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
March 17, 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 [OMG04, OMG05] 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 [BHP] 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

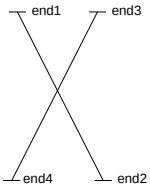

Figure 1.2: Crossing

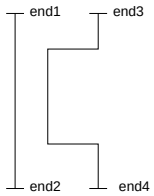

Figure 1.3: Interlaced segments

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.

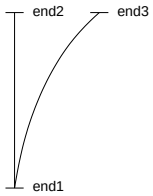

Figure 1.4: Single point

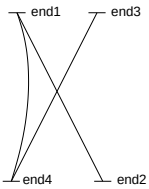

Figure 1.5: Single slip point

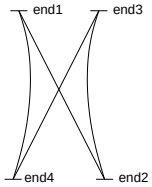

Figure 1.6: Double slip point

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 [OMG04] and [OMG05] 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.

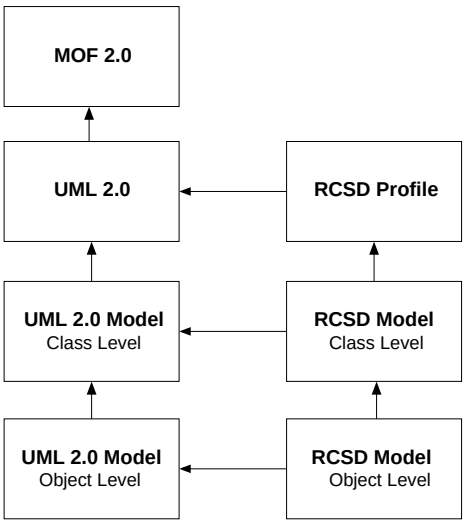

Figure 1.7: RCSD Profile in the UML metalevel context

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 six parts: the definition of primitive datatypes, network elements on class level, associations between these elements, network elements and associations on object level, routes, and top-level constraints.

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.

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*. 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, two 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. Most important is the definition of domain-specific notation. Of course, usual UML notation can be used but is infeasible as we can see in the direct comparison in Section 3.2.

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. 2.34. Again, constraints are used for unambiguous and strict definitions of attributes and suchlike.

The last part of the profile is a set of top-level constraints that describe interrelationships between stereotypes.

Chapter 2

UML 2.0 Profile for the Railway Control System Domain

2.1 Primitives

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.

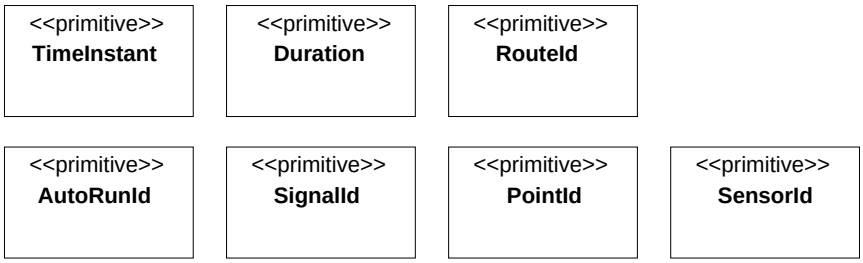

Figure 2.1: Primitive data types

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`.

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 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`.

2.1.4 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`.

2.1.5 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`.

2.1.6 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`.

2.1.7 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 two different kinds of points, *SinglePoints* with one branch and *SlipPoints* where two track segments cross with the possibility of at least one changeover possibility from one segment to the other one. 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.

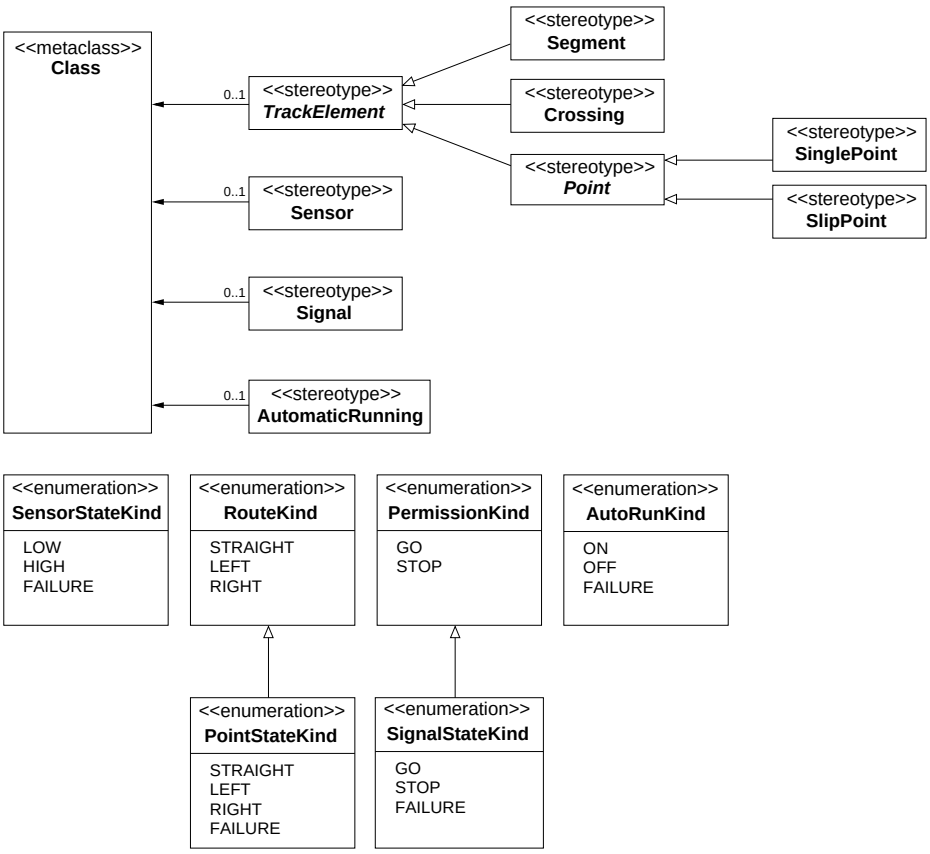

Figure 2.2: Stereotypes and enumerations for modeling the track network of a railway system

2.2.1 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

- Each AutomaticRunning has a mandatory attribute *autoRunId* with type AutoRunId. Its value is constant.

```
ownedAttribute->one(a | a.name='autoRunId' and
      a.oclIsTypeOf(AutoRunId) and
      a.upperBound()=1 and a.lowerBound()=1 and
      a.isReadOnly=true)
```

- Each AutomaticRunning has a mandatory attribute *actualState* with type AutoRunKind.

```
ownedAttribute->one(a | a.name='actualState' and
     a.oclIsTypeOf(AutoRunKind) and
     a.upperBound()=1 and a.lowerBound()=1)
```

- Each AutomaticRunning has an optional attribute *requestedState* with type AutoRunKind. If this attribute exists, the multiplicity is *0..1* or *1*.

```
(ownedAttribute->one(a1 | a1.name='requestedState' and
     a1.oclIsTypeOf(AutoRunKind) and
     a1.upperBound()=1 and a1.lowerBound()≥0)) or
(not ownedAttribute->exists(a2 | a2.name='requestedState'))
```

- Each AutomaticRunning has an optional attribute *requestTime* with type TimeInstant. If this attribute exists, the multiplicity is *1*.

```
(ownedAttribute->one(a1 | a1.name='requestTime' and
     a1.oclIsTypeOf(TimeInstant) and
     a1.upperBound()=1 and a1.lowerBound()=1)) or
(not ownedAttribute->exists(a2 | a2.name='requestedState'))
```

- Each AutomaticRunning has an optional attribute *delta_a* with type Duration. If this attribute exists, its multiplicity is *1* and its value is constant.

```
(ownedAttribute->one(a1 | a1.name='delta_a'
     a1.oclIsTypeOf(Duration) and
     a1.upperBound()=1 and a1.lowerBound()=1 and
     a1.isReadOnly=true)) or
(not ownedAttribute->exists(a2 | a2.name='delta_a'))
```

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

```
ownedAttribute->one(a1 | a1.name='requestedState') implies
     (ownedAttribute->one(a2 | a2.name='requestTime') and
     ownedAttribute->one(a3| a3.name='delta_a')) and
ownedAttribute->one(a4 | a4.name='requestTime') implies
     (ownedAttribute->one(a5 | a5.name='requestedState') and
     ownedAttribute->one(a6 | a6.name='delta_a')) and
ownedAttribute->one(a7 | a7.name='delta_a') implies
     ownedAttribute->one(a8 | a8.name='requestedState') and
     ownedAttribute->one(a9 | a9.name='requestTime'))
```

- All Associations related to an AutomaticRunning are AutoRunAssociations.

```
ownedAttribute->collect(association)->forAll(oclIsTypeOf(AutoRunAssociation))
```

- There is exactly one Association related to an AutomaticRunning.

```
ownedAttribute->collect(association)->size()=1
```

- Each AutomaticRunning has a mandatory attribute *sensor* with type Sensor that is part of an AutoRunAssociation. Its value is constant.

```
ownedAttribute->one(a | a.name='sensor' and
```

```
a.oclIsTypeOf(Sensor) and
a.upperBound()=1 and a.lowerBound()=1 and
a.isReadOnly=true and
a.association.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.

2.2.2 AutoRunKind

Description

AutoRunKind is an enumeration that specifies the valid values for automatic train runnings.

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`.

2.2.3 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

- Each Crossing has an optional attribute *e3Entry* with type Sensor that is part of a SensorAssociation. If the attribute exists, its multiplicity is *0..1* or *1* and its value is constant.

```
(ownedAttribute->one(a1 | a1.name='e3Entry' and
     a1.oclIsTypeOf(Sensor) and
     a1.upperBound()=1 and a1.lowerBound()≥ 0 and
     a1.isReadOnly=true and
     a1.association.oclIsTypeOf(SensorAssociation))) or
(not ownedAttribute->exists(a2 | a2.name='e3Entry'))
```

- Each Crossing has an optional attribute *e4Entry* with type Sensor that is part of a SensorAssociation. If the attribute exists, its multiplicity is *0..1* or *1* and its value is constant.

```
(ownedAttribute->one(a1 | a1.name='e4Entry' and
     a1.oclIsTypeOf(Sensor) and
     a1.upperBound()=1 and a1.lowerBound()≥0 and
     a1.isReadOnly=true and
     a1.association.oclIsTypeOf(SensorAssociation))) or
(not ownedAttribute->exists(a2 | a2.name='e4Entry'))
```

- Each Crossing has an optional attribute *e3Exit* with type Sensor that is part of a SensorAssociation. If the attribute exists, its multiplicity is *0..1* or *1* and its value is constant.

```
(ownedAttribute->one(a1 | a1.name='e3Exit' and
     a1.oclIsTypeOf(Sensor) and
     a1.upperBound()=1 and a1.lowerBound()≥0 and
     a1.isReadOnly=true and
     a1.association.oclIsTypeOf(SensorAssociation))) or
(not ownedAttribute->exists(a2 | a2.name='e3Exit'))
```

- Each Crossing has an optional attribute *e4Exit* with type Sensor that is part of a SensorAssociation. If the attribute exists, its multiplicity is *0..1* or *1* and its value is constant.

```
(ownedAttribute->one(a1 | a1.name='e4Exit' and
     a1.oclIsTypeOf(Sensor) and
     a1.upperBound()=1 and a1.lowerBound()=≥0 and
     a1.isReadOnly=true and
     a1.association.oclIsTypeOf(SensorAssociation))) or
(not ownedAttribute->exists(a2 | a2.name='e4Exit'))
```

- Each Crossing has at least two sensors at end1 and end2.

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

- Each Crossing has at least two sensors at end3 and end4.

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

- Each Crossing has at most four sensors at end3 and end4.

```
ownedAttribute->select(a | a.name='e3Entry' or a.name='e4Entry' or
     a.name='e3Exit' or a.name='e4Exit')->size()≤4
```

- The number of sensors at end3 and end4 is not three, i.e. either two or four.

```
ownedAttribute->select(a | a.name='e3Entry' or a.name='e4Entry' or
     a.name='e3Exit' or a.name='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. 2.3) or interlaced track segments (see Fig. 2.4). They are not sinks or sources of track networks and therefore require sensors at each end.

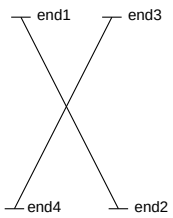

Figure 2.3: Crossed tracks

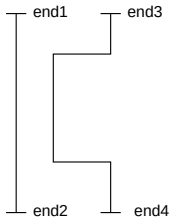

Figure 2.4: Interlaced tracks

Notation

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

2.2.4 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`.

2.2.5 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

- Each Point has a mandatory attribute *pointId* with type PointId. Its value is constant.

```
ownedAttribute->one(a | a.name='pointId' and
      a.oclIsTypeOf(PointId) and
      a.upperBound()=1 and a.lowerBound()=1 and
      a.isReadOnly=true)
```

- Each Point has a mandatory attribute *plus* with type RouteKind. Its value is constant.

```
ownedAttribute->one(a | a.name='plus' and
      a.oclIsTypeOf(RouteKind) and
      a.upperBound()=1 and a.lowerBound()=1 and
      a.isReadOnly=true)
```

- Each Point has a mandatory attribute *minus* with type RouteKind. Its value is constant.

```
ownedAttribute->one(a | a.name='minus' and
      a.oclIsTypeOf(RouteKind) and
      a.upperBound()=1 and a.lowerBound()=1 and
      a.isReadOnly=true)
```

- Each Point has a mandatory attribute *actualState* with type PointStateKind.

```
ownedAttribute->one(a | a.name='actualState' and
      a.oclIsTypeOf(PointStateKind) and
      a.upperBound()=1 and a.lowerBound()=1)
```

- Each Point has a mandatory attribute *requestedState* with type RouteKind.

```
ownedAttribute->one(a | a.name='requestedState' and
      a.oclIsTypeOf(RouteKind) and
      a.upperBound()=1 and a.lowerBound()=1)
```

- Each Point has a mandatory attribute *requestTime* with type TimeInstant.

```
ownedAttribute->one(a | a.name='requestTime' and
      a.oclIsTypeOf(TimeInstant) and
      a.upperBound()=1 and a.lowerBound()=1)
```

- Each Point has a mandatory attribute *delta_p* with type Duration. Its value is constant.

```
ownedAttribute->one(a | a.name='delta_p'
      a.oclIsTypeOf(Duration) and
      a.upperBound()=1 and a.lowerBound()=1 and
      a.isReadOnly=true)
```

- Each Point has an optional attribute *e3Entry* with type Sensor that is part of a SensorAssociation. If this attribute exists, its multiplicity is *0..1* or*1* and its value is constant.

```
(ownedAttribute->one(a1 | a1.name='e3Entry' and
     a1.oclIsTypeOf(Sensor) and
     a1.upperBound()=1 and a1.lowerBound()≥0 and
     a1.isReadOnly=true and
     a1.association.oclIsTypeOf(SensorAssociation))) or
(not ownedAttribute->exists(a2 | a2.name='e3Entry'))
```

- Each Point has an optional attribute *e3Exit* with type Sensor that is part of a SensorAssociation. If this attribute exists, its multiplicity is *0..1* or *1* and its value is constant.

```
(ownedAttribute->one(a1 | a1.name='e3Exit' and
     a1.oclIsTypeOf(Sensor) and
     a1.upperBound()=1 and a1.lowerBound()≥0 and
     a1.isReadOnly=true and
     a1.association.oclIsTypeOf(SensorAssociation))) or
(not ownedAttribute->exists(a2 | a2.name='e3Exit'))
```

- There are at least three associations to sensors.

```
ownedAttribute->collect(association)->select(a | a.oclIsTypeOf(SensorAssociation))->
     size()≥3
```

- There are at most six associations to sensors.

```
ownedAttribute->collect(association)->select(a | a.oclIsTypeOf(SensorAssociation))->
     size()≤6
```

- There is at least one sensor associated to end1.

```
ownedAttribute->select(a | a.name='e1Entry' or a.name='e1Exit')->size()≥1
```

- There are at most two sensors associated to end1.

```
ownedAttribute->select(a | a.name='e1Entry' or a.name='e1Exit')->size()≤2
```

- There is at least one sensor associated to end2.

```
ownedAttribute->select(a | a.name='e2Entry' or a.name='e2Exit')->size()≥1
```

- There are at most two sensors associated to end2.

```
ownedAttribute->select(a | a.name='e2Entry' or a.name='e2Exit')->size()≤2
```

- There is at least one sensor associated to end3.

```
ownedAttribute->select(a | a.name='e3Entry' or a.name='e3Exit')->size()≥1
```

- There are at most two sensors associated to end3.

```
ownedAttribute->select(a | a.name='e3Entry' or a.name='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*.

2.2.6 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`.

2.2.7 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`.

2.2.8 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

- There are at most four sensors associated to end1 and end2.

```
ownedAttribute->select(a | a.name='e1Entry' or a.name='e2Entry' or
     a.name='e1Exit' or a.name='e2Exit')->size()≤4
```

- There is at least one sensor associated to end1 and end2.

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

- The number of sensors associated to end1 and end2 is not three.

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

Semantics

Trains travel on track elements. A segment is a specialization of *TrackElement* with two ends as shown in Fig. 2.5. 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.

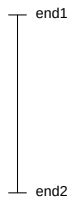

Figure 2.5: Segment

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.

2.2.9 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

- Each Sensor has a mandatory attribute *sensorId* with type SensorId. Its value is constant.

```
ownedAttribute->one(a | a.name='sensorId' and
      a.oclIsTypeOf(SensorId) and
      a.upperBound()=1 and a.lowerBound()=1 and
      a.isReadOnly=true)
```

- Each Sensor has a mandatory attribute *counter* with type Integer.

```
ownedAttribute->one(a | a.name='counter' and
      a.oclIsTypeOf(Integer) and
      a.upperBound()=1 and a.lowerBound()=1)
```

- Each Sensor has a mandatory attribute *actualState* with type SensorStateKind.

```
ownedAttribute->one(a | a.name='actualState' and
      a.oclIsTypeOf(SensorStateKind) and
      a.upperBound()=1 and a.lowerBound()=1)
```

- Each Sensor has a mandatory attribute *delta_l* with type Duration. Its value is constant.

```
ownedAttribute->one(a | a.name='delta_l' and
      a.oclIsTypeOf(Duration) and
      a.upperBound()=1 and a.lowerBound()=1 and
      a.isReadOnly=true)
```

- Each Sensor has a mandatory attribute *sentTime* with type TimeInstant.

```
ownedAttribute->one(a | a.name='sentTime' and
      a.oclIsTypeOf(TimeInstant) and
      a.upperBound()=1 and a.lowerBound()=1)
```

- Each Sensor has a mandatory attribute *delta_tram* with type Duration. Its value is constant.

```
ownedAttribute->one(a | a.name='delta_tram' and
      a.oclIsTypeOf(Integer) and
      a.upperBound()=1 and a.lowerBound()=1 and
      a.isReadOnly=true)
```

- Associations related to Sensors are either SensorAssociations, SignalAssociations, or AutoRunAssociations.

```
ownedAttribute->collect(association)->forAll(a1 |
      a1.oclIsTypeOf(SensorAssociation) or
      a1.oclIsTypeOf(SignalAssociation) or
      a1.oclIsTypeOf(AutoRunAssociation))
```

- The number of track segments associated to a Sensor is two.

```
ownedAttribute->collect(association)->select(a | a.oclIsTypeOf(SensorAssociation)->
      size()=2)
```

- The number of signals associated to a Sensor is at most one.

```
ownedAttribute->collect(association)->select(a | a.oclIsTypeOf(SignalAssociation)->
      size()≤1)
```

- The number of automatic runnings associated to a Sensor is at most one.

```
ownedAttribute->collect(association)->select(a | a.oclIsTypeOf(AutoRunAssociation)->
      size()≤1)
```

- Each Sensor has a mandatory attribute *entry* with type Sensor that is part of a SensorAssociation. Its value is constant.

```
ownedAttribute->one(a | a.name='entry' and
      a.oclIsTypeOf(Sensor) and
      a.upperBound()=1 and a.lowerBound()=1 and
      a.isReadOnly=true and
      a.association.oclIsTypeOf(SensorAssociation))
```

- Each Sensor has a mandatory attribute *exit* with type Sensor that is part of a SensorAssociation. Its value is constant.

```
ownedAttribute->one(a | a.name='exit' and
      a.oclIsTypeOf(Sensor) and
      a.upperBound()=1 and a.lowerBound()=1 and
      a.isReadOnly=true and
      a.association.oclIsTypeOf(SensorAssociation))
```

- Each Sensor has an optional attribute *signal* with type Signal that is part of a SignalAssociation. If this attribute exists, its multiplicity is *0..1* or *1* and its value is constant.

```
(ownedAttribute->one(a | a.name='signal' and
      a.oclIsTypeOf(Signal) and
      a.upperBound()=1 and a.lowerBound()≥0 and
      a.isReadOnly=true and
```

```
      a.association.oclIsTypeOf(SignalAssociation))) or
(not ownedAttribute->exists(a | a.name='signal'))
```

- Each Sensor has an optional attribute *autoRun* with type AutomaticRunning that is part of an AutoRunAssociation. If this attribute exists, its multiplicity is *0..1* or *1* and its value is constant.

```
(ownedAttribute->one(a | a.name='autoRun' and
      a.oclIsTypeOf(AutomaticRunning) and
      a.upperBound()=1 and a.lowerBound()≥0 and
      a.isReadOnly=true and
      a.association.oclIsTypeOf(AutomaticRunning))) or
(not ownedAttribute->exists(a | a.name='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.

2.2.10 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`.

2.2.11 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

- Each Signal has a mandatory attribute *signalId* with type SignalId. Its value is constant.

```
ownedAttribute->one(a | a.name='signalId' and
      a.oclIsTypeOf(SignalId) and
      a.upperBound()=1 and a.lowerBound()=1 and
      a.isReadOnly=true)
```

- Each Signal has a mandatory attribute *actualState* with type SignalStateKind.

```
ownedAttribute->one(a | a.name='actualState' and
      a.upperBound()=1 and a.lowerBound()=1 and
      a.oclIsTypeOf(SignalStateKind))
```

- Each Signal has a mandatory attribute *requestedState* with type PermissionKind.

```
ownedAttribute->one(a | a.name='requestedState' and
      a.upperBound()=1 and a.lowerBound()=1 and
      a.oclIsTypeOf(PermissionKind))
```

- Each Signal has a mandatory attribute *requestTime* with type TimeInstant.

```
ownedAttribute->one(a | a.name='requestTime' and
      a.oclIsTypeOf(TimeInstant) and
      a.upperBound()=1 and a.lowerBound()=1)
```

- Each Signal has a mandatory attribute *delta_s* with type Duration. Its value is constant.

```
ownedAttribute->one(a | a.name='delta_s' and
      a.oclIsTypeOf(Duration) and
      a.upperBound()=1 and a.lowerBound()=1 and
      a.isReadOnly=true)
```

- Each Signal has an optional attribute *limit* with type Integer. If this attribute exists, its multiplicity is *0..1* or *1* and its value is constant.

```
(ownedAttribute->one(a1 | a1.name='limit' and
      a1.oclIsTypeOf(Integer) and
      a1.upperBound()=1 and a1.lowerBound()≥0 and
      a1.isReadOnly=true)) or
(not ownedAttribute->exists(a2 | a2.name='limit'))
```

- Each Signal has an optional attribute *direction* with type RouteKind. If this attribute exists, its multiplicity is *0..1* or *1* and its value is constant.

```
(ownedAttribute->one(a1 | a1.name='direction' and
      a1.oclIsTypeOf(RouteKind) and
```

```
      a1.upperBound()=1 and a1.lowerBound()≥0)) or
(not ownedAttribute->exists(a2 | a2.name='direction'))
```

- Each Signal is only associated to other elements by SignalAssociations.

```
ownedAttribute->collect(association)->forAll(oclIsTypeOf(SignalAssociation))
```

- Each Signal has exactly one SignalAssociation.

```
ownedAttribute->collect(association)->size()=1
```

- Each Signal has a mandatory attribute *sensor* with type Signal that is part of a SignalAssociation.

```
ownedAttribute->one(a | a.name='sensor' and
      a.upperBound()=1 and a.lowerBound()=1 and
      a.oclIsTypeOf(Signal) and
      a.association.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.

2.2.12 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`.

2.2.13 SinglePoint

Description

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

Associations

None.

Attributes

None.

Constraints

- If there is an entry sensor associated to end1, there has to be an exit sensor associated to end2 or end3.

```
ownedAttribute->one(a | a.name='e1Entry') implies
      ownedAttribute->one(a | a.name='e2Exit') or
      ownedAttribute->one(a | a.name='e3Exit')
```

- If there is an exit sensor associated to end 1, there has to be an entry sensor at end2 or end3.
. `ownedAttribute->one(a | a.name='e1Exit') implies`

```
      ownedAttribute->one(a | a.name='e2Entry') or
      ownedAttribute->one(a | a.name='e3Entry')
```

Semantics

SinglePoint is a specialization of *Point*. Is is a point with one branch as shown in Fig. 2.6. Obviously, the branch can be left or right depending on the design of the point.

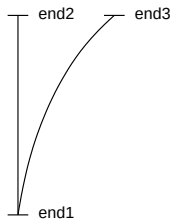

Figure 2.6: Single point

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.

2.2.14 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, a *double slip point* has two changeover possibilities.

Associations

None.

Attributes

None.

Constraints

- Each SlipPoint has an optional attribute *pointIdOpp* with type PointId. If the attribute exists, its multiplicity is *1* and its value is constant.

```
(ownedAttribute->one(a1 | a1.name='pointIdOpp' and
     a1.oclIsTypeOf(PointId) and
     a1.upperBound()=1 and a1.lowerBound()=1 and
     a1.isReadOnly=true)) or
(not ownedAttribute->exists(a2 | a2.name='pointIdOpp'))
```

- Each SlipPoint has an optional attribute *plusOpp* with type RouteKind. If the attribute exists, its multiplicity is *1* and its value is constant.

```
(ownedAttribute->one(a1 | a1.name='plusOpp' and
     a1.oclIsTypeOf(RouteKind) and
     a1.upperBound()=1 and a1.lowerBound()=1 and
     a1.isReadOnly=true)) or
(not ownedAttribute->exists(a2 | a2.name='plusOpp'))
```

- Each SlipPoint has an optional attribute *minusOpp* with type RouteKind. If the attribute exists, its multiplicity is *1* and its value is constant.

```
(ownedAttribute->one(a1 | a1.name='minusOpp' and
     a1.oclIsTypeOf(RouteKind) and
     a1.upperBound()=1 and a1.lowerBound()=1 and
     a1.isReadOnly=true)) or
(not ownedAttribute->exists(a2 | a2.name='minusOpp'))
```

- Each SlipPoint has an optional attribute *actualStateOpp* with type PointStateKind. If the attribute exists, its multiplicity is *1*.

```
(ownedAttribute->one(a1 | a1.name='actualStateOpp' and
     a1.oclIsTypeOf(PointStateKind) and
     a1.upperBound()=1 and a1.lowerBound()=1)) or
(not ownedAttribute->exists(a2 | a2.name='actualStateOpp'))
```

- Each SlipPoint has an optional attribute *requestedStateOpp* with type RouteKind. If the attribute exists, its multiplicity is *1*.

```
(ownedAttribute->one(a1 | a1.name='requestedStateOpp' and
```

```
      a1.oclIsTypeOf(RouteKind) and
      a1.upperBound()=1 and a1.lowerBound()=1)) or
(not ownedAttribute->exists(a2 | a2.name='requestedStateOpp'))
```

- Each SlipPoint has an optional attribute *requestTimeOpp* with type TimeInstant. If the attribute exists, its multiplicity is *1*.

```
(ownedAttribute->one(a1 | a1.name='requestTimeOpp' and
      a1.oclIsTypeOf(TimeInstant) and
      a1.upperBound()=1 and a1.lowerBound()=1)) or
(not ownedAttribute->exists(a2 | a2.name='requestTimeOpp'))
```

- The attributes *pointIdOpp*, *plusOpp*, *minusOpp*, *actualStateOpp*, *requestedStateOpp*, and *requestTimeOpp* are either all present or none of them.

```
ownedAttribute->one(a1 | a1.name='pointIdOpp') implies
      (ownedAttribute->one(a2 | a2.name='plusOpp') and
      ownedAttribute->one(a3 | a3.name='minusOpp') and
      ownedAttribute->one(a4 | a4.name='requestedStateOpp') and
      ownedAttribute->one(a5 | a5.name='actualStateOpp') and
      ownedAttribute->one(a6 | a6.name='requestTimeOpp')) and
ownedAttribute->one(a1 | a1.name='plusOpp') implies
      (ownedAttribute->one(a2 | a2.name='pointIdOpp') and
      ownedAttribute->one(a3 | a3.name='minusOpp') and
      ownedAttribute->one(a4 | a4.name='requestedStateOpp') and
      ownedAttribute->one(a5 | a5.name='actualStateOpp') and
      ownedAttribute->one(a6 | a6.name='requestTimeOpp')) and
ownedAttribute->one(a1 | a1.name='minusOpp') implies
      (ownedAttribute->one(a2 |a2.name='pointIdOpp') and
      ownedAttribute->one(a3 | a3.name='plusOpp') and
      ownedAttribute->one(a4 | a4.name='requestedStateOpp') and
      ownedAttribute->one(a5 | a5.name='actualStateOpp') and
      ownedAttribute->one(a6 | a6.name='requestTimeOpp')) and
ownedAttribute->one(a1 | a1.name='requestedStateOpp') implies
      (ownedAttribute->one(a2 |a2.name='pointIdOpp') and
      ownedAttribute->one(a3 | a3.name='plusOpp') and
      ownedAttribute->one(a4 | a4.name='minusOpp') and
      ownedAttribute->one(a5 | a5.name='actualStateOpp') and
      ownedAttribute->one(a6 | a6.name='requestTimeOpp')) and
ownedAttribute->one(a1 | a1.name='actualStateOpp') implies
      (ownedAttribute->one(a2 |a2.name='pointIdOpp') and
      ownedAttribute->one(a3 | a3.name='plusOpp') and
      ownedAttribute->one(a4 | a4.name='minusOpp') and
      ownedAttribute->one(a5 | a5.name='requestedStateOpp') and
      ownedAttribute->one(a6 | a6.name='requestTimeOpp')) and
ownedAttribute->one(a1 | a1.name='requestTimeOpp') implies
      (ownedAttribute->one(a2 |a2.name='pointIdOpp') and
      ownedAttribute->one(a3 | a3.name='plusOpp') and
      ownedAttribute->one(a4 | a4.name='minusOpp') and
      ownedAttribute->one(a5 | a5.name='requestedStateOpp') and
      ownedAttribute->one(a6 | a6.name='actualStateOpp'))
```

- Each SlipPoint has an optional attribute *e4Entry* with type Senor that is part of a SensorAssociation. If the attribute exists, its multiplicity is *0..1* or *1* and its value is constant.

```
(ownedAttribute->one(a | a1.name='e4Entry' and
      a1.oclIsTypeOf(Sensor) and
```

```
      a1.upperBound()=1 and a1.lowerBound()≥0 and
      a1.isReadOnly=true and
      a1.association.oclIsTypeOf(SensorAssociation))) or
(not ownedAttribute->exists(a2 | a2.name='e4Entry'))
```

- Each SlipPoint has an optional attribute *e4Exit* with type Sensor that is part of a SensorAssociation. If the attribute exists, its multiplicity is *0..1* or *1* and its value is constant.

```
(ownedAttribute->one(a1 | a1.name='e4Exit' and
      a1.oclIsTypeOf(Sensor) and
      a1.upperBound()=1 and a1.lowerBound()≥0 and
      a1.isReadOnly=true and
      a1.association.oclIsTypeOf(SensorAssociation))) or
(not ownedAttribute->exists(a2 | a2.name='e4Exit'))
```

- There is at least one sensor associated to end4.

```
ownedAttribute->select(a | a.name='e4Entry' or a.name='e4Exit')->size()≥1
```

- There are at most two sensors associated to end4.

```
ownedAttribute->select(a | a.name='e4Entry' or a.name='e4Exit')->size()≤2
```

- If an entry sensor is associated to end1, there must be an exit sensor associated to end2 or end4.

```
ownedAttribute->one(a | a.name='e1Entry') implies
      ownedAttribute->one(a | a.name='e2Exit') or
      ownedAttribute->one(a | a.name='e4Exit')
```

- If an exit sensor is associated to end1, there must be an entry sensor associated to end 2 or end4.

```
ownedAttribute->one(a | a.name='e1Exit') implies
      ownedAttribute->one(a | a.name='e2Entry') or
      ownedAttribute->one(a | a.name='e4Entry')
```

- If an entry sensor is associated to end3, there must be either an exit sensor at end4 (single slip point) or at end2 or end4 (double slip point).

```
ownedAttribute->one(a | a.name='e3Entry') implies
      (ownedAttribute->one(a | a.name='e4Exit') and
            ownedAttribute->one(a1 | a1.name='pointIdOpp'))
      or ((ownedAttribute->one(a | a.name='e2Exit')
            or ownedAttribute->one(a | a.name='e4Exit')) and
            ownedAttribute->select(a1 | a1.name='pointIdOpp')->size()=0))
```

- If an exit sensor is associated to end3, there must be either an entry sensor at end4 (single slip point) or at end2 or end4 (double slip point).

```
ownedAttribute->one(a | a.name='e3Exit') implies
      (ownedAttribute->one(a | a.name='e4Entry') and
            ownedAttribute->one(a1 | a1.name='pointIdOpp'))
      or ((ownedAttribute->one(a | a.name='e2Entry')
            or ownedAttribute->one(a | a.name='e4Entry')) and
            ownedAttribute->select(a1 | a1.name='poinIdOpp')->size()=0))
```

Semantics

SlipPoint is a specialization of *Point*. Is is a point that resembles a crossing with at least 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. 2.7), one with two changeover possibilities is called *double slip point* (see Fig. 2.8). In the latter case, a second *id* is needed.

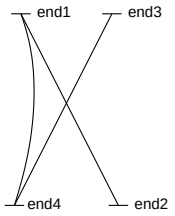

Figure 2.7: Single slip point

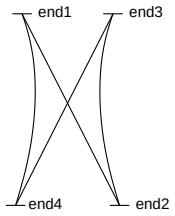

Figure 2.8: Double slip point

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*.

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. Double slip points require more information 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

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

2.2.15 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

- Each TrackElement has a mandatory attribute *maxNumberOfTrains* with type Integer. Its value is constant.

```
ownedAttribute->one(a | a.name='maxNumberOfTrains' and
      a.oclIsTypeOf(Integer) and
```

```
      a.upperBound()=1 and a.lowerBound()=1 and
      a.isReadOnly=true)
```

- All associations to TrackElements are SensorAssociations.

```
ownedAttribute->collect(association)->forAll(oclIsTypeOf(SensorAssociation))
```

- Each TrackElement has an optional attribute *e1Entry* with type Sensor that is part of a SensorAssociation. If the attribute exists, its multiplicity is *0..1* or *1* and its value is constant.

```
(ownedAttribute->one(a1 | a1.name='e1Entry' and
      a1.oclIsTypeOf(Sensor) and
      a1.upperBound()=1 and a1.lowerBound()≥0 and
      a1.isReadOnly=true and
      a1.association.oclIsTypeOf(SensorAssociation))) or
(not ownedAttribute->exists(a2 | a2.name='e1Entry'))
```

- Each TrackElement has an optional attribute *e1Exit* with type Sensor that is part of a SensorAssociation. If the attribute exists, its multiplicity is *0..1* or *1* and its value is constant.

```
(ownedAttribute->one(a1 | a1.name='e1Exit' and
      a1.oclIsTypeOf(Sensor) and
      a1.upperBound()=1 and a1.lowerBound()≥0 and
      a1.isReadOnly=true and
      a1.association.oclIsTypeOf(SensorAssociation))) or
(not ownedAttribute->exists(a2 | a2.name='e1Exit'))
```

- Each TrackElement has an optional attribute *e2Entry* with type Sensor that is part of a SensorAssociation. If the attribute exists, its multiplicity is *0..1* or *1* and its value is constant.

```
(ownedAttribute->one(a1 | a1.name='e2Entry' and
      a1.oclIsTypeOf(Sensor) and
      a1.upperBound()=1 and a1.lowerBound()≥0 and
      a1.isReadOnly=true and
      a1.association.oclIsTypeOf(SensorAssociation))) or
(not ownedAttribute->exists(a2 | a2.name='e2Entry'))
```

- Each TrackElement has an optional attribute *e2Exit* with type Sensor that is part of a SensorAssociation. If the attribute exists, its multiplicity is *0..1* or *1* and its value is constant.

```
(ownedAttribute->one(a1 | a1.name='e2Exit' and
      a1.oclIsTypeOf(Sensor) and
      a1.upperBound()=1 and a1.lowerBound()≥0 and
      a1.isReadOnly=true and
      a1.association.oclIsTypeOf(SensorAssociation))) or
(not ownedAttribute->exists(a2 | a2.name='e2Exit'))
```

- Each TrackElement has an optional attribute *limit* with type Integer. If the attribute exists, its multiplicity is *0..1* or *1* and its value is constant.

```
(ownedAttribute->one(a1 | a1.name='limit' and
      a1.oclIsTypeOf(Integer) and
      a1.upperBound()=1 and a1.lowerBound()≥0 and
      a1.isReadOnly=true)) or
(not ownedAttribute->exists(a2 | a2.name='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*.

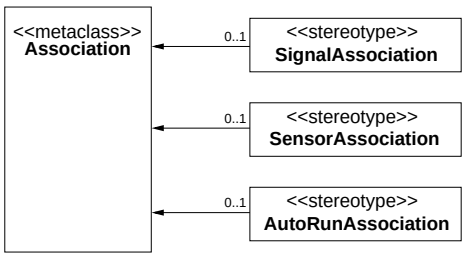

Figure 2.9: Stereotypes for modeling associations between track network elements

2.3.1 AutoRunAssociation

Description

AutoRunAssociations are used to connect sensors to automatic train runnings.

Associations

None.

Attributes

None.

Constraints

- Each AutoRunAssociation is binary.

  ```
  memberEnd->size()=2
  ```

- Each AutoRunAssociation connects two end types.

  ```
  endType->size()=2
  ```

- At one end of each AutoRunAssociation is a Sensor.

  ```
  endType->one(t | t.oclIsTypeOf(Sensor))
  ```

- At one end of each AutoRunAssociation is an AutomaticRunning.

```
endType->one(t | t.oclIsTypeOf(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

- Each SensorAssociation is binary.

```
memberEnd->size()=2
```

- Each SensorAssociation connects two end types.

```
endType->size()=2
```

- At one end of each SensorAssociation is a Sensor.

```
endType->one(t | t.oclIsTypeOf(Sensor))
```

- At one end of each SensorAssociation is a TrackElement.

```
endType->one(t | t.oclIsTypeOf(TrackElement))
```

- If the SensorAssociation connects the entry property of a Sensor with a TrackElement, the respective property of the TrackElement must also specify an entry.

```
memberEnd->exists(m1 | m1.name='entry') implies
     memberEnd->exists(m2 | m2.name='e1Entry' or m2.name='e2Entry' or
          m2.name='e3Entry' or m2.name='e4Entry')
```

- If the SensorAssociation connects the exit property o a Sensor with a TrackElement, the respective property of the TrackElement must also specify an exit.

```
memberEnd->exists(m1 | m1.name='exit') implies
```

```
memberEnd->exists(m2 | m2.name='e1Exit' or m2.name='e2Exit' or
     m2.name='e3Exit' or m2.name='e4Exit')
```

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.

SegmentAssociation 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

- Each SignalAssociation is binary.

  ```
  memberEnd->size()=2
  ```

- Each SignalAssociation connects two end types.

  ```
  endType->size()=2
  ```

- At one end of a SignalAssociation is a Sensor.

  ```
  endType->one(t | t.oclIsTypeOf(Sensor))
  ```

- At one end of a SignalAssociation is a Signal.

  ```
  endType->one(t | t.oclIsTypeOf(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.

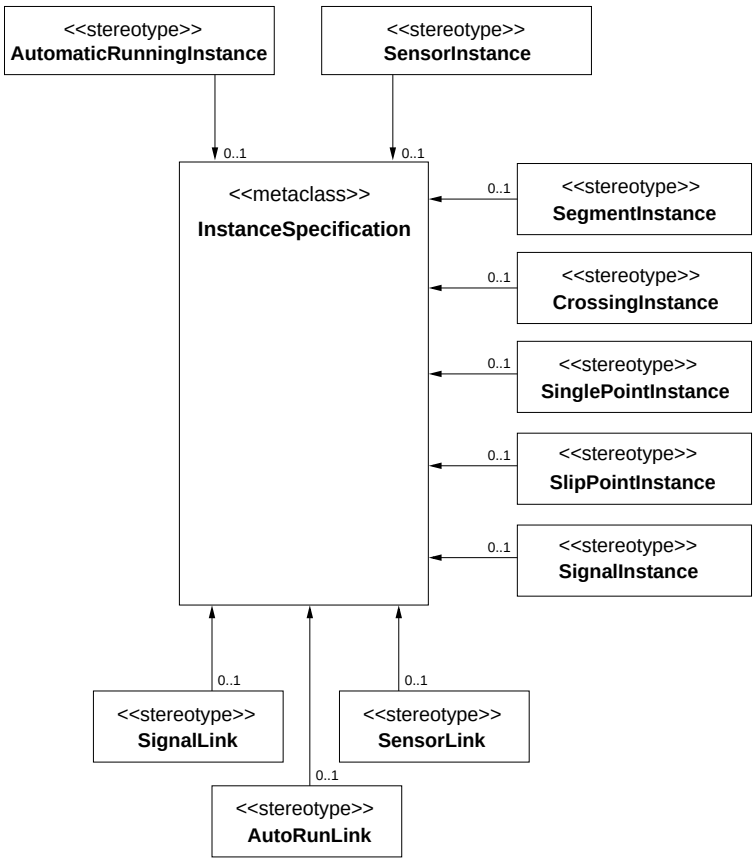

Figure 2.10: Stereotypes for track network element instances

2.4.1 AutomaticRunningInstance

Description

An AutomaticRunningInstance is the instance of an automatic train running.

Associations

None.

Attributes

None.

Constraints

- Each AutomaticRunningInstance is the instance of at least two classifiers.There may be more due to inheritance.

`classifier->size()`≥ 2

- Each AutomaticRunningInstance is the instance of a Class.

  ```
  classifier->one(c | c.oclIsTypeOf(Class))
  ```

- Each AutomaticRunningInstance is the instance of an AutomaticRunning.

  ```
  classifier->one(c | c.oclIsTypeOf(AutomaticRunning))
  ```

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. 2.11), a black box a controlled automatic train running system (see Fig. 2.12). In case that we have a permanent automatic train running instance, the following constraint must hold:

- ```
  slot->select(s | definingFeature.name='actualState' or
        definingFeature.name='requestedState')->forAll(value=ON)
  ```

Figure 2.11: Permanent automatic train running instance

Figure 2.12: 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

- Each AutoRunLink is the instance of two classifiers.

  ```
  classifier->size()=2
  ```

- Each AutoRunLink is the instance of an Association.

  ```
  classifier->one(c | c.oclIsTypeOf(Association))
  ```
  ```

- Each AutoRunLink is the instance of an AutoRunAssociation.

```
classifier->one(c | c.oclIsTypeOf(AutoRunAssociation))
```

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.

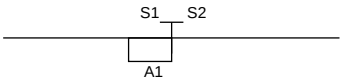

Figure 2.13: Automatic train running instance linked to sensor instance

In Fig. 2.13, *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. 2.14, we can see the same constellation as an object diagram.

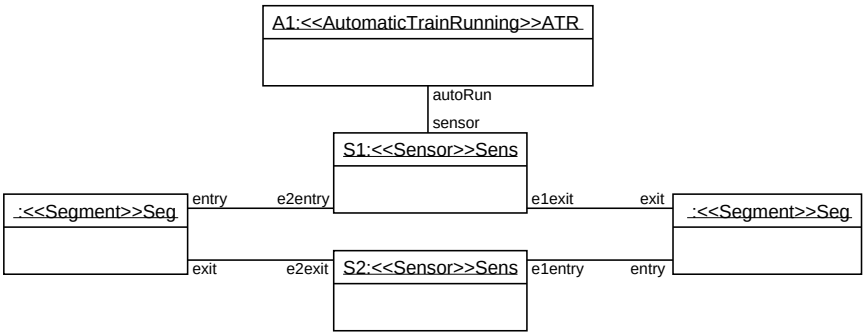

Figure 2.14: Automatic train running instance linked to sensor instance in object notation

2.4.3 CrossingInstance

Description

A CrossingInstance is the instance of a crossing.

Associations

None.

Attributes

None.

Constraints

- Each CrossingInstance is the instance of at least three classifiers. There may be more due to inheritance.

  ```
  classifier->size()≥3
  ```

- Each CrossingInstance is the instance of a Class.

  ```
  classifier->one(c | c.oclIsTypeOf(Class))
  ```

- Each CrossingInstance is the instance of a TrackElement.

  ```
  classifier->one(c | c.oclIsTypeOf(TrackElement))
  ```

- Each CrossingInstance is the instance of a Crossing.

  ```
  classifier->one(c | c.oclIsTypeOf(Crossing))
  ```

- Each CrossingInstance restricts the value of *maxNumberOfTrains* to *1*.

  ```
  slot->select(s1 | s1.definingFeature.name='maxNumberOfTrains')->forAll(s2 | s2.value=1)
  ```

- Each CrossingInstance restricts the value of *limit* to be greater or equal than *0*.

  ```
  slot->select(s1 | s1.definingFeature.name='limit')->forAll(s2 | s2.value≥0)
  ```

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. 2.15 (crossing segments) or in Fig. 2.16. A speed limit can optionally be shown above the crossing just as for segment instances.

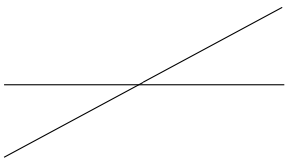

Figure 2.15: Crossing instance

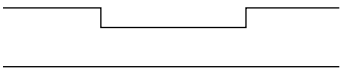

Figure 2.16: Interlaced crossing instance

2.4.4 SegmentInstance

Description

A SegmentInstance is the instance of a segment.

Associations

None.

Attributes

None.

Constraints

- Each SegmentInstance is the instance of at least three classifiers. There may be more due to inheritance.

  `classifier->size()≥3`

- Each SegmentInstance is the instance of a Class.

  `classifier->one(c | c.oclIsTypeOf(Class))`

- Each SegmentInstance is the instance of a TrackElement.

  `classifier->one(c | c.oclIsTypeOf(TrackElement))`

- Each SegmentInstance is the instance of a Segment.

  `classifier->one(c | c.oclIsTypeOf(Segment))`

- Each SegmentInstance restricts the value of *maxNumberOfTrains* to be greater or equal *1*.

  `slot->select(s1 | s1.definingFeature.name='maxNumberOfTrains')->forAll(s2 | s2.value≥1)`

- Each SegmentInstance restricts the value of *limit* to be greater or equal *0*.

  `slot->select(s1 | s1.definingFeature.name='limit')->forAll(s2 | s2.value≥0)`

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. 2.17. 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. 2.18. 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 following constraint holds:

- `slot->select(s1 | s1.definingFeature.name='maxNumberOfTrains')->forAll(s2 | s2.value=1)`

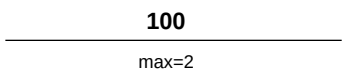

Figure 2.17: Segment instance

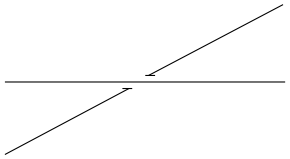

Figure 2.18: Two segment instances that do not cross

- Sink:
```
ownedAttribute->one(a1 | a1.name='e1Entry' or a1.name='e2Entry')->and
ownedAttribute->one(a2 | a2.oclIsTypeOf(SensorAssociation))
```
- Source:
```
ownedAttribute->one(a1 | a1.name='e1Exit' or a1.name='e2Exit')and
ownedAttribute->one(a2 | a2.oclIsTypeOf(SensorAssociation))
```
- Sink/source:
```
(ownedAttribute->select(a1 | a1.name='e1Entry' or a1.name='e1Exit')->size()=2 xor
ownedAttribute->select(a2 | a2.name='e2Entry' or a2.name='e2Exit')->size()=2) and
ownedAttribute->select(a3 | a3.oclIsTypeOf(SensorAssocation))->size()=2
```
- Unidirectional:
```
(ownedAttribute->select(a1 | a1.name='e1Entry' or a1.name='e2Exit')size()=2 xor
ownedAttribute->select(a2 | a2.name='e1Exit' or a2.name='e2Entry')->size()=2) and
ownedAttribute->select(a3 | a3.oclIsTypeOf(SensorAssociation))->size()=2
```
- Bidirectional:
```
ownedAttribute->select(a | a.oclIsTypeOf(SensorAssociation))->size()=4
```

2.4.5 SensorInstance

Description

A SensorInstance is the instance of a sensor.

Associations

None.

Attributes

None.

Constraints

- Each SensorInstance is the instance of at least two classifiers. There may be more due to inheritance.

  ```
  classifier->size()≥2
  ```
- Each SensorInstance is an instance of a Class.

  ```
  classifier->one(c | c.oclIsTypeOf(Class))
  ```
- Each SensorInstance is an instance of a Sensor.

  ```
  classifier->one(c | c.oclIsTypeOf(Sensor))
  ```
- Each SensorInstance restricts the value of *counter* to be greater or equal *0*.

  ```
  slot->select(s1 | s1.definingFeature.name='counter')->forAll(s2 | s2.value ≥ 0)
  ```
- Each SensorInstance restricts the value of *exit* and *entry* to be distinct, i.e. different TrackElements are connected by a Sensor.

  ```
  slot->select(s | s.definingFeature.name='entry' or s.definingFeature.name='exit')->
       forAll(s1,s2 | s1.value <> s2.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. 2.19), b) from the right segment to the left segment (see Fig. 2.20), or c) bidirectionally (see Fig. 2.21). Optionally, black arrows on the segments can be used to make the traveling direction unambigiously clear.

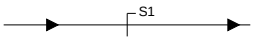

Figure 2.19: Sensor for traveling from the left to the right segment

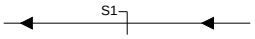

Figure 2.20: Sensor for traveling from the right to the left segment

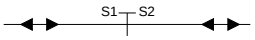

Figure 2.21: Sensor for bidirectional usage

2.4.6 SensorLink

Description

A SensorLink is the instance of a sensor association.

Associations

None.

Attributes

None.

Constraints

- Each SensorLink is an instance of two classifiers.

  ```
  classifier->size()=2
  ```

- Each SensorLink is an instance of an Association.

  ```
  classifier->one(c | c.oclIsTypeOf(Association))
  ```

- Each SensorLink is an instance of a SensorAssociation.

  ```
  classifier->one(c | c.oclIsTypeOf(SensorAssociation))
  ```

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 2.22, 2.23, and 2.24 show the explicit usage in a UML object diagram.

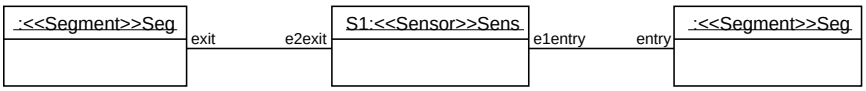

Figure 2.22: Sensor for traveling from the left to the right segment in object notation

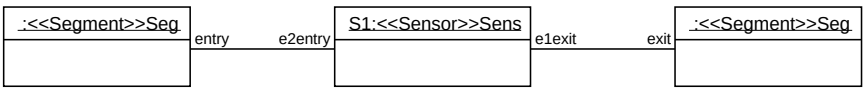

Figure 2.23: Sensor for traveling from the right to the left segment in object notation

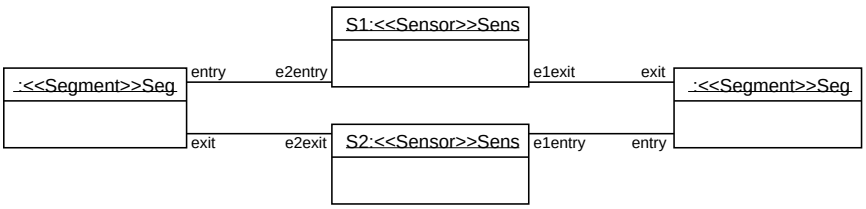

Figure 2.24: Sensor for bidirectional usage in object notation

2.4.7 SignalInstance

Description

A SignalInstance is the instance of a signal.

Associations

None.

Attributes

None.

Constraints

- Each SignalInstance is an instance of at least two classifiers. There may be more due to inheritance.

  `classifier->size()`$\geq$`2`

- Each SignalInstance is an instance of a Class.

  `classifier->one(c | c.oclIsTypeOf(Class))`

- Each SignalInstance is an instance of a Signal.

  `classifier->one(c | c.oclIsTypeOf(Signal))`

- `slot->select(s1 | s1.definingFeature.name='limit')->forAll(s2 | s2.value`$\geq$`0)`

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. 2.25. It is placed in front of the sensor to which it belongs to.

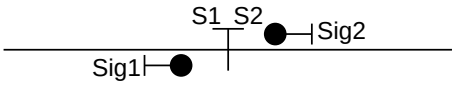

Figure 2.25: Signal on bidirectional segments

There are four different kinds of signals than can be used

- A signal that only gives information *GO* and *STOP* such that:
  `slot->select(s | definingFeature.name='limit')->size()=0 and`
  `slot->select(s | definingFeature.name='direction')->size()=0`

Figure 2.26: Signal that signals *GO* and *STOP*

- A signal that gives additionally speed limits such that:
  `slot->select(s | definingFeature.name='direction')->size()=0`

Figure 2.27: Signal that signals *GO*, *STOP*, and speed limit

- A signal that gives additionally directions such that:
  `slot->select(s | definingFeature.name='limit')->size()=0`

Figure 2.28: Signal that signals *GO*, *STOP*, and direction to go

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

Figure 2.29: Signal with all information possible

2.4.8 SignalLink

Description

A SignalLink is the instance of a signal association.

Associations

None.

Attributes

None.

Constraints

- Each SignalLink is an instance of two classifiers.

  ```
  classifier->size()=2
  ```

- Each SignalLink is an instance of an Association.
  `classifier->one(c | c.oclIsTypeOf(Association))`

- Each SignalLink is an instance of a SignalAssociation.

  ```
  classifier->one(c | c.oclIsTypeOf(SignalAssociation))
  ```

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. 2.25 can be seen explicitly as shown in Fig. 2.30.

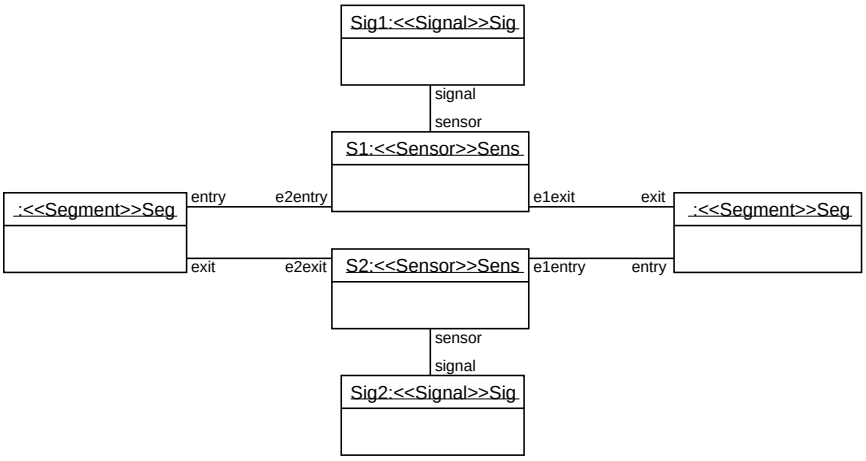

Figure 2.30: Signal on bidirectional segments in object notation

2.4.9 SinglePointInstance

Description

A SinglePointInstance is the instance of a single point.

Associations

None.

Attributes

None.

Constraints

- Each SinglePointInstance is an instance of at least four classifiers. There may be more due to inheritance.

```
classifier->size()≥4
```

- Each SinglePointInstance is an instance of a Class.

```
classifier->one(c | c.oclIsTypeOf(Class))
```

- Each SinglePointInstance is an instance of a TrackElement.

```
classifier->one(c | c.oclIsTypeOf(TrackElement))
```

- Each SinglePointInstance is an instance of a Point.

```
classifier->one(c | c.oclIsTypeOf(Point))
```

- Each SinglePointInstance is an instance of a SinglePoint.

```
classifier->one(c | c.oclIsTypeOf(SinglePoint))
```

- Each SinglePoint restricts the value of *maxNumberOfTrains* to *1*.

```
slot->select(s1 | s1.definingFeature.name='maxNumberOfTrains')->forAll(s2 | s2.value=1)
```

- Each SinglePoint restricts the value of *limit* to be greater or equal *0*.

```
slot->select(s1 | s1.definingFeature.name='limit')->forAll(s2 | s2.value≥0)
```

- One of the two positions *plus* and *minus* must have the value *STRAIGHT*, the other one must have the value *LEFT* or *RIGHT*.

```
slot->select(s1 | s1.definingFeature.name='plus' or s1.definingFeature.name='minus')->
     one(s2 | s2.value=STRAIGHT) and
slot->select(s3 | s3.definingFeature.name='plus' or s3.definingFeature.name='minus')->
     one(s4 | s4.value=LEFT xor s4.value=RIGHT)
```

- Each SinglePointAssociation restricts the values of *actualState* and *requestedState* to the values of *plus* and *minus*.

```
def:  r1:slot->select(s1 | s1.definingFeature.name='actualState' or
     s1.definingFeature.name='requestedState')

def:  r2:  slot->select(s2 | s2.definingFeature.name='plus' or
     s2.definingFeature.name='minus')->collect(value)

     r1->forAll(s3 | r2->including(s3.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. 2.31. At least one position must be marked as *plus* or *minus* position.

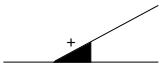

Figure 2.31: Single point instance

2.4.10 SlipPointInstance

Description

A SlipPointInstance is the instance of a slip point.

Associations

None.

Attributes

None.

Constraints

- Each SlipPointInstance is an instance of at least four classifiers. There may be more due to inheritance.

  `classifier->size()≥4`

- Each SlipPointInstance is an instance of a Class.

  `classifier->one(c | c.oclIsTypeOf(Class))`

- Each SlipPointInstance is an instance of a TrackElement.

  `classifier->one(c | c.oclIsTypeOf(TrackElement))`

- Each SlipPointInstance is an instance of a Point.

  `classifier->one(c | c.oclIsTypeOf(Point))`

- Each SlipPointInstance is an instance of a SlipPoint.

  `classifier->one(c | c.oclIsTypeOf(SlipPoint))`

- Each SlipPointInstance restricts the value of *maxNumberOfTrains* to *1*.

  `slot->select(s1 | s1.definingFeature.name='maxNumberOfTrains')->forAll(s2 | s2.value=1)`

- Each SlipPointInstance restricts the value of *limit* to be greater or equal 0.

  `slot->select(s1 | s1.definingFeature.name='limit')->forAll(s2 | s2.value≥0)`

- One of the two positions *plus* and *minus* must have the value *STRAIGHT*, the other one must have the value *LEFT* or *RIGHT*.

```
slot->select(s1 | s1.definingFeature.name='plus' or s1.definingFeature.name='minus')->
     one(s2 | s2.value=STRAIGHT) and
slot->select(s3 | s3.definingFeature.name='plus' or s3.definingFeature.name='minus')->
     one(s4 | s4.value=LEFT xor s4.value=RIGHT)
```

- One of the two positions *plusOpp* and *minusOpp* must have the value *STRAIGHT*. If one of the properties *plus* and *minus* has the value *LEFT*, one of the properties *plusOpp* and *minusOpp* must have the value *RIGHT*.

```
slot->one(s1 | s1.definingFeature.name='plusOpp') and
slot->select(s2 | s2.definingFeature.name='minus' or s2.definingFeature.name='plus')->
     one(s3 | s3.value=LEFT) implies
     slot->select(s4 | s4.definingFeature.name='plusOpp' or
          s4.definingFeature.name='minusOpp')->
               one(s5 | s5.value=RIGHT) and
slot->select(s6 | s6.definingFeature.name='plusOpp' or
          s6.definingFeature.name='minusOpp')->
               one(s7 | s7.value=STRAIGHT)
```

- One of the two positions *plusOpp* and *minusOpp* must have the value *STRAIGHT*. If one of the properties *plus* and *minus* has the value *RIGHT*, one of the properties *plusOpp* and *minusOpp* must have the value *LEFT*.

```
slot->one(s1 | s1.definingFeature.name='plusOpp') and
slot->select(s2 | s2.definingFeature.name='minus' or s2.definingFeature.name='plus')->
     one(s3 | s3.value=RIGHT) implies
     slot->select(s4 | s4.definingFeature.name='plusOpp' or
          s4.definingFeature.name='minusOpp')->
               one(s5 | s5.value=LEFT) and
slot->select(s6 | s6.definingFeature.name='plusOpp' or
          s6.definingFeature.name='minusOpp')->
               one(s7 | s7.value=STRAIGHT)
```

- Each SlipPointAssociation restricts the values of *actualState* and *requestedState* to the values of *plus* and *minus*.

```
def:  r1:slot->select(s1 | s1.definingFeature.name='actualState' or
     s1.definingFeature.name='requestedState')

def:  r2:  slot->select(s2 | s2.definingFeature.name='plus' or
     s2.definingFeature.name='minus')->collect(value)

     r1->forAll(s3 | r2->including(s3.value))
```

- Each SlipPointAssociation restricts the values of *actualStateOpp* and *requestedStateOpp* to the values of *plusOpp* and *minusOpp*.

```
def:  r1:slot->select(s1 | s1.definingFeature.name='actualStateOpp' or
     s1.definingFeature.name='requestedStateOpp')

def:  r2:  slot->select(s2 | s2.definingFeature.name='plusOpp' or
```

```
s2.definingFeature.name='minusOpp')->collect(value)

r1->forAll(s3 | r2->including(s3.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. 2.32 and Fig. 2.33. At least one of the *plus* and *minus* positions of each point has to be marked.

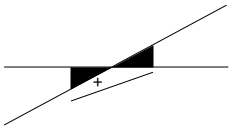

Figure 2.32: Single slip point instance

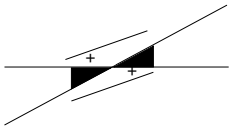

Figure 2.33: Double slip point instance

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.

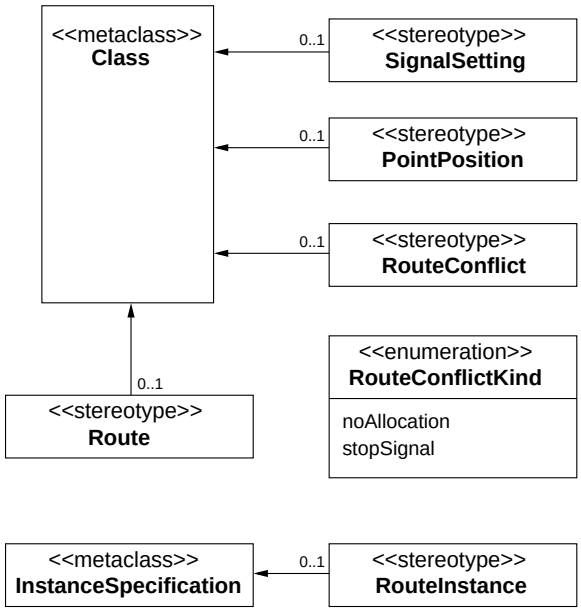

Figure 2.34: Stereotypes for modeling routes in networks

2.5.1 PointPosition

Description

PointPosition is used to model requested positions of point instances.

Associations

None.

Attributes

None.

Constraints

- Each PointPosition has a mandatory attribute *pointId* with type PointId. Its value is constant.

```
ownedAttribute->one(a | a.name='pointId' and
      a.oclIsTypeOf(PointId) and
      a.upperBound()=1 and a.lowerBound()=1 and
      a.isReadOnly=true)
```

- Each PointPosition has a mandatory attribute *pointState* with type RouteKind. Its value is constant.

```
ownedAttribute->one(a | a.name='pointState' and
      a.oclIsTypeOf(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 or as a type. A point position instance is given as a value pair in parenthesis, e.g. `(1, LEFT)`. The first item is the point id, the second item the required point state.

2.5.2 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

- Each Route has a mandatory attribute *routeId* with type RouteId. Its value is constant.

```
ownedAttribute->one(a | a.name='routeId' and
      a.oclIsTypeOf(RouteId) and
```

```
      a.upperBound()=1 and a.lowerBound()=1 and
      a.isReadOnly=true)
```

- Each Route has a mandatory attribute *routeDefinition* with type SensorId. Its multiplicity is *0..** and the series is sequence. The value of *routeDefinition* is constant.

```
ownedAttribute->one(a | a.name='routeDefinition' and
      a.oclIsTypeOf(SensorId) and
      a.upperBound()=* and a.lowerBound()=0 and
      a.isOrdered=true and a.isUnique=false and
      a.isReadOnly=true)
```

- Each Route has a mandatory attribute *routeConflict* with type RouteConflict. Its multiplicity is *0..** and the series is a ordered set. The value of *routeConflict* is constant.

```
ownedAttribute->one(a | a.name='routeConflict' and
      a.oclIsTypeOf(RouteConflict) and
      a.upperBound()=* and a.lowerBound()=0 and
      a.isOrdered=true and a.isUnique=true and
      a.isReadOnly=true)
```

- Each Route has a mandatory attribute *pointPos* with type PointPosition. Its multiplicity is *0..** and the series is a ordered set. The value of *pointPos* is constant.

```
ownedAttribute->one(a | a.name='pointPos' and
      a.oclIsTypeOf(PointPosition) and
      a.upperBound()=* and a.lowerBound()=0 and
      a.isOrdered=true and a.isUnique=true and
      a.isReadOnly=true)
```

- Each Route has a mandatory attribute *signalSetting* with type SignalSetting. The value of *signalSetting* is constant.

```
ownedAttribute->one(a | a.name='signalSetting' and
      a.oclIsTypeOf(SignalSetting) and
      a.upperBound()=1 and a.lowerBound()=1 and
      a.isReadOnly=true)
```

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.

2.5.3 RouteInstance

Description

A RouteInstance is the instance of a route.

Associations

None.

Attributes

None.

Constraints

- Each RouteInstance is the instance of at least two classifiers. There may be more due to inheritance.

```
classifier->size()≥2
```

- Each RouteInstance is an instance of a Class.

```
classifier->one(c | c.oclIsTypeOf(Class))
```

- Each RouteInstance is an instance of a Route.

```
classifier->one(c | c.oclIsTypeOf(Route))
```

- The point positions refer to different points.

```
slot->select(s1 | s1.definingFeature.name='pointPos')->
     select(s2 | s2.definingFeature.name='pointId')->isUnique(s3 | s3.value)
```

- The route conflicts refer to different routes.

```
slot->select(s1 |s1.definingFeature.name='routeConflict')->
     select(s2 | s2.definingFeature.name='routeId')->isUnique(s3 | s3.value))
```

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, e.g. `pointPos=<(1,LEFT),(2,RIGHT),(5,STRAIGHT)>`.

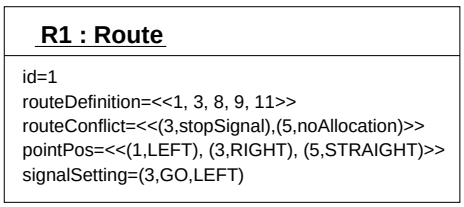

Figure 2.35: Route instance in object notation

2.5.4 RouteConflict

Description

Routes may have conflicts, either because their point and signal settings are incompatible or because they travel on the same segments.

Associations

None.

Attributes

None.

Constraints

- Each RouteConflict has a mandatory attribute *routeId* with type RouteId. Its value is constant.

```
ownedAttribute->one(a | a.name='routeId' and
      a.oclIsTypeOf(routeId) and
      a.upperBound()=1 and a.lowerBound()=1 and
      a.isReadOnly=true)
```

- Each RouteConflict has a mandatory attribute *kind* with type RouteConflictKind. Its value is constant.

```
ownedAttribute->one(a | a.name='kind' and
      a.oclIsTypeOf(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 or as a type. A route conflict instance is given as a value pair in parenthesis, e.g. `(1, noAllocationPossible)`. The first item is the id of the conflicting route, the second item the kind of the conflict.

2.5.5 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`.

2.5.6 SignalSetting

Description

SignalSettings model required signal settings, i.e. required states of signals.

Associations

None.

Attributes

None.

Constraints

- Each SignalSetting has a mandatory attribute *sigId* with type SignalId. Its value is constant.

```
ownedAttribute->one(a | a.name='sigId' and
      a.oclIsTypeOf(SignalId) and
      a.upperBound()=1 and a.lowerBound()=1 and
      a.isReadOnly=true)
```

- Each SignalSetting has a mandatory attribute *sigState* with type PermissionKind. Its value is constant.

```
ownedAttribute->one(a | a.name='sigState' and
      a.oclIsTypeOf(PermissionKind) and
      a.upperBound()=1 and a.lowerBound()=1 and
      a.isReadOnly=true)
```

- Each SignalSetting has a optional attribute *dirState* with type RouteKind. If this attribute exists, its multiplicity is *0..1*, and its value is constant.

```
(ownedAttribute->one(a1 | a1.name='dirState' and
      a1.oclIsTypeOf(RouteKind) and
      a1.upperBound()=1 and a.lowerBound()=0 and
      a1.isReadOnly=true)) or
(not ownedAttribute->exists(a2 | a2.name='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 or as a type. A signal setting instance is given as a value tuple in parenthesis, e.g. `(1, GO, GO_STRAIGHT)`. The first item in the tuple is the signal id, followed by the required signal state to enter a route. Optionally, the third parameter gives the direction.

2.6 Top-Level Constraints

Some constraints that must be fulfilled to achieve a meaningful RCSD model are located on top-level as they describe the interdependencies between different stereotypes.

2.6.1 Identification Numbers

All identification numbers must be unique in the model. That holds for automatic running instances, sensor instances, signal instances, point instances, and route instances.

- The identification numbers of all instances of AutomaticRunningInstance are unique.

```
AutomaticRunningInstance::allInstances()->collect(slots)->
      select(s1 | s1.definingFeature.name='autoRunId')->
            isUnique(s2 | s2.value)
```

- The identification numbers of all instances of SensorInstance are unique.

```
SensorInstance::allInstances()->collect(slots)->
      select(s1 | s1.definingFeature.name='sensorId')->
            isUnique(s2 |s2.value)
```

- The identification numbers of all instances of SignalInstance are unique.

```
SignalInstance::allInstances()->collect(slots)->
      select(s1 | s1.definingFeature.name='signalId')->
            isUnique(s2 | s2.value)
```

- The identification numbers of all instances of SinglePointInstance and SlipPointInstance are unique.

```
SinglePointInstance::allInstances()->collect(slots)->
      union(SlipPointInstance::allInstances()->collect(slots))->
            select(s1 | s1.definingFeature.name='pointId' or
                  s1.definingFeature.name='pointIdOpp')->
                        isUnique(s2 | s2.value)
```

- The identification numbers of all instances of RouteInstance are unique.

```
RouteInstance::allInstances()->collect(slots)->
      select(s1 | s1.definingFeature.name='routeId')->
            isUnique(s2 | s2.value)
```

2.6.2 Sensor Definitions

If two track elements are connected by sensors for bidirectional usage, there are exactly two sensors involved, one for each direction. If the connected segments of two sensors correspond, this must be done so that the sensors work in different directions.

- Two instances of SensorInstance either connect the same TrackElements (referred to by the properties *entry* and *exit*) bidirectionally or not at all.

```
SensorInstance::allInstances()->select(i1,i2 |
      i1->select(s1 | s1.slot.definingFeature.name='entry') =
            i2->select(s2 | s2.slot.definingFeature.name='exit') and
      i1->select(s3 | s3.slot.definingFeature.name='exit') =
```

```
          i2->select(s4 | s4.slot.definingFeature.name='entry'))->
               size()=0 or
SensorInstance::allInstances()->select(i1,i2 |
     i1->select(s1 | s1.slot.definingFeature.name='entry') =
          i2->select(s2 | s2.slot.definingFeature.name='exit') and
     i1->select(s3 | s3.slot.definingFeature.name='exit') =
          i2->select(s4 | s4.slot.definingFeature.name='entry'))->
               size()=2
```

2.6.3 Signal Definitions

Essentially, signals may only point out directions that are possible, i.e. that correspond to possible point positions.

- Signals located at segments can signal only *STRAIGHT* as direction:

```
def:  r1:Segment::allInstances()->select(s1 | s1.definingFeature.name='e1Entry' or
     s1.definingFeature.name='e2Entry')->collect(slots)->
          select(s2 | s2.definingFeature.name='signal')

r1->forAll(s3 | s3->collect(slots)->select(s4 | s4.definingFeature.name='direction')->
     collect(value) = {STRAIGHT})
```

- Signals located at crossings can signal only *STRAIGHT* as direction:

```
def:  r1:Crossing::allInstances()->select(s1 | s1.definingFeature.name='e1Entry'
or
     s1.definingFeature.name='e2Entry' or
     s1.definingFeature.name='e3Entry' or
     s1.definingFeature.name='e4Entry')->collect(slots)->
          select(s2 | s2.definingFeature.name='signal')

r1->forAll(s3 | s3->collect(slots)->select(s4 | s4.definingFeature.name='direction')->
     collect(value) = {STRAIGHT})
```

- Signals located at end2 or end3 of a single point signal only *STRAIGHT* as direction:

```
def:  r1:SinglePoint::allInstances()->select(s1 | s1.definingFeature.name='e2Entry'
or
     s1.definingFeature.name='e3Entry')->collect(slots)->
          select(s2 | s2.definingFeature.name='signal')

r1->forAll(s3 | s3->collect(slots)->select(s4 | s4.definingFeature.name='direction')->
     collect(value) = {STRAIGHT})
```

- Signals located at end1 of a single point can signal only the possible positions of that point as direction:

```
def:  r1:SinglePoint::allInstances()->select(s1 | s1.definingFeature.name='e1Entry')->
     collect(slots)->select(s2 | s2.definingFeature.name='signal')

r1->forAll(s3 | s3->collect(slots)->select(s4 | s4.definingFeature.name='sensor')->
     collect(slots)->select(s5 | s5.definingFeature.name='entry')->
          collect(slots)->select(s6 | s6.definingFeature.name='plus' or
          s6.definingFeature.name='minus')->
```

```
                collect(value)->including(
                     s3->collect(slots)->select(s7 |
                          s7.definingFeature.name='direction')->collect(value)))
```

- Signals located at end2, end3, or end4 of a single slip point can signal only *STRAIGHT* as direction:

```
def:  r1:SlipPoint::allInstances()->select(s1 | s1->select(s2 |
     s2.definingFeature.name='pointIdOpp')->size()=0)

def:  r2:r1->select(s1 | s1.definingFeature.name='e2Entry' or
     s1.definingFeature.name='e3Entry' or
     s1.definingFeature.name='e4Entry')->collect(slots)->
          select(s2 | s2.definingFeature.name='signal')

r2->forAll(s3 | s3->collect(slots)->select(s4 | s4.definingFeature.name='direction')->
     collect(value) = {STRAIGHT})
```

- Signals located at end1 of a single slip point can signal only the possible positions of that point as direction:

```
def:  r1:SlipPoint::allInstances()->select(s1 | s1->select(s2 |
     s2.definingFeature.name='pointIdOpp')->size()=0)

def:  r2:r1->select(s3 | s3.definingFeature.name='e1Entry')->
     collect(slots)->select(s4 | s4.definingFeature.name='signal')

r2->forAll(s5 | s5->collect(slots)->select(s6 | s6.definingFeature.name='sensor')->
     collect(slots)->select(s7 | s7.definingFeature.name='entry')->
          collect(slots)->select(s8 | s8.definingFeature.name='plus' or
          s8.definingFeature.name='minus')->
               collect(value)->including(
                    s4->collect(slots)->select(s9 |
                         s9.definingFeature.name='direction')->collect(value)))
```

- Signals located at end2 or end4 of a double slip point can signal only *STRAIGHT* as direction:

```
def:  r1:SlipPoint::allInstances()->select(s1 | s1->select(s2 |
     s2.definingFeature.name='pointIdOpp')->size()=1)

def:  r2:r1->select(s1 | s1.definingFeature.name='e2Entry' or
     s1.definingFeature.name='e2Entry') or
     s1.definingFeature.name='e4Entry')->collect(slots)->
          select(s2 | s2.definingFeature.name='signal')

r2->forAll(s3 | s3->collect(slots)->select(s4 | s4.definingFeature.name='direction')->
     collect(value) = {STRAIGHT})
```

- Signals located at end1 of a double slip point can signal only possible positions of that point as direction:

```
def:  r1:SlipPoint::allInstances()->select(s1 | s1->select(s2 |
     s2.definingFeature.name='pointIdOpp')->size()=1)
```

```
def:  r2:r1->select(s3 | s3.definingFeature.name='e1Entry')->
      collect(slots)->select(s4 | s4.definingFeature.name='signal')

r2->forAll(s5 | s5->collect(slots)->select(s6 | s6.definingFeature.name='sensor')->
      collect(slots)->select(s7 | s7.definingFeature.name='entry')->
            collect(slots)->select(s8 | s8.definingFeature.name='plus' or
            s8.definingFeature.name='minus')->
                  collect(value)->including(
                        s4->collect(slots)->select(s9 |
                              s9.definingFeature.name='direction')->collect(value)))
```

- Signals located at end3 of a double slip point can signal only possible positions of that point as direction:

```
def:  r1:SlipPoint::allInstances()->select(s1 | s1->select(s2 |
      s2.definingFeature.name='pointIdOpp')->size()=1)

def:  r2:r1->select(s3 | s3.definingFeature.name='e3Entry')->
      collect(slots)->select(s4 | s4.definingFeature.name='signal')

r2->forAll(s5 | s5->collect(slots)->select(s6 | s6.definingFeature.name='sensor')->
      collect(slots)->select(s7 | s7.definingFeature.name='entry')->
            collect(slots)->select(s8 | s8.definingFeature.name='plusOpp' or
            s8.definingFeature.name='minusOpp')->
                  collect(value)->including(
                        s4->collect(slots)->select(s9 |
                              s9.definingFeature.name='direction')->collect(value)))
```

2.6.4 Route Definitions

In case of route definitions, it is required that the requested sensors, points, signals, and routes exist. Point positions and directions must refer to valid point positions and also valid directions.

- The sensors refered to in route definitons by their id exits.

```
def:  r1:RouteInstance::allInstances()->collect(slots)->
      select(s2 | s2.definingFeature='routeDefinition')

def:  r2:SensorInstance::allInstances()->collect(slots)->
      select(s1 | s1.definingFeature.name='sensorId')->collect(value)

r1.forAll(s3 | r2->including(s3.value))
```

- The routes refered to in route conflict definitions by their id exist.

```
def:  r1:RouteInstance::allInstances()->collect(slots)->
      select(s3 | s3.definingFeature.name='routeDefinition')

def:  r2:RouteInstance::allInstances()->collect(slots)->
      select(s1 | s1.definingFeature.name='routeConflict')->collect(slots)->
            select(s2 | s2.definingFeature.name='routeId')->collect(value)
```

```
r1.forAll(s4 | r2->including(s4.value))
```

- The points refered to in point position definitions by their id exist.

```
def:  r1:RouteInstance::allInstances()->collect(slots)->
      select(s1 | s1.definingFeature.name='pointPos')->collect(slots)->
           select(s2 | s2.definingFeature.name='pointId')

def:  r2:SinglePointInstance::allInstances()->collect(slots)->
      select(s3 | s3.definingFeature.name='pointId')->collect(value)

def:  r3:SlipPointInstance::allInstances()->collect(slots)->
      select(s4 | s4.definingFeature.name='pointId'
           or s4.definingFeature.name='pointIdOpp')->collect(value)

r1.forAll(s5 | r2->including(s5.value) or r3->including(s5.value))
```

- The positions refered to in point position definitions belong to existing points (refered to by their id) and have valid positions, i.e. values of the *plus*, *minus*, *plusOpp*, and *minusOpp* properties of the respective point.

```
def:  r1:RouteInstance::allInstances()->collect(slots)->
      select(s1 | s1.definingFeature.name='pointPos')

def:  r2:SinglePointInstance::allInstances()

def:  r3:SlipPointInstance::allInstances()

r1.forAll(s1 |
      r2->exists(s2 |
           s2->collect(slots)->select(s3 | s3.definingFeature.name='plus'
                or s3.definingFeature.name=minus)->collect(value)->
                     including(s1->select(s4 | s4.definingFeature.name='pointState')->
                          collect(value)) and
           s2->collect(slots)->select(s5 | s5.definingFeature.name='pointId')->
                collect(value)->including(s1->select(s6 |
                     s6.definingFeature.name='pointId')->collect(value))) or
      r3->exists(s7 |
           s7->collect(slots)->select(s8 | s8.definingFeature.name='plus'
                or s8.definingFeature.name='minus'
                or s8.definingFeature.name='plusOpp'
                or s8.definingFeature.name='minusOpp')->collect(value)->
                     including(s1->select(s9 | s9.definingFeature.name='pointState')->
                          collect(value)) and
           s7->collect(slots)->select(s10 | s10.definingFeature.name='pointId')->
                collect(value)->including(s1->select(s11 |
                     s11.definingFeature.name='pointId')->collect(value))))
```

- The signals refered to in signal setting definitions by their id exist.

```
def:  r1:RouteInstance::allInstances()->collect(slots)->
      select(s1 | s1.definingFeature.name='signalSetting')->
           select(s2 | s2.definingFeature.name='signalId')
```

```
def:  r2:SignalInstance::allInstances()->collect(slots)->
      select(s3 | s3.definingFeature.name='signalId')->collect(value)

r1.forAll(s4 | r2->including(s4.value))
```

- The directions refered to in signal setting definitions belong to existing signals that are located at a point and correspond to the possible positions of this point.

```
def:  r1:RouteInstance::allInstances()->collect(slots)->
      select(s1 | s1.definingFeature.name='signalSetting')

def:  r2:SignalInstance::allInstances()->collect(slots)->
      select(s2 | s2.definingFeature.name='signalId')->collect(value)

def:  r3:SignalInstance::allInstances()->collect(slots)

r1->forAll(s1 | r2->including(s1->select(s2 | s2.definingFeature.name='signalId')->
      collect(value)) and
      r3->one(s3 | s3->select(s4 | s4.definingFeature.name='signalId')->collect(value)
=
            s1->select(s5 | s5.definingFeature.name='signalId')->collect(value)) and
      r3->select(s6 | s6->select(s7 | s7.definingFeature.name='signalId')->
            collect(value) = s1->select(s8 | s8.definingFeature.name='signalId')->
                  collect(value))->select(s9 | s9.definingFeature.name='sensor')->
                        select(s10 | s10.definingFeature.name='entry')->
                              forAll(oclIsKindOf(Point)) and
      r3->select(s11 | s11->select(s12 | s12.definingFeature.name='signalId)->
            collect(value) = s1->select(s13 | s13.definingFeature.name='signalId')->
                  collect(value))->select(s14 | s14.definingFeature.name='sensor')->
                        select(s15 | s15.definingFeature.name='entry')->
                              select(s16 | s16.definingFeature.name='plus' or
                              s16.definingFeature.name='minus' or
                              s16.definingFeature.name='plusOpp' or
                              s16.definingFeature.name='minusOpp')->collect(value)->
                                    including(s1->select(s17 |
                                          s17.definingFeature.name='dirState')->
                                                collect(value))
```

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.

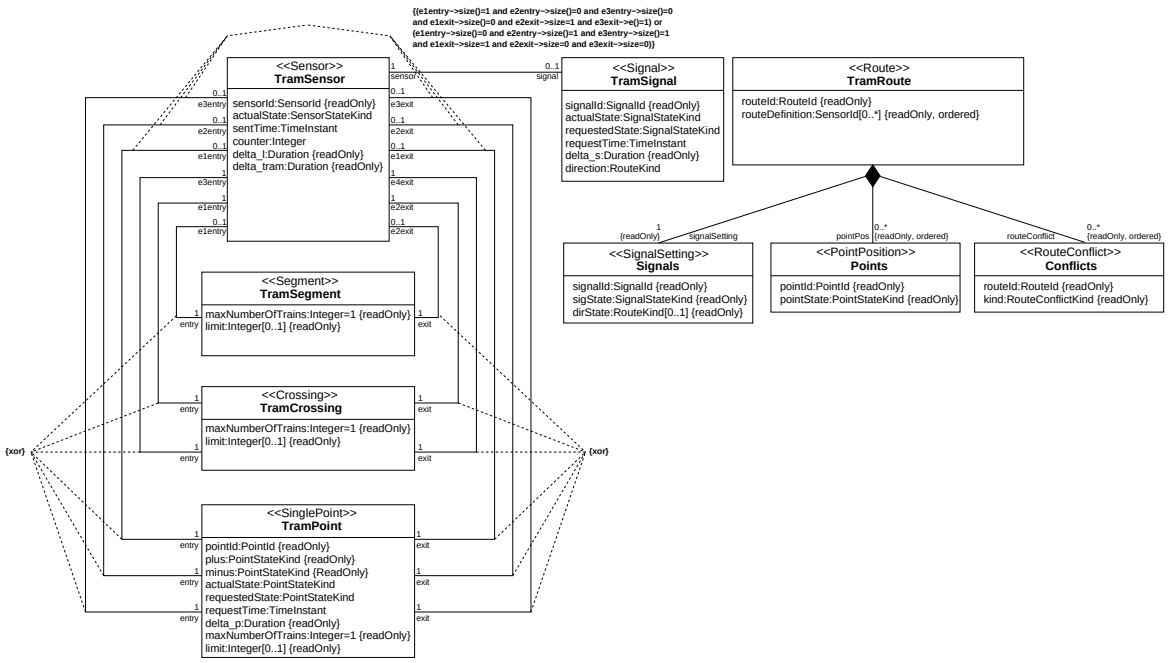

Figure 3.1: Generic tram network

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.

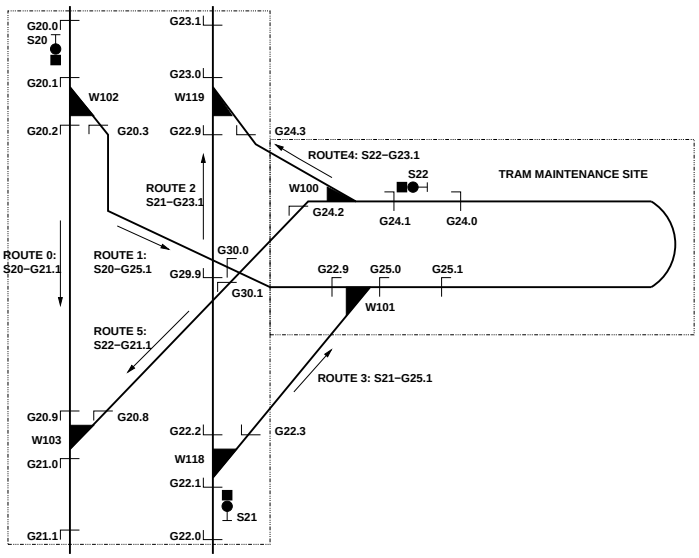

Figure 3.2: Concrete tram network

In Fig. 3.3, the same track network is shown in usual UML notation, i.e. an object diagram. 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.

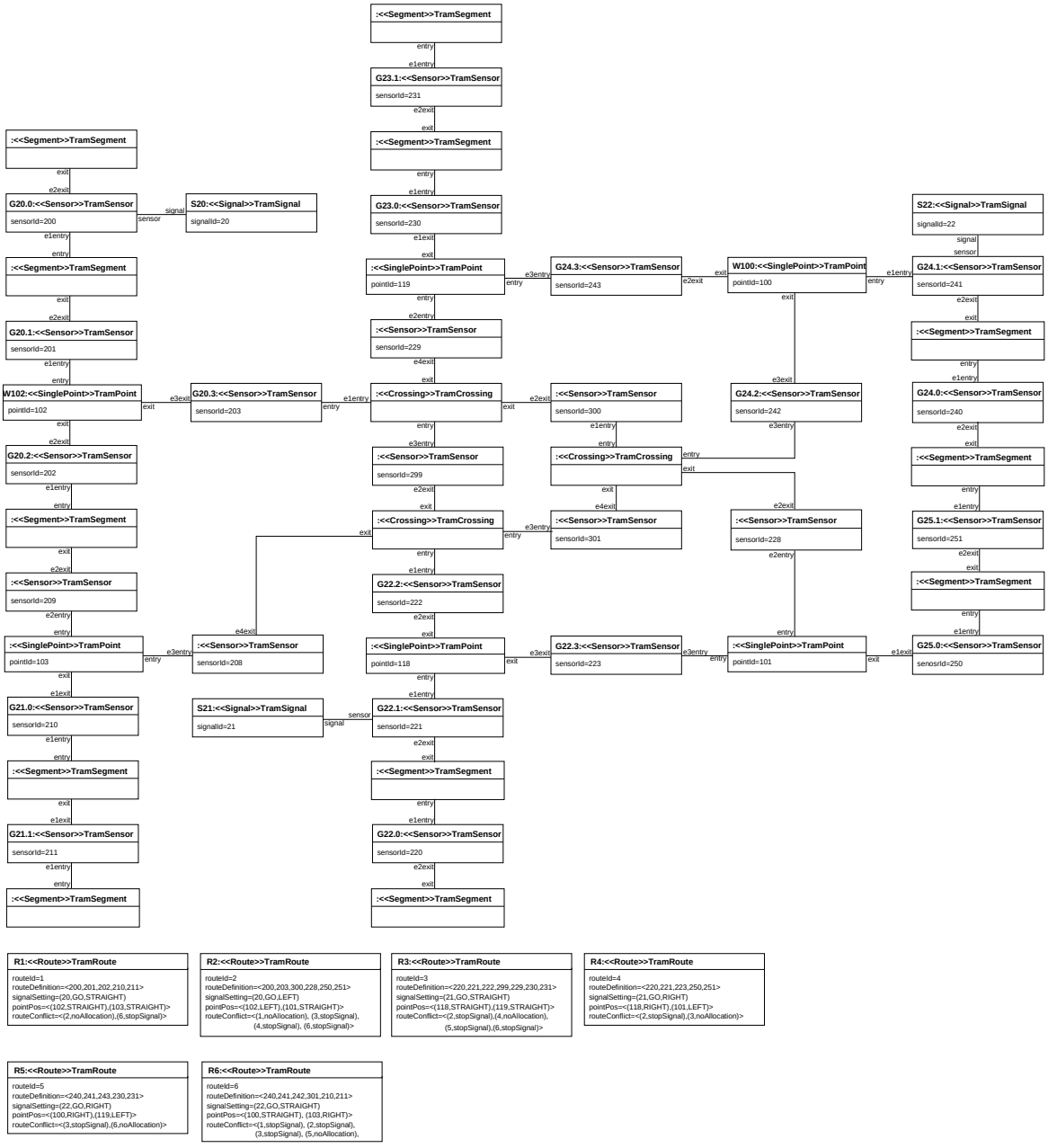

Figure 3.3: Concrete tram network

Bibliography

[BHP] Kirsten Berkenkötter, Ulrich Hannemann, and Jan Peleska. The railway control system domain. Draft, http://www.informatik.uni-bremen.de/agbs/research/RCSD/.

[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 *Lecture Notes in Computer Science*, 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.

[OMG04] Object Management Group. Unified Modeling Language (UML) Specification: Infrastructure, version 2.0. http://www.omg.org/docs/ptc/04-10-14.pdf, October 2004.

[OMG05] Object Management Group. Unified Modeling Language: Superstructure, version 2.0. http://www.omg.org/docs/formal/05-07-04.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.