and of conflicts with other routes. The stereotypes to describe this information are given in Fig. 2.35. Again, constraints are used for unambiguous and strict definitions of properties and their valid values. To give an example, point positions given for a route have to refer to points located at that route.

Chapter 2

UML 2.0 Profile for the Railway Control System Domain

2.1 Primitives and Literals

Primitive data types are used as types of properties, i.e. attributes of classes. We define seven new primitive types, five of them devoted to specific identification numbers, and two of them used for modeling time. *AutoRunId*, *PointId*, *RouteId*, *SensorId*, and *SignalId* are identification types used for points, routes, sensors, and signals. *TimeInstant* and *Duration* are used for modeling points in time and fixed intervals of time.

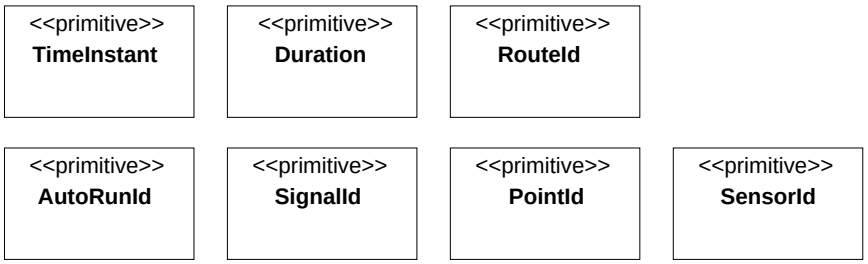

Figure 2.1: Primitive data types

Each primitive data type has its dedicated literal. The literal describes the values a variable of a certain type may take.

2.1.1 AutoRunId

Description

AutoRunId is a primitive type that is used to model identification numbers of automatic train running systems in RCSD.

Semantics

Instances of type AutoRunId have values in $\mathbb{N}$.

Notation

AutoRunId is given as a type, e.g. `aId:AutoRunId`.

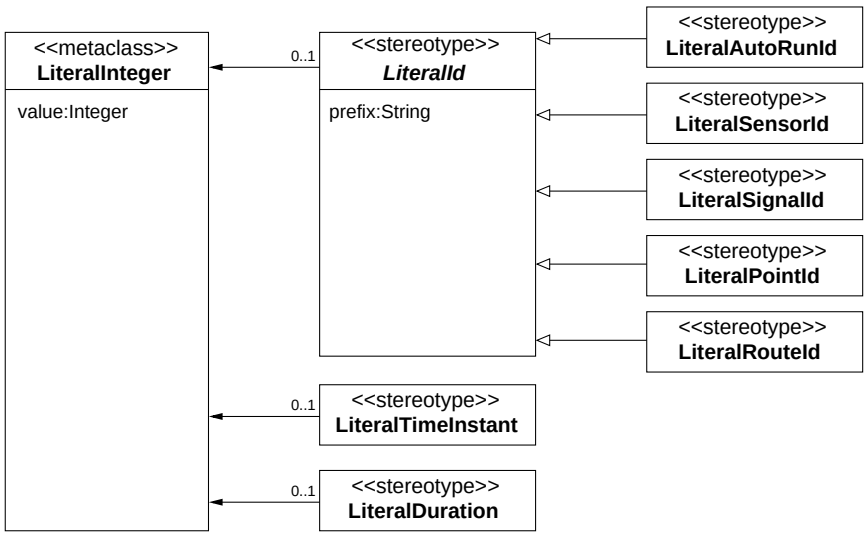

Figure 2.2: Literals of primitive data types

2.1.2 Duration

Description

Duration is a primitive type that is used to model time intervals in RCSD.

Semantics

Instances of type Duration have values in $\mathbb{N}$.

Notation

Duration is given as a type, e.g. `latency:Duration`.

2.1.3 LiteralAutoRunId

Description

LiteralAutoRunId describes valid literals for attributes with type AutoRunId.

Associations

None

Attributes

None.

Constraints

- The prefix is 'A':

```
inv LiteralAutoRunId1:
  prefix = 'A'
```

Semantics

Literals of type LiteralAutoRunId have values in $\mathbb{N}$ and a prefix 'A'.

Notation

LiteralAutoRunId is given as a prefixed value, e.g. `A50`.

2.1.4 LiteralDuration

Description

LiteralDuration describes valid literals for attributes with type Duration.

Associations

None

Attributes

None.

Constraints

- The value of attribute value is a natural.

```
inv LiteralDuration1:
  value >= 0
```

Semantics

Literals of type LiteralDuration have values in $\mathbb{N}$.

Notation

LiteralAutoRunId is given as a value, e.g. `50`.

2.1.5 LiteralId

Description

LiteralId describes valid literals for attributes with identification types (LiteralAutoRunId, LiteralPointId, LiteralSensorId, LiteralSignalId).

Associations

None.

Attributes

- prefix:String

Constraints

- The value of attribute value is a natural.

```
inv LiteralId1:
  value >= 0
```

Semantics

Literals of type LiteralDuration have values in $\mathbb{N}$ and a prefix.

Notation

None. LiteralId is abstract and must be used by its concrete subtypes.

2.1.6 LiteralPointId

Description

LiteralPointId describes valid literals for attributes with type PointId.

Associations

None

Attributes

None.

Constraints

- The prefix is 'P':

```
inv LiteralPointId1:
  prefix = 'P'
```

Semantics

Literals of type LiteralPointId have values in $\mathbb{N}$ and a prefix 'P'.

Notation

LiteralPointId is given as a prefixed value, e.g. `P50`.

2.1.7 LiteralRouteId

Description

LiteralRouteId describes valid literals for attributes with type RouteId.

Associations

None

Attributes

None.

Constraints

- The prefix is 'R':

```
inv LiteralRouteId1:
  prefix = 'R'
```

Semantics

Literals of type LiteralRouteId have values in $\mathbb{N}$ and a prefix 'R'.

Notation

LiteralRouteId is given as a prefixed value, e.g. `R1`.

2.1.8 LiteralSensorId

Description

LiteralSensorId describes valid literals for attributes with type SensorId.

Associations

None

Attributes

None.

Constraints

- The prefix is 'S':

```
inv LiteralSensorId1:
  prefix = 'S'
```

Semantics

Literals of type LiteralSensorId have values in $\mathbb{N}$ and a prefix 'S'.

Notation

LiteralSensorId is given as a prefixed value, e.g. `S11`.

2.1.9 LiteralSignalId

Description

LiteralSignalId describes valid literals for attributes with type SignalId.

Associations

None

Attributes

None.

Constraints

- The prefix is 'Sig':

```
inv LiteralSignalId1:
  prefix = 'Sig'
```

Semantics

Literals of type LiteralSignalId have values in $\mathbb{N}$ and a prefix 'Sig'.

Notation

LiteralSignalId is given as a prefixed value, e.g. `Sig5`.

2.1.10 LiteralTimeInstant

Description

LiteralTimeInstant describes valid literals for attributes with type TimeInstant.

Associations

None

Attributes

None.

Constraints

- The value of attribute value is a natural.

```
inv LiteralTimeInstant1:
  value >= 0
```

Semantics

Literals of type LiteralSignalId have values in $\mathbb{N}$.

Notation

LiteralTimeInstant is given as a value, e.g. `120`.

2.1.11 PointId

Description

PointId is a primitive type that is used to model identification numbers of points in RCSD.

Semantics

Instances of type PointId have values in $\mathbb{N}$.

Notation

PointId is given as a type, e.g. `pId:PointId`.

2.1.12 RouteId

Description

RouteId is a primitive type that is used to model identification numbers of routes in RCSD.

Semantics

Instances of type RouteId have values in $\mathbb{N}$.

Notation

RouteId is given as a type, e.g. `rId:RouteId`.

2.1.13 SensorId

Description

SensorId is a primitive type that is used to model identification numbers of sensors in RCSD.

Semantics

Instances of type SensorId have values in $\mathbb{N}$.

Notation

SensorId is given as a type, e.g. `senId:SensorId`.

2.1.14 SignalId

Description

SignalId is a primitive type that is used to model identification numbers of signals in RCSD.

Semantics

Instances of type SignalId have values in $\mathbb{N}$.

Notation

SignalId is used as a type, e.g. `sigId:SignalId`.

2.1.15 TimeInstant

Description

TimeInstant is a primitive type that is used to model points in time in RCSD.

Semantics

Instances of type TimeInstant have values in $\mathbb{N}$.

Notation

TimeInstant is used as a type, e.g. `requested:TimeInstant`.

2.2 Network Elements

A railway track network consists of *TrackElements* that are either *Segments*, *Crossings*, or *Points*. Additional elements are *Signals*, *Sensors*, and automatic train running systems *AutomaticRunning*.

TrackElements are rails and have at least two ends. At most one entry sensor and one exit sensor may be located at each end of the segment. Each track element has a number of maximal allowed trains at each moment in time and optionally a speed limit (e.g. curved segments).

Segments are either *bidirectional* segments that need one entry and one exit sensor at each end, or *unidirectional* segments that need an entry sensor at one end and an exit sensor at the opposite end. It is also possible, that sensors are located just at one end of the segment. In this case, the segment is either a sink (one entry sensor), a source (one exit sensor), or a combined sink/source (one entry sensor and one exit sensor).

Crossings are track elements with four ends. They consists of two track segments that either cross or are interlaced. Only one train is allowed on a crossing at each point in time. Like simple segments, crossing can be used unidirectionally or bidirectionally and are equipped with sensors.

Points have a *plus* and a *minus* position that are either *STRAIGHT*, *LEFT*, or *RIGHT*. The *plus* position is the default position of the point. The actual state and the requested state are important information about the point. In addition to the correct positions, *FAILURE* is a possible state here. The time of the last request is also memorized. In addition, the time needed to process a request, i.e. a state change, is modeled as *delta_p*. Points have also an id. There are three different kinds of points, *SinglePoints* with one branch, *SlipPoints* where two track segments cross with the possibility of one

changeover from one segment to the other one, and *DoubleSlipPoints* where two track segments cross with two changeover possibilities. All points are identified by their id.

Sensors have an actual state that is either *HIGH*, *LOW*, or *FAILURE*. A counter is used to register passing trains that is stimulated by the switching of the sensor state from *LOW* to *HIGH*. If a passing train is noticed, this information is sent at time *sentTime*. To guarantee the correct detection of passing trains, *delta_l* gives the time the sensor must be in state *HIGH* to notice a train, and *delta_tram* specifies the time that has to pass so that a subsequent train can be detected reliably. Like points, sensors have an id.

Signals provide the train respectively the engine driver with information, i.e. mainly the permission to go or to stop by signaling *GO* and *STOP*. In addition, speed limits and the direction to go (*STRAIGHT*,*LEFT*, *RIGHT*) can be signaled. Each signal has an actual and a requested state and memorizes the time of the last request, just as points. The time needed to fulfill a request is called *delta_s*. Signals are also identified by an id.

If braking of the train has to be enforced, e.g. before a *STOP* signal, automatic train running systems *AutomaticRunning* can be used. These cause braking of the train in case the speed limit is exceeded. They are either permanent or controlled and also identified by an id.

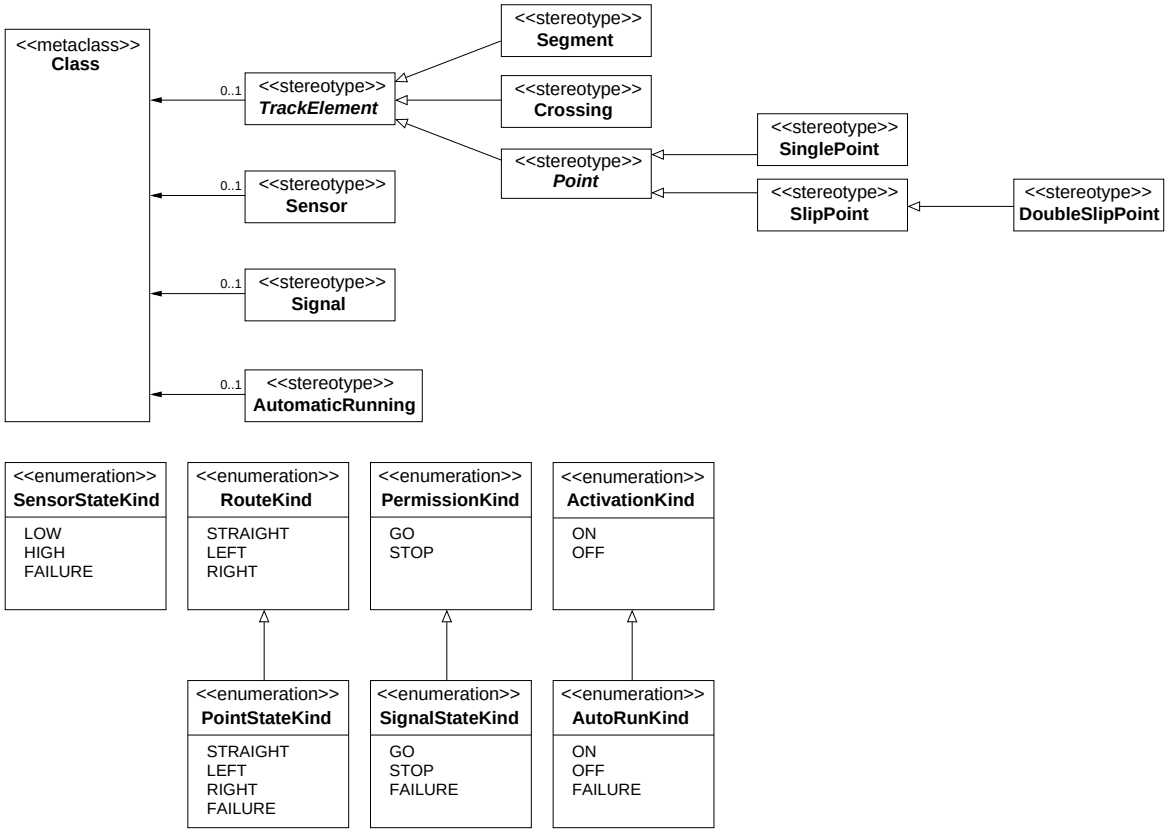

Figure 2.3: Stereotypes and enumerations for modeling the track network of a railway system

2.2.1 ActivationKind

Description

ActivationKind is an enumeration that specifies the valid values for requested states of automatic train runnings.

Semantics

The literals defined by ActivationKind are used as values of properties with type ActivationKind. These literals are:

- `ON`
- `OFF`

Notation

The defined literals are used as values of properties with type ActivationKind, e.g. `requestedState = ON`.

2.2.2 AutomaticRunning

Description

Automatic train running systems invoke automatic braking in case a train exceeds its speed limit, especially if a train is approaching a brake point or a velocity target point, i.e. a signal that enforces *STOP* or a speed limit. Automatic train runnings can be permanent if braking is always required or controlled.

Associations

None.

Attributes

None.

Constraints

- There is a mandatory property autoRunId with type AutoRunId. The property is read-only and has the multiplicity 1.

```
inv AutomaticRunning1:
  ownedAttribute->one(a | a.name->includes('autoRunId') and
                          a.type.name->includes('AutoRunId') and
                          a.upperBound()=1 and
                          a.lowerBound()=1 and
                          a.isReadOnly=true)
```

- There is a mandatory property actualState with type AutoRunKind. The property has the multiplicity 1.

```
inv AutomaticRunning2:
  ownedAttribute->one(a | a.name->includes('actualState') and
                          a.type.name->includes('AutoRunKind') and
                          a.upperBound()=1 and
                          a.lowerBound()=1)
```

- There is an optional property requestedState with type AutoRunKind. If present, the property has the multiplicity 1.

```
inv AutomaticRunning3:
  (ownedAttribute->one(a | a.name->includes('requestedState') and
                           a.type.name->includes('ActivationKind') and
                           a.upperBound()=1 and
                           a.lowerBound()>=0)) or
  (not ownedAttribute->exists(a | a.name->includes('requestedState')))
```

- There is an optional property requestTime with type TimeInstant. If present, the property has the multiplicity 1.

```
inv AutomaticRunning4:
  (ownedAttribute->one(a | a.name->includes('requestTime') and
                           a.type.name->includes('TimeInstant') and
                           a.upperBound()=1 and
                           a.lowerBound()=1)) or
  (not ownedAttribute->exists(a | a.name->includes('requestTime')))
```

- There is an optional property delta_a with type Duration. If present, the property is read-only and has the multiplicity 1.

```
inv AutomaticRunning5:
  (ownedAttribute->one(a | a.name->includes('delta_a') and
                           a.type.name->includes('Duration') and
                           a.upperBound()=1 and
                           a.lowerBound()=1 and
                           a.isReadOnly=true)) or
  (not ownedAttribute->exists(a | a.name->includes('delta_a')))
```

- The properties requested state, requestTime and delta_a are either all present or none of them.

```
inv AutomaticRunning6:
  ownedAttribute->one(a1 | a1.name->includes('requestedState')) implies
    (ownedAttribute->one(a2 | a2.name->includes('requestTime')) and
     ownedAttribute->one(a3 | a3.name->includes('delta_a'))) and
  ownedAttribute->one(a1 | a1.name->includes('requestTime')) implies
    (ownedAttribute->one(a2 | a2.name->includes('requestedState')) and
     ownedAttribute->one(a3 | a3.name->includes('delta_a'))) and
  ownedAttribute->one(a1 | a1.name->includes('delta_a')) implies
    (ownedAttribute->one(a2 | a2.name->includes('requestTime')) and
     ownedAttribute->one(a3 | a3.name->includes('requestedState')))
```

- All outgoing associations are AutoRunAssociations.

```
inv AutomaticRunning7:
  ownedAttribute->collect(outgoingAssociation)->
    forAll(oclIsTypeOf(AutoRunAssociation))
```

- There is exactly one outgoing association.

```
inv AutomaticRunning8:
  ownedAttribute->collect(outgoingAssociation)->size()=1
```

- There is a mandatory property sensor that has an outgoing association of type AutoRunAssociation. The property is read-only and has the multiplicity 1.

```
inv AutomaticRunning9:
  ownedAttribute->one(a | a.name->includes('sensor') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
                          a.isReadOnly = true and
                          a.outgoingAssociation.
                            oclIsTypeOf(AutoRunAssociation))
```

Semantics

AutomaticRunning is a stereotype of *Class*. Automatic braking points are used to enforce stopping of a train in case of exceeded speed limits, especially if a train has to stop. Each automatic running is associated to a sensor by an *AutoRunAssociations*.

If an automatic train running is permanently active, its *actualState* is always *ON*, else it can be also *OFF*. Non-permanently automatic train runnings have a *requestedState* set by the controller at *TimeInstant requestTime*. The time interval needed to switch from the actual to the requested state is *Duration delta_a*. They are identified by an *autoRunId* with type *AutoRunId*.

Notation

AutomaticRunnings are used in class diagrams using the UML class notation. For automatic train running instances, see the respective paragraph.

2.2.3 AutoRunKind

Description

AutoRunKind is an enumeration that specifies the valid values for states of automatic train runnings. It is a specialization of ActivationKind and adds *FAILURE* as possible state.

Semantics

The literals defined by AutoRunKind are used as values of properties with type AutoRunKind. These literals are:

- `ON`
- `OFF`
- `FAILURE`

Notation

The defined literals are used as values of properties with type AutoRunKind, e.g. `actualState = ON`.

2.2.4 Crossing

Description

Crossings are crossings of tracks or interlaced tracks which have in common that only one train is allowed at one moment in time. Crossings consist of two separate tracks with no possibility of a changeover from one track to the other one.

Associations

None.

Attributes

None.

Constraints

- There is an optional property e3Entry that has an outgoing association of type SensorAssociation. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv Crossing1:
  (ownedAttribute->one(a | a.name->includes('e3Entry') and
                           a.upperBound() = 1 and
                           a.lowerBound() >= 0 and
                           a.isReadOnly = true and
                           a.outgoingAssociation.
                             oclIsTypeOf(SensorAssociation))) or
  (not ownedAttribute->exists(a2 | a2.name->includes('e3Entry')))
```

- There is an optional property e3Exit that has an outgoing association of type SensorAssociation. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv Crossing2:
  (ownedAttribute->one(a | a.name->includes('e3Exit') and
                           a.upperBound() = 1 and
                           a.lowerBound() >= 0 and
                           a.isReadOnly = true and
                           a.outgoingAssociation.
                             oclIsTypeOf(SensorAssociation))) or
  (not ownedAttribute->exists(a2 | a2.name->includes('e3Exit')))
```

- There is an optional property e4Entry that has an outgoing association of type SensorAssociation. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv Crossing3:
  (ownedAttribute->one(a | a.name->includes('e4Entry') and
                           a.upperBound() = 1 and
                           a.lowerBound() >= 0 and
                           a.isReadOnly = true and
                           a.outgoingAssociation.
                             oclIsTypeOf(SensorAssociation))) or
  (not ownedAttribute->exists(a2 | a2.name->includes('e4Entry')))
```

- There is an optional property e4Exit that has an outgoing association of type SensorAssociation. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv Crossing4:
  (ownedAttribute->one(a | a.name->includes('e4Exit') and
                           a.upperBound() = 1 and
                           a.lowerBound() >= 0 and
                           a.isReadOnly = true and
                           a.outgoingAssociation.
                             oclIsTypeOf(SensorAssociation))) or
  (not ownedAttribute->exists(a2 | a2.name->includes('e4Exit')))
```

- The number of properties named e1Entry and e2Exit is 2 or/and the number of properties named e1Exit and e2Entry is 2.

```
inv Crossing5:
  ownedAttribute->select(a |
    a.name->includes('e1Entry') or a.name->includes('e2Exit'))->size()=2 or
  ownedAttribute->select(a |
    a.name->includes('e1Exit') or a.name->includes('e2Entry'))->size()=2
```

- The number of properties named e3Entry and e4Exit is 2 or/and the number of properties named e3Exit and e4Entry is 2.

```
inv Crossing6:
  ownedAttribute->select(a |
    a.name->includes('e3Entry') or a.name->includes('e4Exit'))->size()=2 or
  ownedAttribute->select(a |
    a.name->includes('e3Exit') or a.name->includes('e4Entry'))->size()=2
```

- The number of properties named e3Entry, e4Entry, e3Exit, and e4Exit is 4 or less.

```
inv Crossing7:
  ownedAttribute->select(a |
    a.name->includes('e3Entry') or a.name->includes('e4Entry') or
    a.name->includes('e3Exit') or a.name->includes('e4Exit'))->size() <= 4
```

- The number of properties named e3Entry, e4Entry, e3Exit, and e4Exit is not 3.

```
inv Crossing8:
  ownedAttribute->select(a |
    a.name->includes('e3Entry') or a.name->includes('e4Entry') or
    a.name->includes('e3Exit') or a.name->includes('e4Exit'))->size() <> 3
```

Semantics

Crossing is a specialization of *TrackElement* where two segments cross without the possibility to change from one segment to the other one. Each crossing has four ends, *end1* and *end2* mark one segment just as *end3* and *end4* mark the other one. Crossings are formed either by crossing track segments (see Fig. 2.4) or interlaced track segments (see Fig. 2.5). They are not sinks or sources of track networks and therefore require sensors at each end.

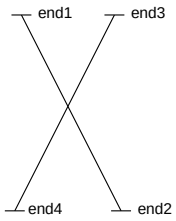

Figure 2.4: Crossed tracks

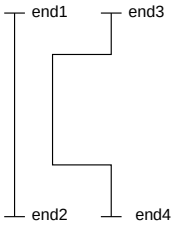

Figure 2.5: Interlaced tracks

Notation

Crossings are used in class diagrams using the UML class notation. For crossing instances, see the respective paragraph.

2.2.5 DoubleSlipPoint

Description

Slip points are used at crossings with a changeover possibility between two almost parallel track segments. A *double slip point* has two changeover possibilities, i.e. it is a crossing with two integrated (opposite) points.

Associations

None.

Attributes

None.

Constraints

- There is a mandatory property pointIdOpp with type PointId. The property is read-only and has the multiplicity 1.

```
inv DoubleSlipPoint1:
  ownedAttribute->one(a | a.name->includes('pointIdOpp') and
                          a.type.name->includes('PointId') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
                          a.isReadOnly = true)
```

- There is a mandatory property plusOpp with type RouteKind. The property is read-only and has the multiplicity 1.

```
inv DoubleSlipPoint2:
  ownedAttribute->one(a | a.name->includes('plusOpp') and
                          a.type.name->includes('RouteKind') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
                          a.isReadOnly = true)
```

- There is a mandatory property minusOpp with type RouteKind. The property is read-only and has the multiplicity 1.

```
inv DoubleSlipPoint3:
  ownedAttribute->one(a | a.name->includes('minusOpp') and
                          a.type.name->includes('RouteKind') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
                          a.isReadOnly = true)
```

- There is a mandatory property actualStateOpp with type PointStateKind. The property has the multiplicity 1.

```
inv DoubleSlipPoint4:
  ownedAttribute->one(a | a.name->includes('actualStateOpp') and
                          a.type.name->includes('PointStateKind') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1)
```

- There is a mandatory property requestedStateOpp with type RouteKind. The property has the multiplicity 1.

```
inv DoubleSlipPoint5:
  ownedAttribute->one(a | a.name->includes('requestedStateOpp') and
                          a.type.name->includes('RouteKind') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1)
```

- There is a mandatory property requestTimeOpp with type TimeInstant. The property has the multiplicity 1.

```
inv DoubleSlipPoint6:
  ownedAttribute->one(a | a.name->includes('requestTimeOpp') and
                          a.type.name->includes('TimeInstant') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1)
```

- If there is one property named e3Entry then there are also two properties named e2Exit and e4Exit.

```
inv DoubleSlipPoint7:
  ownedAttribute->one(a | a.name->includes('e3Entry')) implies
    ownedAttribute->one(a | a.name->includes('e2Exit')) or
    ownedAttribute->one(a | a.name->includes('e4Exit'))
```

- If there is one property named e3Exit then there are also two properties named e2Entry and e4Entry.

```
inv DoubleSlipPoint8:
  ownedAttribute->one(a | a.name->includes('e3Exit')) implies
    ownedAttribute->one(a | a.name->includes('e2Entry')) or
    ownedAttribute->one(a | a.name->includes('e4Entry'))
```

Semantics

DoubleSlipPoint is a specialization of *SlipPoint*. Is is a point that resembles a crossing with two possibilities to change from one segment to the other one (see Fig. 2.6).

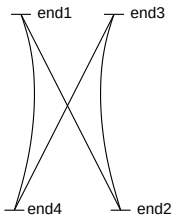

Figure 2.6: Double slip point

As a crossing, the train can travel from *end1* to *end2*, from *end2* to *end1*, from *end3* to *end4*, and from *end4* to *end3*. A single slip point enables traveling from *end1* to *end4* and from *end4* to *end1*. In addition, a double slip point also allows traveling from *end2* to *end3* and from *end3* to *end2*.

At each end of the point must be at least one sensor. The concrete placement of sensors depend on the enabled traveling routes on the point. Double slip points require more information than single slip points as a second point is added, i.e. *plusOpp*, *minusOpp*, *actualStateOpp*, and *requiredStateOpp* are further properties. We assume that both points combined in the double slip point have the same latency. Note that one of the two points has positions *STRAIGHT* and *RIGHT* and the other one positions *STRAIGHT* and *LEFT* as a double slip point consists of two opposite single points.

Notation

DoubleSlipPoints are used in class diagrams using the UML class notation. For double slip point instances, see the respective paragraph.

2.2.6 PermissionKind

Description

PermissionKind is an enumeration that specifies the valid values for signal requests.

Semantics

The literals defined by PermissionKind are used as values of properties with type PermissionKind. These literals are:

- `GO`
- `STOP`

Notation

The defined literals are used as values of properties with type PermissionKind, e.g. `requestedState = GO`.

2.2.7 Point

Description

Points are switches in track networks. Points are either single points or slip points. A single point consists of a stem and a left or right branch. A slip point consists of two crossing segments where a changeover from one segment to the other one is possible in at least one way.

Associations

None.

Attributes

None.

Constraints

- There is a mandatory property pointId with type PointId. The property is read-only and has the multiplicity 1.

```
inv Point1:
  ownedAttribute->one(a | a.name->includes('pointId') and
                          a.type.name->includes('PointId') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
                          a.isReadOnly = true)
```

- There is a mandatory property plus with type RouteKind. The property is read-only and has the multiplicity 1.

```
inv Point2:
  ownedAttribute->one(a | a.name->includes('plus') and
                          a.type.name->includes('RouteKind') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
                          a.isReadOnly = true)
```

- There is a mandatory property minus with type RouteKind. The property is read-only and has the multiplicity 1.

```
inv Point3:
  ownedAttribute->one(a | a.name->includes('minus') and
                          a.type.name->includes('RouteKind') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
                          a.isReadOnly = true)
```

- There is a mandatory property actualState with type PointStateKind. The property has the multiplicity 1.

```
inv Point4:
  ownedAttribute->one(a | a.name->includes('actualState') and
                          a.type.name->includes('PointStateKind') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1)
```

- There is a mandatory property requestedState with type RouteKind. The property has the multiplicity 1.

```
inv Point5:
  ownedAttribute->one(a | a.name->includes('requestedState') and
                          a.type.name->includes('RouteKind') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1)
```

- There is a mandatory property requestTime with type TimeInstant. The property has the multiplicity 1.

```
inv Point6:
  ownedAttribute->one(a | a.name->includes('requestTime') and
                          a.type.name->includes('TimeInstant') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1)
```

- There is a mandatory property delta_p with type Duration. The property is read-only and has the multiplicity 1.

```
inv Point7:
  ownedAttribute->one(a | a.name->includes('delta_p') and
                          a.type.name->includes('Duration') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
                          a.isReadOnly = true)
```

- There is an optional property e3Entry that has an outgoing association of type SensorAssociation. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv Point8:
  (ownedAttribute->one(a | a.name->includes('e3Entry') and
                          a.upperBound() = 1 and
                          a.lowerBound() >= 0 and
                          a.isReadOnly = true and
                          a.outgoingAssociation.
                            oclIsTypeOf(SensorAssociation))) or
  (not ownedAttribute->exists(a2 | a2.name->includes('e3Entry')))
```

- There is an optional property e3Exit that has an outgoing association of type SensorAssociation. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv Point9:
  (ownedAttribute->one(a | a.name->includes('e3Exit') and
                          a.upperBound() = 1 and
                          a.lowerBound() >= 0 and
                          a.isReadOnly = true and
                          a.outgoingAssociation.
                            oclIsTypeOf(SensorAssociation))) or
  (not ownedAttribute->exists(a2 | a2.name->includes('e3Exit')))
```

- There are at least 3 outgoing associations of type SensorAssociation.

```
inv Point10:
  ownedAttribute->collect(outgoingAssociation)->
    select(a | a.oclIsTypeOf(SensorAssociation))->size() >= 3
```

- There are at most 6 outgoing associations of type SensorAssociation.

```
inv Point11:
  ownedAttribute->collect(outgoingAssociation)->
    select(a | a.oclIsTypeOf(SensorAssociation))->size() <= 6
```

- The number of properties named e1Entry and e1Exit is at least 1.

```
inv Point12:
  ownedAttribute->select(a |
    a.name->includes('e1Entry') or a.name->includes('e1Exit'))->size() >= 1
```

- The number of properties named e1Entry and e1Exit is at most 2.

```
inv Point13:
  ownedAttribute->select(a |
    a.name->includes('e1Entry') or a.name->includes('e1Exit'))->size() <= 2
```

- The number of properties named e2Entry and e2Exit is at least 1.

```
inv Point14:
  ownedAttribute->select(a |
    a.name->includes('e2Entry') or a.name->includes('e2Exit'))->size() >= 1
```

- The number of properties named e2Entry and e2Exit is at most 2.

```
inv Point15:
  ownedAttribute->select(a |
    a.name->includes('e2Entry') or a.name->includes('e2Exit'))->size() <= 2
```

- The number of properties named e3Entry and e3Exit is at least 1.

```
inv Point16:
  ownedAttribute->select(a |
    a.name->includes('e3Entry') or a.name->includes('e3Exit'))->size() >= 1
```

- The number of properties named e3Entry and e3Exit is at most 2.

```
inv Point17:
  ownedAttribute->select(a |
    a.name->includes('e3Entry') or a.name->includes('e3Exit'))->size() <= 2
```

Semantics

Point is a specialization of *TrackElement* identified by a *pointId* with type *PointId* . They are used as switch from one track segment to another one. Points have a *plus* position and *minus* position. One of these is always *STRAIGHT* and one is always *LEFT* or *RIGHT* depending on the design of the point. Each point has an *actualState* and a *requestedState* that is either *STRAIGHT*, *LEFT*, or*RIGHT*. The actual state can also be *FAILURE*. The time of the latest request is a *TimeInstant* stored in *requestTime*. The *Duration* needed to switch from one state to another one is *delta_p*.

Sensors are associated to points by *SensorAssociations*. At least one sensor is positioned at each end of the point as a point is not allowed as sink or source of a track network. Points can be used bidirectionally or unidirectionally. In the latter case, two sensors are needed at each end of the point, one entry sensor and one exit sensor.

Notation

None. Point is abstract and must be used by its concrete specializations *SinglePoint* and *SlipPoint*.

2.2.8 PointStateKind

Description

PointStateKind is an enumeration that specifies the valid values for point states. It is a specialization of RouteKind and adds *FAILURE* as possible state.

Semantics

The literals defined by PointStateKind are used as values of properties with type PointStateKind. These literals are:

- `STRAIGHT`
- `LEFT`
- `RIGHT`
- `FAILURE`

Notation

The defined literals are used as values of properties with type PointStateKind, e.g. `actualState = RIGHT`.

2.2.9 RouteKind

Description

RouteKind is an enumeration that specifies the valid values for signals with route indications and point state requests.

Semantics

The literals defined by RouteKind are used as values of properties with type RouteKind. These literals are:

- `STRAIGHT`
- `LEFT`
- `RIGHT`

Notation

The defined literals are used as values of properties with type RouteKind, e.g. `direction = RIGHT`.

2.2.10 Segment

Description

Segment describes a part of a track network with two ends, i.e. straight and curved network elements.

Associations

None.

Attributes

None.

Constraints

- The number of properties named e1Entry, e2Entry, e1Exit, and e2Exit is 4 or less.

```
inv Segment1:
  ownedAttribute->select(a | a.name->includes('e1Entry') or
                             a.name->includes('e2Entry') or
                             a.name->includes('e1Exit') or
                             a.name->includes('e2Exit'))->size() <= 4
```

- The number of properties named e1Entry, e2Entry, e1Exit, and e2Exit is 1 or more.

```
inv Segment2:
  ownedAttribute->select(a | a.name->includes('e1Entry') or
                             a.name->includes('e2Entry') or
                             a.name->includes('e1Exit') or
                             a.name->includes('e2Exit'))->size() >= 1
```

- The number of properties named e1Entry, e2Entry, e1Exit, and e2Exit is not 3.

```
inv Segment3:
  ownedAttribute->select(a | a.name->includes('e1Entry') or
                             a.name->includes('e2Entry') or
                             a.name->includes('e1Exit') or
                             a.name->includes('e2Exit'))->size() <> 3
```

Semantics

Trains travel on track elements. A segment is a specialization of *TrackElement* with two ends as shown in Fig. 2.7. In general, there are three possibilities: traveling from *end1* to *end2*, traveling from *end2* to *end1*, or traveling in both directions. The first two possibilities classify a *unidirectional* segment, the last describes a *bidirectional* segment.

Passing trains have to be detected to allow monitoring of the track network. If there is an entry sensor on one end of the segment, passing the entry sensor means entering the segment. Vice versa, passing the exit sensor at the opposite end means leaving the segment.

Figure 2.7: Segment

A segment may have at most two associations with type *SensorAssociations* to entry sensors and at most two associations to exit sensors: *end1Entry*, *end1Exit*, *end2Entry*, and *end2Exit*. There must be at least one association to a sensor.

A segment that has one entry and one exit sensor at the same end is a sink and source of a network. If there is only one entry sensor associated, the segment is a sink; if there is only one exit sensor associated, the segment is a source.

Notation

Segments are used in class diagrams using the UML class notation. For segment instances, see the respective paragraph.

2.2.11 Sensor

Description

Sensors notice passing trains by changes of their sensor state given by SensorStateKind. If the state changes from *LOW* to *HIGH*, a passing train can be detected.

Associations

None.

Attributes

None.

Constraints

- There is a mandatory property sensorId with type SensorId. The property is read-only and has the multiplicity 1.

```
inv Sensor1:
  ownedAttribute->one(a | a.name->includes('sensorId') and
                          a.type.name->includes('SensorId') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
                          a.isReadOnly = true)
```

- There is a mandatory property counter with type Integer. The property has the multiplicity 1.

```
inv Sensor2:
  ownedAttribute->one(a | a.name->includes('counter') and
                          a.type.name->includes('Integer') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1)
```

- There is a mandatory property actualState with type SensorStateKind. The property has the multiplicity 1.

```
inv Sensor3:
  ownedAttribute->one(a | a.name->includes('actualState') and
                          a.type.name->includes('SensorStateKind') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1)
```

- There is a mandatory property delta_t with type Duration. The property is read-only and has the multiplicity 1.

```
inv Sensor4:
  ownedAttribute->one(a | a.name->includes('delta_t') and
                          a.type.name->includes('Duration') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
                          a.isReadOnly = true)
                          a.lowerBound() = 1)
```

- There is a mandatory property sentTime with type TimeInstant. The property has the multiplicity 1.

```
inv Sensor5:
  ownedAttribute->one(a | a.name->includes('sentTime') and
                          a.type.name->includes('TimeInstant') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1)
                          a.lowerBound() = 1)
```

- There is a mandatory property delta_tram with type Duration. The property is read-only and has the multiplicity 1.

```
inv Sensor6:
  ownedAttribute->one(a | a.name->includes('delta_tram') and
                          a.type.name->includes('Duration') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
                          a.isReadOnly = true)
```

- All outgoing associations are either SensorAssociations, SignalAssociations, or AutoRunAssociations.

```
inv Sensor7:
  ownedAttribute->collect(outgoingAssociation)->forAll(a |
    a.oclIsTypeOf(SensorAssociation) or
    a.oclIsTypeOf(SignalAssociation) or
    a.oclIsTypeOf(AutoRunAssociation) or
    a.isUndefined)
```

- There are at least 2 outgoing SensorAssociations.

```
inv Sensor8:
  ownedAttribute->collect(outgoingAssociation)->select(a |
    a.oclIsTypeOf(SensorAssociation))->size()>= 2
```

- There are at most 10 outgoing SensorAssociations

```
inv Sensor9:
  ownedAttribute->collect(outgoingAssociation)->select(a |
    a.oclIsTypeOf(SensorAssociation))->size()<= 10
```

- There is at most 1 outgoing SignalAssociation.

```
inv Sensor10:
  ownedAttribute->collect(outgoingAssociation)->select(a |
    a.oclIsTypeOf(SignalAssociation))->size() <= 1
```

- There is at most 1 outgoing AutoRunAssociation.

```
inv Sensor11:
  ownedAttribute->collect(outgoingAssociation)->select(a |
    a.oclIsTypeOf(AutoRunAssociation))->size() <= 1
```

- There is a mandatory property entrySeg that has an outgoing association of type SensorAssociation. The property is read-only and has the multiplicity 0..1 or 1.

```
inv Sensor12:
  ownedAttribute->one(a | a.name->includes('entrySeg') and
                          a.upperBound() = 1 and
                          a.lowerBound() >= 0 and
                          a.isReadOnly = true and
                          a.outgoingAssociation.
                            oclIsTypeOf(SensorAssociation))
```

- There is an optional property entryCross that has an outgoing association of type SensorAssociation. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv Sensor13:
  (ownedAttribute->one(a | a.name->includes('entryCross') and
                           a.upperBound() = 1 and
                           a.lowerBound() >= 0 and
                           a.isReadOnly = true and
                           a.outgoingAssociation.
                             oclIsTypeOf(SensorAssociation))) or
  (not ownedAttribute->exists(a | a.name->includes('entryCross')))
```

- There is an optional property entryPoint that has an outgoing association of type SensorAssociation. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv Sensor14:
  (ownedAttribute->one(a | a.name->includes('entryPoint') and
                           a.upperBound() = 1 and
                           a.lowerBound() >= 0 and
                           a.isReadOnly = true and
                           a.outgoingAssociation.
                             oclIsTypeOf(SensorAssociation))) or
  (not ownedAtrribute->exists(a | a.name->includes('entryPoint')))
```

- There is an optional property entrySlPoint that has an outgoing association of type SensorAssociation. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv Sensor15:
  (ownedAttribute->one(a | a.name->includes('entrySlPoint') and
                           a.upperBound() = 1 and
                           a.lowerBound() >= 0 and
                           a.isReadOnly = true and
                           a.outgoingAssociation.
                             oclIsTypeOf(SensorAssociation))) or
  (not ownedAttribute->exists(a | a.name->includes('entrySlPoint')))
```

- There is an optional property entryDbSlPoint that has an outgoing association of type SensorAssociation. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv Sensor16:
  (ownedAttribute->one(a | a.name->includes('entryDbSlPoint') and
                           a.upperBound() = 1 and
                           a.lowerBound() >= 0 and
                           a.isReadOnly = true and
                           a.outgoingAssociation.
                             oclIsTypeOf(SensorAssociation))) or
  (not ownedAttribute->exists(a | a.name->includes('entrySlPoint')))
```

- There is a mandatory property exitSeg that has an outgoing association of type SensorAssociation. The property is read-only and has the multiplicity 0..1 or 1.

```
inv Sensor17:
  ownedAttribute->one(a | a.name->includes('exitSeg') and
                           a.upperBound() = 1 and
                           a.lowerBound() >= 0 and
                           a.isReadOnly = true and
                           a.outgoingAssociation.
                             oclIsTypeOf(SensorAssociation))
```

- There is an optional property exitCross that has an outgoing association of type SensorAssociation. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv Sensor18:
  (ownedAttribute->one(a | a.name->includes('exitCross') and
                           a.upperBound() = 1 and
                           a.lowerBound() >= 0 and
                           a.isReadOnly = true and
                           a.outgoingAssociation.
                             oclIsTypeOf(SensorAssociation))) or
  (not ownedAttribute->exists(a | a.name->includes('exitCross')))
```

- There is an optional property exitPoint that has an outgoing association of type SensorAssociation. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv Sensor19:
  (ownedAttribute->one(a | a.name->includes('exitPoint') and
                           a.upperBound() = 1 and
                           a.lowerBound() >= 0 and
                           a.isReadOnly = true and
                           a.outgoingAssociation.
                             oclIsTypeOf(SensorAssociation))) or
  (not ownedAttribute->exists(a | a.name->includes('exitPoint')))
```

- There is an optional property exitSlPoint that has an outgoing association of type SensorAssociation. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv Sensor20:
  (ownedAttribute->one(a | a.name->includes('exitSlPoint') and
                           a.upperBound() = 1 and
                           a.lowerBound() >= 0 and
                           a.isReadOnly = true and
                           a.outgoingAssociation.
                             oclIsTypeOf(SensorAssociation))) or
  (not ownedAttribute->exists(a | a.name->includes('exitSlPoint')))
```

- There is an optional property exitDbSlPoint that has an outgoing association of type SensorAssociation. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv Sensor21:
  (ownedAttribute->one(a | a.name->includes('exitDbSlPoint') and
                           a.upperBound() = 1 and
                           a.lowerBound() >= 0 and
                           a.isReadOnly = true and
                           a.outgoingAssociation.
                             oclIsTypeOf(SensorAssociation))) or
  (not ownedAttribute->exists(a | a.name->includes('exitSlPoint')))
```

- There is an optional property signal that has an outgoing association of type SignalAssociation. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv Sensor22:
  (ownedAttribute->one(a | a.name->includes('signal') and
                           a.upperBound() = 1 and
                           a.lowerBound() >= 0 and
                           a.isReadOnly = true and
                           a.outgoingAssociation.
                             oclIsTypeOf(SignalAssociation))) or
  (not ownedAttribute->exists(a | a.name->includes('signal')))
```

- There is an optional property autoRun that has an outgoing association of type AutoRunAssociation. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv Sensor23:
  (ownedAttribute->one(a | a.name->includes('autoRun') and
                           a.upperBound() = 1 and
                           a.lowerBound() >= 0 and
                           a.isReadOnly = true and
                           a.outgoingAssociation.
                             oclIsTypeOf(AutoRunAssociation))) or
  (not ownedAttribute->exists(a | a.name->includes('autoRun')))
```

Semantics

Sensor is a stereotype of *Class*. Each sensor requires the following properties: the state *actualState* of the sensor that is either *HIGH*, *LOW*, or *FAILURE*, a *counter* with type *Integer* that is incremented if a passing train has been detected, the *Duration delta_l* needed to detected a train reliably, the *TimeInstant sentTime* at which this detection has occurred and is signaled, and the minimal *Duration delta_tram* between two passing trains to guarantee reliable detection of both of them. Sensors also have a *sensorId* with type *SensorId*.

A Sensor is associated by a *SensorAssociations* to two track elements. The properties *entry* and *exit* model these relationships.

A Sensor is associated to at most one signal by a *SignalAssociation* and at most one automatic train running system by an *AutoRunAssociation*.

Notation

Sensors are used in class diagrams using the UML class notation. For sensor instances, see the respective paragraph.

2.2.12 SensorStateKind

Description

SensorStateKind is an enumeration that specifies the valid values for sensor states.

Semantics

The literals defined by SensorStateKind are used as values of properties with type SensorStateKind. These literals are:

- `HIGH`
- `LOW`
- `FAILURE`

Notation

The defined literals are used as values of properties with type SensorStateKind, e.g. `actualState = HIGH`.

2.2.13 Signal

Description

Signals are used to provide important information to the engine driver, i.e. speed limitations, driving directions, the permission to go, or the need to stop.

Associations

None.

Attributes

None.

Constraints

- There is a mandatory property signalId with type SignalId. The property is read-only and has the multiplicity 1.

```
inv Signal1:
  ownedAttribute->one(a | a.name->includes('signalId') and
                          a.type.name->includes('SignalId') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
                          a.isReadOnly = true)
```

- There is a mandatory property actualState with type SignalStateKind. The property has the multiplicity 1.

```
inv Signal2:
  ownedAttribute->one(a | a.name->includes('actualState') and
                          a.type.name->includes('SignalStateKind') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1)
```

- There is a mandatory property requestedState with type PermissionKind. The property has the multiplicity 1.

```
inv Signal3:
  ownedAttribute->one(a | a.name->includes('requestedState') and
                          a.type.name->includes('PermissionKind') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1)
```

- There is a mandatory property requestTime with type TimeInstant. The property has the multiplicity 1.

```
inv Signal4:
  ownedAttribute->one(a | a.name->includes('requestTime') and
                          a.type.name->includes('TimeInstant') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1)
```

- There is a mandatory property delta_s with type Duration. The property is read-only and has the multiplicity 1.

```
inv Signal5:
  ownedAttribute->one(a | a.name->includes('delta_s') and
                          a.type.name->includes('Duration') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
                          a.isReadOnly = true)
```

- There is an optional property limit with type Integer. If present, the property has the multiplicity 0..1 or 1.

```
inv Signal6:
  (ownedAttribute->one(a | a.name->includes('limit') and
                          a.type.name->includes('Integer') and
                          a.upperBound() = 1 and
                          a.lowerBound() >= 0)) or
  (not ownedAttribute->exists(a | a.name->includes('limit')))
```

- There is an optional property direction with type RouteKind. If present, the property has the multiplicity 0..1 or 1.

```
inv Signal7:
  (ownedAttribute->one(a | a.name->includes('direction') and
                          a.type.name->includes('RouteKind') and
                          a.upperBound() = 1 and
                          a.lowerBound() >= 0)) or
  (not ownedAttribute->exists(a | a.name->includes('direction')))
```

- There is an optional property requestedDir with type RouteKind. If present, the property has the multiplicity 0..1 or 1.

```
inv Signal8:
  (ownedAttribute->one(a | a.name->includes('requestedDir') and
                          a.type.name->includes('RouteKind') and
                          a.upperBound() = 1 and
                          a.lowerBound() >= 0)) or
  (not ownedAttribute->exists(a | a.name->includes('requestedDir')))
```

- The properties direction and requestedDir are either both present or none of them.

```
inv Signal9:
  ownedAttribute->one(a1 | a1.name->includes('direction')) implies
    ownedAttribute->one(a2 | a2.name->includes('requestedDir'))  and
  ownedAttribute->one(a1 | a1.name->includes('requestedDir')) implies
    ownedAttribute->one(a2 | a2.name->includes('direction'))
```

- All outgoing associations are SignalAssociations.

```
inv Signal10:
  ownedAttribute->forAll(a |
    a.outgoingAssociation.oclIsTypeOf(SignalAssociation) or
    (a.outgoingAssociation->size() = 0 ))
```

- There is exactly one outgoing SignalAssociation.

```
inv Signal11:
  ownedAttribute->collect(outgoingAssociation)->select(a |
    a.oclIsTypeOf(SignalAssociation))->size() = 1
```

- There is an optional property sensor that has an outgoing association of type SignalAssociation. If present, the property is read-only and has the multiplicity 1.

```
inv Signal12:
  ownedAttribute->one(a | a.name->includes('sensor') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
                          a.isReadOnly = true and
                          a.outgoingAssociation.
                            oclIsTypeOf(SignalAssociation))
```

Semantics

Signal is a stereotype of *Class* . Each signal has an *actualState* and a *requestedState*. The *TimeInstant* at which a request is received is *requestTime* while *Duration delta_s* gives the time needed to switch from one state to another one. Signals can give further information to the engine driver, i.e. a speed *limit* and the *direction* in case the signal is located before a point.

A signal is always located at a sensor. This relationship is modeled by a *SignalAssociation*. There are no other associations.

Notation

Signals are used in class diagrams using the UML class notation. For signal instances, see the respective paragraph.

2.2.14 SignalStateKind

Description

SignalStateKind is an enumeration that specifies the valid values for signal states. It is a specialization of PermissionKind and adds *FAILURE* as possible value.

Semantics

The literals defined by SignalStateKind are used as values of properties with type SignalStateKind. These literals are:

- `GO`
- `STOP`
- `FAILURE`

Notation

The defined literals are used as values of properties with type SignalStateKind, e.g. `actualState = GO`.

2.2.15 SinglePoint

Description

Single points are points with a stem and a left or right branch.

Associations

None.

Attributes

None.

Constraints

- If there is one property named e1Entry then there are also two properties named e2Exit and e3Exit.

```
inv SinglePoint1:
  ownedAttribute->one(a | a.name->includes('e1Entry')) implies
    ownedAttribute->one(a | a.name->includes('e2Exit')) and
    ownedAttribute->one(a | a.name->includes('e3Exit'))
```

- If there is one property named e1Exit then there are also two properties named e2Entry and e3Entry.

```
inv SinglePoint2:
  ownedAttribute->one(a | a.name->includes('e1Exit')) implies
    ownedAttribute->one(a | a.name->includes('e2Entry')) and
    ownedAttribute->one(a | a.name->includes('e3Entry'))
```

Semantics

SinglePoint is a specialization of *Point*. Is is a point with one branch as shown in Fig. 2.8. Obviously, the branch can be left or right depending on the design of the point.

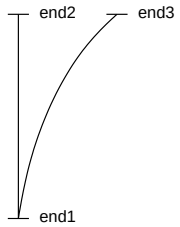

Figure 2.8: Single point

Trains can travel on several ways over the point: if the point can be entered at *end1*, it can be left either at *end2* or *end3* depending on the actual state. If the point is used bidirectionally, possible ways are from *end2* to *end1* or from *end3* to *end1*. For each way on a point, the respective entry sensors and exit sensors must be present.

Notation

SinglePoints are used in class diagrams using the UML class notation. For single point instances, see the respective paragraph.

2.2.16 SlipPoint

Description

Slip points are used at crossings with a changeover possibility between two almost parallel track segments. A *single slip point* has one changeover possibility, i.e. it is a crossing with one integrated point.

Associations

None.

Attributes

None.

Constraints

- There is an optional property e4Entry that has an outgoing association of type SensorAssociation. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv SlipPoint1:
  (ownedAttribute->one(a | a.name->includes('e4Entry') and
                           a.upperBound() = 1 and
                           a.lowerBound() >= 0 and
                           a.isReadOnly = true and
```

```
                          a.outgoingAssociation.
                            oclIsTypeOf(SensorAssociation))) or
  (not ownedAttribute->exists(a2 | a2.name->includes('e4Entry')))
```

- There is an optional property e4Exit that has an outgoing association of type SensorAssociation. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv SlipPoint2:
  (ownedAttribute->one(a | a.name->includes('e4Exit') and
                          a.upperBound() = 1 and
                          a.lowerBound() >= 0 and
                          a.isReadOnly = true and
                          a.outgoingAssociation.
                            oclIsTypeOf(SensorAssociation))) or
  (not ownedAttribute->exists(a2 | a2.name->includes('e4Exit')))
```

- The number of properties named e4Entry and e4Exit is at least 1.

```
inv SlipPoint3:
  ownedAttribute->select(a |
    a.name->includes('e4Entry') or a.name->includes('e4Exit'))->size() >= 1
```

- The number of properties named e4Entry and e4Exit is at most 2.

```
inv SlipPoint4:
  ownedAttribute->select(a |
    a.name->includes('e4Entry') or a.name->includes('e4Exit'))->size() <= 2
```

- If there is one property named e1Entry then there are also two properties named e2Exit and e4Exit.

```
inv SlipPoint5:
  ownedAttribute->one(a | a.name->includes('e1Entry')) implies
    ownedAttribute->one(a | a.name->includes('e2Exit')) and
    ownedAttribute->one(a | a.name->includes('e4Exit'))
```

- If there is one property named e1Exit then there are also two properties named e2Entry and e4Entry.

```
inv SlipPoint6:
  ownedAttribute->one(a | a.name->includes('e1Exit')) implies
    ownedAttribute->one(a | a.name->includes('e2Entry')) and
    ownedAttribute->one(a | a.name->includes('e4Entry'))
```

- If there is one property named e3Entry then there are also one property e4Exit.

```
inv SlipPoint7:
  ownedAttribute->one(a | a.name->includes('e3Entry')) implies
    ownedAttribute->one(a | a.name->includes('e4Exit'))
```

- If there is one property named E3Exit then there is also one property e4Entry.

```
inv SlipPoint8:
  ownedAttribute->one(a | a.name->includes('e3Exit')) implies
    ownedAttribute->one(a | a.name->includes('e4Entry'))
```

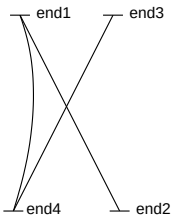

Figure 2.9: Single slip point

Semantics

SlipPoint is a specialization of *Point*. Is is a point that resembles a crossing with one possibility to change from one segment to the other one. A slip point with one changeover possibility is called *single slip point* (see Fig. 2.9).

As a crossing, the train can travel from *end1* to *end2*, from *end2* to *end1*, from *end3* to *end4*, and from *end4* to *end3*. A single slip point enables traveling from *end1* to *end4* and from *end4* to *end1*.

Single slip points require sensors at the fourth end of the point, namely *e4Entry* and *e4Exit*. At each end of the point must be at least one sensor. The concrete placement of sensors depend on the enabled traveling routes on the point.

Notation

SlipPoints are used in class diagrams using the UML class notation. For slip point instances, see the respective paragraph.

2.2.17 TrackElement

Description

The track network consists of track elements, i.e. segments and points. A track element has at least two ends that serve as connection points between different elements. Passing trains are monitored by sensors located at each of the two ends of the segment.

Associations

None.

Attributes

None.

Constraints

- There is a mandatory property maxNumberOfTrains with type Integer. The property is read-only and has the multiplicity 1.

```
inv TrackElement1:
  ownedAttribute->one(a | a.name->includes('maxNumberOfTrains') and
                          a.type.name->includes('Integer') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
                          a.isReadOnly = true)
```

- All outgoing associations are SensorAssociations.

```
inv TrackElement2:
  ownedAttribute->collect(outgoingAssociation)->
```

```
    forAll(a | a.oclIsTypeOf(SensorAssociation) or
               a.isUndefined)
```

- There is an optional property e1Entry that has an outgoing association of type SensorAssociation. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv TrackElement3:
  (ownedAttribute->one(a | a.name->includes('e1Entry') and
                           a.upperBound() = 1 and
                           a.lowerBound() >= 0 and
                           a.isReadOnly = true and
                           a.outgoingAssociation.
                             oclIsTypeOf(SensorAssociation))) or
  (not ownedAttribute->exists(a2 | a2.name->includes('e1Entry')))
```

- There is an optional property e1Exit that has an outgoing association of type SensorAssociation. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv TrackElement4:
  (ownedAttribute->one(a | a.name->includes('e1Exit') and
                           a.upperBound() = 1 and
                           a.lowerBound() >= 0 and
                           a.isReadOnly = true and
                           a.outgoingAssociation.
                             oclIsTypeOf(SensorAssociation))) or
  (not ownedAttribute->exists(a2 | a2.name->includes('e1Exit')))
```

- There is an optional property e2Entry that has an outgoing association of type SensorAssociation. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv TrackElement5:
  (ownedAttribute->one(a | a.name->includes('e2Entry') and
                           a.upperBound() = 1 and
                           a.lowerBound() >= 0 and
                           a.isReadOnly = true and
                           a.outgoingAssociation.
                             oclIsTypeOf(SensorAssociation))) or
  (not ownedAttribute->exists(a2 | a2.name->includes('e2Entry')))
```

- There is an optional property e2Exit that has an outgoing association of type SensorAssociation. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv TrackElement6:
  (ownedAttribute->one(a | a.name->includes('e2Exit') and
                           a.upperBound() = 1 and
                           a.lowerBound() >= 0 and
                           a.isReadOnly = true and
                           a.outgoingAssociation.
                             oclIsTypeOf(SensorAssociation))) or
  (not ownedAttribute->exists(a2 | a2.name->includes('e2Exit')))
```

- There is an optional property limit with type Integer. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv TrackElement7:
  (ownedAttribute->one(a | a.name->includes('limit') and
                           a.type.name->includes('Integer') and
                           a.upperBound() = 1 and
                           a.lowerBound() >= 0 and
                           a.isReadOnly = true)) or
  (not ownedAttribute->exists(a2 | a2.name->includes('limit')))
```

Semantics

TrackElement is a stereotype with *Class* as its metaclass. Each track element has a maximal number of trains *maxNumberOfTrains* as a property. At each end of the track element there may be one entry sensor and one exit sensor, given as *e1Entry*, *e1Exit*, *e2Entry*, and *e2Exit*. In addition, a fixed speed *limit* can be defined per track element. All associations related to *TrackElement* are *SensorAssociations*.

Notation

None. TrackElement is abstract and must be used by its concrete specializations.

2.3 Associations

Associations are used to connect track network elements. *SensorAssociations* are used to associate sensors to track elements, *SignalAssociations* are used to associate signals to sensors, while automatic train runnings are also connected to sensors but by means of *AutoRunAssociations*.

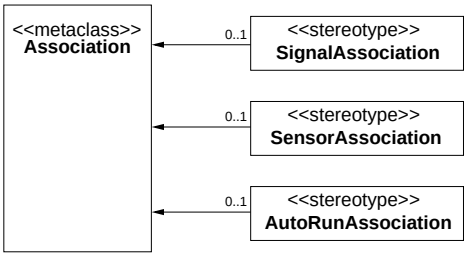

Figure 2.10: Stereotypes for modeling associations between track network elements

2.3.1 AutoRunAssociation

Description

AutoRunAssociations are used to connect sensors to automatic train runnings.

Associations

None.

Attributes

None.

Constraints

- There are 2 ends.

```
inv AutoRunAssociation1:
  memberEnd->size()=2
```

- There are 2 end types.

```
inv AutoRunAssociation2:
  endType->size()=2
```

- One end type is Sensor.

```
inv AutoRunAssociation3:
  endType->one(t | t.oclIsKindOf(Sensor))
```

- One end type is AutomaticRunning.

```
inv AutoRun4:
  endType->one(t | t.oclIsKindOf(AutomaticRunning))
```

Semantics

AutoRunAssociations connect sensors and automatic train runnings, respectively properties of them. They are always binary.

Notation

AutoRunAssociations are used in class diagrams using the UML association notation. For instances, called *AutoRunLink*, see the respective paragraph.

2.3.2 SensorAssociation

Description

SensorAssociations are used to connect sensors to track elements.

Associations

None.

Attributes

None.

Constraints

- There are at least 2 ends.

```
inv SensorAssociation1:
  memberEnd->size()>=2
```

- There are at most 5 ends.

```
inv SensorAssociation2:
  memberEnd->size()<=5
```

- There are 2 end types.

```
inv SensorAssociation3:
  endType->size() = 2
```

- One end type is Sensor.

```
inv SensorAssociation4:
  endType->one(t | t.oclIsKindOf(Sensor))
```

- If there is an end entrySeg or exitSeg, there is one end type Segment.

```
inv SensorAssociation5:
  memberEnd->exists(m1 | m1.name->includes('entrySeg') or
                         m1.name->includes('exitSeg')) implies
    endType->one(t | t.oclIsTypeOf(Segment))
```

- If there is an end entryCross or exitCross, there is one end type Crossing.

```
inv SensorAssociation6:
  memberEnd->exists(m1 | m1.name->includes('entryCross') or
                         m1.name->includes('exitCross')) implies
    endType->one(t | t.oclIsTypeOf(Crossing))
```

- If there is an end entryPoint or exitPoint, there is one end type SinglePoint.

```
inv SensorAssociation7:
  memberEnd->exists(m1 | m1.name->includes('entryPoint') or
                         m1.name->includes('exitPoint')) implies
    endType->one(t | t.oclIsTypeOf(SinglePoint))
```

- If there is an end entrySlPoint or exitSlPoint, there is one end type SlipPoint.

```
inv SensorAssociation8:
  memberEnd->exists(m1 | m1.name->includes('entrySlPoint') or
                         m1.name->includes('exitSlPoint')) implies
    endType->one(t | t.oclIsTypeOf(SlipPoint))
```

- If there is an end entryDbSlPoint or exitDbSlPoint, there is one end type DoubleSlipPoint.

```
inv SensorAssociation9:
  memberEnd->exists(m1 | m1.name->includes('entryDbSlPoint') or
                         m1.name->includes('exitDbSlPoint')) implies
    endType->one(t | t.oclIsTypeOf(DoubleSlipPoint))
```

- If there is an end entrySeg, the other ends are named e1Entry or e2Entry. All other ends belong to the same Class.

```
inv SensorAssociation10:
  memberEnd->exists(m1 | m1.name->includes('entrySeg')) implies
    (let p:Property =
       memberEnd->select(m1 | m1.name->includes('entrySeg'))->first
     in
      memberEnd->excluding(p)->
        forAll(m3 | m3.name->includes('e1Entry') or
                    m3.name->includes('e2Entry')) and
      memberEnd->excluding(p)->
         forAll(m3,m4 | m3.owningClass = m4.owningClass)
     )
```

- If there is an end entryCross, the other ends are named e1Entry, e2Entry, e3Entry, or e4Entry. All other ends belong to the same Class.

```
inv SensorAssociation11:
  memberEnd->exists(m1 | m1.name->includes('entryCross')) implies
    (let p:Property =
       memberEnd->select(m1 | m1.name->includes('entryCross'))->first
     in
       memberEnd->excluding(p)->
         forAll(m3 | m3.name->includes('e1Entry') or
                     m3.name->includes('e2Entry') or
                     m3.name->includes('e3Entry') or
                     m3.name->includes('e4Entry')) and
      memberEnd->excluding(p)->
         forAll(m3,m4 | m3.owningClass = m4.owningClass)
    )
```

- If there is an end entryPoint, the other ends are named e1Entry, e2Entry, or e3Entry. All other ends belong to the same Class.

```
inv SensorAssociation12:
  memberEnd->exists(m1 | m1.name->includes('entryPoint')) implies
    (let p:Property =
       memberEnd->select(m1 | m1.name->includes('entryPoint'))->first
     in
      memberEnd->excluding(p)->
        forAll(m3 | m3.name->includes('e1Entry') or
                    m3.name->includes('e2Entry') or
                    m3.name->includes('e3Entry')) and
      memberEnd->excluding(p)->
        forAll(m3,m4 | m3.owningClass = m4.owningClass)
    )
```

- If there is an end entrySlPoint, the other ends are named e1Entry, e2Entry, e3Entry, or e4Entry. All other ends belong to the same Class.

```
inv SensorAssociation13:
  memberEnd->exists(m1 | m1.name->includes('entrySlPoint')) implies
    (let p:Property =
       memberEnd->select(m1 | m1.name->includes('entrySlPoint'))->first
     in
       memberEnd->excluding(p)->
         forAll(m3 | m3.name->includes('e1Entry') or
                     m3.name->includes('e2Entry') or
                     m3.name->includes('e3Entry') or
                     m3.name->includes('e4Entry')) and
       memberEnd->excluding(p)->
         forAll(m3,m4 | m3.owningClass = m4.owningClass)
    )
```

- If there is an end entryDbSlPoint, the other ends are named e1Entry, e2Entry, e3Entry, or e4Entry. All other ends belong to the same Class.

```
inv SensorAssociation14:
  memberEnd->exists(m1 | m1.name->includes('entryDbSlPoint')) implies
    (let p:Property =
       memberEnd->select(m1 | m1.name->includes('entryDbSlPoint'))->first
     in
       memberEnd->excluding(p)->
         forAll(m3 | m3.name->includes('e1Entry') or
                     m3.name->includes('e2Entry') or
                     m3.name->includes('e3Entry') or
                     m3.name->includes('e4Entry')) and
       memberEnd->excluding(p)->
         forAll(m3,m4 | m3.owningClass = m4.owningClass)
    )
```

- If there is an end exitSeg, the other ends are named e1Exit or e2Exit. All other ends belong to the same Class.

```
inv SensorAssociation15:
  memberEnd->exists(m1 | m1.name->includes('exitSeg')) implies
    (let p:Property =
       memberEnd->select(m1 | m1.name->includes('exitSeg'))->first
     in
       memberEnd->excluding(p)->
         forAll(m3 | m3.name->includes('e1Exit') or
```

```
                    m3.name->includes('e2Exit')) and
      memberEnd->excluding(p)->
        forAll(m3,m4 | m3.owningClass = m4.owningClass)
    )
```

- If there is an end exitCross, the other ends are named e1Exit, e2Exit, e3Exit, or e4Exit. All other ends belong to the same Class.

```
inv SensorAssociation16:
  memberEnd->exists(m1 | m1.name->includes('exitCross')) implies
    (let p:Property =
       memberEnd->select(m1 | m1.name->includes('exitCross'))->first
     in
       memberEnd->excluding(p)->
         forAll(m3 | m3.name->includes('e1Exit') or
                     m3.name->includes('e2Exit') or
                     m3.name->includes('e3Exit') or
                     m3.name->includes('e4Exit')) and
       memberEnd->excluding(p)->
         forAll(m3,m4 | m3.owningClass = m4.owningClass)
    )
```

- If there is an end exitPoint, the other ends are named e1Exit, e2Exit, or e3Exit. All other ends belong to the same Class.

```
inv SensorAssociation17:
  memberEnd->exists(m1 | m1.name->includes('exitPoint')) implies
    (let p:Property =
       memberEnd->select(m1 | m1.name->includes('exitPoint'))->first
     in
       memberEnd->excluding(p)->
         forAll(m3 | m3.name->includes('e1Exit') or
                     m3.name->includes('e2Exit') or
                     m3.name->includes('e3Exit')) and
       memberEnd->excluding(p)->
         forAll(m3,m4 | m3.owningClass = m4.owningClass)
    )
```

- If there is an end exitSlPoint, the other ends are named e1Exit, e2Exit, e3Exit, or e4Exit. All other ends belong to the same Class.

```
inv SensorAssociation18:
  memberEnd->exists(m1 | m1.name->includes('exitSlPoint')) implies
    (let p:Property =
       memberEnd->select(m1 | m1.name->includes('exitSlPoint'))->first
     in
       memberEnd->excluding(p)->
         forAll(m3 | m3.name->includes('e1Exit') or
                     m3.name->includes('e2Exit') or
                     m3.name->includes('e3Exit') or
                    m3.name->includes('e4Exit')) and
       memberEnd->excluding(p)->
         forAll(m3,m4 | m3.owningClass = m4.owningClass)
    )
```

- If there is an end exitDbSlPoint, the other ends are named e1Exit, e2Exit, e3Exit, or e4Exit. All other ends belong to the same Class.

```
inv SensorAssociation19:
```

```
memberEnd->exists(m1 | m1.name->includes('exitDbSlPoint')) implies
  (let p:Property =
     memberEnd->select(m1 | m1.name->includes('exitDbSlPoint'))->first
   in
     memberEnd->excluding(p)->
       forAll(m3 | m3.name->includes('e1Exit') or
                   m3.name->includes('e2Exit') or
                   m3.name->includes('e3Exit') or
                  m3.name->includes('e4Exit')) and
     memberEnd->excluding(p)->
       forAll(m3,m4 | m3.owningClass = m4.owningClass)
```

Semantics

SensorAssociations connect track elements and sensors, respectively properties of them. They are always binary. Exit properties of sensors are always connected to exit properties of points and segments. Vice versa, entry properties of sensors are connected to entry properties of segments.

SensorAssociation is used to create a railway network by associating its different parts.

Notation

SensorAssociations are used in class diagrams using the UML association notation. For instances, called *SensorLink*, see the respective paragraph.

2.3.3 SignalAssociation

Description

A SignalAssociation connects one signals to one sensor. Passing trains must obey signals associated to exit sensors of the current track segment in their driving direction.

Associations

None.

Attributes

None.

Constraints

- There are 2 ends.

```
inv SignalAssociation1:
  memberEnd->size() = 2
```

- There are 2 end types.

```
inv SignalAssociation2:
  endType->size() = 2
```

- One end type is Sensor.

```
inv SignalAssociation3:
  endType->one(t | t.oclIsKindOf(Sensor))
```

- One end type is Signal.

```
inv SignalAssociation4:
  endType->one(t | t.oclIsKindOf(Signal))
```

Semantics

SignalAssociation connects a signal to a sensor.

Notation

SignalAssociations are used in class diagrams using the UML association notation. For instances, called *SignalLink*, see the respective paragraph.

2.4 Instances

Each of the track network elements has its specific instance stereotype as this is needed, e.g. to add additional notation. This makes it possible to use (a) usual UML object symbols for these instances or (b) use the specific railway domain notation. The same holds for the three kinds of association specified above.

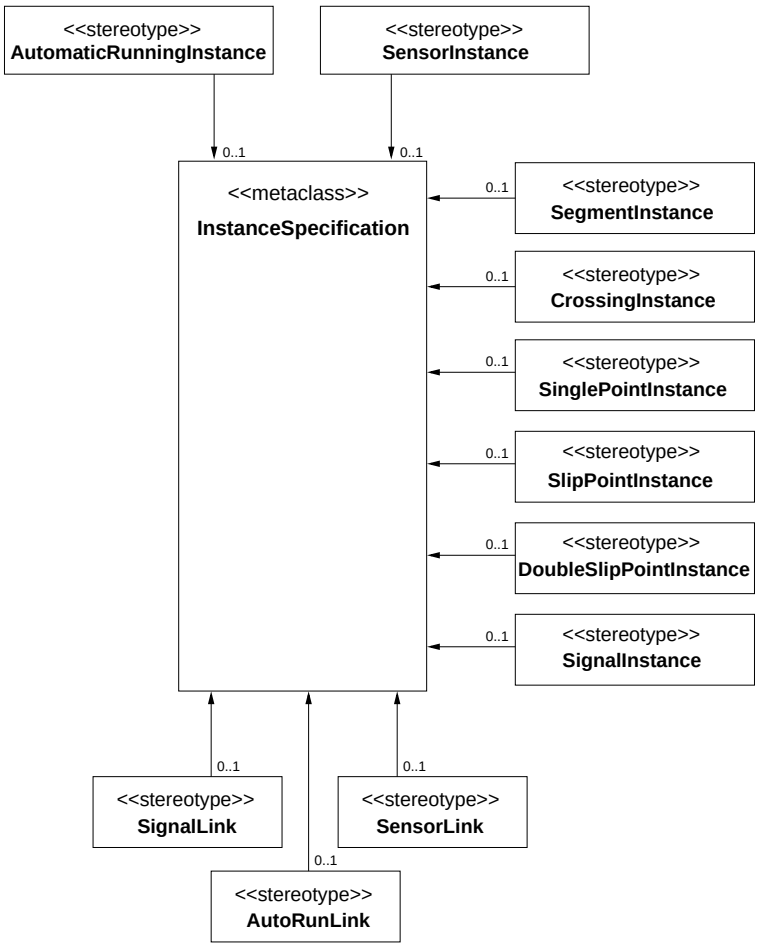

Figure 2.11: Stereotypes for track network element instances

2.4.1 AutomaticRunningInstance

Description

An AutomaticRunningInstance is the instance of an automatic train running.

Associations

None.

Attributes

None.

Constraints

- There is one classifier instantiated.

```
inv AutomaticRunningInstance1:
  classifier->size() = 1
```

- The instantiated classifier is a kind of Class.

```
inv AutomaticRunningInstance2:
  classifier->one(oclIsKindOf(Class))
```

- The instantiated classifier has the type AutomaticRunning.

```
inv AutomaticRunningInstance3:
  classifier->one(oclIsTypeOf(AutomaticRunning))
```

- There is a mandatory slot autoRunId. The value is given by a LiteralAutoRunId.

```
inv AutomaticRunningInstance4:
  slot->one(s1 | s1.definingFeature.name->includes('autoRunId') and
                 s1.value->size()= 1 and
                 s1.value->first.oclIsTypeOf(LiteralAutoRunId))
```

- There is an optional slot requestedState. If present, the value is taken from the enumeration ActivationKind.

```
inv AutomaticRunningInstance5:
  slot->select(s1 | s1.definingFeature.name->includes('requestedState'))->
    forAll(s2 |  s2.value->size()= 1 and
                 s2.value->first().oclAsType(InstanceValue).
                   instance.oclIsTypeOf(EnumerationLiteral) and
                 s2.value->first().oclAsType(InstanceValue).
                   instance.oclAsType(EnumerationLiteral).enumeration.name->
                     includes('ActivationKind'))
```

- There is a mandatory slot actualState. The value is taken from the enumeration AutoRunKind.

```
inv AutomaticRunningInstance6:
  slot->one(s1 | s1.definingFeature.name->includes('actualState') and
                 s1.value->size()= 1 and
                 s1.value->first().oclAsType(InstanceValue).
                   instance.oclIsTypeOf(EnumerationLiteral) and
                 s1.value->first().oclAsType(InstanceValue).
                   instance.oclAsType(EnumerationLiteral).enumeration.name->
                     includes('AutoRunKind'))
```

- There is an optional slot requestTime. If present, the value is given by a LiteralTimeInstant.

```
inv AutomaticRunningInstance7:
  slot->select(s1 | s1.definingFeature.name->includes('requestTime'))->
    forAll(s2 | s2.value->size()= 1 and
                s2.value->first.oclIsTypeOf(LiteralTimeInstant))
```

- There is an optional slot delta_a. If present, the value is given by a LiteralDuration.

```
inv AutomaticRunningInstance8:
  slot->select(s1 | s1.definingFeature.name->includes('delta_a'))->
    forAll(s2 | s2.value->size()= 1 and
                s2.value->first.oclIsTypeOf(LiteralDuration))
```

- There is an optional slot sensor. If present, the slot is the end of an AutoRunLink.

```
inv AutomaticRunningInstance9:
  (slot->select(s1 | s1.definingFeature.name->includes('sensor') and
                     s1.value->size()= 1 and
                     s1.value->first().oclIsTypeOf(InstanceValue) and
                     s1.value->first().oclAsType(InstanceValue).instance.
                     oclIsTypeOf(AutoRunLink))->size() = 1) or
  (not slot->exists(s1 | s1.definingFeature.name->includes('sensor')))
```

- If the slot requestedState is present, there have to be slots for requestTime and delta_a.

```
inv AutomaticRunningInstance10:
  self.classifier->asSequence->first.oclAsType(Class).ownedAttribute->
    one(p | p.name->includes('requestedState') and
            p.lower = Set{1}) implies
  slot->select(s1 | (s1.definingFeature.name->includes('requestedState') or
                     s1.definingFeature.name->includes('requestTime') or
                     s1.definingFeature.name->includes('delta_a')) and
                     s1.value->size() = 1)->size()=3
```

- The slots requestedState, requestTime, and delta_a are either all present or none of them.

```
inv AutomaticRunningInstance11:
  slot->select(s1 | (s1.definingFeature.name->includes('requestedState') or
                     s1.definingFeature.name->includes('requestTime') or
                     s1.definingFeature.name->includes('delta_a')) and
                     s1.value->size() = 1)->size()=0 or
  slot->select(s1 | (s1.definingFeature.name->includes('requestedState') or
                     s1.definingFeature.name->includes('requestTime') or
                     s1.definingFeature.name->includes('delta_a')) and
                     s1.value->size() = 1)->size()=3
```

- The slot autoRunId has a different value for every instance of AutoRunInstance.

```
inv AutomaticRunningInstance12:
  AutomaticRunningInstance.allInstances->collect(slot)->asSet->flatten->
    select(s | s.definingFeature.name->includes('autoRunId'))->
      iterate(s:Slot;
              result:Set(LiteralAutoRunId) =
                oclEmpty(Set(LiteralAutoRunId)) |
              result->including(s.value->first.oclAsType(LiteralAutoRunId)))->
                isUnique(value)
```

Semantics

Given by the transformation from a concrete network to a labeled transition system.

Notation

An AutomaticRunningInstance is either depicted as a UML object or as a white or black box underneath the sensor to which it is associated in a track layout diagram. A white box denotes a permanent automatic train running system (see Fig. 2.12), a black box a controlled automatic train running system (see Fig. 2.13).

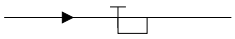

Figure 2.12: Permanent automatic train running instance

Figure 2.13: Controlled automatic train running instance

2.4.2 AutoRunLink

Description

An AutoRunLink is the instance of an automatic running association.

Associations

None.

Attributes

None.

Constraints

- There is one classifier instantiated.

```
inv AutoRunLink1:
  classifier->size() = 1
```

- The instantiated classifier is a kind of Association.

```
inv AutoRunLink2:
  classifier->one(oclIsKindOf(Association))
```

- The instantiated classifier has the type AutoRunAssociation.

```
inv AutoRunLink3:
  classifier->one(oclIsTypeOf(AutoRunAssociation))
```

- There are two slots.

```
inv AutoRunLink4:
  slot->size()=2
```

Semantics

Given by the transformation from a concrete network to a labeled transition system.

Notation

An AutoRunLink is either depicted as a UML link or implicitly in a track layout diagram. The placement of the automatic train running instance depends on the driving directions of the segments at which connecting sensor the automatic train running instance is placed.

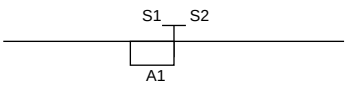

Figure 2.14: Automatic train running instance linked to sensor instance

In Fig. 2.14, *S2* is the exit sensor of the left track segment and the entry sensor of the right one. Vice versa, *S1* is the exit sensor of the right track segment and the entry sensor of the left one. The automatic train running *A1* is placed at sensor *S1*, i.e. braking is invoked if the train travels from the right to the left segment and exceeds the current speed limit at sensor *S1*. The current speed limit is either given by a signal placed at the same sensor or by a fixed speed limit of the entry segment of the sensor.

In Fig. 2.15, we can see the same constellation as an object diagram.

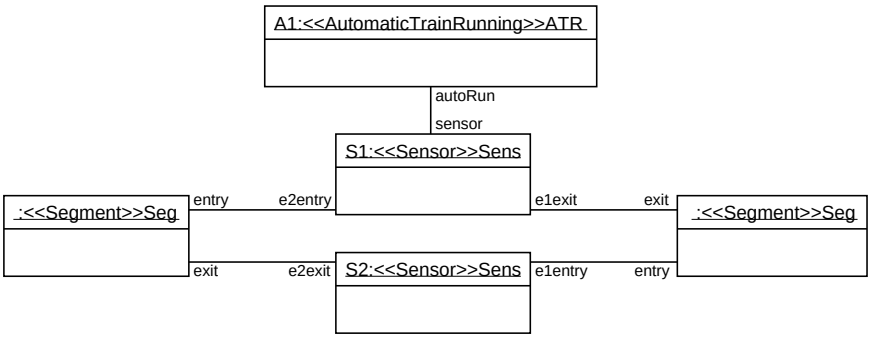

Figure 2.15: Automatic train running instance linked to sensor instance in object notation

2.4.3 CrossingInstance

Description

A CrossingInstance is the instance of a crossing.

Associations

None.

Attributes

None.

Constraints

- There is one classifier instantiated.

```
inv CrossingInstance1:
  classifier->size() = 1
```

- The instantiated classifier is a kind of Class.

```
inv CrossingInstance2:
  classifier->one(oclIsKindOf(Class))
```

- The instantiated classifier is a kind of TrackElement.

```
inv CrossingInstance3:
  classifier->one(oclIsKindOf(TrackElement))
```

- The instantiated classifier has the type Crossing.

```
inv CrossingInstance4:
  classifier->one(oclIsTypeOf(Crossing))
```

- There is a mandatory slot maxNumberOfTrains. The value is given by a LiteralInteger.

```
inv CrossingInstance5:
  slot->one(s1 | s1.definingFeature.name->includes('maxNumberOfTrains') and
                 s1.value->size()= 1 and
                 s1.value->first.oclIsTypeOf(LiteralInteger) and
                 s1.value->first()->oclAsType(LiteralInteger).value = 1)
```

- There is an optional slot limit. If present, the value is given by a LiteralInteger.

```
inv CrossingInstance6:
  slot->one(s1 | s1.definingFeature.name->includes('limit') and
                 s1.value->size()= 1 and
                 s1.value->first.oclIsTypeOf(LiteralInteger) and
                 s1.value->first()->oclAsType(LiteralInteger).value >= 0) or
  not slot->exists(s1 | s1.definingFeature.name->includes('limit'))
```

- The slots e1Entry, e3Entry, e2Exit, and e4Exit are mandatory. Each slot is an end of a SensorLink.

```
inv CrossingInstance7:
  slot->select(s1 | (s1.definingFeature.name->includes('e1Entry') or
                     s1.definingFeature.name->includes('e3Entry') or
                     s1.definingFeature.name->includes('e2Exit') or
                     s1.definingFeature.name->includes('e4Exit')) and
                     s1.value->size() = 1 and
                     s1.value->first().oclIsTypeOf(InstanceValue) and
                     s1.value->first().oclAsType(InstanceValue).instance.
                       oclIsTypeOf(SensorLink))->size = 4
```

- The slots e1Entry, e3Entry, e2Exit, and e4Exit either exist all or not of them. If present, each slot is an end of a SensorLink.

```
inv CrossingInstance8:
  (slot->select(s1 | (s1.definingFeature.name->includes('e2Entry') or
                     s1.definingFeature.name->includes('e4Entry') or
                     s1.definingFeature.name->includes('e1Exit') or
                     s1.definingFeature.name->includes('e3Exit')) and
                     s1.value->size() = 1 and
                     s1.value->first().oclIsTypeOf(InstanceValue) and
                     s1.value->first().oclAsType(InstanceValue).instance.
                       oclIsTypeOf(SensorLink))->size = 4) or
  (slot->select(s1 | (s1.definingFeature.name->includes('e2Entry') or
                     s1.definingFeature.name->includes('e4Entry') or
                     s1.definingFeature.name->includes('e1Exit') or
                     s1.definingFeature.name->includes('e3Exit')) and
                     s1.value->size() = 1 and
                     s1.value->first().oclIsTypeOf(InstanceValue) and
                     s1.value->first().oclAsType(InstanceValue).instance.
                       oclIsTypeOf(SensorLink))->size = 0)
```

- If the instantiated classifier has a mandatory attribute e2Entry, there have to be slots named e2entry and e1Exit. If present, each slot is an end of a SensorLink.

```
inv CrossingInstance9:
  self.classifier->asSequence->first.oclAsType(Class).ownedAttribute->
    one(p | p.name->includes('e2Entry') and
                p.lower = Set{1}) implies
  slot->select(s1 | (s1.definingFeature.name->includes('e2Entry') or
```

```
                    s1.definingFeature.name->includes('e1Exit')) and
                    s1.value->size() = 1 and
                    s1.value->first().oclIsTypeOf(InstanceValue) and
                    s1.value->first().oclAsType(InstanceValue).instance.
                      oclIsTypeOf(SensorLink))->size = 2
```

- If the instantiated classifier has a mandatory attribute e2Entry, there have to be slots named e2entry and e1Exit. If present, each slot is an end of a SensorLink.

```
inv CrossingInstance10:
  self.classifier->asSequence->first.oclAsType(Class).ownedAttribute->
    one(p | p.name->includes('e4Entry') and
                p.lower = Set{1}) implies
  slot->select(s1 | (s1.definingFeature.name->includes('e4Entry') or
                    s1.definingFeature.name->includes('e3Exit')) and
                    s1.value->size() = 1 and
                    s1.value->first().oclIsTypeOf(InstanceValue) and
                    s1.value->first().oclAsType(InstanceValue).instance.
                      oclIsTypeOf(SensorLink))->size = 2
```

Semantics

Given by the transformation from a concrete network to a labeled transition system.

Notation

A CrossingInstance is either depicted as a UML object or as a symbol as shown in Fig. 2.16 (crossing segments) or in Fig. 2.17. A speed limit can optionally be shown above the crossing just as for segment instances.

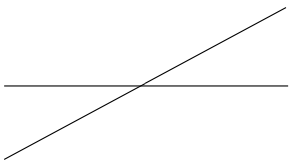

Figure 2.16: Crossing instance

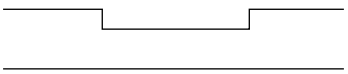

Figure 2.17: Interlaced crossing instance

2.4.4 DoubleSlipPointInstance

Description

A DoubleSlipPointInstance is the instance of a double slip point.

Associations

None.

Attributes

None.

Constraints

- There is one classifier instantiated.

```
inv DoubleSlipPointInstance1:
  classifier->size() = 1
```

- The instantiated classifier is a kind of Class.

```
inv DoubleSlipPointInstance2:
  classifier->one(oclIsKindOf(Class))
```

- The instantiated classifier is a kind of TrackElement.

```
inv DoubleSlipPointInstance3:
  classifier->one(oclIsKindOf(TrackElement))
```

- The instantiated classifier is a kind of Point.

```
inv DoubleSlipPointInstance4:
  classifier->one(oclIsKindOf(Point))
```

- The instantiated classifier is a kind of SlipPoint.

```
inv DoubleSlipPointInstance5:
  classifier->one(oclIsTypeOf(SlipPoint))
```

- The instantiated classifier has the type DoubleSlipPoint.

```
inv DoubleSlipPointInstance6:
  classifier->one(oclIsTypeOf(DoubleSlipPoint))
```

- There is a mandatory slot pointId. The value is given by a

```
 LiteralPointId.
inv DoubleSlipPointInstance7:
  slot->one(s1 | s1.definingFeature.name->includes('pointId') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralPointId))
```

- There is a mandatory slot pointIdOpp. The value is given by a LiteralPointId.

```
inv DoubleSlipPointInstance8:
  slot->one(s1 | s1.definingFeature.name->includes('pointIdOpp') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralPointId))
```

- There is a mandatory slot plus. The value is taken from the enumeration RouteKind.

```
inv DoubleSlipPointInstance9:
  slot->one(s1 | s1.definingFeature.name->includes('plus') and
                 s1.value->size()= 1 and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclIsTypeOf(EnumerationLiteral) and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclAsType(EnumerationLiteral).enumeration.name->
                     includes('RouteKind'))
```

- There is a mandatory slot minus. The value is taken from the enumeration RouteKind.

```
inv DoubleSlipPointInstance10:
  slot->one(s1 | s1.definingFeature.name->includes('minus') and
                 s1.value->size()= 1 and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclIsTypeOf(EnumerationLiteral) and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclAsType(EnumerationLiteral).enumeration.name->
                     includes('RouteKind'))
```

- There is a mandatory slot plusOpp. The value is taken from the enumeration RouteKind.

```
inv DoubleSlipPointInstance11:
  slot->one(s1 | s1.definingFeature.name->includes('plusOpp') and
                 s1.value->size()= 1 and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclIsTypeOf(EnumerationLiteral) and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclAsType(EnumerationLiteral).enumeration.name->
                     includes('RouteKind'))
```

- There is a mandatory slot minusOpp. The value is taken from the enumeration RouteKind.

```
inv DoubleSlipPointInstance12:
  slot->one(s1 | s1.definingFeature.name->includes('minusOpp') and
                 s1.value->size()= 1 and
                 s1.value->first().oclAsType(InstanceValue).instance.
                    oclIsTypeOf(EnumerationLiteral) and
                 s1.value->first().oclAsType(InstanceValue).instance.
                    oclAsType(EnumerationLiteral).enumeration.name->
                      includes('RouteKind'))
```

- There is a mandatory slot maxNumberOfTrains. The value is given by a LiteralInteger.

```
inv DoubleSlipPointInstance13:
  slot->one(s1 | s1.definingFeature.name->includes('maxNumberOfTrains') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralInteger) and
                 s1.value->first().oclAsType(LiteralInteger).value = 1)
```

- There is a mandatory slot delta_p. The value is given by a LiteralDuration.

```
inv DoubleSlipPointInstance14:
  slot->one(s1 | s1.definingFeature.name->includes('delta_p') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralDuration))
```

- There is a mandatory slot actualState. The value is taken from the enumeration PointStateKind.

```
inv DoubleSlipPointInstance15:
  slot->one(s1 | s1.definingFeature.name->includes('actualState') and
                 s1.value->size()= 1 and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclIsTypeOf(EnumerationLiteral) and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclAsType(EnumerationLiteral).enumeration.name->
                     includes('PointStateKind'))
```

- There is a mandatory slot requestedState. The value is taken from the enumeration RouteKind.

```
inv DoubleSlipPointInstance16:
  slot->one(s1 | s1.definingFeature.name->includes('requestedState') and
                 s1.value->size()= 1 and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclIsTypeOf(EnumerationLiteral) and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclAsType(EnumerationLiteral).enumeration.name->
                     includes('RouteKind'))
```

- There is a mandatory slot requestTime. The value is given by a LiteralTimeInstant.

```
inv DoubleSlipPointInstance17:
  slot->one(s1 | s1.definingFeature.name->includes('requestTime') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralTimeInstant))
```

- There is a mandatory slot actualStateOpp. The value is taken from the enumeration PointStateKind.

```
inv DoubleSlipPointInstance18:
  slot->one(s1 | s1.definingFeature.name->includes('actualStateOpp') and
                 s1.value->size()= 1 and
                 s1.value->first().oclAsType(InstanceValue).instance.
                    oclIsTypeOf(EnumerationLiteral) and
                 s1.value->first().oclAsType(InstanceValue).instance.
                    oclAsType(EnumerationLiteral).enumeration.name->
                      includes('PointStateKind'))
```

- There is a mandatory slot requestedStateOpp. The value is taken from the enumeration RouteKind.

```
inv DoubleSlipPointInstance19:
  slot->one(s1 | s1.definingFeature.name->includes('requestedStateOpp') and
                 s1.value->size()= 1 and
                 s1.value->first().oclAsType(InstanceValue).instance.
                    oclIsTypeOf(EnumerationLiteral) and
                 s1.value->first().oclAsType(InstanceValue).instance.
                    oclAsType(EnumerationLiteral).enumeration.name->
                      includes('RouteKind'))
```

- There is a mandatory slot requestTimeOpp. The value is given by a LiteralTimeInstant.

```
inv DoubleSlipPointInstance20:
  slot->one(s1 | s1.definingFeature.name->includes('requestTimeOpp') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralTimeInstant))
```

- There is an optional slot requestTime. If present, the value is given by a LiteralInteger.

```
inv DoubleSlipPointInstance21:
  slot->select(s1 | s1.definingFeature.name->includes('limit'))->
    forAll(s2 | s2.value->size()= 1 and
                s2.value->first().oclIsTypeOf(LiteralInteger) and
                s2.value->first()->oclAsType(LiteralInteger).value >= 0)
```

- One of the slots plus and minus has the value STRAIGHT, the other one the value LEFT or RIGHT.

```
inv DoubleSlipPointInstance22:
  slot->select(s1 | s1.definingFeature.name->includes('minus') or
                    s1.definingFeature.name->includes('plus'))->
    one(s2 | s2.value->size()= 1 and
             s2.value->first().oclIsTypeOf(InstanceValue) and
             s2.value->first().oclAsType(InstanceValue).instance.name->
               includes('STRAIGHT')) and
  slot->select(s1 | s1.definingFeature.name->includes('minus') or
                    s1.definingFeature.name->includes('plus'))->
    one(s2 | s2.value->size()= 1 and
             s2.value->first().oclIsTypeOf(InstanceValue) and
             (s2.value->first().oclAsType(InstanceValue).instance.name->
               includes('LEFT') or
              s2.value->first()->oclAsType(InstanceValue).instance.name->
               includes('RIGHT')))
```

- One of the slots plusOpp and minusOpp has the value STRAIGHT, the other one the value LEFT or RIGHT.

```
inv DoubleSlipPointInstance23:
  slot->select(s1 | s1.definingFeature.name->includes('minusOpp') or
                    s1.definingFeature.name->includes('plusOpp'))->
    one(s2 | s2.value->size()= 1 and
             s2.value->first().oclIsTypeOf(InstanceValue) and
             s2.value->first().oclAsType(InstanceValue).instance.name->
               includes('STRAIGHT')) and
  slot->select(s1 | s1.definingFeature.name->includes('minusOpp') or
                    s1.definingFeature.name->includes('plusOpp'))->
    one(s2 | s2.value->size()= 1 and
             s2.value->first().oclIsTypeOf(InstanceValue) and
             (s2.value->first().oclAsType(InstanceValue).instance.name->
               includes('LEFT') or
              s2.value->first()->oclAsType(InstanceValue).instance.name->
               includes('RIGHT')))
```

- If one of the slots plus or minus has he value LEFT, then also one of the slots plusOpp or minusOpp has the value LEFT. Vice versa, if one of the slots plus or minus has the value RIGHT, then also one of the slots plusOpp or minusOpp has the value RIGHT.

```
inv DoubleSlipPointInstance24:
  slot->select(s1 | s1.definingFeature.name->includes('plus') or
                    s1.definingFeature.name->includes('minus'))->
    one(s2 | s2.value->first().oclAsType(InstanceValue).instance.name->
               includes('LEFT')) implies
      slot->select(s1 | s1.definingFeature.name->includes('plusOpp') or
                        s1.definingFeature.name->includes('minusOpp'))->
        one(s2 | s2.value->first().oclAsType(InstanceValue).instance.name->
          includes('LEFT')) and
  slot->select(s1 | s1.definingFeature.name->includes('plus') or
                    s1.definingFeature.name->includes('minus'))->
    one(s2 | s2.value->first().oclAsType(InstanceValue).instance.name->
               includes('RIGHT')) implies
      slot->select(s1 | s1.definingFeature.name->includes('plusOpp') or
                        s1.definingFeature.name->includes('minusOpp'))->
        one(s2 | s2.value->first().oclAsType(InstanceValue).instance.name->
          includes('RIGHT'))
```

- The slots e1Entry, e2Exit, e4Exit exist either all or none of them. If present, each slot is the end of a SensorLink.

```
inv DoubleSlipPointInstance25:
  (slot->select(s1 | (s1.definingFeature.name->includes('e1Entry') or
                      s1.definingFeature.name->includes('e2Exit') or
                      s1.definingFeature.name->includes('e4Exit')) and
                     s1.value->size()= 1 and
                     s1.value->first().oclIsTypeOf(InstanceValue) and
                     s1.value->first().oclAsType(InstanceValue).instance.
                     oclIsTypeOf(SensorLink))->size() = 3) or
   (slot->select(s1 | (s1.definingFeature.name->includes('e1Entry') or
                      s1.definingFeature.name->includes('e2Exit') or
                      s1.definingFeature.name->includes('e4Exit')) and
                     s1.value->size()= 1 and
                     s1.value->first().oclIsTypeOf(InstanceValue) and
                     s1.value->first().oclAsType(InstanceValue).instance.
                     oclIsTypeOf(SensorLink))->size() = 0)
```

- The slots e1Exit, e2Entry, e4Entry exist either all or none of them. If present, each slot is the end of a SensorLink.

```
inv DoubleSlipPointInstance26:
  (slot->select(s1 | (s1.definingFeature.name->includes('e1Exit') or
                      s1.definingFeature.name->includes('e2Entry') or
                      s1.definingFeature.name->includes('e4Entry')) and
                     s1.value->size()= 1 and
                     s1.value->first().oclIsTypeOf(InstanceValue) and
                     s1.value->first().oclAsType(InstanceValue).instance.
                       oclIsTypeOf(SensorLink))->size() = 3) or
  (slot->select(s1 | (s1.definingFeature.name->includes('e1Exit') or
                      s1.definingFeature.name->includes('e2Entry') or
                      s1.definingFeature.name->includes('e4Entry')) and
                      s1.value->size()= 1 and
                    s1.value->first().oclIsTypeOf(InstanceValue) and
                    s1.value->first().oclAsType(InstanceValue).instance.
                    oclIsTypeOf(SensorLink))->size() = 0)
```

- The slots e3Entry, e2Exit, e4Exit exist either all or none of them. If present, each slot is the end of a SensorLink.

```
inv DoubleSlipPointInstance27:
  (slot->select(s1 | (s1.definingFeature.name->includes('e3Entry') or
                      s1.definingFeature.name->includes('e2Exit') or
                      s1.definingFeature.name->includes('e4Exit')) and
                     s1.value->size()= 1 and
                     s1.value->first().oclIsTypeOf(InstanceValue) and
                     s1.value->first().oclAsType(InstanceValue).instance.
                     oclIsTypeOf(SensorLink))->size() = 3) or
   (slot->select(s1 | (s1.definingFeature.name->includes('e3Entry') or
                      s1.definingFeature.name->includes('e2Exit') or
                      s1.definingFeature.name->includes('e4Exit')) and
                     s1.value->size()= 1 and
                     s1.value->first().oclIsTypeOf(InstanceValue) and
                     s1.value->first().oclAsType(InstanceValue).instance.
                     oclIsTypeOf(SensorLink))->size() = 0)
```

- The slots e3Exit, e2Entry, e4Entry exist either all or none of them. If present, each slot is the end of a SensorLink.

```
inv DoubleSlipPointInstance28:
  (slot->select(s1 | (s1.definingFeature.name->includes('e3Exit') or
                      s1.definingFeature.name->includes('e2Entry') or
                      s1.definingFeature.name->includes('e4Entry')) and
                     s1.value->size()= 1 and
                     s1.value->first().oclIsTypeOf(InstanceValue) and
                     s1.value->first().oclAsType(InstanceValue).instance.
                       oclIsTypeOf(SensorLink))->size() = 3) or
  (slot->select(s1 | (s1.definingFeature.name->includes('e3Exit') or
                      s1.definingFeature.name->includes('e2Entry') or
                      s1.definingFeature.name->includes('e4Entry')) and
                      s1.value->size()= 1 and
                    s1.value->first().oclIsTypeOf(InstanceValue) and
                    s1.value->first().oclAsType(InstanceValue).instance.
                    oclIsTypeOf(SensorLink))->size() = 0)
```

- There are at least 4 slots named e1Entry, e1Exit, e2Entry, e2Exit, e3Entry, e3Exit, e4Entry, or e4Exit. If present, each slot is the end of a SensorLink.

```
inv DoubleSlipPointInstance29:
  slot->select(s1 | (s1.definingFeature.name->includes('e1Entry') or
                     s1.definingFeature.name->includes('e1Exit') or
                     s1.definingFeature.name->includes('e2Entry') or
                     s1.definingFeature.name->includes('e2Exit') or
                     s1.definingFeature.name->includes('e3Entry') or
                     s1.definingFeature.name->includes('e3Exit') or
                     s1.definingFeature.name->includes('e4Entry') or
                     s1.definingFeature.name->includes('e4Exit')) and
                    s1.value->size()= 1 and
                    s1.value->first().oclIsTypeOf(InstanceValue) and
                    s1.value->first().oclAsType(InstanceValue).instance.
                       oclIsTypeOf(SensorLink))->size() >= 4
```

- The slot pointId has a different value for every instance of SinglePointInstance, SlipPointInstance, and DoubleSlipPointInstance.

```
inv DoubleSlipPointInstance30:
  DoubleSlipPointInstance.allInstances->collect(slot)->asSet->flatten->
    select(s | s.definingFeature.name->includes('pointId') or
               s.definingFeature.name->includes('pointIdOpp'))->
      union(SinglePointInstance.allInstances->collect(slot)->asSet->flatten->
        select(s | s.definingFeature.name->includes('pointId'))
      )->
      union(SlipPointInstance.allInstances->collect(slot)->asSet->flatten->
        select(s | s.definingFeature.name->includes('pointId'))
      )->
      iterate(s:Slot;
              result:Set(LiteralPointId) =
                oclEmpty(Set(LiteralPointId)) |
              result->including(s.value->first.oclAsType(LiteralPointId)))->
                isUnique(value)
```

Semantics

Given by the transformation from a concrete network to a labeled transition system.

Notation

A DoubleSlipPointInstance is either depicted as a UML object or as a symbol as shown in Fig. 2.18. At least one of the *plus* and *minus* positions of each point has to be marked.

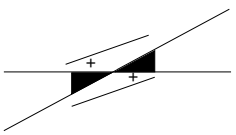

Figure 2.18: Double slip point instance

2.4.5 SegmentInstance

Description

A SegmentInstance is the instance of a segment.

Associations

None.

Attributes

None.

Constraints

- There is one classifier instantiated.

```
inv SegmentInstance1:
  classifier->size() = 1
```

- The instantiated classifier is a kind of Class.

```
inv SegmentInstance2:
  classifier->one(oclIsKindOf(Class))
```

- The instantiated classifier is a kind of TrackElement.

```
inv SegmentInstance3:
  classifier->one(oclIsKindOf(TrackElement))
```

- The instantiated classifier has the type Segment.

```
inv SegmentInstance4:
  classifier->one(oclIsTypeOf(Segment))
```

- There is a mandatory slot maxNumberOfTrains. The value is given by a LiteralInteger.

```
inv SegmentInstance5:
  slot->one(s1 | s1.definingFeature.name->includes('maxNumberOfTrains') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralInteger) and
                 s1.value->first()->oclAsType(LiteralInteger).value = 1)
```

- There is an optional slot limit. If present, the value is given by a LiteralInteger.

```
inv SegmentInstance6:
  slot->one(s1 | s1.definingFeature.name->includes('limit') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralInteger) and
                 s1.value->first()->oclAsType(LiteralInteger).value >= 0) or
  not slot->exists(s1 | s1.definingFeature.name->includes('limit'))
```

- At least 1 and at most 2 of the slots e1Entry and e2Exit exist. If present, each slot is the end of a SensorLink.

```
inv SegmentInstance7:
  slot->select(s1 | (s1.definingFeature.name->includes('e1Entry') or
                     s1.definingFeature.name->includes('e2Exit')) and
                    s1.value->size()= 1 and
                    s1.value->first().oclIsTypeOf(InstanceValue) and
                    s1.value->first().oclAsType(InstanceValue).instance.
                    oclIsTypeOf(SensorLink))->size()=1 or
  slot->select(s1 | (s1.definingFeature.name->includes('e1Entry') or
                     s1.definingFeature.name->includes('e2Exit')) and
                    s1.value->size()= 1 and
                    s1.value->first().oclIsTypeOf(InstanceValue) and
                    s1.value->first().oclAsType(InstanceValue).instance.
                    oclIsTypeOf(SensorLink))->size()=2
```

- At most 2 of the slots e2Entry and e1Exit exist. If present, each slot is the end of a SensorLink.

```
inv SegmentInstance8:
  slot->select(s1 | (s1.definingFeature.name->includes('e2Entry') or
                     s1.definingFeature.name->includes('e1Exit')) and
                    s1.value->size()= 1 and
                    s1.value->first().oclIsTypeOf(InstanceValue) and
                    s1.value->first().oclAsType(InstanceValue).instance.
                    oclIsTypeOf(SensorLink))->size()<=1 or
  slot->select(s1 | (s1.definingFeature.name->includes('e2Entry') or
                     s1.definingFeature.name->includes('e1Exit')) and
                    s1.value->size()= 1 and
                    s1.value->first().oclIsTypeOf(InstanceValue) and
                    s1.value->first().oclAsType(InstanceValue).instance.
                    oclIsTypeOf(SensorLink))->size()=2
```

- The number of the slots named e1Entry, e1Exit, e2Entry, or e2Exit is not 3. If present, each slot is the end of a SensorLink.

```
inv SegmentInstance9:
  slot->select(s1 | (s1.definingFeature.name->includes('e1Entry') or
                     s1.definingFeature.name->includes('e1Exit') or
                     s1.definingFeature.name->includes('e2Entry') or
                     s1.definingFeature.name->includes('e2Exit')) and
                    s1.value->size()= 1 and
                    s1.value->first().oclIsTypeOf(InstanceValue) and
                    s1.value->first().oclAsType(InstanceValue).instance.
                    oclIsTypeOf(SensorLink))->size() <> 3
```

Semantics

Given by the transformation from a concrete network to a labeled transition system.

Notation

A SegmentInstance is either depicted as a UML object or as a symbol as shown in Fig. 2.19. Note that it is possible that two segments cross in a track layout diagram but do not form a crossing e.g. if one segment is located on a bridge. In this case, one of the segments is drawn interrupted as shown in Fig. 2.20. Optionally, the current speed limit can be shown above the segment. If no speed limit is shown, there is no limit. The maximal number of trains is shown under the segment and marked with *max=*. If it is not shown, the default value is *1*.

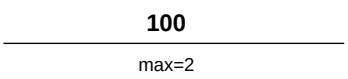

Figure 2.19: Segment instance

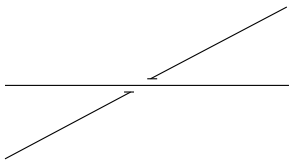

Figure 2.20: Two segment instances that do not cross

- Sink:

```
slot->one(s1 | s1.definingFeature.name->includes('e2Entry')) and
slot->select( s2 | s2.definingFeature.name->includes('e1Entry') or
                   s2.definignFeature.name->includes('e1Exit') or
                   s2.definingFeature.name->includes('e2Exit'))->size() = 0
```

- Source:

```
slot->one(s1 | s1.definingFeature.name->includes('e1Entry')) and
slot->select( s2 | s2.definingFeature.name->includes('e2Entry') or
                   s2.definignFeature.name->includes('e1Exit') or
                   s2.definingFeature.name->includes('e2Exit'))->size() = 0
```

- Sink/source:

```
(slot->one(s1 | s1.definingFeature.name->includes('e1Entry')) and
 slot->one(s2 | s2.definingFeature.name->includes('e1Exit')) and
 slot->select( s2 | s2.definingFeature.name->includes('e2Entry') or
                   s2.definingFeature.name->includes('e2Exit'))->size() = 0
)
xor
(slot->one(s1 | s1.definingFeature.name->includes('e2Entry')) and
 slot->one(s2 | s2.definingFeature.name->includes('e2Exit')) and
 slot->select(s3 | s3.definingFeature.name->includes('e1Entry') or
                   s3.definingFeature.name->includes('e1Exit'))->size() = 0
)
```

- Unidirectional:

```
(slot->one(s1 | s1.definingFeature.name->includes('e1Entry')) and
 slot->one(a2 | s2.definingFeature.name->includes('e2Exit')) and
 slot->select( s2 | s2.definingFeature.name->includes('e2Entry') or
                   s2.definignFeature.name->includes('e1Exit'))->size() = 0
)
xor
(slot->one(s1 | s1.definingFeature.name->includes('e2Entry')) and
 slot->one(a2 | s2.definingFeature.name->includes('e1Exit')) and
 slot->select( s2 | s2.definingFeature.name->includes('e1Entry') or
                   s2.definignFeature.name->includes('e2Exit'))->size() = 0
)
```

- Bidirectional:

```
slot->one(s1 | s1.definingFeature.name->includes('e1Entry')) and
slot->one(s1 | s1.definingFeature.name->includes('e2Entry')) and
slot->one(s1 | s1.definingFeature.name->includes('e1Exit')) and
slot->one(s1 | s1.definingFeature.name->includes('e2Exit'))
```

2.4.6 SensorInstance

Description

A SensorInstance is the instance of a sensor.

Associations

None.

Attributes

None.

Constraints

- There is one classifier instantiated.

```
inv SensorInstance1:
  classifier->size() = 1
```

- The instantiated classifier is a kind of Class.

```
inv SensorInstance2:
  classifier->one(oclIsKindOf(Class))
```

- The instantiated classifier has the type Sensor.

```
inv SensorInstance3:
  classifier->one(oclIsTypeOf(Sensor))
```

- There is a mandatory slot sensorId. The value is given by a LiteralSensorId.

```
inv SensorInstance4:
  slot->one(s1 | s1.definingFeature.name->includes('sensorId') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralSensorId))
```

- There is a mandatory slot actualState. The value is taken from the enumeration SensorStateKind.

```
inv SensorInstance5:
  slot->one(s1 | s1.definingFeature.name->includes('actualState') and
                 s1.value->size()= 1 and
                 s1.value->first().oclAsType(InstanceValue).
                   instance.oclIsTypeOf(EnumerationLiteral) and
                 s1.value->first().oclAsType(InstanceValue).
                   instance.oclAsType(EnumerationLiteral).enumeration.name->
                     includes('SensorStateKind'))
```

- There is a mandatory slot sentTime. The value is given by a LiteralTimeInstant.

```
inv SensorInstance6:
  slot->one(s1 | s1.definingFeature.name->includes('sentTime') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralTimeInstant))
```

- There is a mandatory slot counter. The value is given by a LiteralInteger.

```
inv SensorInstance7:
  slot->one(s1 | s1.definingFeature.name->includes('counter') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralInteger) and
                 s1.value->first().oclAsType(LiteralInteger).value >= 0)
```

- There is a mandatory slot delta_t. The value is given by a LiteralDuration.

```
inv SensorInstance8:
  slot->one(s1 | s1.definingFeature.name->includes('delta_t') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralDuration))
```

- There is a mandatory slot delta_tram. The value is given by a LiteralDuration.

```
inv SensorInstance9:
  slot->one(s1 | s1.definingFeature.name->includes('delta_tram') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralDuration))
```

- The number of slots named entrySeg, entryCross, entryPoint, entrySlPoint, or entryDbSlPoint is 1. The slot has a defined value.

```
inv SensorInstance10:
  slot->select(s1 | s1.definingFeature.name->includes('entrySeg') or
                    s1.definingFeature.name->includes('entryCross') or
                    s1.definingFeature.name->includes('entryPoint') or
                    s1.definingFeature.name->includes('entrySlPoint') or
                    s1.definingFeature.name->includes('entryDbSlPoint'))->
    size()=1 and
  slot->select(s1 | s1.definingFeature.name->includes('entrySeg') or
                    s1.definingFeature.name->includes('entryCross') or
                    s1.definingFeature.name->includes('entryPoint') or
                    s1.definingFeature.name->includes('entrySlPoint') or
                    s1.definingFeature.name->includes('entryDbSlPoint'))->
    asSequence->first().value->first().oclAsType(InstanceValue).
      instance.isDefined
```

- The number of slots named exitSeg, exitCross, exitPoint, exitSlPoint, or exitDbSlPoint is 1. The slot has a defined value.

```
inv SensorInstance11:
  slot->select(s1 | s1.definingFeature.name->includes('exitSeg') or
                    s1.definingFeature.name->includes('exitCross') or
                    s1.definingFeature.name->includes('exitPoint') or
                    s1.definingFeature.name->includes('exitSlPoint') or
                    s1.definingFeature.name->includes('exitDbSlPoint'))->
    size()=1 and
  slot->select(s1 | s1.definingFeature.name->includes('exitSeg') or
                    s1.definingFeature.name->includes('exitCross') or
                    s1.definingFeature.name->includes('exitPoint') or
                    s1.definingFeature.name->includes('exitSlPoint') or
                    s1.definingFeature.name->includes('exitDbSlPoint'))->
    asSequence->first().value->first().oclAsType(InstanceValue).
      instance.isDefined
```

- The values of the slots entrySeg, entryCross, entryPoint, entrySlPoint, entryDbSlPoint, exitSeg, exitCross, exitPoint, exitSlPoint, and exitDbSlPoint are distint, i.e. they refer to different instances.

```
inv SensorInstance12:
  slot->select(s1 | s1.definingFeature.name->includes('entrySeg') or
                    s1.definingFeature.name->includes('entryCross') or
                    s1.definingFeature.name->includes('entryPoint') or
                    s1.definingFeature.name->includes('entrySlPoint') or
                    s1.definingFeature.name->includes('entryDbSlPoint') or
                    s1.definingFeature.name->includes('exitSeg') or
                    s1.definingFeature.name->includes('exitCross') or
                    s1.definingFeature.name->includes('exitPoint') or
                    s1.definingFeature.name->includes('exitSlPoint') or
                    s1.definingFeature.name->includes('exitDbSlPoint'))->
    forAll(s1,s2 | s1 <> s2 implies s1.value <> s2.value)
```

- There is an optional slot signal. If present, the slot is the end of an SignalLink.

```
inv SensorInstance13:
  (slot->select(s1 | s1.definingFeature.name->includes('signal') and
                     s1.value->size()= 1 and
                     s1.value->first().oclIsTypeOf(InstanceValue) and
                     s1.value->first().oclAsType(InstanceValue).instance.
                     oclIsTypeOf(SignalLink))->size() = 1) or
  (not slot->exists(s1 | s1.definingFeature.name->includes('signal')))
```

- There is an optional slot autoRun. If present, the slot is the end of an AutoRunLink.

```
inv SensorInstance14:
  (slot->select(s1 | s1.definingFeature.name->includes('autoRun') and
                     s1.value->size()= 1 and
                     s1.value->first().oclIsTypeOf(InstanceValue) and
                     s1.value->first().oclAsType(InstanceValue).instance.
                     oclIsTypeOf(AutoRunLink))->size() = 1) or
  (not slot->exists(s1 | s1.definingFeature.name->includes('autoRun')))
```

- The slot sensorId has a different value for every instance of SensorInstance.

```
inv SensorInstance15:
  SensorInstance.allInstances->collect(slot)->asSet->flatten->
    select(s | s.definingFeature.name->includes('sensorId'))->
      iterate(s:Slot;
              result:Set(LiteralSensorId) =
                oclEmpty(Set(LiteralSensorId)) |
              result->including(s.value->first.oclAsType(LiteralSensorId)))->
                isUnique(value)
```

- If the slot entrySeg, entryCross, entryPoint, entrySlPoint, or entryDbSlPoint of one instance of SensorInstance has the same value as the slot exitSeg, exitCross, exitPoint, exitSlPoint, or exitDbSlPoint of another instance of SensorInstance, then the value of the slot exitSeg, exitCross, exitPoint, exitSlPoint, or exitDbSlipPoint of the first instance has to be the same as the value of the slot entrySeg, entryCross, entryPoint, entrySlPoint, or entryDbSlPoint of the second instance.

```
inv SensorInstance16:
  SensorInstance.allInstances->forAll(i1,i2 |
    (i1.slot->select(s | s.definingFeature.name->includes('entrySeg') or
                         s.definingFeature.name->includes('entryCross') or
                         s.definingFeature.name->includes('entryPoint') or
                         s.definingFeature.name->includes('entrySlPoint') or
                         s.definingFeature.name->includes('entryDbSlPOint'))->
      asSequence->first().value =
     i2.slot->select(s | s.definingFeature.name->includes('exitSeg') or
                         s.definingFeature.name->includes('exitCross') or
                         s.definingFeature.name->includes('exitPoint') or
                         s.definingFeature.name->includes('exitSlPoint') or
                         s.definingFeature.name->includes('exitDbSlPoint'))->
      asSequence->first().value) implies
    (i1.slot->select(s | s.definingFeature.name->includes('exitSeg') or
                         s.definingFeature.name->includes('exitCross') or
                         s.definingFeature.name->includes('exitPoint') or
                         s.definingFeature.name->includes('exitSlPoint') or
                         s.definingFeature.name->includes('exitDbSlPoint'))->
      asSequence->first().value =
     i2.slot->select(s | s.definingFeature.name->includes('entrySeg') or
                         s.definingFeature.name->includes('entryCross') or
                         s.definingFeature.name->includes('entryPoint') or
                         s.definingFeature.name->includes('entrySlPoint') or
                         s.definingFeature.name->includes('entryDbSlPoint'))->
      asSequence->first().value))
```

Semantics

Given by the transformation from a concrete network to a labeled transition system.

Notation

A SensorInstance is either depicted as a UML object or as a symbol in a track layout diagram. Each sensor separates two segments. The alignment of bar on top of the separating line denotes the traveling direction: a) from the left segment to the right segment (see Fig. 2.21), b) from the right segment to the left segment (see Fig. 2.22), or c) bidirectionally (see Fig. 2.23). Optionally, black arrows on the segments can be used to make the traveling direction unambigiously clear.

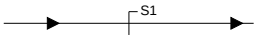

Figure 2.21: Sensor for traveling from the left to the right segment

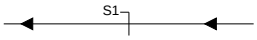

Figure 2.22: Sensor for traveling from the right to the left segment

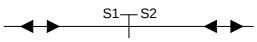

Figure 2.23: Sensor for bidirectional usage

2.4.7 SensorLink

Description

A SensorLink is the instance of a sensor association.

Associations

None.

Attributes

None.

Constraints

- There is one classifier instantiated.

```
inv SensorLink1:
  classifier->size() = 1
```

- The instantiated classifier is a kind of Association.

```
inv SensorLink2:
  classifier->one(oclIsKindOf(Association))
```

- The instantiated classifier has the type SensorAssociation.

```
inv SensorLink3:
  classifier->one(oclIsTypeOf(SensorAssociation))
```

- There are two slots.

```
inv SensorLink4:
  slot->size()=2
```

Semantics

Given by the transformation from a concrete network to a labeled transition system.

Notation

A SensorLink is either depicted as a UML link or implicitly in a track layout diagram. The figures 2.24, 2.25, and 2.26 show the explicit usage in a UML object diagram.

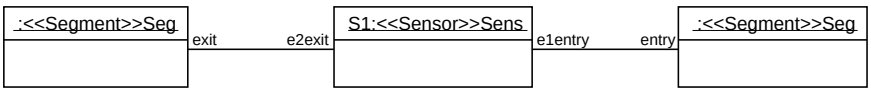

Figure 2.24: Sensor for traveling from the left to the right segment in object notation

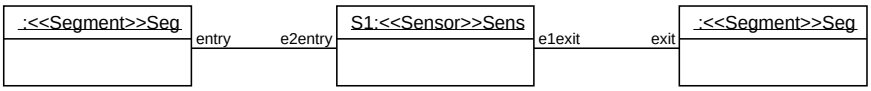

Figure 2.25: Sensor for traveling from the right to the left segment in object notation

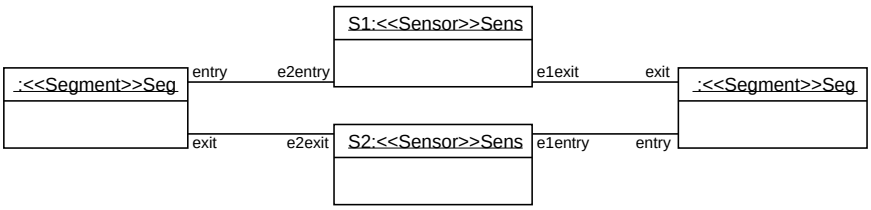

Figure 2.26: Sensor for bidirectional usage in object notation

2.4.8 SignalInstance

Description

A SignalInstance is the instance of a signal.

Associations

None.

Attributes

None.

Constraints

- There is one classifier instantiated.

```
inv SignalInstance1:
  classifier->size() = 1
```

- The instantiated classifier is a kind of Class.

```
inv SignalInstance2:
  classifier->one(oclIsKindOf(Class))
```

- The instantiated classifier has the type Signal.

```
inv SignalInstance3:
  classifier->one(oclIsTypeOf(Signal))
```

- There is a mandatory slot signalId. The value is given by a LiteralSignalId.

```
inv SignalInstance4:
  slot->one(s1 | s1.definingFeature.name->includes('signalId') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralSignalId))
```

- There is a mandatory slot actualState. The value is taken from the enumeration SignalStateKind.

```
inv SignalInstance5:
  slot->one(s1 | s1.definingFeature.name->includes('actualState') and
                 s1.value->size()= 1 and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclIsTypeOf(EnumerationLiteral) and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclAsType(EnumerationLiteral).enumeration.name->
                     includes('SignalStateKind'))
```

- There is a mandatory slot requestedState. The value is taken from the enumeration PermissionKind.

```
inv SignalInstance6:
  slot->one(s1 | s1.definingFeature.name->includes('requestedState') and
                 s1.value->size()= 1 and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclIsTypeOf(EnumerationLiteral) and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclAsType(EnumerationLiteral).enumeration.name->
                     includes('PermissionKind'))
```

- There is a mandatory slot requestTime. The value is given by a LiteralTimeInstant.

```
inv SignalInstance7:
  slot->one(s1 | s1.definingFeature.name->includes('requestTime') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralTimeInstant))
```

- There is a mandatory slot delta_s. The value is given by a LiteralDuration.

```
inv SignalInstance8:
  slot->one(s1 | s1.definingFeature.name->includes('delta_s') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralDuration))
```

- There is an optional slot direction. If present, the value is taken from the enumeration RouteKind.

```
inv SignalInstance9:
  slot->select(s1 | s1.definingFeature.name->includes('direction'))->
    forAll(s2 | s2.value->size()= 1 and
                s2.value->first().oclAsType(InstanceValue).instance.
                  oclIsTypeOf(EnumerationLiteral) and
                s2.value->first().oclAsType(InstanceValue).instance.
                  oclAsType(EnumerationLiteral).enumeration.name->
                    includes('RouteKind')) and
  slot->select(s1 | s1.definingFeature.name->includes('direction'))->size <= 1
```

- There is an optional slot requestedDir. If present, the value is taken from the enumeration RouteKind.

```
inv SignalInstance10:
  slot->select(s1 | s1.definingFeature.name->includes('requestedDir'))->
    forAll(s2 | s2.value->size()= 1 and
                s2.value->first().oclAsType(InstanceValue).instance.
                  oclIsTypeOf(EnumerationLiteral) and
                s2.value->first().oclAsType(InstanceValue).instance.
                  oclAsType(EnumerationLiteral).enumeration.name->
                    includes('RouteKind')) and
  slot->select(s1 | s1.definingFeature.name->includes('requestedDir'))->size <= 1
```

- There is a mandatory slot sensor. The slot is the end of a SignalLink.

```
inv SignalInstance11:
  slot->select(s1 | (s1.definingFeature.name->includes('sensor')) and
                    s1.value->size()= 1 and
                    s1.value->first().oclIsTypeOf(InstanceValue) and
                    s1.value->first().oclAsType(InstanceValue).instance.
                    oclIsTypeOf(SignalLink))->size() = 1
```

- The slots requestedDir and direction are either both present or none of them.

```
inv SignalInstance12:
  slot->select(s1 | (s1.definingFeature.name->includes('requestedDir') or
                     s1.definingFeature.name->includes('direction')) and
                     s1.value->size() = 1)->size()=0 or
  slot->select(s1 | (s1.definingFeature.name->includes('requestedDir') or
                     s1.definingFeature.name->includes('direction')) and
                     s1.value->size() = 1)->size()=2
```

- The slot signalId has a different value for every instance of SignalInstance.

```
inv SignalInstance13:
  SignalInstance.allInstances->collect(slot)->asSet->flatten->
    select(s | s.definingFeature.name->includes('signalId'))->
      iterate(s:Slot;
              result:Set(LiteralSignalId) =
                oclEmpty(Set(LiteralSignalId)) |
              result->including(s.value->first.oclAsType(LiteralSignalId)))->
                isUnique(value)
```

- If an instance of SignalInstance is located at a SegmentInstance, the values of slots direction and requestedDir are always STRAIGHT.

```
inv SignalInstance14:
  slot->select(s1 | s1.definingFeature.name->includes('direction'))->size = 1 and
  slot->select(s | s.definingFeature.name->includes('sensor'))->
    asSequence->first().value->first.oclAsType(InstanceValue).instance.slot->
      select(s | s.definingFeature.oclAsType(Property).owningClass.
        oclIsTypeOf(Signal))->asSequence->first().value->
          first.oclAsType(InstanceValue).instance.slot->
            select(s | s.definingFeature.name->includes('entrySeg'))->size()=1
  implies
  slot->select(s | s.definingFeature.name->includes('direction'))->
    asSequence->first().value->first()->
      oclAsType(InstanceValue).instance.name->includes('STRAIGHT') and
  slot->select(s | s.definingFeature.name->includes('requestedDir'))->
    asSequence->first().value->first()->
      oclAsType(InstanceValue).instance.name->includes('STRAIGHT')
```

- If an instance of SignalInstance is located at a CrossingInstance, the values of slots direction and requestedDir are always STRAIGHT.

```
inv SignalInstance15:
  slot->select(s1 | s1.definingFeature.name->includes('direction'))->size = 1 and
  slot->select(s | s.definingFeature.name->includes('sensor'))->
    asSequence->first().value->first.oclAsType(InstanceValue).instance.slot->
      select(s | s.definingFeature.oclAsType(Property).owningClass.
        oclIsTypeOf(Signal))->asSequence->first().value->
          first.oclAsType(InstanceValue).instance.slot->
            select(s | s.definingFeature.name->includes('entryCross'))->
              size()=1
  implies
  slot->select(s | s.definingFeature.name->includes('direction'))->
    asSequence->first().value->first()->
      oclAsType(InstanceValue).instance.name->includes('STRAIGHT') and
  slot->select(s | s.definingFeature.name->includes('requested'))->
    asSequence->first().value->first()->
      oclAsType(InstanceValue).instance.name->includes('STRAIGHT')
```

- If an instance of SignalInstance is located at slot e1Entry of a SinglePointInstance, the values of slots direction and requestedDir have to refer to a valid position of that point (given by slots plus and minus). If not, the actualState of that instance of SinglePointInstance has to be FAILURE.

```
inv SignalInstance16:
  slot->select(s1 | s1.definingFeature.name->includes('direction'))->size = 1 and
  slot->select(s | s.definingFeature.name->includes('sensor'))->
    asSequence->first().value->first.oclAsType(InstanceValue).instance.slot->
      select(s | s.definingFeature.oclAsType(Property).owningClass.
        oclIsTypeOf(Signal))->asSequence->first().value->
          first.oclAsType(InstanceValue).instance.slot->
            select(s | s.definingFeature.name->includes('entryPoint'))->
              asSequence->first().value->first().oclAsType(InstanceValue).
                 instance.slot->select(s | s.definingFeature.
                    oclAsType(Property).owningClass.oclIsTypeOf(SinglePoint))->
                     asSequence->first().value->first.oclAsType(InstanceValue).
                        instance.slot->select(s |
                         s.definingFeature.name->includes('e1Entry'))->size()=1

  implies
    (SinglePointInstance.allInstances->exists(p | p.slot->select(s |
      s.definingFeature.name->includes('e1Entry'))->asSequence->first().
        value.oclAsType(InstanceValue).instance.slot->select(s |
          s.definingFeature.oclAsType(Property).owningClass.
            oclIsKindOf(Sensor))->first().value->first().
              oclAsType(InstanceValue).instance.slot->select(s |
                s.definingFeature.name->includes('signal'))->asSequence->
                  first().value->first().oclAsType(InstanceValue).
                    instance.slot->select(s |
                       s.definingFeature.oclAsType(Property).owningClass.
                         oclIsKindOf(Sensor))->asSequence->first().value->
                           first().oclAsType(InstanceValue).instance = self
     and
     (p.slot->select(s | s.definingFeature.name->includes('minus') or
                         s.definingFeature.name->includes('plus'))->
       iterate(s:Slot;
               ret:Set(String)=oclEmpty(Set(String)) |
                 ret->including(s.value->first().oclAsType(InstanceValue).
                   instance.name->asSequence->first()))->includes(
```

```
        slot->select(s | s.definingFeature.name->includes('direction'))->
          asSequence->first().value->first().oclAsType(InstanceValue).
          instance.name->asSequence->first()) and
      p.slot->select(s | s.definingFeature.name->includes('minus') or
                         s.definingFeature.name->includes('plus'))->
       iterate(s:Slot;
               ret:Set(String)=oclEmpty(Set(String)) |
                 ret->including(s.value->first().oclAsType(InstanceValue).
                   instance.name->asSequence->first()))->includes(
         slot->select(s | s.definingFeature.name->includes('requestedDir'))->
           asSequence->first().value->first().oclAsType(InstanceValue).
           instance.name->asSequence->first())
     ) or
     slot->select(s | s.definingFeature.name->includes('actualState'))->
       asSequence->first().value->first().oclAsType(InstanceValue).
         instance.name->includes('FAILURE')
      ))
```

- If an instance of SignalInstance is located at slot e2Entry or e3Entry of a SinglePointInstance, the values of slots direction and requestedDir have to be STRAIGHT. If not, the actualState of that instance of SinglePointInstance has to be FAILURE.

```
inv SignalInstance17:
  slot->select(s1 | s1.definingFeature.name->includes('direction'))->size = 1 and
  slot->select(s | s.definingFeature.name->includes('sensor'))->
    asSequence->first().value->first.oclAsType(InstanceValue).instance.slot->
      select(s | s.definingFeature.oclAsType(Property).owningClass.
        oclIsTypeOf(Signal))->asSequence->first().value->
          first.oclAsType(InstanceValue).instance.slot->
            select(s | s.definingFeature.name->includes('entryPoint'))->
              asSequence->first().value->first().oclAsType(InstanceValue).
                    instance.slot->select(s | s.definingFeature.
                    oclAsType(Property).owningClass.oclIsTypeOf(SinglePoint))->
                     asSequence->first().value->first.oclAsType(InstanceValue).
                      instance.slot->select(s |
                       s.definingFeature.name->includes('e2Entry') or
                       s.definingFeature.name->includes('e3Entry'))->size()=1
  implies
  (slot->select(s | s.definingFeature.name->includes('direction'))->
     asSequence->first().value->first()->
       oclAsType(InstanceValue).instance.name->includes('STRAIGHT') and
   slot->select(s | s.definingFeature.name->includes('requestedDir'))->
     asSequence->first().value->first()->
       oclAsType(InstanceValue).instance.name->includes('STRAIGHT')
  ) or
  slot->select(s | s.definingFeature.name->includes('actualState'))->
       asSequence->first().value->first().oclAsType(InstanceValue).
         instance.name->includes('FAILURE')
```

- If an instance of SignalInstance is located at slot e1Entry of a SlipPointInstance, the values of slots direction and requestedDir have to refer to a valid position of that point (given by slots plus and minus). If not, the actualState of that instance of SlipPointInstance has to be FAILURE.

```
inv SignalInstance18:
  slot->select(s1 | s1.definingFeature.name->includes('direction'))->size = 1 and
  slot->select(s | s.definingFeature.name->includes('sensor'))->
    asSequence->first().value->first.oclAsType(InstanceValue).instance.slot->
      select(s | s.definingFeature.oclAsType(Property).owningClass.
        oclIsTypeOf(Signal))->asSequence->first().value->
```

```
          first.oclAsType(InstanceValue).instance.slot->
            select(s | s.definingFeature.name->includes('entrySlPoint'))->
              asSequence->first().value->first().oclAsType(InstanceValue).
                 instance.slot->select(s | s.definingFeature.
                    oclAsType(Property).owningClass.oclIsTypeOf(SlipPoint))->
                     asSequence->first().value->first.oclAsType(InstanceValue).
                        instance.slot->select(s |
                         s.definingFeature.name->includes('e1Entry'))->size()=1

  implies
    (SlipPointInstance.allInstances->exists(p | p.slot->select(s |
      s.definingFeature.name->includes('e1Entry'))->asSequence->first().
        value.oclAsType(InstanceValue).instance.slot->select(s |
          s.definingFeature.oclAsType(Property).owningClass.
            oclIsKindOf(Sensor))->first().value->first().
              oclAsType(InstanceValue).instance.slot->select(s |
                s.definingFeature.name->includes('signal'))->asSequence->
                  first().value->first().oclAsType(InstanceValue).
                    instance.slot->select(s |
                       s.definingFeature.oclAsType(Property).owningClass.
                         oclIsKindOf(Sensor))->asSequence->first().value->
                           first().oclAsType(InstanceValue).instance = self
     and
     (p.slot->select(s | s.definingFeature.name->includes('minus') or
                         s.definingFeature.name->includes('plus'))->
       iterate(s:Slot;
               ret:Set(String)=oclEmpty(Set(String)) |
                 ret->including(s.value->first().oclAsType(InstanceValue).
                   instance.name->asSequence->first()))->includes(
         slot->select(s | s.definingFeature.name->includes('direction'))->
           asSequence->first().value->first().oclAsType(InstanceValue).
           instance.name->asSequence->first()) and
      p.slot->select(s | s.definingFeature.name->includes('minus') or
                         s.definingFeature.name->includes('plus'))->
       iterate(s:Slot;
               ret:Set(String)=oclEmpty(Set(String)) |
                 ret->including(s.value->first().oclAsType(InstanceValue).
                   instance.name->asSequence->first()))->includes(
         slot->select(s | s.definingFeature.name->includes('requestedDir'))->
           asSequence->first().value->first().oclAsType(InstanceValue).
           instance.name->asSequence->first())
     ) or
     slot->select(s | s.definingFeature.name->includes('actualState'))->
       asSequence->first().value->first().oclAsType(InstanceValue).
         instance.name->includes('FAILURE')
))
```

- If an instance of SignalInstance is located at slot e2Entry or e3Entry of a SlipPointInstance, the values of slots direction and requestedDir have to be STRAIGHT. If not, the actualState of that instance of SlipPointInstance has to be FAILURE.

```
inv SignalInstance19:
  slot->select(s1 | s1.definingFeature.name->includes('direction'))->size = 1 and
  slot->select(s | s.definingFeature.name->includes('sensor'))->
    asSequence->first().value->first.oclAsType(InstanceValue).instance.slot->
      select(s | s.definingFeature.oclAsType(Property).owningClass.
        oclIsTypeOf(Signal))->asSequence->first().value->
          first.oclAsType(InstanceValue).instance.slot->
            select(s | s.definingFeature.name->includes('entrySlPoint'))->
              asSequence->first().value->first().oclAsType(InstanceValue).
```

```
                  instance.slot->select(s | s.definingFeature.
                  oclAsType(Property).owningClass.oclIsTypeOf(SlipPoint))->
                   asSequence->first().value->first.oclAsType(InstanceValue).
                    instance.slot->select(s |
                     s.definingFeature.name->includes('e2Entry') or
                     s.definingFeature.name->includes('e3Entry'))->size()=1
implies
(slot->select(s | s.definingFeature.name->includes('direction'))->
  asSequence->first().value->first()->
    oclAsType(InstanceValue).instance.name->includes('STRAIGHT') and
 slot->select(s | s.definingFeature.name->includes('requestedDir'))->
  asSequence->first().value->first()->
    oclAsType(InstanceValue).instance.name->includes('STRAIGHT')
) or
slot->select(s | s.definingFeature.name->includes('actualState'))->
     asSequence->first().value->first().oclAsType(InstanceValue).
       instance.name->includes('FAILURE')
```

- If an instance of SignalInstance is located at slot e4Entry of a SlipPointInstance, the values of slots direction and requestedDir have to refer to a valid position of that point (given by slots plus and minus), strictly speaking the opposite direction. If not, the actualState of that instance of SlipPointInstance has to be FAILURE.

```
inv SignalInstance20:
  slot->select(s1 | s1.definingFeature.name->includes('direction'))->size = 1 and
  slot->select(s | s.definingFeature.name->includes('sensor'))->
    asSequence->first().value->first.oclAsType(InstanceValue).instance.slot->
      select(s | s.definingFeature.oclAsType(Property).owningClass.
        oclIsTypeOf(Signal))->asSequence->first().value->
          first.oclAsType(InstanceValue).instance.slot->
            select(s | s.definingFeature.name->includes('entrySlPoint'))->
              asSequence->first().value->first().oclAsType(InstanceValue).
                 instance.slot->select(s | s.definingFeature.
                    oclAsType(Property).owningClass.oclIsTypeOf(SlipPoint))->
                     asSequence->first().value->first.oclAsType(InstanceValue).
                        instance.slot->select(s |
                         s.definingFeature.name->includes('e4Entry'))->size()=1

  implies
    (SlipPointInstance.allInstances->exists(p | p.slot->select(s |
      s.definingFeature.name->includes('e4Entry'))->asSequence->first().
        value.oclAsType(InstanceValue).instance.slot->select(s |
          s.definingFeature.oclAsType(Property).owningClass.
            oclIsKindOf(Sensor))->first().value->first().
              oclAsType(InstanceValue).instance.slot->select(s |
                s.definingFeature.name->includes('signal'))->asSequence->
                  first().value->first().oclAsType(InstanceValue).
                    instance.slot->select(s |
                       s.definingFeature.oclAsType(Property).owningClass.
                         oclIsKindOf(Sensor))->asSequence->first().value->
                           first().oclAsType(InstanceValue).instance = self
     and
     (p.slot->select(s | s.definingFeature.name->includes('minus') or
                         s.definingFeature.name->includes('plus'))->
       iterate(s:Slot;
               ret:Set(String)=oclEmpty(Set(String)) |
                 ret->including(s.value->first().oclAsType(InstanceValue).
                   instance.name->asSequence->first()))->
                     excluding('STRAIGHT')->
       excludes(
```

```
        slot->select(s | s.definingFeature.name->includes('direction'))->
          asSequence->first().value->first().oclAsType(InstanceValue).
          instance.name->asSequence->first()) and
     p.slot->select(s | s.definingFeature.name->includes('minus') or
                        s.definingFeature.name->includes('plus'))->
      iterate(s:Slot;
              ret:Set(String)=oclEmpty(Set(String)) |
                ret->including(s.value->first().oclAsType(InstanceValue).
                  instance.name->asSequence->first()))->
                    excluding('STRAIGHT')->
      excludes(
        slot->select(s | s.definingFeature.name->includes('requestedDir'))->
          asSequence->first().value->first().oclAsType(InstanceValue).
          instance.name->asSequence->first())
    ) or
    slot->select(s | s.definingFeature.name->includes('actualState'))->
      asSequence->first().value->first().oclAsType(InstanceValue).
        instance.name->includes('FAILURE')
))
```

- If an instance of SignalInstance is located at slot e1Entry of a DoubleSlipPointInstance, the values of slots direction and requestedDir have to refer to a valid position of that point (given by slots plus and minus). If not, the actualState of that instance of DoubleSlipPointInstance has to be FAILURE.

```
inv SignalInstance21:
  slot->select(s1 | s1.definingFeature.name->includes('direction'))->size = 1 and
  slot->select(s | s.definingFeature.name->includes('sensor'))->
    asSequence->first().value->first.oclAsType(InstanceValue).instance.slot->
      select(s | s.definingFeature.oclAsType(Property).owningClass.
        oclIsTypeOf(Signal))->asSequence->first().value->
          first.oclAsType(InstanceValue).instance.slot->
            select(s | s.definingFeature.name->includes('entryDbSlPoint'))->
              asSequence->first().value->first().oclAsType(InstanceValue).
                 instance.slot->select(s | s.definingFeature.
                    oclAsType(Property).owningClass.oclIsTypeOf(DoubleSlipPoint))->
                     asSequence->first().value->first.oclAsType(InstanceValue).
                        instance.slot->select(s |
                         s.definingFeature.name->includes('e1Entry'))->size()=1

  implies
    (SlipPointInstance.allInstances->exists(p | p.slot->select(s |
      s.definingFeature.name->includes('e1Entry'))->asSequence->first().
        value.oclAsType(InstanceValue).instance.slot->select(s |
          s.definingFeature.oclAsType(Property).owningClass.
            oclIsKindOf(Sensor))->first().value->first().
              oclAsType(InstanceValue).instance.slot->select(s |
                s.definingFeature.name->includes('signal'))->asSequence->
                  first().value->first().oclAsType(InstanceValue).
                    instance.slot->select(s |
                       s.definingFeature.oclAsType(Property).owningClass.
                         oclIsKindOf(Sensor))->asSequence->first().value->
                           first().oclAsType(InstanceValue).instance = self
     and
     (p.slot->select(s | s.definingFeature.name->includes('minus') or
                         s.definingFeature.name->includes('plus'))->
       iterate(s:Slot;
               ret:Set(String)=oclEmpty(Set(String)) |
                 ret->including(s.value->first().oclAsType(InstanceValue).
                   instance.name->asSequence->first()))->includes(
```

```
        slot->select(s | s.definingFeature.name->includes('direction'))->
          asSequence->first().value->first().oclAsType(InstanceValue).
          instance.name->asSequence->first()) and
      p.slot->select(s | s.definingFeature.name->includes('minus') or
                         s.definingFeature.name->includes('plus'))->
       iterate(s:Slot;
               ret:Set(String)=oclEmpty(Set(String)) |
                 ret->including(s.value->first().oclAsType(InstanceValue).
                   instance.name->asSequence->first()))->includes(
         slot->select(s | s.definingFeature.name->includes('requestedDir'))->
           asSequence->first().value->first().oclAsType(InstanceValue).
           instance.name->asSequence->first())
     ) or
     slot->select(s | s.definingFeature.name->includes('actualState'))->
       asSequence->first().value->first().oclAsType(InstanceValue).
         instance.name->includes('FAILURE')
))
```

- If an instance of SignalInstance is located at slot e1Entry of a DoubleSlipPointInstance, the values of slots direction and requestedDir have to refer to a valid position of that point (given by slots plusOpp and minusOpp). If not, the actualState of that instance of DoubleSlipPointInstance has to be FAILURE.

```
inv SignalInstance22:
  slot->select(s1 | s1.definingFeature.name->includes('direction'))->size = 1 and
  slot->select(s | s.definingFeature.name->includes('sensor'))->
    asSequence->first().value->first.oclAsType(InstanceValue).instance.slot->
      select(s | s.definingFeature.oclAsType(Property).owningClass.
        oclIsTypeOf(Signal))->asSequence->first().value->
          first.oclAsType(InstanceValue).instance.slot->
            select(s | s.definingFeature.name->includes('entryDbSlPoint'))->
              asSequence->first().value->first().oclAsType(InstanceValue).
                 instance.slot->select(s | s.definingFeature.
                    oclAsType(Property).owningClass.oclIsTypeOf(DoubleSlipPoint))->
                     asSequence->first().value->first.oclAsType(InstanceValue).
                        instance.slot->select(s |
                         s.definingFeature.name->includes('e2Entry'))->size()=1

  implies
    (SlipPointInstance.allInstances->exists(p | p.slot->select(s |
      s.definingFeature.name->includes('e2Entry'))->asSequence->first().
        value.oclAsType(InstanceValue).instance.slot->select(s |
          s.definingFeature.oclAsType(Property).owningClass.
            oclIsKindOf(Sensor))->first().value->first().
              oclAsType(InstanceValue).instance.slot->select(s |
                s.definingFeature.name->includes('signal'))->asSequence->
                  first().value->first().oclAsType(InstanceValue).
                    instance.slot->select(s |
                       s.definingFeature.oclAsType(Property).owningClass.
                         oclIsKindOf(Sensor))->asSequence->first().value->
                           first().oclAsType(InstanceValue).instance = self
     and
     (p.slot->select(s | s.definingFeature.name->includes('minusOpp') or
                         s.definingFeature.name->includes('plusOpp'))->
       iterate(s:Slot;
               ret:Set(String)=oclEmpty(Set(String)) |
                 ret->including(s.value->first().oclAsType(InstanceValue).
                   instance.name->asSequence->first()))->includes(
         slot->select(s | s.definingFeature.name->includes('direction'))->
           asSequence->first().value->first().oclAsType(InstanceValue).
```

```
          instance.name->asSequence->first()) and
      p.slot->select(s | s.definingFeature.name->includes('minusOpp') or
                         s.definingFeature.name->includes('plusOpp'))->
       iterate(s:Slot;
               ret:Set(String)=oclEmpty(Set(String)) |
                 ret->including(s.value->first().oclAsType(InstanceValue).
                   instance.name->asSequence->first()))->includes(
         slot->select(s | s.definingFeature.name->includes('requestedDir'))->
           asSequence->first().value->first().oclAsType(InstanceValue).
           instance.name->asSequence->first())
     ) or
     slot->select(s | s.definingFeature.name->includes('actualState'))->
       asSequence->first().value->first().oclAsType(InstanceValue).
         instance.name->includes('FAILURE')
))
```

- If an instance of SignalInstance is located at slot e3Entry of a DoubleSlipPointInstance, the values of slots direction and requestedDir have to refer to a valid position of that point (given by slots plusOpp and minusOpp), strictly speaking the opposite direction. If not, the actualState of that instance of DoubleSlipPointInstance has to be FAILURE.

```
inv SignalInstance23:
  slot->select(s1 | s1.definingFeature.name->includes('direction'))->size = 1 and
  slot->select(s | s.definingFeature.name->includes('sensor'))->
    asSequence->first().value->first.oclAsType(InstanceValue).instance.slot->
      select(s | s.definingFeature.oclAsType(Property).owningClass.
        oclIsTypeOf(Signal))->asSequence->first().value->
          first.oclAsType(InstanceValue).instance.slot->
            select(s | s.definingFeature.name->includes('entryDbSlPoint'))->
              asSequence->first().value->first().oclAsType(InstanceValue).
                instance.slot->select(s | s.definingFeature.
                   oclAsType(Property).owningClass.oclIsTypeOf(DoubleSlipPoint))->
                    asSequence->first().value->first.oclAsType(InstanceValue).
                       instance.slot->select(s |
                        s.definingFeature.name->includes('e3Entry'))->size()=1

  implies
    (SlipPointInstance.allInstances->exists(p | p.slot->select(s |
      s.definingFeature.name->includes('e3Entry'))->asSequence->first().
        value.oclAsType(InstanceValue).instance.slot->select(s |
          s.definingFeature.oclAsType(Property).owningClass.
            oclIsKindOf(Sensor))->first().value->first().
              oclAsType(InstanceValue).instance.slot->select(s |
                s.definingFeature.name->includes('signal'))->asSequence->
                  first().value->first().oclAsType(InstanceValue).
                    instance.slot->select(s |
                       s.definingFeature.oclAsType(Property).owningClass.
                         oclIsKindOf(Sensor))->asSequence->first().value->
                           first().oclAsType(InstanceValue).instance = self
     and
     (p.slot->select(s | s.definingFeature.name->includes('minusOpp') or
                         s.definingFeature.name->includes('plusOpp'))->
       iterate(s:Slot;
               ret:Set(String)=oclEmpty(Set(String)) |
                 ret->including(s.value->first().oclAsType(InstanceValue).
                   instance.name->asSequence->first()))->
                     excluding('STRAIGHT')->
       excludes(
         slot->select(s | s.definingFeature.name->includes('direction'))->
           asSequence->first().value->first().oclAsType(InstanceValue).
```

```
        instance.name->asSequence->first()) and
    p.slot->select(s | s.definingFeature.name->includes('minusOpp') or
                       s.definingFeature.name->includes('plusOpp'))->
     iterate(s:Slot;
             ret:Set(String)=oclEmpty(Set(String)) |
               ret->including(s.value->first().oclAsType(InstanceValue).
                 instance.name->asSequence->first()))->
                   excluding('STRAIGHT')->
     excludes(
       slot->select(s | s.definingFeature.name->includes('requestedDir'))->
         asSequence->first().value->first().oclAsType(InstanceValue).
         instance.name->asSequence->first())
   ) or
   slot->select(s | s.definingFeature.name->includes('actualState'))->
     asSequence->first().value->first().oclAsType(InstanceValue).
       instance.name->includes('FAILURE')
))
```

- If an instance of SignalInstance is located at slot e3Entry of a DoubleSlipPointInstance, the values of slots direction and requestedDir have to refer to a valid position of that point (given by slots plus and minus), strictly speaking the opposite direction. If not, the actualState of that instance of DoubleSlipPointInstance has to be FAILURE.

```
inv SignalInstance24:
  slot->select(s1 | s1.definingFeature.name->includes('direction'))->size = 1 and
  slot->select(s | s.definingFeature.name->includes('sensor'))->
    asSequence->first().value->first.oclAsType(InstanceValue).instance.slot->
      select(s | s.definingFeature.oclAsType(Property).owningClass.
        oclIsTypeOf(Signal))->asSequence->first().value->
          first.oclAsType(InstanceValue).instance.slot->
            select(s | s.definingFeature.name->includes('entryDbSlPoint'))->
              asSequence->first().value->first().oclAsType(InstanceValue).
                 instance.slot->select(s | s.definingFeature.
                    oclAsType(Property).owningClass.oclIsTypeOf(DoubleSlipPoint))->
                     asSequence->first().value->first.oclAsType(InstanceValue).
                        instance.slot->select(s |
                         s.definingFeature.name->includes('e4Entry'))->size()=1

  implies
    (SlipPointInstance.allInstances->exists(p | p.slot->select(s |
      s.definingFeature.name->includes('e4Entry'))->asSequence->first().
        value.oclAsType(InstanceValue).instance.slot->select(s |
          s.definingFeature.oclAsType(Property).owningClass.
            oclIsKindOf(Sensor))->first().value->first().
              oclAsType(InstanceValue).instance.slot->select(s |
                s.definingFeature.name->includes('signal'))->asSequence->
                 first().value->first().oclAsType(InstanceValue).
                   instance.slot->select(s |
                      s.definingFeature.oclAsType(Property).owningClass.
                        oclIsKindOf(Sensor))->asSequence->first().value->
                          first().oclAsType(InstanceValue).instance = self
     and
     (p.slot->select(s | s.definingFeature.name->includes('minus') or
                         s.definingFeature.name->includes('plus'))->
       iterate(s:Slot;
               ret:Set(String)=oclEmpty(Set(String)) |
                 ret->including(s.value->first().oclAsType(InstanceValue).
                   instance.name->asSequence->first()))->
                     excluding('STRAIGHT')->
       excludes(
```

```
        slot->select(s | s.definingFeature.name->includes('direction'))->
          asSequence->first().value->first().oclAsType(InstanceValue).
          instance.name->asSequence->first()) and
     p.slot->select(s | s.definingFeature.name->includes('minus') or
                        s.definingFeature.name->includes('plus'))->
      iterate(s:Slot;
              ret:Set(String)=oclEmpty(Set(String)) |
                ret->including(s.value->first().oclAsType(InstanceValue).
                  instance.name->asSequence->first()))->
                    excluding('STRAIGHT')->
      excludes(
        slot->select(s | s.definingFeature.name->includes('requestedDir'))->
          asSequence->first().value->first().oclAsType(InstanceValue).
          instance.name->asSequence->first())
    ) or
    slot->select(s | s.definingFeature.name->includes('actualState'))->
      asSequence->first().value->first().oclAsType(InstanceValue).
        instance.name->includes('FAILURE')
))
```

Semantics

Given by the transformation from a concrete network to a labeled transition system.

Notation

A SignalInstance is either depicted as a UML object or as a symbol in track layout diagram (see Fig. 2.27. It is placed in front of the sensor to which it belongs to.

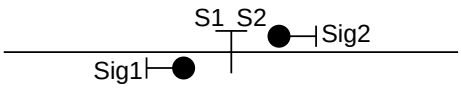

Figure 2.27: Signal on bidirectional segments

There are four different kinds of signals than can be used

- A signal that only gives information *GO* and *STOP* such that:

```
not slot->exists(s1 | s1.definingFeature.name->includes('limit') or
                      s1.definingFeature.name->includes('direction'))
```

Figure 2.28: Signal that signals *GO* and *STOP*

- A signal that gives additionally speed limits such that:

```
not slot->exists(s1 | s1.definingFeature.name->includes('direction')) and
slot->one(s2 | s2.definingFeature.name->includes('limit'))
```

Figure 2.29: Signal that signals *GO*, *STOP*, and speed limit

- A signal that gives additionally directions such that:

```
not slot->exists(s1 | s1.definingFeature.name->includes('limit')) and
slot->one(s2 | s2.definingFeature.name->includes('direction'))
```

Figure 2.30: Signal that signals *GO*, *STOP*, and direction to go

- A signal that gives additionally both speed limits and directions:

```
slot->one(s1 | s1.definingFeature.name->includes('limit')) and
slot->one(s2 | s2.definingFeature.name->includes('direction'))
```

Figure 2.31: Signal with all information possible

2.4.9 SignalLink

Description

A SignalLink is the instance of a signal association.

Associations

None.

Attributes

None.

Constraints

- There is one classifier instantiated.

```
inv SignalLink1:
  classifier->size() = 1
```

- The instantiated classifier is a kind of Association.

```
inv SignalLink2:
  classifier->one(oclIsKindOf(Association))
```

- The instantiated classifier has the type SignalAssociation.

```
inv SignalLink3:
  classifier->one(oclIsTypeOf(SignalAssociation))
```

- There are two slots.

```
inv SignalLink4:
  slot->size()=2
```

Semantics

Given by the transformation from a concrete network to a labeled transition system.

Notation

A SignalLink is either depicted as a UML link or implicity in track layout diagrams by placing the signal near to the associated sensor. In object notation, the associations in Fig. 2.27 can be seen explicitly as shown in Fig. 2.32.

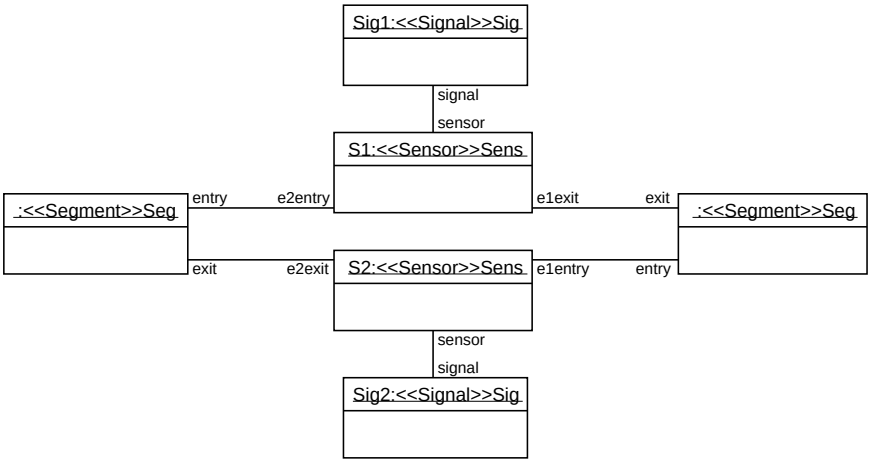

Figure 2.32: Signal on bidirectional segments in object notation

2.4.10 SinglePointInstance

Description

A SinglePointInstance is the instance of a single point.

Associations

None.

Attributes

None.

Constraints

- There is one classifier instantiated.

```
inv SinglePointInstance1:
  classifier->size() = 1
```

- The instantiated classifier is a kind of Class.

```
inv SinglePointInstance2:
  classifier->one(oclIsKindOf(Class))
```

- The instantiated classifier is a kind of TrackElement.

```
inv SinglePointInstance3:
  classifier->one(oclIsKindOf(TrackElement))
```

- The instantiated classifier is a kind of Point.

```
inv SinglePointInstance4:
  classifier->one(oclIsKindOf(Point))
```

- The instantiated classifier has the type SinglePoint.

```
inv SinglePointInstance5:
  classifier->one(oclIsTypeOf(SinglePoint))
```

- There is a mandatory slot pointId. The value is given by a LiteralPointId.

```
inv SinglePointInstance6:
  slot->one(s1 | s1.definingFeature.name->includes('pointId') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralPointId))
```

- There is a mandatory slot maxNumberOfTrains. The value is given by a LiteralInteger.

```
inv SinglePointInstance7:
  slot->one(s1 | s1.definingFeature.name->includes('maxNumberOfTrains') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralInteger) and
                 s1.value->first().oclAsType(LiteralInteger).value = 1)
```

- There is a mandatory slot plus. The value is taken from the enumeration RouteKind.

```
inv SinglePointInstance8:
  slot->one(s1 | s1.definingFeature.name->includes('plus') and
                 s1.value->size()= 1 and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclIsTypeOf(EnumerationLiteral) and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclAsType(EnumerationLiteral).enumeration.name->
                     includes('RouteKind'))
```

- There is a mandatory slot minus. The value is taken from the enumeration RouteKind.

```
inv SinglePointInstance9:
  slot->one(s1 | s1.definingFeature.name->includes('minus') and
                 s1.value->size()= 1 and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclIsTypeOf(EnumerationLiteral) and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclAsType(EnumerationLiteral).enumeration.name->
                     includes('RouteKind'))
```

- There is a mandatory slot actualState. The value is taken from the enumeration PointStateKind.

```
inv SinglePointInstance10:
  slot->one(s1 | s1.definingFeature.name->includes('actualState') and
                 s1.value->size()= 1 and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclIsTypeOf(EnumerationLiteral) and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclAsType(EnumerationLiteral).enumeration.name->
                     includes('PointStateKind'))
```

- There is a mandatory slot requested. The value is taken from the enumeration RouteKind.

```
inv SinglePointInstance11:
  slot->one(s1 | s1.definingFeature.name->includes('requestedState') and
                 s1.value->size()= 1 and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclIsTypeOf(EnumerationLiteral) and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclAsType(EnumerationLiteral).enumeration.name->
                     includes('RouteKind'))
```

- There is a mandatory slot requestTime. The value is given by a LiteralTimeInstant.

```
inv SinglePointInstance12:
  slot->one(s1 | s1.definingFeature.name->includes('requestTime') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralTimeInstant))
```

- There is a mandatory slot delta_p. The value is given by a LiteralDuration.

```
inv SinglePointInstance13:
  slot->one(s1 | s1.definingFeature.name->includes('delta_p') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralDuration))
```

- There is an optional slot limit. If present, the value is given by a LiteralInteger.

```
inv SinglePointInstance14:
  slot->one(s1 | s1.definingFeature.name->includes('limit') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralInteger) and
                 s1.value->first.oclAsType(LiteralInteger).value >= 0) or
  not slot->exists(s1 | s1.definingFeature.name->includes('limit'))
```

- The slots e1Entry, e2Exit, e3Exit exist either all or none of them. If present, each slot is the end of a SensorLink.

```
inv SinglePointInstance15:
  (slot->select(s1 | (s1.definingFeature.name->includes('e1Entry') or
                      s1.definingFeature.name->includes('e2Exit') or
                      s1.definingFeature.name->includes('e3Exit')) and
                     s1.value->size()= 1 and
                     s1.value->first().oclIsTypeOf(InstanceValue) and
                     s1.value->first().oclAsType(InstanceValue).instance.
                     oclIsTypeOf(SensorLink))->size() = 3) or
   (slot->select(s1 | (s1.definingFeature.name->includes('e1Entry') or
                       s1.definingFeature.name->includes('e2Exit') or
                       s1.definingFeature.name->includes('e3Exit')) and
                      s1.value->size()= 1 and
                      s1.value->first().oclIsTypeOf(InstanceValue) and
                      s1.value->first().oclAsType(InstanceValue).instance.
                      oclIsTypeOf(SensorLink))->size() = 0)
```

- The slots e1Exit, e2Entry, e3Entry exist either all or none of them. If present, each slot is the end of a SensorLink.

```
inv SinglePointInstance16:
  (slot->select(s1 | (s1.definingFeature.name->includes('e1Exit') or
                      s1.definingFeature.name->includes('e2Entry') or
                      s1.definingFeature.name->includes('e3Entry')) and
                     s1.value->size()= 1 and
                     s1.value->first().oclIsTypeOf(InstanceValue) and
                     s1.value->first().oclAsType(InstanceValue).instance.
                       oclIsTypeOf(SensorLink))->size() = 3) or
  (slot->select(s1 | (s1.definingFeature.name->includes('e1Exit') or
                     s1.definingFeature.name->includes('e2Entry') or
                     s1.definingFeature.name->includes('e3Entry')))->size()=0)
```

- There are at least three slots named e1Entry, e1Exit, e2Entry, e2Exit, e3Entry, or e3Exit. If present, each slot is the end of a SensorLink.

```
inv SinglePointInstance17:
  slot->select(s1 | (s1.definingFeature.name->includes('e1Entry') or
                     s1.definingFeature.name->includes('e1Exit') or
                     s1.definingFeature.name->includes('e2Entry') or
                     s1.definingFeature.name->includes('e2Exit') or
                     s1.definingFeature.name->includes('e3Entry') or
                     s1.definingFeature.name->includes('e3Exit')) and
                    s1.value->size()= 1 and
                    s1.value->first().oclIsTypeOf(InstanceValue) and
                    s1.value->first().oclAsType(InstanceValue).instance.
                       oclIsTypeOf(SensorLink))->size() >= 3
```

- One of the slots plus and minus has the value STRAIGHT, the other one the value LEFT or RIGHT.

```
inv SinglePointInstance18:
  slot->select(s1 | s1.definingFeature.name->includes('minus') or
                    s1.definingFeature.name->includes('plus'))->
    one(s2 | s2.value->size()= 1 and
             s2.value->first().oclIsTypeOf(InstanceValue) and
             s2.value->first().oclAsType(InstanceValue).instance.name->
               includes('STRAIGHT')) and
  slot->select(s1 | s1.definingFeature.name->includes('minus') or
                    s1.definingFeature.name->includes('plus'))->
    one(s2 | s2.value->size()= 1 and
             s2.value->first().oclIsTypeOf(InstanceValue) and
             (s2.value->first().oclAsType(InstanceValue).instance.name->
               includes('LEFT') or
              s2.value->first()->oclAsType(InstanceValue).instance.name->
               includes('RIGHT')))
```

- The value of the slot requestedState must be a valid direction of that point, i.e. a value defined by slot plus or minus.

```
inv SinglePointInstance19:
   let
      r1:Set(Slot) = slot->select(s1 |
        s1.definingFeature.name->includes('requestedState'))
   in
     let
        r2:Set(String) = slot->select(s2 |
          s2.definingFeature.name->includes('plus') or
          s2.definingFeature.name->includes('minus'))->
            collect(value.oclAsType(InstanceValue).instance.name)->flatten->
              asSet
     in
       r1->forAll(s3 | r2->includes(s3.value->first()->
         oclAsType(InstanceValue).instance.name->asSequence->first()))
```

- The value of the slot actualState must be a valid direction of that point, i.e. a value defined by slot plus or minus. If not, the actualState has the value FAILURE.

```
inv SinglePointInstance20:
   let
      r1:Set(Slot) = slot->select(s1 |
                s1.definingFeature.name->includes('actualState'))
   in
     let
        r2:Set(String) = slot->select(s2 |
          s2.definingFeature.name->includes('plus') or
```

```
        s2.definingFeature.name->includes('minus'))->
          collect(value.oclAsType(InstanceValue).instance.name)->flatten->
            asSet
    in
      r1->forAll(s3 | r2->includes(s3.value->first()->
                        oclAsType(InstanceValue).instance.name->asSequence->
                          first()) or
                      s3.value->first()->oclAsType(InstanceValue).instance.
                        name->includes('FAILURE'))
```

- The slot pointId has a different value for every instance of SinglePointInstance, SlipPointInstance, and DoubleSlipPointInstance.

```
inv SinglePointInstance21:
  SinglePointInstance.allInstances->collect(slot)->asSet->flatten->
    select(s | s.definingFeature.name->includes('pointId'))->
      union(SlipPointInstance.allInstances->collect(slot)->asSet->flatten->
        select(s | s.definingFeature.name->includes('pointId'))
      )->
      union(DoubleSlipPointInstance.allInstances->collect(slot)->asSet->flatten->
        select(s | s.definingFeature.name->includes('pointId') or
                   s.definingFeature.name->includes('pointIdOpp'))
      )->
      iterate(s:Slot;
              result:Set(LiteralPointId) =
                oclEmpty(Set(LiteralPointId)) |
              result->including(s.value->first.oclAsType(LiteralPointId)))->
                isUnique(value)
```

Semantics

Given by the transformation from a concrete network to a labeled transition system.

Notation

A SinglePointInstance is either depicted as a UML object or as a symbol as shown in Fig. 2.33. At least one position must be marked as *plus* or *minus* position.

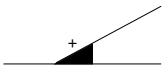

Figure 2.33: Single point instance

2.4.11 SlipPointInstance

Description

A SlipPointInstance is the instance of a slip point.

Associations

None.

Attributes

None.

Constraints

- There is one classifier instantiated.

```
inv SlipPointInstance1:
  classifier->size() = 1
```

- The instantiated classifier is a kind of Class.

```
inv SlipPointInstance2:
  classifier->one(oclIsKindOf(Class))
```

- The instantiated classifier is a kind of TrackElement.

```
inv SlipPointInstance3:
  classifier->one(oclIsKindOf(TrackElement))
```

- The instantiated classifier is a kind of Point.

```
inv SlipPointInstance4:
  classifier->one(oclIsKindOf(Point))
```

- The instantiated classifier is a kind of SlipPoint.

```
inv SlipPointInstance5:
  classifier->one(oclIsTypeOf(SlipPoint))
```

- There is a mandatory slot pointId. The value is given by a LiteralPointId.

```
inv SlipPointInstance6:
  slot->one(s1 | s1.definingFeature.name->includes('pointId') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralPointId))
```

- There is a mandatory slot plus. The value is taken from the enumeration RouteKind.

```
inv SlipPointInstance7:
  slot->one(s1 | s1.definingFeature.name->includes('plus') and
                 s1.value->size()= 1 and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclIsTypeOf(EnumerationLiteral) and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclAsType(EnumerationLiteral).enumeration.name->
                     includes('RouteKind'))
```

- There is a mandatory slot minus. The value is taken from the enumeration RouteKind.

```
inv SlipPointInstance8:
  slot->one(s1 | s1.definingFeature.name->includes('minus') and
                 s1.value->size()= 1 and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclIsTypeOf(EnumerationLiteral) and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclAsType(EnumerationLiteral).enumeration.name->
                     includes('RouteKind'))
```

- There is a mandatory slot maxNumberOfTrains. The value is given by a LiteralInteger.

```
inv SlipPointInstance9:
  slot->one(s1 | s1.definingFeature.name->includes('maxNumberOfTrains') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralInteger) and
                 s1.value->first().oclAsType(LiteralInteger).value = 1)
```

- There is a mandatory slot delta_p. The value is given by a LiteralDuration.

```
inv SlipPointInstance10:
  slot->one(s1 | s1.definingFeature.name->includes('delta_p') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralDuration))
```

- There is a mandatory slot actualState. The value is taken from the enumeration PointStateKind.

```
inv SlipPointInstance11:
  slot->one(s1 | s1.definingFeature.name->includes('actualState') and
                 s1.value->size()= 1 and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclIsTypeOf(EnumerationLiteral) and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclAsType(EnumerationLiteral).enumeration.name->
                     includes('PointStateKind'))
```

- There is a mandatory slot requestedState. The value is taken from the enumeration RouteKind.

```
inv SlipPointInstance12:
  slot->one(s1 | s1.definingFeature.name->includes('requestedState') and
                 s1.value->size()= 1 and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclIsTypeOf(EnumerationLiteral) and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclAsType(EnumerationLiteral).enumeration.name->
                     includes('RouteKind'))
```

- There is a mandatory slot requestTime. The value is given by a LiteralTimeInstant.

```
inv SlipPointInstance13:
  slot->one(s1 | s1.definingFeature.name->includes('requestTime') and
                 s1.value->size()= 1 and
                 s1.value->first().oclIsTypeOf(LiteralTimeInstant))
```

- There is a mandatory slot limit. The value is given by a LiteralInteger.

```
inv SlipPointInstance14:
  slot->select(s1 | s1.definingFeature.name->includes('limit'))->
    forAll(s2 | s2.value->size()= 1 and
                s2.value->first().oclIsTypeOf(LiteralInteger) and
                s2.value->first()->oclAsType(LiteralInteger).value >= 0)
```

- One of the slots plus and minus has the value STRAIGHT, the other one the value LEFT or RIGHT.

```
inv SlipPointInstance15:
  slot->select(s1 | s1.definingFeature.name->includes('minus') or
                    s1.definingFeature.name->includes('plus'))->
    one(s2 | s2.value->size()= 1 and
             s2.value->first().oclIsTypeOf(InstanceValue) and
             s2.value->first().oclAsType(InstanceValue).instance.name->
```

```
              includes('STRAIGHT')) and
  slot->select(s1 | s1.definingFeature.name->includes('minus') or
                    s1.definingFeature.name->includes('plus'))->
    one(s2 | s2.value->size()= 1 and
             s2.value->first().oclIsTypeOf(InstanceValue) and
             (s2.value->first().oclAsType(InstanceValue).instance.name->
               includes('LEFT') or
              s2.value->first()->oclAsType(InstanceValue).instance.name->
               includes('RIGHT')))
```

- The slots e1Entry, e2Exit, e4Exit exist either all or none of them. If present, each slot is the end of a SensorLink.

```
inv SlipPointInstance16:
  (slot->select(s1 | (s1.definingFeature.name->includes('e1Entry') or
                      s1.definingFeature.name->includes('e2Exit') or
                      s1.definingFeature.name->includes('e4Exit')) and
                     s1.value->size()= 1 and
                     s1.value->first().oclIsTypeOf(InstanceValue) and
                     s1.value->first().oclAsType(InstanceValue).instance.
                     oclIsTypeOf(SensorLink))->size() = 3) or
   (slot->select(s1 | (s1.definingFeature.name->includes('e1Entry') or
                      s1.definingFeature.name->includes('e2Exit') or
                      s1.definingFeature.name->includes('e4Exit')) and
                     s1.value->size()= 1 and
                     s1.value->first().oclIsTypeOf(InstanceValue) and
                     s1.value->first().oclAsType(InstanceValue).instance.
                     oclIsTypeOf(SensorLink))->size() = 0)
```

- The slots e1Exit, e2Entry, e4Entry exist either all or none of them. If present, each slot is the end of a SensorLink.

```
inv SlipPointInstance17:
  (slot->select(s1 | (s1.definingFeature.name->includes('e1Exit') or
                      s1.definingFeature.name->includes('e2Entry') or
                      s1.definingFeature.name->includes('e4Entry')) and
                     s1.value->size()= 1 and
                     s1.value->first().oclIsTypeOf(InstanceValue) and
                     s1.value->first().oclAsType(InstanceValue).instance.
                     oclIsTypeOf(SensorLink))->size() = 3) or
   (slot->select(s1 | (s1.definingFeature.name->includes('e1Exit') or
                      s1.definingFeature.name->includes('e2Entry') or
                      s1.definingFeature.name->includes('e4Entry')) and
                     s1.value->size()= 1 and
                     s1.value->first().oclIsTypeOf(InstanceValue) and
                     s1.value->first().oclAsType(InstanceValue).instance.
                     oclIsTypeOf(SensorLink))->size() = 0)
```

- The slots e3Entry and e4Exit exist either both or none of them. If present, each slot is the end of a SensorLink.

```
inv SlipPointInstance18:
  (slot->select(s1 | (s1.definingFeature.name->includes('e4Exit') or
                      s1.definingFeature.name->includes('e3Entry')) and
                     s1.value->size()= 1 and
                     s1.value->first().oclIsTypeOf(InstanceValue) and
                     s1.value->first().oclAsType(InstanceValue).instance.
                     oclIsTypeOf(SensorLink))->size() = 2) or
   (slot->select(s1 | (s1.definingFeature.name->includes('e4Exit') or
                      s1.definingFeature.name->includes('e3Entry')) and
```

```
                    s1.value->size()= 1 and
                    s1.value->first().oclIsTypeOf(InstanceValue) and
                    s1.value->first().oclAsType(InstanceValue).instance.
                    oclIsTypeOf(SensorLink))->size() = 0)
```

- The slots e3Exit and e4Entry exist either both or none of them. If present, each slot is the end of a SensorLink.

```
inv SlipPointInstance19:
  (slot->select(s1 | (s1.definingFeature.name->includes('e4Entry') or
                      s1.definingFeature.name->includes('e3Exit')) and
                    s1.value->size()= 1 and
                    s1.value->first().oclIsTypeOf(InstanceValue) and
                    s1.value->first().oclAsType(InstanceValue).instance.
                    oclIsTypeOf(SensorLink))->size() = 2) or
   (slot->select(s1 | (s1.definingFeature.name->includes('e4Entry') or
                      s1.definingFeature.name->includes('e3Exit')) and
                    s1.value->size()= 1 and
                    s1.value->first().oclIsTypeOf(InstanceValue) and
                    s1.value->first().oclAsType(InstanceValue).instance.
                    oclIsTypeOf(SensorLink))->size() = 0)
```

- There are at least 4 slots named e1Entry, e1Exit, e2Entry, e2Exit, e3Entry, e3Exit, e4Entry, or e4Exit. If present, each slot is the end of a SensorLink.

```
inv SlipPointInstance20:
  slot->select(s1 | (s1.definingFeature.name->includes('e1Entry') or
                     s1.definingFeature.name->includes('e1Exit') or
                     s1.definingFeature.name->includes('e2Entry') or
                     s1.definingFeature.name->includes('e2Exit') or
                     s1.definingFeature.name->includes('e3Entry') or
                     s1.definingFeature.name->includes('e3Exit') or
                     s1.definingFeature.name->includes('e4Entry') or
                     s1.definingFeature.name->includes('e4Exit')) and
                    s1.value->size()= 1 and
                    s1.value->first().oclIsTypeOf(InstanceValue) and
                    s1.value->first().oclAsType(InstanceValue).instance.
                       oclIsTypeOf(SensorLink))->size() >= 4
```

- The slot pointId has a different value for every instance of SinglePointInstance, SlipPointInstance, and DoubleSlipPointInstance.

```
inv SlipPointInstance21:
  SlipPointInstance.allInstances->collect(slot)->asSet->flatten->
    select(s | s.definingFeature.name->includes('pointId'))->
      union(SinglePointInstance.allInstances->collect(slot)->asSet->flatten->
        select(s | s.definingFeature.name->includes('pointId'))
      )->
      union(DoubleSlipPointInstance.allInstances->collect(slot)->asSet->flatten->
        select(s | s.definingFeature.name->includes('pointId') or
                   s.definingFeature.name->includes('pointIdOpp'))
      )->
      iterate(s:Slot;
              result:Set(LiteralPointId) =
                oclEmpty(Set(LiteralPointId)) |
              result->including(s.value->first.oclAsType(LiteralPointId)))->
                isUnique(value)
```

Semantics

Given by the transformation from a concrete network to a labeled transition system.

Notation

A SlipPointInstance is either depicted as a UML object or as a symbol as shown in Fig. 2.34. At least one of the *plus* and *minus* positions of each point has to be marked.

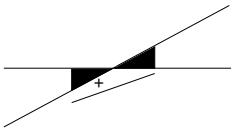

Figure 2.34: Single slip point instance

2.5 Routes

Route definitions are based on a track network description. They describe routes for trains on the basis of sensor sequences. The setting of the first signal for entering the route must be given, just as the states of all points needed for safe travel on the route. Furthermore, conflicts with other routes are memorized to guarantee safe traffic on the track network.

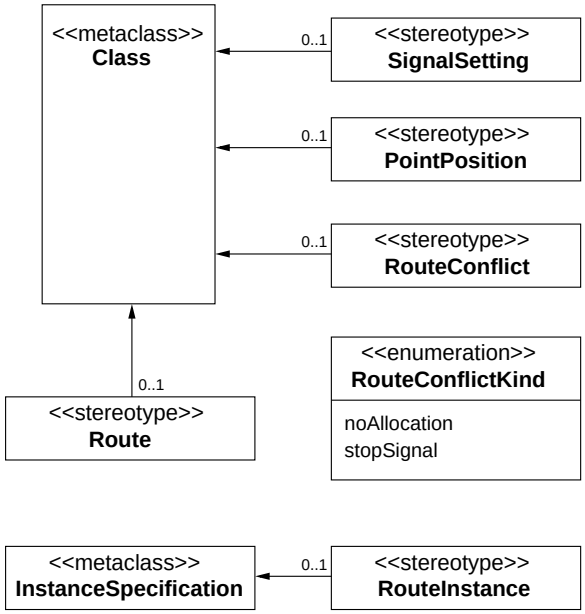

Figure 2.35: Stereotypes for modeling routes in networks

2.5.1 PointPosition

Description

PointPosition is used to model requested positions of point instances.

Associations

None.

Attributes

None.

Constraints

- There is a mandatory property pointId with type PointId. The property is read-only and has the multiplicity 1.

```
inv PointPosition1:
  ownedAttribute->one(a | a.name->includes('pointId') and
                          a.type.name->includes('PointId') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
                          a.isReadOnly = true)
```

- There is a mandatory property pointState with type RouteKind. The property is read-only and has the multiplicity 1.

```
inv PointPosition2:
  ownedAttribute->one(a | a.name->includes('pointState') and
                          a.type.name->includes('RouteKind') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
                          a.isReadOnly = true)
```

Semantics

PointPosition combines the *pointId* with type *PointId* of a point and the required *pointState* with type *RouteKind* as route.

Notation

PointPosition is used in class digrams using the UML class notation.

2.5.2 PointPositionInstance

Description

A PointPositionInstance is the instance of a point position.

Associations

None.

Attributes

None.

Constraints

- There is one classifier instantiated.

```
inv PointPositionInstance1:
  classifier->size() = 1
```

- The instantiated classifier is a kind of Class.

```
inv PointPositionInstance2:
  classifier->one(oclIsKindOf(Class))
```

- The instantiated classifier has the type PointPosition.

```
inv PointPositionInstance3:
  classifier->one(oclIsTypeOf(PointPosition))
```

- There is a mandatory slot pointId. The value is given by a LiteralPointId.

```
inv PointPositionInstance4:
  slot->one(s1 | s1.definingFeature.name->includes('pointId') and
                 s1.value->size()= 1 and
                 s1.value->first.oclIsTypeOf(LiteralPointId))
```

- There is a mandatory slot pointId. The value is taken from the enumeration RouteKind.

```
inv PointPositionInstance5:
  slot->one(s1 | s1.definingFeature.name->includes('pointState') and
                 s1.value->size()= 1 and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclIsTypeOf(EnumerationLiteral) and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclAsType(EnumerationLiteral).enumeration.name->
                     includes('RouteKind'))
```

- The value of slot pointId refers to a value of slot pointId or pointIdOpp of one SinglePointInstance, SlipPointInstance, or DoubleSlipPointInstance. In other words, the referenced point exists.

```
inv PointPositionInstance6:
  let id:Integer =
  slot->select(s | s.definingFeature.name->includes('pointId'))->
    asSequence->first().value->first.oclAsType(LiteralPointId).value
  in
    SinglePointInstance.allInstances->exists(p |
      p.slot->select(s | s.definingFeature.name->includes('pointId'))->
        asSequence->first().value->first().
          oclAsType(LiteralPointId).value = id) or
    SlipPointInstance.allInstances->exists(p |
      p.slot->select(s | s.definingFeature.name->includes('pointId'))->
        asSequence->first().value->first().
          oclAsType(LiteralPointId).value = id) or
    DoubleSlipPointInstance.allInstances->exists(p |
      p.slot->select(s | s.definingFeature.name->includes('pointId'))->
        asSequence->first().value->first().
          oclAsType(LiteralPointId).value = id) or
    DoubleSlipPointInstance.allInstances->exists(p |
      p.slot->select(s | s.definingFeature.name->includes('pointIdOpp'))->
        asSequence->first().value->first().
          oclAsType(LiteralPointId).value = id)
```

- The value of slot pointState refers to a valid position of the single point referenced by pointId. The valid positions of that point are given by the values of slots plus and minus.

```
inv PointPositionInstance7:
let id:Integer =
  slot->select(s | s.definingFeature.name->includes('pointId'))->
    asSequence->first().value->first.oclAsType(LiteralPointId).value
  in
  (
    SinglePointInstance.allInstances->exists(p |
      p.slot->select(s | s.definingFeature.name->includes('pointId'))->
        asSequence()->first().value->first().oclAsType(LiteralPointId).value =
```

```
      id)
    implies
    (
    let p:SinglePointInstance =
      SinglePointInstance.allInstances->select(i | i.slot->
        select(s | s.definingFeature.name->includes('pointId'))->
          asSequence()->first().value->first().oclAsType(LiteralPointId).value =
            id)->asSequence()->first()
    in
      p.slot->select(s |
      s.definingFeature.name->includes('plus'))->asSequence()->
        first().value->first().oclAsType(InstanceValue).
               instance.name->union(
        p.slot->select(s |
          s.definingFeature.name->includes('minus'))->asSequence()->
            first().value->first().oclAsType(InstanceValue).
              instance.name
        )->includes(self.slot->select(s |
        s.definingFeature.name->includes('pointState'))->
          asSequence->first().value->first().oclAsType(InstanceValue).
            instance.name->asSequence->first())
    )
  )
```

- The value of slot pointState refers to a valid position of the slip point referenced by pointId. The valid positions of that point are given by the values of slots plus and minus.

```
inv PointPositionInstance8:
let id:Integer =
  slot->select(s | s.definingFeature.name->includes('pointId'))->
    asSequence->first().value->first.oclAsType(LiteralPointId).value
  in
  (
    SlipPointInstance.allInstances->exists(p |
      p.slot->select(s | s.definingFeature.name->includes('pointId'))->
        asSequence()->first().value->first().oclAsType(LiteralPointId).value =
        id)
    implies
    (
    let p:SlipPointInstance =
      SlipPointInstance.allInstances->select(i | i.slot->
        select(s | s.definingFeature.name->includes('pointId'))->
          asSequence()->first().value->first().oclAsType(LiteralPointId).value =
            id)->asSequence()->first()
    in
      p.slot->select(s |
      s.definingFeature.name->includes('plus'))->asSequence()->
        first().value->first().oclAsType(InstanceValue).
               instance.name->union(
        p.slot->select(s |
          s.definingFeature.name->includes('minus'))->asSequence()->
            first().value->first().oclAsType(InstanceValue).
              instance.name
        )->includes(self.slot->select(s |
        s.definingFeature.name->includes('pointState'))->
          asSequence->first().value->first().oclAsType(InstanceValue).
            instance.name->asSequence->first())
    )
  )
```

- The value of slot pointState refers to a valid position of the first point of a double slip point

referenced by pointId. The valid positions of that point are given by the values of slots plus and minus.

```
inv PointPositionInstance9:
let id:Integer =
  slot->select(s | s.definingFeature.name->includes('pointId'))->
    asSequence->first().value->first.oclAsType(LiteralPointId).value
  in
  (
    DoubleSlipPointInstance.allInstances->exists(p |
      p.slot->select(s | s.definingFeature.name->includes('pointId'))->
        asSequence()->first().value->first().oclAsType(LiteralPointId).value =
        id)
    implies
    (
    let p:DoubleSlipPointInstance =
      DoubleSlipPointInstance.allInstances->select(i | i.slot->
        select(s | s.definingFeature.name->includes('pointId'))->
          asSequence()->first().value->first().oclAsType(LiteralPointId).value =
            id)->asSequence()->first()
    in
      p.slot->select(s |
      s.definingFeature.name->includes('plus'))->asSequence()->
        first().value->first().oclAsType(InstanceValue).
               instance.name->union(
        p.slot->select(s |
          s.definingFeature.name->includes('minus'))->asSequence()->
            first().value->first().oclAsType(InstanceValue).
              instance.name
        )->includes(self.slot->select(s |
        s.definingFeature.name->includes('pointState'))->
          asSequence->first().value->first().oclAsType(InstanceValue).
            instance.name->asSequence->first())
    )
  )
```

- The value of slot pointState refers to a valid position of the second point of a double slip point referenced by pointId. The valid positions of that point are given by the values of slots plusOpp and minusOpp.

```
inv PointPositionInstance10:
let id:Integer =
  slot->select(s | s.definingFeature.name->includes('pointId'))->
    asSequence->first().value->first.oclAsType(LiteralPointId).value
  in
  (
    DoubleSlipPointInstance.allInstances->exists(p |
      p.slot->select(s | s.definingFeature.name->includes('pointIdOpp'))->
        asSequence()->first().value->first().oclAsType(LiteralPointId).value =
        id)
    implies
    (
    let p:DoubleSlipPointInstance =
      DoubleSlipPointInstance.allInstances->select(i | i.slot->
        select(s | s.definingFeature.name->includes('pointIdOpp'))->
          asSequence()->first().value->first().oclAsType(LiteralPointId).value =
            id)->asSequence()->first()
    in
      p.slot->select(s |
      s.definingFeature.name->includes('plusOpp'))->asSequence()->
```

```
        first().value->first().oclAsType(InstanceValue).
               instance.name->union(
        p.slot->select(s |
          s.definingFeature.name->includes('minusOpp'))->asSequence()->
            first().value->first().oclAsType(InstanceValue).
              instance.name
        )->includes(self.slot->select(s |
        s.definingFeature.name->includes('pointState'))->
          asSequence->first().value->first().oclAsType(InstanceValue).
            instance.name->asSequence->first())
    )
  )
```

Semantics

Given by the transformation from a concrete network to a labeled transition system.

Notation

A PointPositionInstance is either depicted as a UML object or as a tuple, e.g. `(P5,STRAIGHT)`. The first item of the tuple is the point id, the second one the point position.

2.5.3 Route

Description

Routes define ways through the track network by sequences of sensors to be passed, the first signal setting needed to enter the route, the requested point positions, and the identifications of conflicting routes.

Associations

None.

Attributes

None.

Constraints

- There is a mandatory property routeId with type RouteId. The property is read-only and has the multiplicity 1.

```
inv Route1:
  ownedAttribute->one(a | a.name->includes('routeId') and
                          a.type.name->includes('RouteId') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
                          a.isReadOnly = true)
```

- There is a mandatory property routeDefinition with type SensorId. The property is read-only and ordered and has the multiplicity 1..*, where * is a natural greater than 1.

```
inv Route2:
  ownedAttribute->one(a | a.name->includes('routeDefinition') and
                          a.type.name->includes('SensorId') and
                          a.upperBound() >= 1 and
                          a.lowerBound() = 1 and
                          a.isOrdered = true and
                          a.isUnique = false and
                          a.isReadOnly = true)
```

- There is a mandatory property routeConflict with type RouteConflict. The property is read-only and has the multiplicity 0..*, where * is a natural greater than 1. The property is ordered and each value has to be unique. There is an outgoing association whose opposite end is a RouteConflict.

```
inv Route3:
  ownedAttribute->one(a | a.name->includes('routeConflict') and
                          a.upperBound() >= 1 and
                          a.lowerBound() = 0 and
                          a.isOrdered = true and
                          a.isUnique = true and
                          a.isReadOnly = true and
                          a.outgoingAssociation->size()=1 and
                          a.outgoingAssociation.memberEnd->size()=2 and
                          a.outgoingAssociation.memberEnd->excluding(a).
                            name->includes('route') and
                          a.outgoingAssociation.memberEnd->excluding(a)->
                            first.owningClass.oclIsTypeOf(RouteConflict))
```

- There is a mandatory property pointPos with type PointPosition. The property is read-only and has the multiplicity 0..*, where * is a natural greater than 1. The property is ordered and each value has to be unique. There is an outgoing association whose opposite end is a PointPosition.

```
inv Route4:
  ownedAttribute->one(a | a.name->includes('pointPos') and
                          a.upperBound() >= 1 and
                          a.lowerBound() = 0 and
                          a.isOrdered = true and
                          a.isUnique = true and
                          a.isReadOnly = true and
                          a.outgoingAssociation->size()=1 and
                          a.outgoingAssociation.memberEnd->size()=2 and
                          a.outgoingAssociation.memberEnd->excluding(a).
                            name->includes('route') and
                          a.outgoingAssociation.memberEnd->excluding(a)->
                            first.owningClass.oclIsTypeOf(PointPosition)
                         )
```

- There is a mandatory property signalSetting with type SignalSetting. The property is read-only and has the multiplicity 1. The property is ordered and each value has to be unique. There is an outgoing association whose opposite end is a SignalSetting.

```
inv Route5:
  ownedAttribute->one(a | a.name->includes('signalSetting') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
                          a.isReadOnly = true and
                          a.outgoingAssociation->size()=1 and
                          a.outgoingAssociation.memberEnd->size()=2 and
                          a.outgoingAssociation.memberEnd->excluding(a).
                            name->includes('route') and
                          a.outgoingAssociation.memberEnd->excluding(a)->
                            first.owningClass.oclIsTypeOf(SignalSetting)
                        )
```

Semantics

Routes own one property *routeDefinition* with type *SensorId* that gives a sequence of sensor identifications defining a route through the network. The signal setting at the beginning of a route is given

as property *signalSetting* with the stereotype *SignalSetting* as type. The point positions required for a route are defined by an ordered set named *pointPos* with type *PointPosition*. Also, *routeConflict*, a an ordered set with type *RouteConflict*, is needed as information to avoid conflicting routes. Routes have also a *routeId* of type *RouteId*.

Notation

Route is used in class diagrams using the UML class notation.

2.5.4 RouteInstance

Description

A RouteInstance is the instance of a route.

Associations

None.

Attributes

None.

Constraints

- There is one classifier instantiated.

```
inv RouteInstance1:
  classifier->size() = 1
```

- The instantiated classifier is a kind of Class.

```
inv RouteInstance2:
  classifier->one(oclIsKindOf(Class))
```

- The instantiated classifier has the type Route.

```
inv RouteInstance3:
  classifier->one(oclIsTypeOf(Route))
```

- There is a mandatory slot routeId. The value is given by a LiteralRouteId.

```
inv RouteInstance4:
  slot->one(s1 | s1.definingFeature.name->includes('routeId') and
                 s1.value->size()= 1 and
                 s1.value->first.oclIsTypeOf(LiteralRouteId))
```

- There is a mandatory slot routeDefinition. The value is given by a sequence of ast least 2 LiteralSensorIds.

```
inv RouteInstance5:
  slot->one(s1 | s1.definingFeature.name->includes('routeDefinition') and
                 s1.value->size() >= 2 and
                 s1.value->forAll(oclIsTypeOf(LiteralSensorId)))
```

- There is a mandatory slot signalSetting. The value is given by a SignalSettingInstance.

```
inv RouteInstance6:
  slot->one(s1 | s1.definingFeature.name->includes('signalSetting') and
                 s1.value->size() = 1 and
                 s1.value->first.oclIsTypeOf(InstanceValue) and
                 s1.value->first.oclAsType(InstanceValue).instance.
                   slot->select(s2 | s2.value->first().
                     oclAsType(InstanceValue).instance.
                       oclIsKindOf(SignalSettingInstance))->size() = 1)
```

- There is an optional slot routeConflict. If present, the value is given by a sequence of RouteConflictInstances.

```
inv RouteInstance7:
  slot->select(s1 | s1.definingFeature.name->includes('routeConflict'))->
    forAll(s2 | s2.value->size() = 1 and
                s2.value->forAll(oclIsTypeOf(InstanceValue)) and
                s2.value->forAll(i | i.oclAsType(InstanceValue).instance.
                  slot->select(s2 | s2.value->first().
                    oclAsType(InstanceValue).instance.
                      oclIsKindOf(RouteConflictInstance))->size() >= 1))
```

- There is an optional slot pointPosition. If present, the value is given by a sequence of PointPositionInstances.

```
inv RouteInstance8:
  slot->select(s1 | s1.definingFeature.name->includes('pointPos'))->
    forAll(s2 | s2.value->size() = 1 and
                s2.value->forAll(oclIsTypeOf(InstanceValue)) and
                s2.value->forAll(i | i.oclAsType(InstanceValue).instance.
                  slot->select(s2 | s2.value->first().
                    oclAsType(InstanceValue).instance.
                      oclIsKindOf(PointPositionInstance))->size() >= 1 ))
```

- The value of slot pointId refers to a value of slot pointId of one RouteInstance. In other words, the referenced route exists.

```
inv RouteInstance9:
  RouteInstance.allInstances->collect(slot)->asSet->flatten->
    select(s | s.definingFeature.name->includes('routeId'))->
      iterate(s:Slot;
              result:Set(LiteralRouteId) =
                oclEmpty(Set(LiteralRouteId)) |
              result->including(s.value->first.oclAsType(LiteralRouteId)))->
                isUnique(value)
```

- All values of slot routeDefinition refer to values of slot sensorId of one SensorInstance. In other words, the referenced sensors exist.

```
inv RouteInstance10:
  let i:Set(Integer) =
    slot->select(s | s.definingFeature.name->includes('routeDefinition'))->
      asSequence->first().value->
        iterate(v:ValueSpecification;
                result:Set(Integer)=oclEmpty(Set(Integer)) |
                result->including(v.oclAsType(LiteralSensorId).value))
  in
    i->forAll(id |
      SensorInstance.allInstances->exists(sens |
        sens.slot->select(s | s.definingFeature.name->includes('sensorId'))->
          asSequence->first().value->first().
            oclAsType(LiteralSensorId).value = id))
```

- Each pair of sensor ids given by the sequence routeDefinition are the entry and exit sensor of one track element. In other words, the sequence of sensor ids defines a complete route without gaps.

```
inv RouteInstance11:
  let i:Sequence(Integer) =
    slot->select(s | s.definingFeature.name->includes('routeDefinition'))->
      asSequence->first().value->
        iterate(v:ValueSpecification;
                result:Sequence(Integer)=oclEmpty(Sequence(Integer)) |
                result->including(v.oclAsType(LiteralSensorId).value))
  in
    (let sensors:Sequence(SensorInstance) =
      i->iterate(id:Integer;
                 result:Sequence(SensorInstance) =
                 oclEmpty(Sequence(SensorInstance)) |
                 result->including(SensorInstance.allInstances->select(
                   sens | sens.slot->select(s | s.definingFeature.name->
                     includes('sensorId'))->asSequence->first().value->
                      first().oclAsType(LiteralSensorId).value = id)->
                        asSequence->first()))
      in
        Sequence{1..sensors->size()-1}->iterate
          (nr:Integer;
           result:Boolean = true |
           if (sensors->at(nr).slot->
               select(s2 | s2.definingFeature.name->includes('entrySeg') or
                           s2.definingFeature.name->includes('entryCross') or
                            s2.definingFeature.name->includes('entryPoint'))->
                  asSequence->first.value.oclAsType(InstanceValue).instance.
                    slot->select(s3 | s3.definingFeature.oclAsType(Property).
                      owningClass.oclIsKindOf(TrackElement))->
                        first().value.oclAsType(InstanceValue).instance =
               sensors->at(nr+1).slot->
               select(s2 | s2.definingFeature.name->includes('exitSeg') or
                           s2.definingFeature.name->includes('exitCross') or
                           s2.definingFeature.name->includes('exitPoint'))->
                 asSequence->first.value.oclAsType(InstanceValue).instance.
                   slot->select(s3 | s3.definingFeature.oclAsType(Property).
                     owningClass.oclIsKindOf(TrackElement))->
                       first().value.oclAsType(InstanceValue).instance)
           then result
           else false
           endif)
    )
```

- Each route referenced by slot routeId in the sequence of route conflicts refers to a different route. In other words, it is not possible to define different route conflicts for one and the same route.

```
inv RouteInstance12:
    let i:Sequence(LiteralRouteId) =
      slot->select(s | s.definingFeature.name->includes('routeConflict'))->
        asSequence->first().value->first().oclAsType(InstanceValue).instance.
          slot->select(s2 | s2.definingFeature.oclAsType(Property).
            owningClass.oclIsKindOf(RouteConflict))->
              iterate(s:Slot;
                      result:Sequence(LiteralRouteId)=
                        oclEmpty(Sequence(LiteralRouteId)) |
                      result->including(
```

```
                    s.value->first().oclAsType(InstanceValue).instance.slot->
                      select(s2 |
                        s2.definingFeature.name->includes('routeId'))->
   asSequence->first().value->first().
                          oclAsType(LiteralRouteId)
                )
            )
    in
    i->isUnique(value)
```

- No routeId defined by a route conflict refers to the value of route id of this route instance. In other words, it is not possible for a route to have a conflict with itself.

```
inv RouteInstance13:
    let i:Sequence(Integer) =
      slot->select(s | s.definingFeature.name->includes('routeConflict'))->
        asSequence->first().value->first().oclAsType(InstanceValue).instance.
          slot->select(s2 | s2.definingFeature.oclAsType(Property).
            owningClass.oclIsKindOf(RouteConflict))->
              iterate(s:Slot;
                      result:Sequence(Integer)=oclEmpty(Sequence(Integer)) |
                      result->including(
                        s.value->first().oclAsType(InstanceValue).instance.slot->
                          select(s2 |
                            s2.definingFeature.name->includes('routeId'))->
                              asSequence->first().value->first().
                                oclAsType(LiteralRouteId).value
                )
            )
    in
    i->excludes(self.slot->select(s |
       s.definingFeature.name->includes('routeId'))->asSequence->first().value->first().
         oclAsType(LiteralRouteId).value)
```

- The value of slot sigId of the defined signalSetting refers to a value of signalId of one signal instance. This signal instance must be located at a sensor instance whose sensorId is contained in the routeDefinition of this route instance. In other words, the referenced signal of the signal setting must be located at a sensor that belongs to the defined route.

```
inv RouteInstance14:
    let r:Sequence(Integer) =
    slot->select(s | s.definingFeature.name->includes('routeDefinition'))->
      asSequence->first().value->
        iterate(v:ValueSpecification;
                result:Sequence(Integer)=oclEmpty(Sequence(Integer)) |
                result->including(v.oclAsType(LiteralSensorId).value))
    in
    (
      let i:Integer =
        slot->select(s | s.definingFeature.name->includes('signalSetting'))->
          asSequence->first().value->first().oclAsType(InstanceValue).instance.slot->
            select(s2 | s2.definingFeature.oclAsType(Property).owningClass.
              oclIsKindOf(SignalSetting))->
                asSequence->first().value->first().oclAsType(InstanceValue).
                  instance.slot->select(s2 |
                    s2.definingFeature.name->includes('sigId'))->asSequence->
                      first().value->first().oclAsType(LiteralSignalId).value
      in
      (
        SensorInstance.allInstances->one(sens |
```

```
        r->includes(sens.slot->select(s |
        s.definingFeature.name->includes('sensorId'))->asSequence->first().
          value->first().oclAsType(LiteralSensorId).value)
        and
        sens.slot->select(s | s.definingFeature.name->includes('signal'))->
          asSequence->first().value->first().oclAsType(InstanceValue).
            instance.slot->select(s2 | s2.definingFeature.oclAsType(Property).
             owningClass.oclIsKindOf(Sensor))->asSequence->first().value->first().
               oclAsType(InstanceValue).instance.slot-select(s3 |
                 s3.definingFeature.name->includes('signalId'))->asSequence->
                   first().value->first().oclAsType(LiteralSignalId).value = i
        )
    )
  )
```

- Each point referenced by slot pointId in the sequence of point positions refers to a different point. In other words, it is not possible to define different point positions for one and the same point.

```
inv RouteInstance15:
    let i:Sequence(LiteralPointId) =
      slot->select(s | s.definingFeature.name->includes('pointPos'))->
        asSequence->first().value->first().oclAsType(InstanceValue).instance.
          slot->select(s2 | s2.definingFeature.oclAsType(Property).
            owningClass.oclIsKindOf(PointPosition))->
              iterate(s:Slot;
                      result:Sequence(LiteralPointId)=
                        oclEmpty(Sequence(LiteralPointId)) |
                      result->including(
                        s.value->first().oclAsType(InstanceValue).instance.slot->
                          select(s2 |
                             s2.definingFeature.name->includes('pointId'))->
                               asSequence->first().value->first().
                                 oclAsType(LiteralPointId)
                     )
               )
    in
    i->isUnique(value)
```

- The value of slot pointId of the defined pointPosition refers to a value of pointId of one point instance. This point instance must be located at a sensor instance whose sensorId is contained in the routeDefinition of this route instance. In other words, the referenced points of the sequence of point positions must be located at a sensor that belongs to the defined route.

```
inv RouteInstance16:
let r:Sequence(Integer) =
    slot->select(s | s.definingFeature.name->includes('routeDefinition'))->
      asSequence->first().value->
        iterate(v:ValueSpecification;
                result:Sequence(Integer)=oclEmpty(Sequence(Integer)) |
                result->including(v.oclAsType(LiteralSensorId).value))
    in
    (
      let ids:Sequence(Integer) =
        slot->select(s | s.definingFeature.name->includes('pointPos'))->
          asSequence->first().value->first().oclAsType(InstanceValue).
            instance.slot->select(s2 |
            s2.definingFeature.oclAsType(Property).owningClass.
              oclIsKindOf(PointPosition))->
                iterate(s:Slot;
```

```
                    result:Sequence(Integer) =
                      oclEmpty(Sequence(Integer)) |
                    result->including(s.value->first().oclAsType(InstanceValue).
                      instance.slot->select(s2 |
                        s2.definingFeature.name->includes('pointId'))->
                          asSequence->first().value->first().
                            oclAsType(LiteralPointId).value))
  in
  (
    ids->forAll(i |
    SensorInstance.allInstances->one(sens |
      r->includes(sens.slot->select(s |
      s.definingFeature.name->includes('sensorId'))->asSequence->first().value->
        first().oclAsType(LiteralSensorId).value)
      and
      sens.slot->select(s | s.definingFeature.name->includes('entryPoint'))->
        asSequence->first().value->
          first().oclAsType(InstanceValue).instance.slot->select(s2 |
            s2.definingFeature.oclAsType(Property).owningClass.
              oclIsKindOf(Point))->asSequence->first().value->first().
                oclAsType(InstanceValue).instance.slot->select(s3 |
                  s3.definingFeature.name->includes('pointId'))->
    asSequence->first().value->first().
                      oclAsType(LiteralPointId).value = i
    )
    )
  )
)
```

Semantics

Given by the transformation from a concrete network to a labeled transition system.

Notation

A RouteInstance is either depicted as a UML object or in table notation.

As an object, the three sequences *routeDefinition*, *pointPos*, and *routeConflict* are denoted as sequences (see Fig. 2.36), e.g. `pointPos=<(P1,LEFT),(P2,RIGHT),(P5,STRAIGHT)>`.

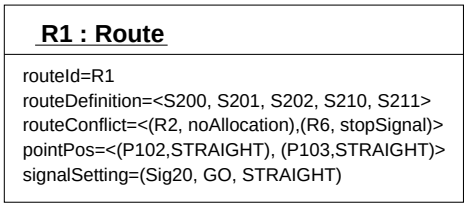

Figure 2.36: Route instance in object notation

In table notation, there are four tables: one describing all routes (see Tab. 2.1), one describing all point positions for each route (see Tab. 2.2), one describing all signal settings (see Tab. 2.3), and one describing all conflicts (see Tab. 2.4).

2.5.5 RouteConflict

Description

Routes may have conflicts, either because their point and signal settings are incompatible or because they travel on the same segments.

Route	Sensor Sequence
R1	<S200,S201,S202,S210,S211>
R2	<S200,S203,S300,S228,S250,S251>
R3	<S220,S221,S222,S299,S229,S230,S231>
R4	<S220,S221,S223,S250,S251>
R5	<S240,S241,S243,S230,S231>
R6	<S240,S241,S242,S301,S210,S211>

Table 2.1: Route definitions

Route	P100	P101	P102	P103	P118	P119
R1			STRAIGHT	STRAIGHT		
R2		STRAIGHT	LEFT			
R3					STRAIGHT	STRAIGHT
R4		LEFT			RIGHT	
R5	RIGHT					LEFT
R6	STRAIGHT			RIGHT		

Table 2.2: Point positions

Route	Signal	Setting
R1	Sig20	(GO, STRAIGHT)
R2	Sig20	(GO, LEFT)
R3	Sig21	(GO, STRAIGHT)
R4	Sig21	(GO, RIGHT)
R5	Sig22	(GO, RIGHT)
R6	Sig22	(GO, STRAIGHT)

Table 2.3: Signal settings

Route	R1	R2	R3	R4	R5	R6
1		noAllocation				stopSignal
2	noAllocation		stopSignal	stopSignal		stopSignal
3		stopSignal		no Allocation	stopSignal	stopSignal
4		stopSignal	noAllocation			
5			stopSignal			noAllocation
6	stopSignal	stopSignal	stopSignal		noAllocation	

Table 2.4: Route conflicts

Associations

None.

Attributes

None.

Constraints

- There is a mandatory property routeId with type RouteId. The property is read-only and has the multiplicity 1.

```
inv RouteConflict1:
  ownedAttribute->one(a | a.name->includes('routeId') and
                          a.type.name->includes('RouteId') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
```

```
                          a.isReadOnly = true)
```

- There is a mandatory property kind with type RouteConflictKind. The property is read-only and has the multiplicity 1.

```
inv RouteConflict2:
  ownedAttribute->one(a | a.name->includes('kind') and
                          a.type.name->includes('RouteConflictKind') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
                          a.isReadOnly = true)
```

Semantics

RouteConflict is defined by the *routeId* with type *RouteId* of a conflicting route and the *kind* that is a *RouteConflictKind*.

Notation

RouteConflict is used in class diagrams using the UML class notation.

2.5.6 RouteConflictInstance

Description

A RouteConflictInstance is the instance of a route conflict.

Associations

None.

Attributes

None.

Constraints

- There is one classifier instantiated.

```
inv RouteConflictInstance1:
  classifier->size() = 1
```

- The instantiated classifier is a kind of Class.

```
inv RouteConflictInstance2:
  classifier->one(oclIsKindOf(Class))
```

- The instantiated classifier has the type RouteConflict.

```
inv RouteConflictInstance3:
  classifier->one(oclIsTypeOf(RouteConflict))
```

- There is a mandatory slot routeId. The value is given by a LiteralRouteId.

```
inv RouteConflictInstance4:
  slot->one(s1 | s1.definingFeature.name->includes('routeId') and
                    s1.value->size() = 1 and
                    s1.value->first.oclIsTypeOf(LiteralRouteId))
```

- There is a mandatory slot kind. The value is taken from the enumeration RouteConflict Kind.

```
inv RouteConflictInstance5:
  slot->one(s1 | s1.definingFeature.name->includes('kind') and
                    s1.value->size() = 1 and
                    s1.value->first().oclAsType(InstanceValue).instance.
                      oclIsTypeOf(EnumerationLiteral) and
                    s1.value->first().oclAsType(InstanceValue).instance.
                       oclAsType(EnumerationLiteral).enumeration.name->
                       includes('RouteConflictKind'))
```

- The value of slot routeId refers to a value of slot routeId of one RouteInstance. In other words, the referenced route exists.

```
inv RouteConflictInstance6:
  let id:Set(Integer) =
  slot->select(s | s.definingFeature.name->includes('routeId'))->
    iterate(s:Slot; result:Set(Integer) = oclEmpty(Set(Integer)) |
      result->including(s.value->first().oclAsType(LiteralRouteId).value))
  in
    id->forAll(i |
      RouteInstance.allInstances->exists(r |
        r.slot->select(s | s.definingFeature.name->includes('routeId'))->
          asSequence->first().value->first().
            oclAsType(LiteralRouteId).value = i))
```

Semantics

Given by the transformation from a concrete network to a labeled transition system.

Notation

A RouteConflictInstance is either depicted as a UML object or as a tuple, e.g. `(R1,noAllocationPossible)`. The first item of the tupel is the route id, the second one the conflict kind.

2.5.7 RouteConflictKind

Description

RouteConflictKind describes the two kind of possible conflicts between routes: either the routes cannot be allocated at the same time due to incompatible point or signal states, or the routes can be allocated at the same time but only one train may proceed while all other trains on the conflicting routes have to wait due to signals with state *STOP*.

Semantics

The literals defined by RouteConflictKind are used as values of properties with type *RouteConflictKind*. These literals are:

- `noAllocation`
- `stopSignal`

Notation

The defined literals are used as values of properties with type *RouteConflictKind*, e.g. `kind = noAllocation`.

2.5.8 SignalSetting

Description

SignalSettings model required signal settings, i.e. required states of signals.

Associations

None.

Attributes

None.

Constraints

- There is a mandatory property sigId with type SignalId. The property is read-only and has the multiplicity 1.

```
inv SignalSetting1:
  ownedAttribute->one(a | a.name->includes('sigId') and
                          a.type.name->includes('SignalId') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
                          a.isReadOnly = true)
```

- There is a mandatory property sigState with type PermissionKind. The property is read-only and has the multiplicity 1.

```
inv SignalSetting2:
  ownedAttribute->one(a | a.name->includes('sigState') and
                          a.type.name->includes('PermissionKind') and
                          a.upperBound() = 1 and
                          a.lowerBound() = 1 and
                          a.isReadOnly = true)
```

- There is an optional property dirState with type RouteKind. If present, the property is read-only and has the multiplicity 0..1 or 1.

```
inv SignalSetting3:
  (ownedAttribute->one(a | a.name->includes('dirState') and
                           a.type.name->includes('RouteKind') and
                           a.upperBound()=1 and
                           a.lowerBound() >= 0 and
                           a.isReadOnly = true)) or
  (not ownedAttribute->exists(a | a.name->includes('dirState')))
```

Semantics

SignalSetting has three properties, the *sigId* of a signal that must be set to enter a route, the required signal state *sigState*, and optionally the required direction *dirState*.

Notation

SignalSetting is used in class diagrams using the UML class notation.

2.5.9 SignalSettingInstance

Description

A SignalSettingInstance is the instance of a signal setting.

Associations

None.

Attributes

None.

Constraints

- There is one classifier instantiated.

```
inv SignalSettingInstance1:
  classifier->size() = 1
```

- The instantiated classifier is a kind of Class.

```
inv SignalSettingInstance2:
  classifier->one(oclIsKindOf(Class))
```

- The instantiated classifier has the type SignalSetting.

```
inv SignalSettingInstance3:
  classifier->one(oclIsTypeOf(SignalSetting))
```

- There is a mandatory slot sigId. The value is given by a LiteralSignalId.

```
inv SignalSettingInstance4:
  slot->one(s1 | s1.definingFeature.name->includes('sigId') and
                 s1.value->size()= 1 and
                 s1.value->first.oclIsTypeOf(LiteralSignalId))
```

- There is a mandatory slot sigState. The value is taken from the enumeration PermissionKind.

```
inv SignalSettingInstance5:
  slot->one(s1 | s1.definingFeature.name->includes('sigState') and
                 s1.value->size()= 1 and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclIsTypeOf(EnumerationLiteral) and
                 s1.value->first().oclAsType(InstanceValue).instance.
                   oclAsType(EnumerationLiteral).enumeration.name->
                     includes('PermissionKind'))
```

- There is an optional slot dirState.If present,the value is taken from the enumeration RouteKind.

```
inv SignalSettingInstance6:
  slot->select(s1 | s1.definingFeature.name->includes('dirState'))->
    forAll(s2 | s2.value->size()= 1 and
                s2.value->first().oclAsType(InstanceValue).instance.
                  oclIsTypeOf(EnumerationLiteral) and
                s2.value->first().oclAsType(InstanceValue).instance.
                  oclAsType(EnumerationLiteral).enumeration.name->
                    includes('RouteKind')) and
  slot->select(s1 | s1.definingFeature.name->includes('dirState'))->size <= 1
```

- The value of slot sigId refers to a value of slot signalId of one SignalInstance. In other words, the referenced signal exists.

```
inv SignalSettingInstance7:
  let id:Integer =
  slot->select(s | s.definingFeature.name->includes('sigId'))->
    asSequence->first().value->first.oclAsType(LiteralSignalId).value
  in
    SignalInstance.allInstances->exists(sig |
      sig.slot->select(s | s.definingFeature.name->includes('signalId'))->
        asSequence->first().value->first().
          oclAsType(LiteralSignalId).value = id)
```

- The value of slot sigDir is STRAIGHT if the signal instance referenced by sigId is located at a segment instance.

```
inv SignalSettingInstance8:
  let id:Integer =
  slot->select(s | s.definingFeature.name->includes('sigId'))->
    asSequence->first().value->first.oclAsType(LiteralSignalId).value
  in
    SignalInstance.allInstances->exists(sig |
      sig.slot->select(s | s.definingFeature.name->includes('signalId'))->
        asSequence->first().value->first().
          oclAsType(LiteralSignalId).value = id and
      (sig.slot->select(s | s.definingFeature.name->includes('sensor'))->
        asSequence->first().value->first.oclAsType(InstanceValue).instance.
          slot->select(s | s.definingFeature.oclAsType(Property).owningClass.
            oclIsTypeOf(Signal))->asSequence->first().value->
              first.oclAsType(InstanceValue).instance.slot->
                select(s | s.definingFeature.name->includes('entrySeg'))->
                  size()=1))
       implies
         self.slot->select(s | s.definingFeature.name->
           includes('sigDir'))->asSequence->first().value->first()->
             oclAsType(InstanceValue).instance.name->includes('STRAIGHT')
```

- The value of slot sigDir is STRAIGHT if the signal instance referenced by sigId is located at a crossing instance.

```
inv SignalSettingInstance9:
  let id:Integer =
  slot->select(s | s.definingFeature.name->includes('sigId'))->
    asSequence->first().value->first.oclAsType(LiteralSignalId).value
  in
    SignalInstance.allInstances->exists(sig |
      sig.slot->select(s | s.definingFeature.name->includes('signalId'))->
        asSequence->first().value->first().
          oclAsType(LiteralSignalId).value = id and
      (sig.slot->select(s | s.definingFeature.name->includes('sensor'))->
        asSequence->first().value->first.oclAsType(InstanceValue).instance.
          slot->select(s | s.definingFeature.oclAsType(Property).owningClass.
            oclIsTypeOf(Signal))->asSequence->first().value->
              first.oclAsType(InstanceValue).instance.slot->
                select(s | s.definingFeature.name->includes('entryCross'))->
                  size()=1))
       implies
         self.slot->select(s | s.definingFeature.name->
           includes('sigDir'))->asSequence->first().value->first()->
             oclAsType(InstanceValue).instance.name->includes('STRAIGHT')
```

- If the signal instance referenced by sigId is located at slot e1Entry of a single point instance, the value of slot sigDir is consistent with the possible directions at that point that are given by slot plus and minus of that single point instance.

```
inv SignalSettingInstance10:
  let id:Integer =
  slot->select(s | s.definingFeature.name->includes('sigId'))->
    asSequence->first().value->first.oclAsType(LiteralSignalId).value
  in
    SignalInstance.allInstances->exists(sig |
      sig.slot->select(s | s.definingFeature.name->includes('signalId'))->
        asSequence->first().value->first().
          oclAsType(LiteralSignalId).value = id and
      (sig.slot->select(s | s.definingFeature.name->includes('sensor'))->
        asSequence->first().value->first.oclAsType(InstanceValue).instance.
          slot->select(s | s.definingFeature.oclAsType(Property).owningClass.
            oclIsTypeOf(Signal))->asSequence->first().value->
              first.oclAsType(InstanceValue).instance.slot->
                select(s | s.definingFeature.name->includes('entryPoint'))->
                  asSequence->first().value->first().oclAsType(InstanceValue).
                    instance.slot->select(s | s.definingFeature.
                    oclAsType(Property).owningClass.oclIsTypeOf(SinglePoint))->
                     asSequence->first().value->first.oclAsType(InstanceValue).
                      instance.slot->select(s |
                       s.definingFeature.name->includes('e1Entry'))->size()=1))
     implies
      SinglePointInstance.allInstances->exists(p | p.slot->select(s |
      s.definingFeature.name->includes('e1Entry'))->asSequence->first().
        value.oclAsType(InstanceValue).instance.slot->select(s |
          s.definingFeature.oclAsType(Property).owningClass.
            oclIsKindOf(Sensor))->first().value->first().
              oclAsType(InstanceValue).instance.slot->select(s |
                s.definingFeature.name->includes('signal'))->asSequence->
                  first().value->first().oclAsType(InstanceValue).
                    instance.slot->select(s |
                       s.definingFeature.oclAsType(Property).owningClass.
                         oclIsKindOf(Sensor))->asSequence->first().value->
                           first().oclAsType(InstanceValue).instance.slot->
                             select(s | s.definingFeature.name->
                               includes('signalId'))->asSequence->first().
                                 value.oclAsType(LiteralSignalId).value->first() = id
     and
     p.slot->select(s | s.definingFeature.name->includes('plus'))->
       asSequence->first().value->first().oclAsType(InstanceValue).
         instance.name->union(
           p.slot->select( s | s.definingFeature.name->includes('minus'))->
             asSequence->first().value->first().oclAsType(InstanceValue).
               instance.name)->includes(
                 self.slot->select(s | s.definingFeature.name->
                   includes('sigDir'))->asSequence->first().value->
                     first()->oclAsType(InstanceValue).instance.name->
                       asSequence->first()))
```

- If the signal instance referenced by sigId is located at slot e2Entry or e3Entry of a single point instance, the value of slot sigDir is STRAIGHT.

```
inv SignalSettingInstance11:
  let id:Integer =
  slot->select(s | s.definingFeature.name->includes('sigId'))->
    asSequence->first().value->first.oclAsType(LiteralSignalId).value
  in
    SignalInstance.allInstances->exists(sig |
      sig.slot->select(s | s.definingFeature.name->includes('signalId'))->
        asSequence->first().value->first().
```

```
        oclAsType(LiteralSignalId).value = id and
     (sig.slot->select(s | s.definingFeature.name->includes('sensor'))->
       asSequence->first().value->first.oclAsType(InstanceValue).instance.
         slot->select(s | s.definingFeature.oclAsType(Property).owningClass.
           oclIsTypeOf(Signal))->asSequence->first().value->
             first.oclAsType(InstanceValue).instance.slot->
               select(s | s.definingFeature.name->includes('entryPoint'))->
                 asSequence->first().value->first().oclAsType(InstanceValue).
                   instance.slot->select(s | s.definingFeature.
                   oclAsType(Property).owningClass.oclIsTypeOf(SinglePoint))->
                    asSequence->first().value->first.oclAsType(InstanceValue).
                     instance.slot->select(s |
                      s.definingFeature.name->includes('e2Entry') or
                      s.definingFeature.name->includes('e3Entry'))->size()=1))
   implies
    self.slot->select(s | s.definingFeature.name->
         includes('sigDir'))->asSequence->first().value->first()->
           oclAsType(InstanceValue).instance.name->includes('STRAIGHT')
```

- If the signal instance referenced by sigId is located at slot e1Entry of a slip point instance, the value of slot sigDir is consistent with the possible directions at that point that are given by slot plus and minus of that slip point instance.

```
inv SignalSettingInstance12:
  let id:Integer =
  slot->select(s | s.definingFeature.name->includes('sigId'))->
    asSequence->first().value->first.oclAsType(LiteralSignalId).value
  in
    SignalInstance.allInstances->exists(sig |
      sig.slot->select(s | s.definingFeature.name->includes('signalId'))->
        asSequence->first().value->first().
          oclAsType(LiteralSignalId).value = id and
      (sig.slot->select(s | s.definingFeature.name->includes('sensor'))->
        asSequence->first().value->first.oclAsType(InstanceValue).instance.
          slot->select(s | s.definingFeature.oclAsType(Property).owningClass.
            oclIsTypeOf(Signal))->asSequence->first().value->
              first.oclAsType(InstanceValue).instance.slot->
                select(s | s.definingFeature.name->includes('entrySlPoint'))->
                  asSequence->first().value->first().oclAsType(InstanceValue).
                    instance.slot->select(s | s.definingFeature.
                    oclAsType(Property).owningClass.oclIsTypeOf(SlipPoint))->
                     asSequence->first().value->first.oclAsType(InstanceValue).
                      instance.slot->select(s |
                       s.definingFeature.name->includes('e1Entry'))->size()=1))
     implies
      SlipPointInstance.allInstances->exists(p | p.slot->select(s |
      s.definingFeature.name->includes('e1Entry'))->asSequence->first().
        value.oclAsType(InstanceValue).instance.slot->select(s |
          s.definingFeature.oclAsType(Property).owningClass.
            oclIsKindOf(Sensor))->first().value->first().
              oclAsType(InstanceValue).instance.slot->select(s |
                s.definingFeature.name->includes('signal'))->asSequence->
                  first().value->first().oclAsType(InstanceValue).
                    instance.slot->select(s |
                       s.definingFeature.oclAsType(Property).owningClass.
                         oclIsKindOf(Sensor))->asSequence->first().value->
                           first().oclAsType(InstanceValue).instance.slot->
                             select(s | s.definingFeature.name->
                               includes('signalId'))->asSequence->first().
                                 value.oclAsType(LiteralSignalId).value->first() = id
```

```
    and
    p.slot->select(s | s.definingFeature.name->includes('plus'))->
      asSequence->first().value->first().oclAsType(InstanceValue).
        instance.name->union(
          p.slot->select( s | s.definingFeature.name->includes('minus'))->
            asSequence->first().value->first().oclAsType(InstanceValue).
              instance.name)->includes(
                self.slot->select(s | s.definingFeature.name->
                  includes('sigDir'))->asSequence->first().value->
                    first()->oclAsType(InstanceValue).instance.name->
                      asSequence->first()))
```

- If the signal instance referenced by sigId is located at slot e2Entry or e3Entry of a single point instance, the value of slot sigDir is STRAIGHT.

```
inv SignalSettingInstance13:
  let id:Integer =
  slot->select(s | s.definingFeature.name->includes('sigId'))->
    asSequence->first().value->first.oclAsType(LiteralSignalId).value
  in
    SignalInstance.allInstances->exists(sig |
      sig.slot->select(s | s.definingFeature.name->includes('signalId'))->
        asSequence->first().value->first().
          oclAsType(LiteralSignalId).value = id and
      (sig.slot->select(s | s.definingFeature.name->includes('sensor'))->
        asSequence->first().value->first.oclAsType(InstanceValue).instance.
          slot->select(s | s.definingFeature.oclAsType(Property).owningClass.
            oclIsTypeOf(Signal))->asSequence->first().value->
              first.oclAsType(InstanceValue).instance.slot->
                select(s | s.definingFeature.name->includes('entrySlPoint'))->
                  asSequence->first().value->first().oclAsType(InstanceValue).
                    instance.slot->select(s | s.definingFeature.
                    oclAsType(Property).owningClass.oclIsTypeOf(SlipPoint))->
                     asSequence->first().value->first.oclAsType(InstanceValue).
                      instance.slot->select(s |
                       s.definingFeature.name->includes('e2Entry') or
                       s.definingFeature.name->includes('e3Entry'))->size()=1))
     implies
      self.slot->select(s | s.definingFeature.name->
           includes('sigDir'))->asSequence->first().value->first()->
             oclAsType(InstanceValue).instance.name->includes('STRAIGHT')
```

- If the signal instance referenced by sigId is located at slot e4Entry of a slip point instance, the value of slot sigDir is consistent with the possible directions at that point that are given by slot plus and minus of that slip point instance. If tbe plus and minus positions of that point allow for directions RIGHT and STRAIGHT, sigDir may be LEFT or STRAIGHT as the slip point is entered from the opposite end. Vice versa, if the plus and minus positions allow for directions LEFT and STRAIGHT, sig DIR may be RIGHT or STRAIGHT.

```
inv SignalSettingInstance14:
  let id:Integer =
  slot->select(s | s.definingFeature.name->includes('sigId'))->
    asSequence->first().value->first.oclAsType(LiteralSignalId).value
  in
    SignalInstance.allInstances->exists(sig |
      sig.slot->select(s | s.definingFeature.name->includes('signalId'))->
        asSequence->first().value->first().
          oclAsType(LiteralSignalId).value = id and
      (sig.slot->select(s | s.definingFeature.name->includes('sensor'))->
        asSequence->first().value->first.oclAsType(InstanceValue).instance.
```

```
        slot->select(s | s.definingFeature.oclAsType(Property).owningClass.
          oclIsTypeOf(Signal))->asSequence->first().value->
            first.oclAsType(InstanceValue).instance.slot->
              select(s | s.definingFeature.name->includes('entrySlPoint'))->
                asSequence->first().value->first().oclAsType(InstanceValue).
                  instance.slot->select(s | s.definingFeature.
                  oclAsType(Property).owningClass.oclIsTypeOf(SlipPoint))->
                   asSequence->first().value->first.oclAsType(InstanceValue).
                    instance.slot->select(s |
                     s.definingFeature.name->includes('e4Entry'))->size()=1))
    implies
     SlipPointInstance.allInstances->exists(p | p.slot->select(s |
     s.definingFeature.name->includes('e4Entry'))->asSequence->first().
       value.oclAsType(InstanceValue).instance.slot->select(s |
         s.definingFeature.oclAsType(Property).owningClass.
           oclIsKindOf(Sensor))->first().value->first().
             oclAsType(InstanceValue).instance.slot->select(s |
               s.definingFeature.name->includes('signal'))->asSequence->
                 first().value->first().oclAsType(InstanceValue).
                   instance.slot->select(s |
                      s.definingFeature.oclAsType(Property).owningClass.
                        oclIsKindOf(Sensor))->asSequence->first().value->
                          first().oclAsType(InstanceValue).instance.slot->
                            select(s | s.definingFeature.name->
                              includes('signalId'))->asSequence->first().
                                value.oclAsType(LiteralSignalId).value->first() = id
    and
    p.slot->select(s | s.definingFeature.name->includes('plus'))->
      asSequence->first().value->first().oclAsType(InstanceValue).
        instance.name->union(
          p.slot->select( s | s.definingFeature.name->includes('minus'))->
            asSequence->first().value->first().oclAsType(InstanceValue).
              instance.name)->excluding('STRAIGHT')->excludes(
                self.slot->select(s | s.definingFeature.name->
                  includes('sigDir'))->asSequence->first().value->
                    first()->oclAsType(InstanceValue).instance.name->
                      asSequence->first()))
```

- If the signal instance referenced by sigId is located at slot e1Entry of a double slip point instance, the value of slot sigDir is consistent with the possible directions at that point that are given by slot plus and minus of that double slip point instance.

```
inv SignalSettingInstance15:
  let id:Integer =
  slot->select(s | s.definingFeature.name->includes('sigId'))->
    asSequence->first().value->first.oclAsType(LiteralSignalId).value
  in
    SignalInstance.allInstances->exists(sig |
      sig.slot->select(s | s.definingFeature.name->includes('signalId'))->
        asSequence->first().value->first().
          oclAsType(LiteralSignalId).value = id and
      (sig.slot->select(s | s.definingFeature.name->includes('sensor'))->
        asSequence->first().value->first.oclAsType(InstanceValue).instance.
          slot->select(s | s.definingFeature.oclAsType(Property).owningClass.
            oclIsTypeOf(Signal))->asSequence->first().value->
              first.oclAsType(InstanceValue).instance.slot->
                select(s | s.definingFeature.name->includes('entryDbSlPoint'))->
                  asSequence->first().value->first().oclAsType(InstanceValue).
                    instance.slot->select(s | s.definingFeature.
                    oclAsType(Property).owningClass.oclIsTypeOf(DoubleSlipPoint))->
```

```
                  asSequence->first().value->first.oclAsType(InstanceValue).
                   instance.slot->select(s |
                    s.definingFeature.name->includes('e1Entry'))->size()=1))
    implies
     DoubleSlipPointInstance.allInstances->exists(p | p.slot->select(s |
     s.definingFeature.name->includes('e1Entry'))->asSequence->first().
       value.oclAsType(InstanceValue).instance.slot->select(s |
         s.definingFeature.oclAsType(Property).owningClass.
           oclIsKindOf(Sensor))->first().value->first().
             oclAsType(InstanceValue).instance.slot->select(s |
               s.definingFeature.name->includes('signal'))->asSequence->
                 first().value->first().oclAsType(InstanceValue).
                   instance.slot->select(s |
                     s.definingFeature.oclAsType(Property).owningClass.
                       oclIsKindOf(Sensor))->asSequence->first().value->
                         first().oclAsType(InstanceValue).instance.slot->
                           select(s | s.definingFeature.name->
                             includes('signalId'))->asSequence->first().
                               value.oclAsType(LiteralSignalId).value->first() = id
    and
    p.slot->select(s | s.definingFeature.name->includes('plus'))->
      asSequence->first().value->first().oclAsType(InstanceValue).
        instance.name->union(
          p.slot->select( s | s.definingFeature.name->includes('minus'))->
            asSequence->first().value->first().oclAsType(InstanceValue).
              instance.name)->includes(
                self.slot->select(s | s.definingFeature.name->
                  includes('sigDir'))->asSequence->first().value->
                    first()->oclAsType(InstanceValue).instance.name->
                      asSequence->first()))
```

- If the signal instance referenced by sigId is located at slot e2Entry of a double slip point instance, the value of slot sigDir is consistent with the possible directions at that point that are given by slot plusOpp and minusOpp of that double slip point instance.

```
inv SignalSettingInstance16:
  let id:Integer =
  slot->select(s | s.definingFeature.name->includes('sigId'))->
    asSequence->first().value->first.oclAsType(LiteralSignalId).value
  in
    SignalInstance.allInstances->exists(sig |
      sig.slot->select(s | s.definingFeature.name->includes('signalId'))->
        asSequence->first().value->first().
          oclAsType(LiteralSignalId).value = id and
      (sig.slot->select(s | s.definingFeature.name->includes('sensor'))->
        asSequence->first().value->first.oclAsType(InstanceValue).instance.
          slot->select(s | s.definingFeature.oclAsType(Property).owningClass.
            oclIsTypeOf(Signal))->asSequence->first().value->
              first.oclAsType(InstanceValue).instance.slot->
                select(s | s.definingFeature.name->includes('entryDbSlPoint'))->
                  asSequence->first().value->first().oclAsType(InstanceValue).
                    instance.slot->select(s | s.definingFeature.
                    oclAsType(Property).owningClass.oclIsTypeOf(DoubleSlipPoint))->
                     asSequence->first().value->first.oclAsType(InstanceValue).
                      instance.slot->select(s |
                       s.definingFeature.name->includes('e2Entry'))->size()=1))
    implies
     DoubleSlipPointInstance.allInstances->exists(p | p.slot->select(s |
     s.definingFeature.name->includes('e2Entry'))->asSequence->first().
       value.oclAsType(InstanceValue).instance.slot->select(s |
```

```
        s.definingFeature.oclAsType(Property).owningClass.
          oclIsKindOf(Sensor))->first().value->first().
            oclAsType(InstanceValue).instance.slot->select(s |
              s.definingFeature.name->includes('signal'))->asSequence->
                first().value->first().oclAsType(InstanceValue).
                  instance.slot->select(s |
                    s.definingFeature.oclAsType(Property).owningClass.
                      oclIsKindOf(Sensor))->asSequence->first().value->
                        first().oclAsType(InstanceValue).instance.slot->
                          select(s | s.definingFeature.name->
                            includes('signalId'))->asSequence->first().
                              value.oclAsType(LiteralSignalId).value->first() = id
    and
    p.slot->select(s | s.definingFeature.name->includes('plusOpp'))->
      asSequence->first().value->first().oclAsType(InstanceValue).
        instance.name->union(
          p.slot->select( s | s.definingFeature.name->includes('minusOpp'))->
            asSequence->first().value->first().oclAsType(InstanceValue).
              instance.name)->includes(
                self.slot->select(s | s.definingFeature.name->
                  includes('sigDir'))->asSequence->first().value->
                    first()->oclAsType(InstanceValue).instance.name->
                      asSequence->first()))
```

- If the signal instance referenced by sigId is located at slot e4Entry of a double slip point instance, the value of slot sigDir is consistent with the possible directions at that point that are given by slot plus and minus of that double slip point instance. If tbe plus and minus positions of that point allow for directions RIGHT and STRAIGHT, sigDir may be LEFT or STRAIGHT as the double slip point is entered from the opposite end. Vice versa, if the plus and minus positions allow for directions LEFT and STRAIGHT, sig DIR may be RIGHT or STRAIGHT.

```
inv SignalSettingInstance17:
  let id:Integer =
  slot->select(s | s.definingFeature.name->includes('sigId'))->
    asSequence->first().value->first.oclAsType(LiteralSignalId).value
  in
    SignalInstance.allInstances->exists(sig |
      sig.slot->select(s | s.definingFeature.name->includes('signalId'))->
        asSequence->first().value->first().
          oclAsType(LiteralSignalId).value = id and
      (sig.slot->select(s | s.definingFeature.name->includes('sensor'))->
        asSequence->first().value->first.oclAsType(InstanceValue).instance.
          slot->select(s | s.definingFeature.oclAsType(Property).owningClass.
            oclIsTypeOf(Signal))->asSequence->first().value->
              first.oclAsType(InstanceValue).instance.slot->
                select(s | s.definingFeature.name->includes('entryDbSlPoint'))->
                  asSequence->first().value->first().oclAsType(InstanceValue).
                    instance.slot->select(s | s.definingFeature.
                    oclAsType(Property).owningClass.oclIsTypeOf(DoubleSlipPoint))->
                     asSequence->first().value->first.oclAsType(InstanceValue).
                      instance.slot->select(s |
                       s.definingFeature.name->includes('e4Entry'))->size()=1))
    implies
     DoubleSlipPointInstance.allInstances->exists(p | p.slot->select(s |
     s.definingFeature.name->includes('e4Entry'))->asSequence->first().
       value.oclAsType(InstanceValue).instance.slot->select(s |
         s.definingFeature.oclAsType(Property).owningClass.
           oclIsKindOf(Sensor))->first().value->first().
             oclAsType(InstanceValue).instance.slot->select(s |
               s.definingFeature.name->includes('signal'))->asSequence->
```

```
                first().value->first().oclAsType(InstanceValue).
                  instance.slot->select(s |
                    s.definingFeature.oclAsType(Property).owningClass.
                      oclIsKindOf(Sensor))->asSequence->first().value->
                        first().oclAsType(InstanceValue).instance.slot->
                          select(s | s.definingFeature.name->
                            includes('signalId'))->asSequence->first().
                              value.oclAsType(LiteralSignalId).value->first() = id
    and
    p.slot->select(s | s.definingFeature.name->includes('plus'))->
      asSequence->first().value->first().oclAsType(InstanceValue).
        instance.name->union(
          p.slot->select( s | s.definingFeature.name->includes('minus'))->
            asSequence->first().value->first().oclAsType(InstanceValue).
              instance.name)->excluding('STRAIGHT')->excludes(
                self.slot->select(s | s.definingFeature.name->
                  includes('sigDir'))->asSequence->first().value->
                    first()->oclAsType(InstanceValue).instance.name->
                      asSequence->first()))
```

- If the signal instance referenced by sigId is located at slot e3Entry of a double slip point instance, the value of slot sigDir is consistent with the possible directions at that point that are given by slot plusOpp and minusOpp of that double slip point instance. If tbe plusOpp and minusOpp positions of that point allow for directions RIGHT and STRAIGHT, sigDir may be LEFT or STRAIGHT as the double slip point is entered from the opposite end. Vice versa, if the plusOpp and minusOpp positions allow for directions LEFT and STRAIGHT, sig DIR may be RIGHT or STRAIGHT.

```
inv SignalSettingInstance18:
  let id:Integer =
  slot->select(s | s.definingFeature.name->includes('sigId'))->
    asSequence->first().value->first.oclAsType(LiteralSignalId).value
  in
    SignalInstance.allInstances->exists(sig |
      sig.slot->select(s | s.definingFeature.name->includes('signalId'))->
        asSequence->first().value->first().
          oclAsType(LiteralSignalId).value = id and
      (sig.slot->select(s | s.definingFeature.name->includes('sensor'))->
        asSequence->first().value->first.oclAsType(InstanceValue).instance.
          slot->select(s | s.definingFeature.oclAsType(Property).owningClass.
            oclIsTypeOf(Signal))->asSequence->first().value->
              first.oclAsType(InstanceValue).instance.slot->
                select(s | s.definingFeature.name->includes('entryDbSlPoint'))->
                  asSequence->first().value->first().oclAsType(InstanceValue).
                    instance.slot->select(s | s.definingFeature.
                    oclAsType(Property).owningClass.oclIsTypeOf(DoubleSlipPoint))->
                     asSequence->first().value->first.oclAsType(InstanceValue).
                      instance.slot->select(s |
                       s.definingFeature.name->includes('e3Entry'))->size()=1))
    implies
     DoubleSlipPointInstance.allInstances->exists(p | p.slot->select(s |
     s.definingFeature.name->includes('e3Entry'))->asSequence->first().
       value.oclAsType(InstanceValue).instance.slot->select(s |
         s.definingFeature.oclAsType(Property).owningClass.
           oclIsKindOf(Sensor))->first().value->first().
             oclAsType(InstanceValue).instance.slot->select(s |
               s.definingFeature.name->includes('signal'))->asSequence->
                 first().value->first().oclAsType(InstanceValue).
                   instance.slot->select(s |
                     s.definingFeature.oclAsType(Property).owningClass.
```

```
                    oclIsKindOf(Sensor))->asSequence->first().value->
                      first().oclAsType(InstanceValue).instance.slot->
                        select(s | s.definingFeature.name->
                          includes('signalId'))->asSequence->first().
                            value.oclAsType(LiteralSignalId).value->first() = id
and
p.slot->select(s | s.definingFeature.name->includes('plusOpp'))->
  asSequence->first().value->first().oclAsType(InstanceValue).
    instance.name->union(
      p.slot->select( s | s.definingFeature.name->includes('minusOpp'))->
        asSequence->first().value->first().oclAsType(InstanceValue).
          instance.name)->excluding('STRAIGHT')->excludes(
            self.slot->select(s | s.definingFeature.name->
              includes('sigDir'))->asSequence->first().value->
                first()->oclAsType(InstanceValue).instance.name->
                  asSequence->first()))
```

Semantics

Given by the transformation from a concrete network to a labeled transition system.

Notation

A SignalSettingInstance is either depicted as a UML object or as a tuple, e.g. `(Sig1,GO, LEFT)` or `(Sig2,GO)`. The first item of the tuple is the signal id, the second one the signal state, and the last one the (optional) direction.

Chapter 3

Tram Specification Using the Profile

The stereotypes and data types defined in the profile are used in UML diagrams. A class diagram is used to model a concrete problem in the railway domain, e.g. trams. The concrete track networks are object diagrams related to the class diagram.

3.1 Generic Track Network

In our example, a tram track network is given in a class diagram as shown in Fig. 3.1.

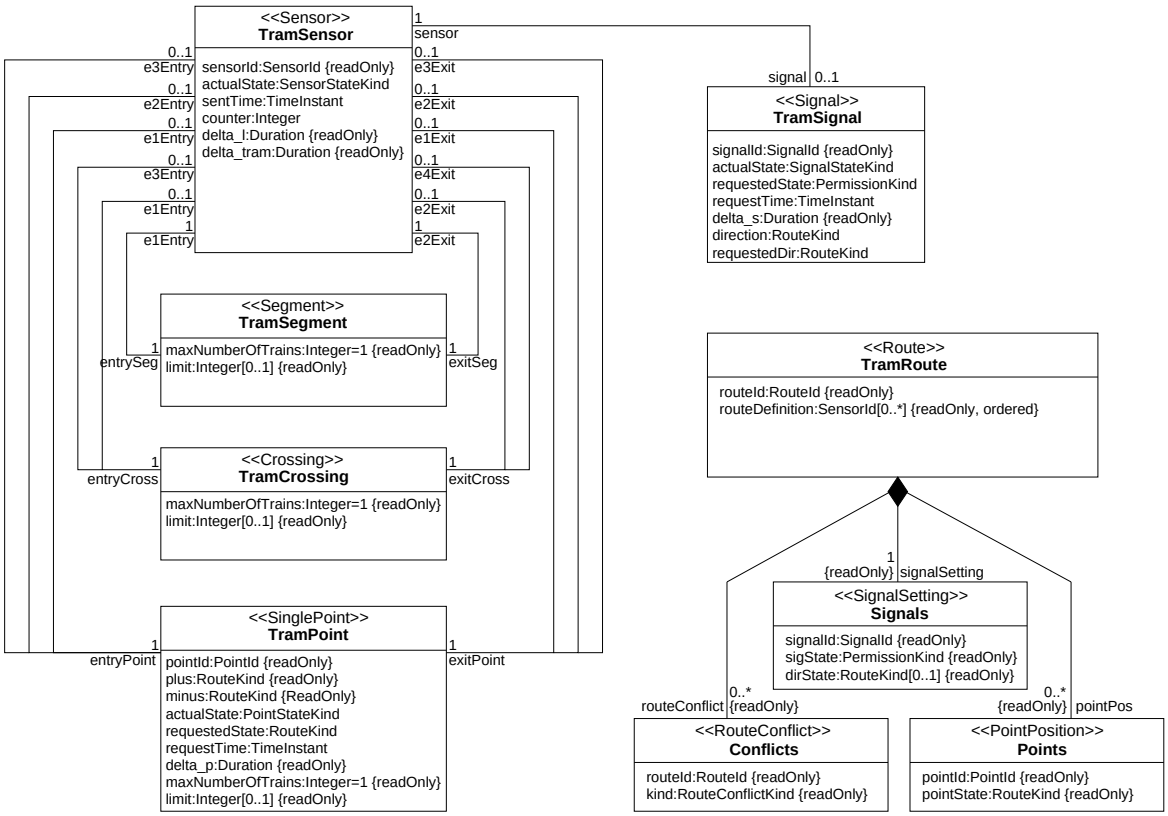

Figure 3.1: Generic tram network

The interrelationships between the different stereotypes from RCSD are concretized for trams:

there are no automatic running systems and no slip points, all segments are used unidirectionally, and signals do not use speed limits. The maximal number of trains allowed on each segment is 1.

3.2 Concrete Track Network

The network description of a concrete tram track network to be controlled is an *object diagram* that is based on the class diagram given above. We show a track layout diagram and route definitions as an object diagram in RCSD profile notation in Fig. 3.2. This diagram has to be complemented by a table that gives property values for constant such as *delta_s* for *TramSignal*.

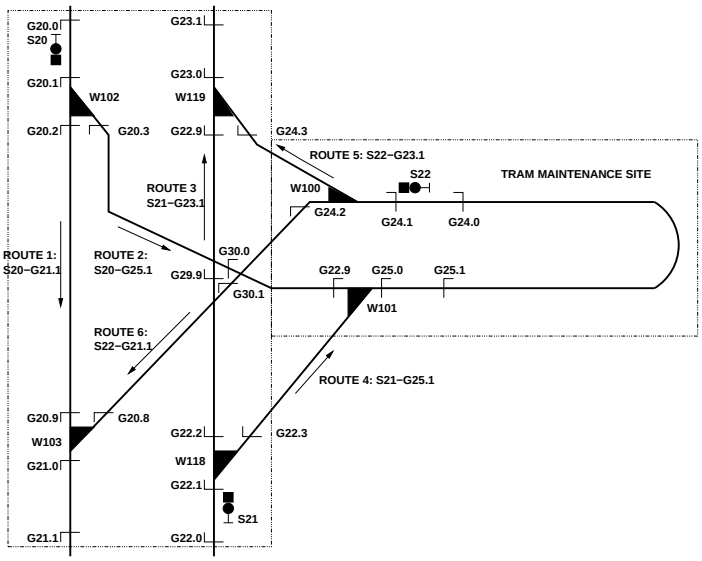

Figure 3.2: Concrete tram network

In Fig. 3.3, the same track network is shown in usual UML notation, i.e. an object diagram. For the sake of brevity, several values for properties have been left out, e.g. *requestedState* and *requestTime* of *TramSignal*. Note that this object diagram is not fully specified without these values.

Comparing the two figures, it is obvious that the RCSD profile notation is more comprehensible and therefore preferable in the communication process between domain experts and software designers.

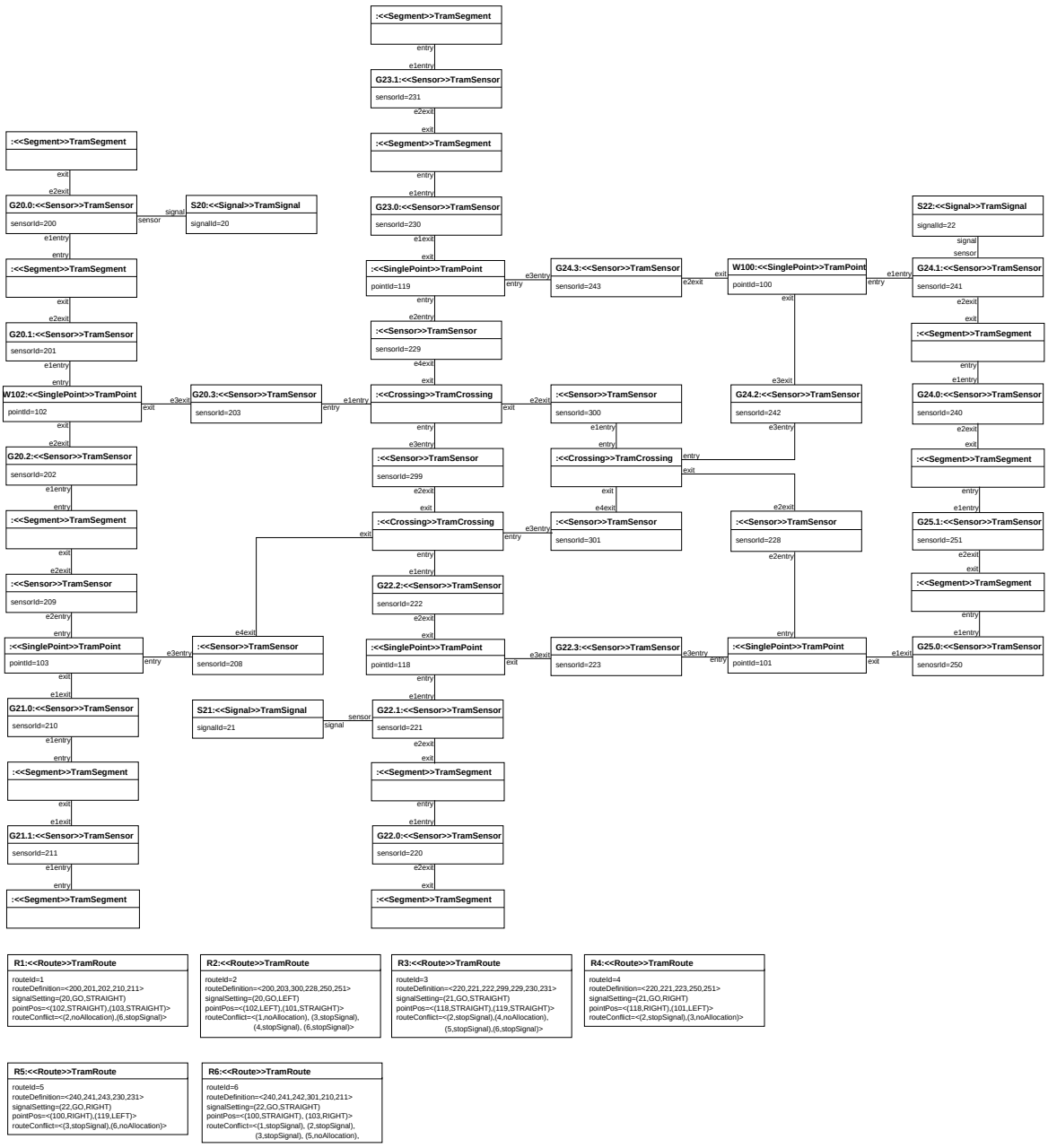

Figure 3.3: Concrete tram network

Bibliography

[FKvV98] W. J. Fokkink, G. P. Kolk, and S. F. M. van Vlijmen. EURIS, a specification method for distributed interlockings. In *Proceedings of SAFECOMP '98*, volume 1516 of *LNCS*, pages 296–305. Springer-Verlag, 1998.

[HP02] A. E. Haxthausen and J. Peleska. A Domain Specific Language for Railway Control Systems. In *Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology, (IDPT2002), Pasadena, California*, June 23-28 2002.

[HP03] A. E. Haxthausen and J. Peleska. Automatic Verification, Validation and Test for Railway Control Systems based on Domain-Specific Descriptions. In *Proceedings of the 10th IFAC Symposium on Control in Transportation Systems.* Elsevier Science Ltd, Oxford, 2003.

[Hun06] Hardi Hungar. UML-basierte entwicklung sicherheitskritische systeme im bahnbereich. In *Dagstuhl Workshop MBEES - Modellbasierte Entwicklung eingebetteter Systeme*, Informatik Bericht, pages 63–64, TU Braunschweig, Jan 2006.

[JPD04] Anne E. Haxthausen Jan Peleska, Daniel Große and Rolf Drechsler. Automated Verification for Train Control Systems. In Eckehard Schnieder and Géza Tarnai, editors, *FORMS/FORMAT 2004. Formal Methods for Automation and Safety in Railway and Automotive Systems*, pages 296–303, Braunschweig, December 2004. Proceedings of Symposium FORMS/FORMAT 2004, Braunschweig, Germany, 2nd and 3rd December 2004. ISBN 3-9803363-8-7.

[OMG05a] Object Management Group. Unified Modeling Language: Superstructure, version 2.0. http://www.omg.org/docs/formal/05-07-04.pdf, July 2005.

[OMG05b] Object Management Group. Unified Modeling Language (UML) Specification: Infrastructure, version 2.0. http://www.omg.org/docs/ptc/04-10-14.pdf, July 2005.

[OMG06] Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification. http://www.omg.org/docs/formal/06-01-01.pdf, January 2006.

[Pac02] Joern Pachl. *Railway Operation and Control.* VTD Rail Publishing, Mountlake Terrace (USA), 2002. ISBN 0-9719915-1-0.

[PBH00] J. Peleska, A. Baer, and A. E. Haxthausen. Towards Domain-Specific Formal Specification Languages for Railway Control Systems. In *Proceedings of the 9th IFAC Symposium on Control in Transportation Systems 2000, June 13-15, 2000, Braunschweig, Germany*, pages 147–152, 2000.

[PHK+06] Jan Peleska, Anne E. Haxthausen, Sebastian Kinder, Daniel Große, and Rolf Drechsler. Model-driven development and verification in the railway domain. 2006. submitted to SAFECOMP2006.

[rai] A grand challenge for computing science: Towards a domain theory of railways. http://www.railwaydomain.org.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language – Reference Manual, 2nd edition.* Addison-Wesley, July 2004.