Ausgewählte Beispiele zum RailML®-Format (Auszug)

RailML-Version: 2.0-2.2
Erstausgabe: Juli 2010
ältere Ausgaben: 03.08.2010, 19.03.2012, 26.03.2012, 16.04.2012, 22.05.2012, 16.01.2013
Stand: 08.12.2014

In diesem Dokument werden in loser Folge und willkürlicher Auswahl typische Beispielsituationen aus dem Eisenbahnbetrieb und deren Abbildung in RailML vorgestellt. Besonderes Augenmerk liegt dabei auf solchen Situationen, die neu in RailML 2.x sind oder deren Abbildung sich geändert hat. Die Auflistung ist in keinerlei Hinsicht vollständig – die Möglichkeiten von RailML sind weitaus umfangreicher, als hier dargestellt werden kann.

Sofern nicht anders genannt, sind die hier angegebenen Beispiele für die RailML-Schemenversionen ab 2.0 gültig. Einige Beispiele beziehen sich auf erst nach Version 2.0 eingeführte Attribute, was dann explizit genannt wird.

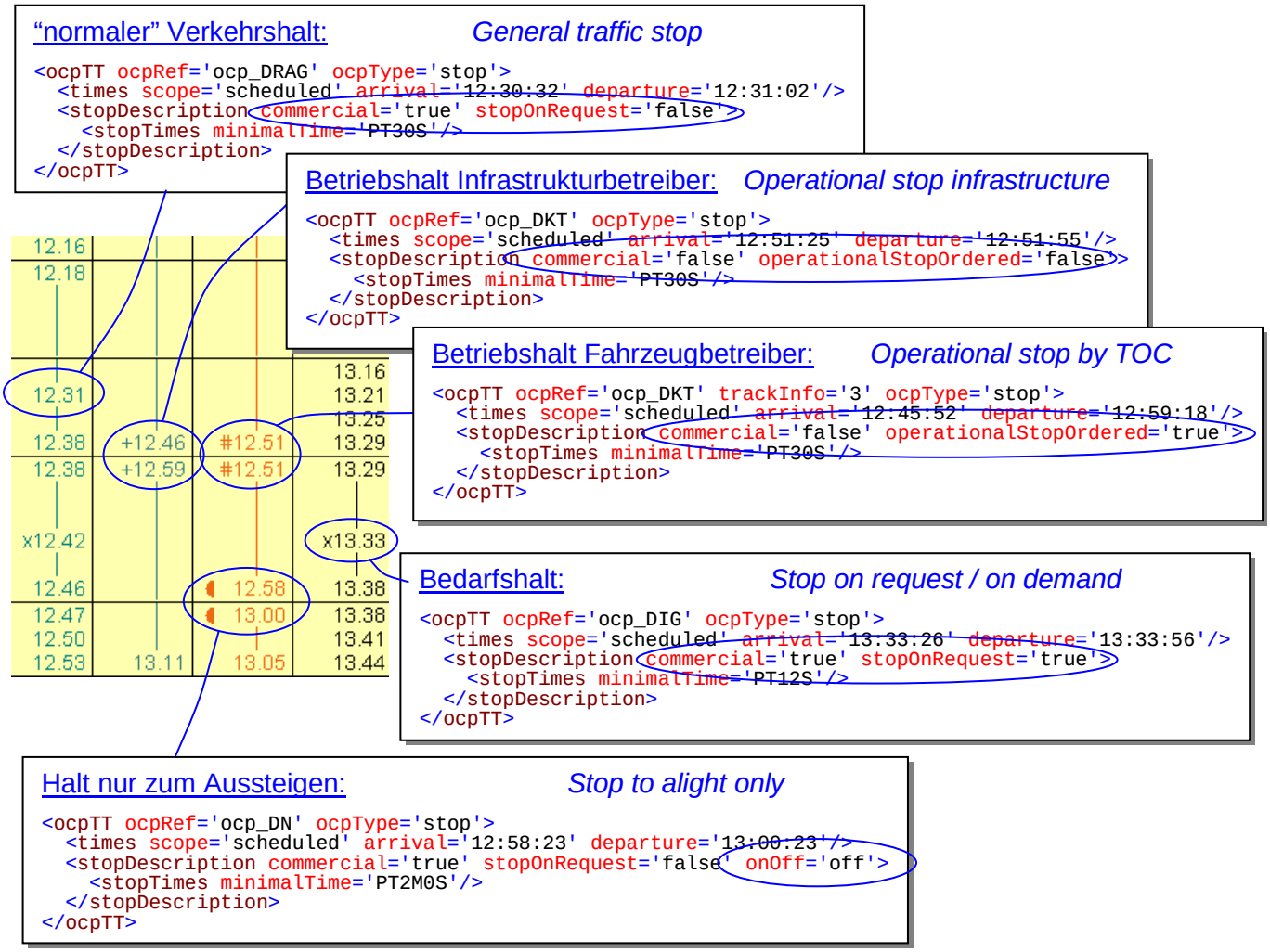
In der Regel sind die Dateien, aus denen hier beispielhaft zitiert wird, unter

www.irfp.de/deutsch/fbs/schnittstelle_railml.html

sowohl als RailML-Dateien als auch als PDF-Dateien (Tabellenfahr- und Umlaufpläne) zum Herunterladen verfügbar.

Themenübersicht

Verschiedene Haltearten



Das Attribut stopOnRequest ist nur anzugeben, wenn *commercial=true* ist.
Das Attribut operationalStopOrdered[1] ist nur anzugeben, wenn *commercial=false* ist.

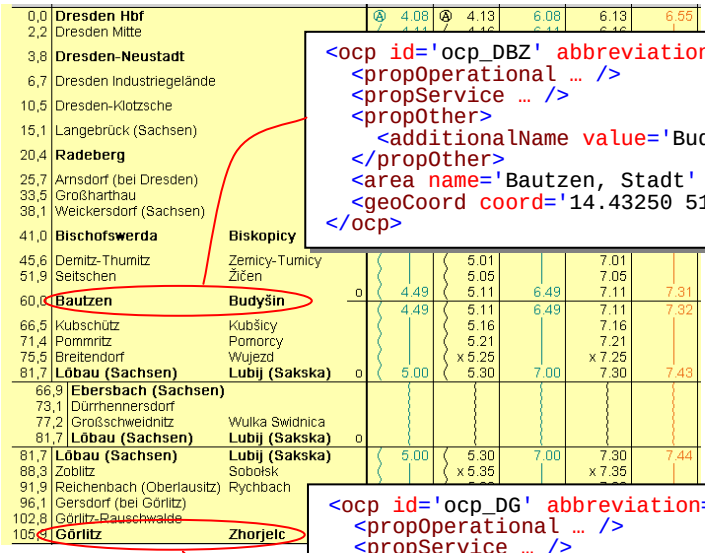
Es ist nicht vorgesehen, mehrere Haltearten gleichzeitig anzugeben. Entsprechend den Gepflogenheiten im Eisenbahnbetrieb gelten:

- Wenn Gründe für einen Verkehrs- und einen Betriebshalt gleichzeitig vorliegen, wird ein Verkehrshalt angegeben.
- Wenn ein Betriebshalt sowohl durch den Infrastrukturbetreiber als auch durch den Fahrzeugbetreiber erforderlich wird, wird er als Betriebshalt durch Fahrzeugbetreiber ausgewiesen (bestellter Betriebshalt).

Ein Bedarfshalt ist eine besondere Form des Verkehrshaltes.

[1] Das Attribut operationalStopOrdered wurde mit RailML 2.2 eingeführt.

Alternative Stationsnamen

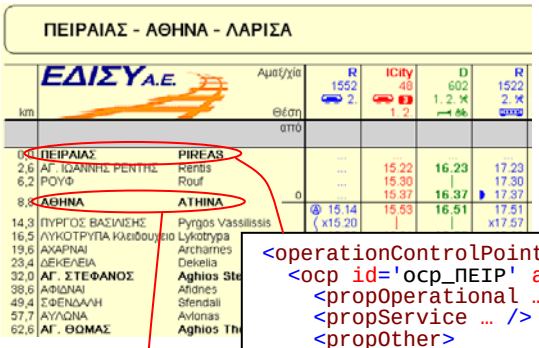



```
<ocp id='ocp_DBZ' abbreviation='DBZ' number='8010026' name='Bautzen'>
  <propOperational … />
  <propService … />
  <propOther>
    <additionalName value='Budyšin' type='trafficName' xml:lang='hsb'/>
  </propOther>
  <area name='Bautzen, Stadt' number='14272010' zip='2625'/>
  <geoCoord coord='14.43250 51.17220 201.47'/>
</ocp>
```

```
<ocp id='ocp_DG' abbreviation='DG' number='8010131' name='Görlitz'>
  <propOperational … />
  <propService … />
  <propOther>
    <additionalName value='Bft. Görlitz Pbf.' type='operationalName'/>
    <additionalName value='Zhorjelc' type='trafficName' xml:lang='pl'/>
  </propOther>
  <area name='Görlitz, Stadt  KfS' number='14263000' zip='2826'/>
  <geoCoord coord='14.98306 51.15072 209.42'/>
</ocp>
```

Mit dem Typ **operationalName** kann ein alternativer Name angegeben werden, der für innerbetriebliche Zwecke zu verwenden ist. Mit dem Typ **trafficName** können mehrere alternative Namen angegeben werden, die zur Veröffentlichung geeignet sind. Diese können sich hinsichtlich Sprache und/oder Zeichensatz unterscheiden, wie aus den Beispielen ersichtlich ist.

Die Angabe des xml:lang-Attributs ist optional und nur selten vorhanden oder gar notwendig. Sofern das Attribut verwendet wird, sind die Sprachcodes nach ISO 639 anzugeben.



Wie nebenstehendes Beispiel zeigt, kann die Unterscheidung alternativer Stationsnamen auch allein im Zeichensatz und nicht in der Sprache liegen. Im Beispiel sind beide angegebenen Namen neugriechisch, jedoch einmal mit griechischen und einmal mit lateinischen Buchstaben.

```
<operationControlPoints>
  <ocp id='ocp_ΠΕΙΡ' abbreviation='ΠΕΙΡ' name='ΠΕΙΡΑΙΑΣ'>
    <propOperational … />
    <propService … />
    <propOther>
      <additionalName value='Πειραιεύς' type='trafficName' xml:lang='grc'/>
      <additionalName value='ΠΕΙΡΑΙΑΣ' type='trafficName' xml:lang='ell'/>
      <additionalName value='PIREAS' type='trafficName' xml:lang='ell'/>
      <additionalName value='Piraeus' type='trafficName' xml:lang='en'/>
      <additionalName value='Piräus' type='trafficName' xml:lang='de'/>
      <additionalName value='Le Pirée' type='trafficName' xml:lang='fr'/>
```

```
<ocp id='ocp_ΑΘΗΝ' abbreviation='ΑΘΗΝ' name='ΑΘΗΝΑ'>
  <propOperational … />
  <propService … />
  <propOther>
    <additionalName value='Ἀθῆναι' type='trafficName' xml:lang='grc'/>
    <additionalName value='ΑΘΗΝΑ' type='trafficName' xml:lang='ell'/>
    <additionalName value='ATHINA' type='trafficName' xml:lang='ell'/>
    <additionalName value='Athens' type='trafficName' xml:lang='en'/>
    <additionalName value='Athen' type='trafficName' xml:lang='de'/>
    <additionalName value='Athènes' type='trafficName' xml:lang='fr'/>
  </propOther>
</ocp>
```

Zugarten, Zuggattungen, Produkte und Fahrgastfahrten

Innerhalb der Elemente <train> und <trainPart> gibt es in RailML derzeit kein Attribut, das unmittelbar erkennen lässt, von welcher Zuggattung ein Zug(teil) ist oder ob es ein Reisezug (Fahrgastfahrt, personenbefördernd) ist. Vielmehr muss hierfür das Attribut categoryRef des Elements <trainPart> „zurückverfolgt“ werden:

```
<trainPart id='tp_222' name='222' line='EC 200' trainNumber='222' processStatus='planned'
                          timetablePeriodRef='ttp_2020_21' categoryRef='cat_EC'>
```

Die Zugarten sind innerhalb der Struktur <timetable> in einer eigenen Liste zusammengefasst. Hier können die optionalen Attribute trainUsage und deadRun zu Rate gezogen werden, um festzustellen, ob eine Zugart i. d. R. für Reiseverkehr vorgesehen ist:

```
<categories>
  <category id='cat_SEV' code='SEV' name='Schienenersatzverkehr'/>
  <category id='cat_OBB' code='OBB' name='Oberlausitz-Bahn' trainUsage='passenger'/>
  <category id='cat_EC' code='EC' name='EuroCity' description='Schnellfahrende Reisezüge
            im internationalen Verkehr mit besonderem Komfort' trainUsage='passenger'/>
  <category id='cat_OBE' code='OBE' name='Oberlausitz-Express' trainUsage='passenger'/>
  <category id='cat_OBC' code='OBC' name='Oberlausitz-City' trainUsage='passenger'/>
  <category id='cat_D' code='D' name='Schnellzug mit Durchgangswagen' description=
                'Schnellfahrende Reisezüge des Fernverkehrs' trainUsage='passenger'/>
  <category id='cat_Os' code='Os'/>
  <category id='cat_CS' code='CS' name='Ganzzug' trainUsage='goods'/>
  <category id='cat_S' code='S' name='Stadtschnellbahn' description='Reisezüge des
                linienbezogenen Ballungsverkehrs mit Systemhalten im dichten
                  Takt unter S-Bahn-Tarifanwendung' trainUsage='passenger'/>
  <category id='cat_FZ' code='FZ' name='FrachtZubringer' trainUsage='goods'/>
  <category id='cat_Lt' code='Lt' name='Leertriebwagen' deadrun='true'/>
  <category id='cat_P' code='P' name='Personenzug' description='Reisezüge des
                Binnenverkehrs der ÖBB auf DB-Infrastruktur und Züge der
                 ÖBB und ČD im Korridorverkehr' trainUsage='passenger'/>
</categories>
```

Es bleibt hier dem lesenden Programm überlassen, wie es damit umgeht, wenn diese Attribute nicht angegeben sind: Es kann einen Standard-Fall annehmen oder beim Anwender nachfragen oder diese Attribute zwingend erfordern, indem es eine Fehlermeldung ausgibt.

Das bei <trainPart> angegebene Attribut categoryRef verweist auf Zugarten (categories), die in der Praxis üblicherweise als **Produkte** bezeichnet und auch veröffentlicht werden. Hierbei ist zu beachten, dass ein Zug gleichzeitig mehrere Produkte haben kann, da verschiedene Zugteile unterschiedliche Produkte haben können (s. a. Kapitel *Allgemeines zu Zügen und Zugteilen*). Dies kommt durchaus auch praktisch vor, z. B. die zwischen Erfurt und Plaue (Thüringen) vereinigt verkehrenden Züge der Erfurter Bahn (Produktbezeichnung *EB*) und Süd-Thüringen-Bahn (Produktbezeichnung *STB*) oder die bei der ÖBB gelegentlich an Züge mit der Produktbezeichnung *RailJet* angehangenen Verstärkungswagen, die jedoch als *IC* vermarktet werden.

Dem gegenüber steht die traditionelle **Zuggattung** als etwas eher Betriebsinternes, wobei ein Zug in einem Streckenabschnitt nur eine Zuggattung haben kann. Für diese Zuggattung gibt es in RailML das Attribut categoryRef im Element <train>, und zwar unter <trainPartSequence>, da auch die betriebliche Gattung innerhalb des Laufwegs eines Zuges wechseln kann. Praktische Beispiele für solche Wechsel wären die vielen Fernreisezüge, die vor und nach ihrem eigentlichen (veröffentlichten) Zuglauf als Leerreisezüge zu- und abgeführt werden, wie das z. B. zwischen Berlin-Grunewald, Lehrter Bahnhof und Rummelsburg üblich ist:

```
<train id='tro_141' type='operational' trainNumber='141' scope='primary'>
  <trainPartSequence sequence='1' categoryRef='cat_IC'>   <!-- IC Schiphol - Berlin -->
    <trainPartRef ref='tp_141_XNSP-BHF' position='1'/>
  </trainPartSequence>
  <trainPartSequence sequence='2' categoryRef='cat_Lr'>  <!-- Leerzug zur Abstellung -->
    <trainPartRef ref='tp_141_BHF-BRGBA' position='1'/>
  </trainPartSequence>
</train>
```

Das Attribut categoryRef unter <train> für die betriebliche Zuggattung wird üblicher Weise nur bei betrieblichen Zügen (Elemente <train> mit Ausprägung type='operational') angegeben.

Es gibt in RailML derzeit keine explizite Unterscheidung zwischen Produkt und Gattung – ein Element <category> kann sowohl als Zuggattung (d. h. referenziert von einem <train>) als auch als Produkt (d. h. referenziert von einem <trainPart>) verwendet werden.

Auch hier bleibt es dem lesenden Programm überlassen, wie es mit theoretisch möglichen Widersprüchen innerhalb der inhaltlichen Redundanz umgeht, falls ein Zug(teil) über seine <trainPart>.categoryRef z. B. als personenbefördernd, über seine <train>.categoryRef jedoch als nicht personenbefördernd dargestellt wird. Es wird empfohlen, sich für verkehrliche Eigenschaften (wie z. B. *personenbefördernd*) am Produkt zu orientieren, für eventuelle betriebliche Eigenschaften (wie vielleicht categoryPriority) jedoch an der Gattung.

Um einen Zug oder Zugteil explizit als personenbefördernd oder nicht personenbefördernd zu deklarieren, muss dieser zwingend über categoryRef auf eine entsprechend definierte <category> verweisen. In diesem Zusammenhang ist jedoch zu beachten, dass gelegentlich das sogenannte „**Überschreiben von Sitzplätzen**“ vorkommt: Hierbei wird das Element <passengerUsage> unter <trainPart> benutzt, um die aus einer Formation (Attribut formationRef) „geerbten“ Platzkapazitäten zu überschreiben, d. h. im Einzelfall zu korrigieren. Falls die Platzkapazitäten dabei auf 0 korrigiert werden, folgt daraus implizit, dass der Zugteil in gewissem Sinne nicht mehr personenbefördernd sein kann – u. U. obwohl seine categoryRef etwas anderes suggeriert. Diese Möglichkeit wird z. B. benutzt, um abgesperrte Wagen innerhalb eines Zuges auszudrücken (ein Teil der Wagen = Zugteil darf von Reisenden ganz normal benutzt werden, ein anderer Zugteil ist jedoch abgesperrt, z. B. weil diese Wagen nicht an einen Bahnsteig passen).

```
<trainPart id='tp_xyz' name='141' … categoryRef='cat_IC'>
   <formationTT formationRef='fmt_abc' … >
     <passengerUsage>
       <places category='class1' count='0'/>
       <places category='class2' count='0'/>
     </passengerUsage>
   </formationTT>
```

Es kann zwar alternativ auch durch Zuweisung eines „Leerzug-Produkts“ zu diesem Zugteil ausgedrückt werden, dass die Wagen als abgesperrt gelten, jedoch ist das nicht immer üblich und oft von Gepflogenheiten des Anwenders abhängig (und kann daher nicht immer durch das schreibende Programm sichergestellt werden). Ebenso sollte i. d. R. nicht der gesamte Zug, sondern nur einige seiner Teile, durch Überschreiben auf 0 Plätze gesetzt sein. Dennoch wird lesenden Programmen empfohlen, beim Feststellen von Eigenschaften wie *personenbefördernd* oder *zu veröffentlichen* auch das Überschreiben der Plätze zu prüfen und ggf. darauf zu reagieren.

Besonderer Hinweis zum Export von RailML aus dem Fahrplanbearbeitungssystem FBS:

Damit die <categories> korrekt und vollständig von FBS nach RailML ausgegeben werden, ist es notwendig, alle verwendeten Zuggattungen in FBS zu konfigurieren. Das ist normalerweise (bei Verwendung des mitgelieferten Zuggattungsverzeichnisses FBS.znv) schon der Fall. Falls jedoch eine bisher noch nicht registrierte Gattung bzw. Produktbezeichnung verwendet wird, sollte geprüft werden, ob diese korrekt (z. B. für Reiseverkehr) konfiguriert ist. Das ist mit dem FBS-Editor für Zuggattungsverzeichnisse möglich (s. Menü Programme → ZNV). Dort sollte sie in der linken Liste eingetragen sein und darin eine Verwendung (eine Art von Reiseverkehr oder Güterverkehr) eingeschaltet sein.

Um das sicherzustellen, gibt es in der FBS-RailML-Schnittstelle unter *inhaltliche Prüfungen* die Option *alle Züge müssen eine definierte Zuggattung haben*. Wenn das Zielprogramm die Fahrgastfahrten korrekt feststellen soll, empfehlen wir, diese Prüfung beim Export aus FBS immer eingeschaltet zu lassen und bei eventuellen Warnmeldungen die Zuggattungen entsprechend zu konfigurieren.

Kilometrierung von Gleisen und Strecken

Aus historischen Gründen ist die Kilometrierung einer Eisenbahnstrecke oft nicht ununterbrochen fortlaufend. Zum Beispiel durch Korrekturen der Streckenführung oder Eignerwechsel kann sie beliebige Unstetigkeitsstellen aufweisen („springen“) und/oder die Zählrichtung wechseln („fallen“).

In RailML wird die praktische, historische (z. B. an Hektometertafeln angeschriebene) Kilometrierung einer Strecke als **absolute Kilometrierung** bezeichnet. Zur genauen Bestimmung von Entfernungen ist aber auch eine ununterbrochene Kilometrierung notwendig. Diese ist i. d. R. virtuell, d. h. in der Praxis nicht direkt erkennbar. Sie wird in RailML als **relative Kilometrierung** bezeichnet. Die relative Kilometrierung eines Gleises (RailML: Attribut pos) ist immer ununterbrochen fortlaufend steigend, jedoch nicht unbedingt bei 0 beginnend.

Der Zusammenhang zwischen relativer und absoluter Kilometrierung erschließt sich durch die **mileageChanges**-Struktur. Die Angabe der absoluten Kilometrierung an den Betriebsstellen (crossSection) ist dazu redundant.

In dem Sonderfall, bei dem die absoluten Kilometrierung auf der gesamten Strecke identisch ist mit der relativen Kilometrierung, kann die gesamte mileageChanges-Struktur in der RailML-Datei fehlen. Es sind dann alle absoluten (außen sichtbaren) Kilometrierungen als identisch zur jeweiligen relativen Position anzusetzen.

Das folgende Beispiel definiert ein Streckengleis, welches
(1) am Anfang fallend kilometriert ist, beginnend mit dem absoluten Kilometer 9,430
(2) nach 9424 Metern, also bei absolutem km 0,006, auf den neuen km-Wert 23,376 springt und von dort an steigend weiterläuft,
(3) nach 18809 Metern (vom Anfang der Strecke), also bei bisherigem absoluten km 32,761 (= 23.376 + 18.809 - 9.424) auf den neuen km-Wert 33,391 springt und von dort an wiederum fallend weiterläuft

```
<track id='tr_11.5107_1' name='OBW-OWT' type='mainTrack'>
  <trackTopology>
    <trackBegin id='trn_OBW_11.5107_1' pos='0' absPos='9430'>
      <macroscopicNode ocpRef='ocp_OBW'/>
    </trackBegin>
    <trackEnd id='trn_OWT_11.5107_1' pos='18820' absPos='33380'>
      <macroscopicNode ocpRef='ocp_OWT'/>
    </trackEnd>
    <mileageChanges>
      <mileageChange id='mch_11.07_0' absPos='9430' pos='0' dir='down'/>          (1)
      <mileageChange id='mch_11.07_1' absPosIn='6' absPos='23376' pos='9424' dir='up'/>   (2)
      <mileageChange id='mch_11.07_2' absPosIn='32761' absPos='33391' pos='18809' dir='down'/>   (3)
    </mileageChanges>
    <crossSections>
      <crossSection id='trn_OBW' pos='1290' absPos='8140' ocpRef='ocp_OBW_76A'/>
      <crossSection id='trn_OSML' pos='3570' absPos='5860' ocpRef='ocp_OSML'/>
      <crossSection id='trn_OPT' pos='5580' absPos='3850' ocpRef='ocp_OPT'/>
      <crossSection id='trn_ONKW_H' pos='8930' absPos='500' ocpRef='ocp_ONKW_H'/>
      <crossSection id='trn_ONKW' pos='9430' absPos='23382' ocpRef='ocp_ONKW'/>
      <crossSection id='trn_ONKW' pos='10300' absPos='24252' ocpRef='ocp_ONKW_A'/>
      <crossSection id='trn_ONKO' pos='13460' absPos='27412' ocpRef='ocp_ONKO'/>
      <crossSection id='trn_OWT_N' pos='17945' absPos='31897' ocpRef='ocp_OWT_N'/>
    </crossSections>
  </trackTopology>
</track>
```

Die Pos. 1 ist nur in FBS-RailML (interne Version 2.0.5) vorhanden. Es ist ersichtlich, wie die Elemente <trackBegin>, <trackEnd> und <crossSection> in ihren absPos-Attributen die jeweils an dieser Position geltende absolute Kilometrierung angeben.

Das Attribut absPosIn gibt auf redundante Art die „alte“ (bisherige) absolute Kilometrierung an einem <mileageChange>-Element an. Dieser Wert ließe sich immer auch aus dem vorherigen <mileageChange>-Element berechnen.

Es sei angemerkt, dass die Ausprägungen *up* und *down* des Attributs dir numerisch („hochzählen“, „runterzählen“) zu interpretieren sind und damit vom britischen Sprachgebrauch abweichen, wo *up* = „in Richtung London“ und *down* = „weg von London“ bedeuten.

Mit dem optionalen Attribut type kann die Art Kilometrierungssprungs angegeben werden (sofern es sich dabei nicht um einen Kilometrierungsrichtungswechsel[2] handelt). Dies bezieht sich terminologisch auf die in der Praxis unterschiedenen Begriffe

„**Fehlerstelle**“: type = *missing*
„**Überglänge**“: type = *overlapping*

Das folgende Beispiel definiert eine Kilometrierung, die von km 32,567 auf km 32,600 springt. Da es sich um *steigende* Kilometrierung handelt (dir=up) und die Werte zwischen 32,567 und 32,600 also fehlen, handelt es sich um eine **Fehlerstelle**:

```
<mileageChanges>
  <mileageChange id='mch_80.6653_0' absPos='32500' pos='0' dir='up'/> [3]
  <mileageChange id='mch_80.6653_1' absPosIn='32567' type='missing' absPos='32600'
                                                                    pos='67' dir='up'/>
</mileageChanges>
```

Das folgende Beispiel definiert eine Kilometrierung, die von km 22,233 auf km 21,200 „zurückspringt“. Da es sich um *steigende* Kilometrierung handelt (dir=up) kommen die Werte zwischen 21,200 und 22,233 doppelt vor, es handelt sich also um eine **Überlänge**:

```
<mileageChanges>
  <mileageChange id='mch_80.6383.1_1_0' absPos='6693' pos='0' dir='up'/> [3]
  <mileageChange id='mch_80.6383.1_1_1' absPosIn='22233' type='overlapping' absPos='21200'
                                                                    pos='15540' dir='up'/>
</mileageChanges>
```

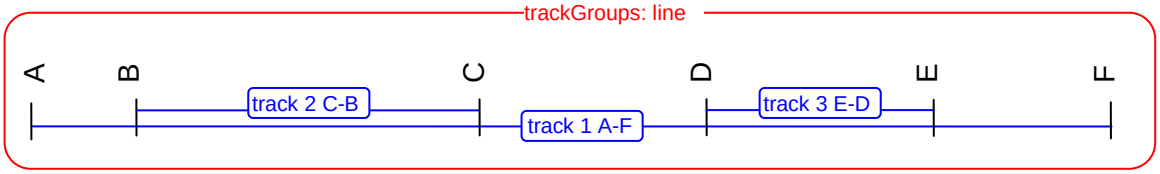
[2] Ein Kilometrierungsrichtungswechsel kommt auf Strecken der DB Netz AG und vielen anderen EIU nicht vor, da diese in solchen Fällen eine neue Streckennummer vergeben. Damit wird der Fall in RailML auf zwei tracks aufgeteilt.

[3] Die initiale Zeile ist nur in FBS-RailML (interne Version 2.0.5) vorhanden. Sie wird hier nur wiedergegeben zum leichteren Verständnis. Mit den derzeitigen RailML-Schemen ist die initiale Kilometrierungsrichtung nur durch Auswerten von <trackBegin>.absPos und <mileageChange>.absPosIn ermittelbar. Dies soll (u. a.) ab RailML 3.0 geändert werden.

Fahrt auf dem Gegengleis

In RailML gibt es zunächst keine explizite Kennzeichnung der Regelfahrrichtung eines Gleises. Auch bei Gruppierung der Gleise (RailML: tracks) zu Strecken (RailML: trackGroups) sind alle Gleise zunächst gleichwertig.

Beispiel für die Gruppierung von drei (Strecken-)Gleisen zu einer Strecke:



In dieser Philosophie der Gruppierung von Gleisen zu Strecken ergibt sich eine Regelfahrrichtung – sofern überhaupt notwendig – nur sehr indirekt z. B. durch entsprechende sicherungstechnische Ausrüstung bzw. das Fehlen derselben (was in RailML nur erkennbar ist, sofern das entsprechende sicherungstechische Teilschema verwendet wird) oder gar nur durch das überwiegende Vorkommen von Zügen einer Fahrtrichtung.

Sofern dennoch die Angabe einer Regelfahrrichtung gewünscht wird, ist dies mit dem Attribut **mainDir** möglich:

```
<track id='tr_80.6214_1' name='DOZ U-DZ' type='mainTrack' mainDir='up'>
  <trackTopology>
    <trackBegin id='trn_DOZ_U_80.6214_1' pos='52670' absPos='37902'>
      <macroscopicNode ocpRef='ocp_DOZ_U'/>
    </trackBegin>
    <trackEnd id='trn_DZ_80.6214_1' pos='63827' absPos='26745'>
      <macroscopicNode ocpRef='ocp_DZ'/>
    </trackEnd>
    <mileageChanges>
      <mileageChange id='mch_80.6214_1_52670' absPos='37902' pos='52670' dir='down'/>
    </mileageChanges>
    <crossSections>
      <crossSection id='trn_DNOW_80.6214_1' pos='54102' absPos='36470' ocpRef='ocp_DNOW'/>
      <crossSection id='trn_DMHD_G_80.6214_1' pos='57147' absPos='33425' ocpRef='ocp_DMHD_G'/>
      <crossSection id='trn_DMHD_80.6214_1' pos='57872' absPos='32700' ocpRef='ocp_DMHD'/>
      <crossSection id='trn_DMHD_A_80.6214_1' pos='58484' absPos='32088' ocpRef='ocp_DMHD_A'/>
      <crossSection id='trn_DZ_Z_80.6214_1' pos='61782' absPos='28790' ocpRef='ocp_DZ_Z'/>
      <crossSection id='trn_DZ_T_80.6214_1' pos='62672' absPos='27900' ocpRef='ocp_DZ_T'/>
    </crossSections>
  </trackTopology>
</track>
```

Das Attribut mainDir hat allerdings keine technische Bedeutung, d. h. es folgt keinerlei technische bzw. infrastrukturelle Eigenschaft des Gleises aus seinem Wert.

Zu beachten ist, dass mainDir sich auf die *relative* Kilometrierungsrichtung bezieht und damit vom Attribut mileageChange.dir abweicht, wo sich dir natürlich auf die absolute Kilometrierungsrichtung bezieht. Der Grund für die Festlegung auf relative Kilometrierungsrichtung ist, dass sich mainDir immer auf das *gesamte* Gleis bezieht, während die absolute Kilometrierungsrichtung innerhalb des Gleises mehrfach wechseln kann und damit nicht eindeutig wäre.

Im konkreten Beispiel wird das Gleis in der Regel **in Richtung aufsteigender relativer Kilometrierung** gebraucht (mainDir=up), währenddessen die absolute Kilometrierung fallend ist (dir=down). Das Gleis wird folglich in der Regel in Richtung fallender absoluter Kilometrierung gebraucht.

Ebenso wie die Regelfahrrichtung ergibt sich bei der RailML-Philosophie eine **Zugfahrt auf dem Gegengleis** zunächst indirekt, indem ein Zugteil über sein Attribut trackRef ein Gleis referenziert, welches offensichtlich nicht das für diese Fahrtrichtung gewöhnliche ist.

Um hier vereinfachend ein Erkennen der Fahrt auf dem Gegengleis zu ermöglichen, ohne die Infrastruktur dereferenzieren zu müssen, kann das optionale Attribut **trackInfo** verwendet werden. Es wird empfohlen, hier

trackInfo=1 für Regelgleis,
trackInfo=2 für Gegengleis

zu verwenden und das Attribut nur auf zweigleisigen Streckenabschnitten anzugeben. Dies ist jedoch keine zwingende RailML-Vorschrift.

```
<ocpTT ocpRef='ocp_DKT_W4' ocpType='pass'>
  <times scope='scheduled' departure='13:36:46'/>
  <sectionTT section='DKTW4-DLAB' lineRef='ln_80.6212' trackRef='tr_80.6212_2' trackInfo='1'/>
</ocpTT>
<ocpTT ocpRef='ocp_DLAB' ocpType='pass'>
  <times scope='scheduled' departure='13:40:36'/>
  <sectionTT section='DLAB-DRAG' lineRef='ln_80.6212' trackRef='tr_80.6212_2' trackInfo='1'/>
</ocpTT>
<ocpTT ocpRef='ocp_DRAG' ocpType='pass'>
  <times scope='scheduled' departure='13:46:18'/>
  <sectionTT section='DRAG-DAF W' lineRef='ln_80.6212' trackRef='tr_80.6212_1' trackInfo='2'/>
</ocpTT>
<ocpTT ocpRef='ocp_DAF_W' ocpType='pass'>
  <times scope='scheduled' departure='13:49:29'/>
  <sectionTT section='DAF W-DAF' lineRef='ln_80.6212' trackRef='tr_80.6212_1' trackInfo='2'/>
</ocpTT>
<ocpTT ocpRef='ocp_DAF' ocpType='pass'>
  <times scope='scheduled' departure='13:50:51'/>
  <sectionTT section='DAF-DSETB' lineRef='ln_80.6212' trackRef='tr_80.6212_2' trackInfo='1'/>
</ocpTT>
<ocpTT ocpRef='ocp_DSET_B' ocpType='pass'>
  <times scope='scheduled' departure='13:55:14'/>
  <sectionTT section='DSETB-DGRT' lineRef='ln_80.6212' trackRef='tr_80.6212_2' trackInfo='1'/>
</ocpTT>
<ocpTT ocpRef='ocp_DGRT' ocpType='pass'>
  <times scope='scheduled' departure='13:57:40'/>
  <sectionTT section='DGRT-DBW' lineRef='ln_80.6212' trackRef='tr_80.6212_2' trackInfo='1'/>
</ocpTT>
```

Im konkreten Beispiel fährt der Zug bis DRAG und ab DAF auf dem Regelgleis einer zweigleisigen Strecke entgegen der Kilometrierung, was in diesem Falle Streckengleis 2 (`tr_80.6212_2`) ist. Das Regelgleis ist zusätzlich erkennbar an trackInfo=1. Im Abschnitt DRAG-DAF fährt er jedoch auf dem Gegengleis, hier erkennbar indirekt an trackRef =`tr_80.6212_1` sowie direkt an trackInfo=2.

Hinweis: Der hier gezeigte Ausschnitt bezieht sich auf FBS-RailML 2.05. trackRef wird ab RailML 2.1 als Unter-Element und nicht als Attribut ausgegeben; inhaltlich treffen die Aussagen aber weiterhin unverändert zu.

Allgemeines zu Zügen und Zugteilen

Die Grundphilosophie der RailML-*timetable*-Version 2.0 ist, den vielfältigen Praxisanforderungen des Eisenbahnbetriebs wie z. B. Flügeln von Zügen, Verstärken, Wagenzugdurchlauf, Einsatz von Kurswagen usw. durch „Zerlegen“ von Zügen in kleinste, unteilbare Einheiten gerecht zu werden. Diese kleinsten unteilbaren Zugeinheiten werden **trainParts** genannt. Die eigentlichen Zuginformationen wie Zeiten, Fahrzeuge usw. sind an den Zugteilen zu finden. Die **Zug**-Struktur (**<train>**) in RailML hingegen fasst lediglich Zugteile zu Zügen zusammen, enthält aber darüber hinausgehend i. d. R. keine weiteren Informationen.

Während sich innerhalb eines Zuglaufes (RailML-Element train) z. B. die Verkehrstage oder die Anzahl Fahrzeuge ändern können, sind innerhalb eines Elements trainPart immer alle Eigenschaften konstant.

Die RailML-Elemente **train** können je einen betrieblichen *oder* verkehrlichen Zug enthalten. Wenn das train-Element einen betrieblichen Zug enthält, ist dessen Attribut type=*operational.* Wenn das train-Element einen verkehrlichen Zug enthält, ist dessen Attribut type=*commercial.*

Das Zusammensetzen aus **betrieblicher Sicht** ergibt Züge, wie sie der Eisenbahner und das Eisenbahninfrastrukturunternehmen „sehen“, jedoch nicht unbedingt wie sie der Reisende sieht. Die Grundeigenschaft betrieblicher Züge ist, dass zu einem Zeitpunkt auf einem Streckengleis nur ein Zug fahren kann. Dieser Zug muss klar durch eine betriebliche Zugnummer definiert sein. Dieser Sichtweise kommt teilweise ein Sicherheitsaspekt zu (z. B. Verständigung zwischen den Betriebsstellen); sie wird daher als grundlegend empfunden.

Das Zusammensetzen aus **verkehrlicher Sicht** ergibt „Züge“, wie der Reisende sie i. d. R. sieht und wie sie z. B. in Aushang- oder Tabellenfahrplänen (Kursbuch) oder elektronischen Fahrplanmedien angegeben werden. Hierbei können zu einem Zeitpunkt auf (quasi) einem Streckengleis mehrere (verkehrliche) Züge gleichzeitig „fahren“: In einem Tabellenfahrplan werden gekuppelte Züge durch zwei benachbarte Spalten mit gleichen Zeiten dargestellt. In einem Aushangfahrplan können zwei Zeilen mit gleicher Abfahrtszeit und gleichem Gleis vorhanden sein, wobei beide Zeilen unterschiedliche Ziele angeben, die Zugteile aber erst im späteren Zuglauf getrennt werden. Bedingt durch die Sichtweise trifft der Begriff „Zug“ hier nur im weitesten Sinne zu: So müssen „verkehrliche Züge“ nicht unbedingt ein Triebfahrzeug haben (z. B. Kurswagen).

Jeder Zugteil wird i. d. R. durch genau je einen betrieblichen und einen verkehrlichen Zug erfasst.

Jeder Zug (**train**-Element) führt alle Zugteile, aus denen er besteht, in seiner Struktur trainPartRef mit ref und position auf. Ein Zug kann dabei entweder gleichzeitig (an einer Stelle seines Zuglaufs) oder nacheinander (in aufeinanderfolgenden Abschnitten seines Zuglaufs) aus verschiedenen Teilen bestehen. Der Wert trainPartRef.position zählt die Stellung im Zug an einer Stelle im Zuglauf. Es kann daher mehrere trainPartRef-Einträge mit gleicher position geben, wenn die entsprechenden Zugteile an verschiedenen Abschnitten im Zuglauf vorkommen.

Zu beachten ist, dass trainPartRef.position nicht unbedingt die tatsächliche Stellung im Zug angibt - durch unterschiedliche Verkehrstage der Zugteile können Teile mit niedrigerem position-Wert an bestimmten Tagen im Zugverband „fehlen“.

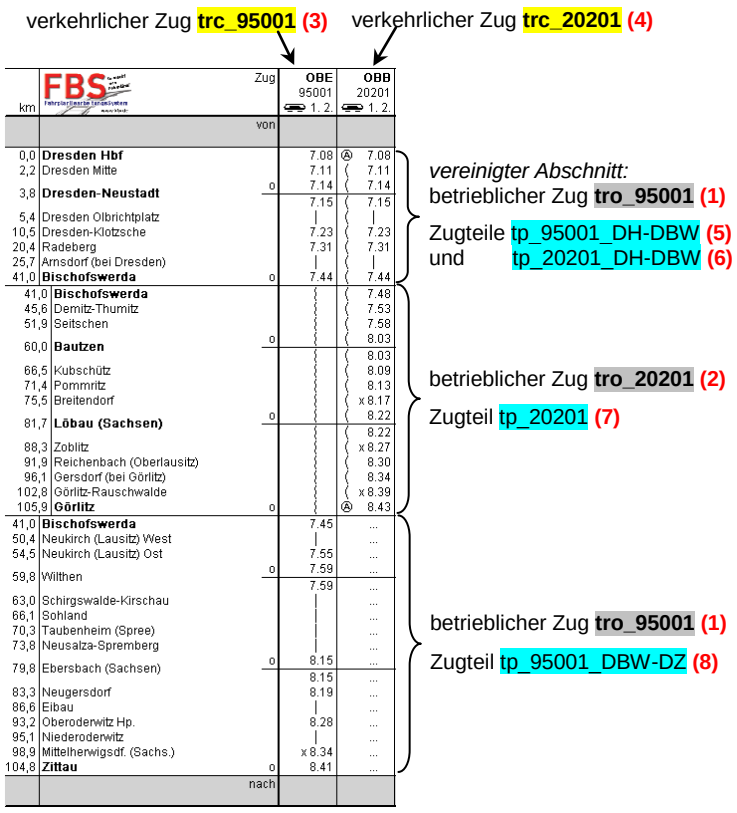
Mehrere Zugteile an einer Stelle eines Zuglaufs kommen z. B. beim (abschnittsweisen) Verstärken oder Flügeln von Zügen vor. Aufeinanderfolgende Abschnitte eines Zuglaufs kommen z. B. vor, wenn unterwegs Verkehrstage wechseln.

Flügeln von Zügen

Unter Flügeln von Zügen ist hier das vereinigte Verkehren zweier Zugteile (meist Triebwagen) zu verstehen, die auf einem Teilabschnitt ihres Laufweges auch unabhängig voneinander verkehren. Dies ist im Prinzip nicht zu unterscheiden vom klassischen Begriff des *Doppelzuges*. Hingegen stellen die bei DB Netz gebräuchlichen Begriffe *Start-* und *Zielflügel* etwas anderes dar, worauf unter *Mehrfachzuglauf/Ergänzungsfahrpläne* eingegangen wird.

Beispiel 1

Zunächst ist festzuhalten, dass beim Flügeln von Zügen ein wichtiger Unterschied zwischen der **betrieblichen** und der **verkehrlichen** Sichtweise auf die Züge besteht: Betrieblich (insbesondere aus Sicht des Eisenbahninfrastrukturunternehmens (EIU)) gibt es im vereinigten Abschnitt (im Beispiel: Dresden Hbf – Bischofswerda) nur *einen* Zug, während es aus verkehrlicher Sicht zwei (vereinigte) Züge sind. Die im Tabellenfahrplan im Spaltenkopf und in anderen verkehrlichen Auskunftsmedien angegebene Nummer ist daher *nicht* identisch mit der eigentlichen betrieblichen Zugnummer. Dieser Unterschied ist oft Grund für Verwirrung, da der Begriff *Zugnummer* hier nicht immer „sauber“ verwendet wird.



In RailML werden die Züge zunächst in kleinste, nicht weiter unterteilbare Einheiten zerlegt. Diese kleinsten unteilbaren Zugeinheiten werden Zugteile genannt. Die eigentlichen Zuginformationen wie Zeiten, Fahrzeuge usw. sind an den Zugteilen zu finden. Zugteile werden dann sowohl zu betrieblichen als auch zu verkehrlichen Zügen zusammengesetzt.

Das Attribut *id* der Züge und Zugteile dient nur der eindeutigen Bezeichnung und darf nicht interpretiert werden. Aus dem Attribut *trainNumber* der betrieblichen Züge ist erkennbar, unter welcher (offiziellen) Zugnummer der Zug beim EIU geführt wird.

Die betrieblichen und verkehrlichen Züge können am Attribut type (=*operational* oder *commercial*) unterschieden werden.

Auszüge aus der zugehörigen RailML-Datei:

1
```
<train id='tro_95001' type='operational' trainNumber='95001'>
  <trainPartSequence sequence='1'>
    <trainPartRef ref='tp_95001_DH-DBW' position='1'/>
    <trainPartRef ref='tp_20201_DH-DBW' position='2'/>
  </trainPartSequence>
  <trainPartSequence sequence='2'>
    <trainPartRef ref='tp_95001_DBW-DZ' position='1'/>
  </trainPartSequence>
</train>
```

2
```
<train id='tro_20201' type='operational' trainNumber='20201'>
  <trainPartSequence sequence='1'>
    <trainPartRef ref='tp_20201' position='1'/>
  </trainPartSequence>
</train>
```

3
```
<train id='trc_95001' name='95001' type='commercial'>
  <trainPartSequence sequence='1'>
    <trainPartRef ref='tp_95001_DH-DBW' position='1'/>
  </trainPartSequence>
  <trainPartSequence sequence='2'>
    <trainPartRef ref='tp_95001_DBW-DZ' position='1'/>
  </trainPartSequence>
</train>
```

4
```
<train id='trc_20201' name='20201' type='commercial'>
  <trainPartSequence sequence='1'>
    <trainPartRef ref='tp_20201_DH-DBW' position='2'/>
  </trainPartSequence>
  <trainPartSequence sequence='2'>
    <trainPartRef ref='tp_20201' position='1'/>
  </trainPartSequence>
</train>
```

5
```
<trainPart id='tp_95001_DH-DBW' trainNumber='95001' categoryRef='cat_OBE'>
  <formationTT formationRef='fmt_0'/>
  <operatingPeriodRef ref='opp_1'/>
  <ocpsTT>
    <ocpTT ocpRef='ocp_DH' ocpType='begin'>
      <times scope='scheduled' departure='07:08:18'/>
    </ocpTT>
    …
    <ocpTT ocpRef='ocp_DN' ocpType='stop'>
      <times scope='scheduled' arrival='07:14:07' departure='07:15:07'/>
    </ocpTT>
    …
    <ocpTT ocpRef='ocp_DBW' trackInfo='6' ocpType='end'>
      <times scope='scheduled' arrival='07:43:31'/>
    </ocpTT>
  </ocpsTT>
</trainPart>
```

ocp_DH = Dresden Hbf
ocp_DN = Dresden-Neustadt
ocp_DBW = Bischofswerda

6
```
<trainPart id='tp_20201_DH-DBW' trainNumber='95001' categoryRef='cat_OBB'>
  <formationTT formationRef='fmt_0'/>
  <operatingPeriodRef ref='opp_0'/>
  <ocpsTT>
    <ocpTT ocpRef='ocp_DH' ocpType='begin'>
      <times scope='scheduled' departure='07:08:18'/>
    </ocpTT>
    …
    <ocpTT ocpRef='ocp_DN' ocpType='stop'>
      <times scope='scheduled' arrival='07:14:07' departure='07:15:07'/>
    </ocpTT>
    …
    <ocpTT ocpRef='ocp_DBW' trackInfo='6' ocpType='end'>
      <times scope='scheduled' arrival='07:43:31'/>
    </ocpTT>
  </ocpsTT>
</trainPart>
```

ocp_DH = Dresden Hbf
ocp_DN = Dresden-Neustadt
ocp_DBW = Bischofswerda

7
```
<trainPart id='tp_20201' trainNumber='20201' categoryRef='cat_OBB'>
  <formationTT formationRef='fmt_0'/>
  <operatingPeriodRef ref='opp_0'/>
  <ocpsTT>
    <ocpTT ocpRef='ocp_DBW' trackInfo='6' ocpType='begin'>
      <times scope='scheduled' departure='07:48:18'/>
    </ocpTT>
    …
    <ocpTT ocpRef='ocp_DBZ' ocpType='stop'>
      <times scope='scheduled' arrival='08:03:23' departure='08:03:53'/>
    </ocpTT>
    …
    <ocpTT ocpRef='ocp_DL' ocpType='stop'>
      <times scope='scheduled' arrival='08:22:15' departure='08:22:45'/>
    </ocpTT>
    …
    <ocpTT ocpRef='ocp_DG' ocpType='end'>
      <times scope='scheduled' arrival='08:42:30'/>
    </ocpTT>
  </ocpsTT>
</trainPart>
```

ocp_DBW = Bischofswerda
ocp_DBZ = Bautzen
ocp_DL = Löbau
ocp_DG = Görlitz

8
```
<trainPart id='tp_95001_DBW-DZ' trainNumber='95001' categoryRef='cat_OBE'>
  <formationTT formationRef='fmt_0'/>
  <operatingPeriodRef ref='opp_1'/>
  <ocpsTT>
    <ocpTT ocpRef='ocp_DBW' trackInfo='6' ocpType='begin'>
      <times scope='scheduled' departure='07:45:31'/>
    </ocpTT>
    …
    <ocpTT ocpRef='ocp_DEB' trackInfo='1' ocpType='stop'>
      <times scope='scheduled' arrival='08:14:47' departure='08:15:17'/>
    </ocpTT>
    …
    <ocpTT ocpRef='ocp_DZ' ocpType='end'>
      <times scope='scheduled' arrival='08:40:40'/>
    </ocpTT>
  </ocpsTT>
</trainPart>
```

ocp_DBW = Bischofswerda
ocp_DEB = Ebersbach
ocp_DZ = Zittau

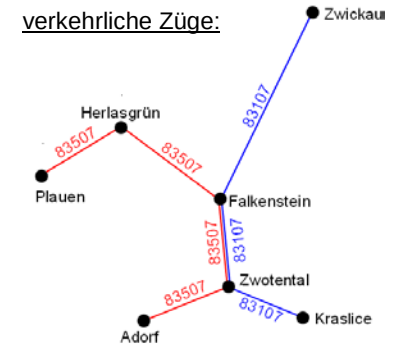
Aus den Elementen trainPartSequence der Züge ist die Reihenfolge der Laufwegabschnitte („serielle“ Verkettung der Zugteile) erkennbar. Das Attribut sequence enthält die laufende Nummer des Laufwegabschnitts. Sofern in einem Laufwegabschnitt mehr als ein Zugteil vorkommt (hier: tro_95001 *sequence* 1), enthalten die Attribute position die Reihenfolge der Zugteile im Zugverband. (Zu beachten ist jedoch, dass position nicht unbedingt die tatsächliche Stellung im Zug angibt - durch unterschiedliche Verkehrstage der Zugteile können Teile mit niedrigerem position-Wert an bestimmten Tagen im Zugverband „fehlen“.)

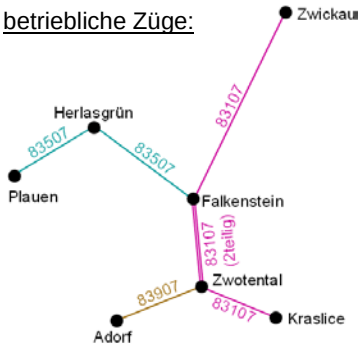
Beispiel 2

Im folgenden Beispiel wird der Unterschied zwischen betrieblicher und verkehrlicher Zugnummer besonders deutlich.

verkehrlicher Zug **83107** verkehrlicher Zug **83507**

km	Zug	VL 83107	VL 83507	
	von	Zwickau Zentrum		
0,0	**Zwickau (Sachsen) Hbf.**	7.16	...	betrieblicher Zug **83107**
13,9	Voigtsgrün	|	...	
22,7	Lengenfeld (Vogtland)	7.38	...	
28,3	Rodewisch	7.44	...	
30,4	Auerbach (Vogtl) unt. Hp.	7.47	...	
35,3	**Falkenstein (Vogtland)** o	7.52	...	
-1,3	**Plauen (Vogtland) ob. Bf.**	|	7.18	betrieblicher Zug **83507**
13,3	Herlasgrün	|	7.32	
24,9	Eich (Sachsen)	|	x 7.46	
30,6	Auerbach (Vogtl.) ob. Hp.	|	x 7.52	
31,9	Auerbach (Vogtland)	|	x 7.53	
35,3	**Falkenstein (Vogtland)** o	|	7.58	
35,3	**Falkenstein (Vogtland)**	8.02	8.02	*vereinigter Abschnitt:* betrieblicher Zug **83107** (2teilig)
42,1	Grünbach	x 8.09	x 8.09	
45,6	Muldenberg	x 8.12	x 8.12	
47,1	Muldenberg Floßplatz	x 8.14	x 8.14	
52,7	Schöneck (Vogtland)	8.21	8.21	
53,6	Schöneck (V.) Ferienpark	x 8.22	x 8.22	
58,0	**Zwotental** o	8.27	8.27	
58,0	**Zwotental**	8.32	|	betrieblicher Zug **83107**
61,5	Zwota-Zechenbach	x 8.37	|	
63,8	Zwota	x 8.40	|	
66,3	Klingenthal	8.43	|	
69,5	Kraslice-Pod vlekem	x 8.47	|	
70,7	**Kraslice** o	8.49	|	
58,0	**Zwotental**	...	8.34	betrieblicher Zug **83907**
61,0	Gunzen	...	x 8.37	
67,2	Siebenbrunn	...	x 8.47	
70,9	**Adorf (Vogtland)** o	...	8.52	
	nach			






```
<train id='tro_83107' type='operational' trainNumber='83107' scope='primary'>
  <trainPartSequence sequence='1'>
    <trainPartRef ref='tp_83107_DZWM-DFA' position='1'/>
  </trainPartSequence>
  <trainPartSequence sequence='2'>
    <trainPartRef ref='tp_83107_DFA-DZA' position='1'/>
    <trainPartRef ref='tp_83507_DFA-DZA' position='2'/>
  </trainPartSequence>
  <trainPartSequence sequence='3'>
    <trainPartRef ref='tp_83107_DZA-XTKE' position='1'/>
  </trainPartSequence>
</train>
```

```
<train id='trc_83107.1' name='83107' type='commercial'>
  <trainPartSequence sequence='1'>
    <trainPartRef ref='tp_83107_DLFD-DFA' position='1'/>
  </trainPartSequence>
  <trainPartSequence sequence='2'>
    <trainPartRef ref='tp_83107_DFA-DZA' position='1'/>
  </trainPartSequence>
  <trainPartSequence sequence='3'>
    <trainPartRef ref='tp_83107_DZA-XTKE' position='1'/>
  </trainPartSequence>
</train>
```

```
<train id='tro_83507' type='operational' trainNumber='83507' scope='primary'>
  <trainPartSequence sequence='1'>
    <trainPartRef ref='tp_83507' position='1'/>
  </trainPartSequence>
</train>
```

```
<train id='tro_83907' type='operational' trainNumber='83907' scope='primary'>
  <trainPartSequence sequence='1'>
    <trainPartRef ref='tp_83507.1' position='1'/>
    <trainPartRef ref='tp_83507.2' position='2'/>
  </trainPartSequence>
</train>
```

Allgemeines zu Datumsbezügen, Gültigkeitsperiode und Feiertagen

Im Falle des Datenaustauschs von konkreten Fahrplänen verfügen die in einer RailML-Datei enthaltenen Fahrplandaten i. d. R. auch über einen Datumsbezug. Der Datumsbezug wird im Element **timetablePeriod** und optional ergänzend in den Elementen **operatingPeriod** hergestellt.

Typischer Weise enthält ein Element timetablePeriod

- Beginn und Ende der Gültigkeitsperiode in den Attributen startDate und endDate,
- eine optionale Aufzählung von Feiertagen innerhalb der Gültigkeitsperiode.

```
<timetablePeriod id='ttp_2020_21' name='2020/21' startDate='2020-12-13' endDate='2021-12-11'>
  <holidays>
    <holiday holidayDate='2020-12-25'/>
    <holiday holidayDate='2020-12-26'/>
    <holiday holidayDate='2021-01-01'/>
    <holiday holidayDate='2021-04-02'/>
    <holiday holidayDate='2021-04-04'/>
    <holiday holidayDate='2021-04-05'/>
    <holiday holidayDate='2021-05-01'/>
    <holiday holidayDate='2021-05-13'/>
    <holiday holidayDate='2021-05-23'/>
    <holiday holidayDate='2021-05-24'/>
    <holiday holidayDate='2021-10-03'/>
    <holiday holidayDate='2021-10-31'/>
    <holiday holidayDate='2021-11-17'/>
  </holidays>
</timetablePeriod>
```

Im einfachsten Fall enthält nur das eine vorhandene Element timetablePeriod einen Datumsbezug über startDate+endDate und ggf. noch über holidays. Die Elemente operatingPeriod können ohne weitere Angabe eines konkreten Datums beliebige Wochentage aus dem Gültigkeitsbereich referenzieren (s. nachfolgende Beispiele).

Die Angabe der Feiertage ist die Bezugsbasis für die Unterelemente operatingDayDeviance der Elemente opertingPeriod. Die Elemente holidays und operatingDayDeviance sind daher zunächst nur gemeinsam sinnvoll anzuwenden. Wenn keine Elemente operatingDayDeviance angegeben sind, würden holidays nicht ausgewertet. Es ist aber dennoch denkbar, die Feiertage nur der Vollständigkeit halber anzugeben.

Eine RailML-Datei kann, muss aber nicht über eine Gültigkeitsperiode verfügen. Eine RailML-Datei kann auch mehrere Gültigkeitsperioden beinhalten, wobei sich jede operatingPeriod und damit auch jeder Zug und Zugteil immer nur auf eine Gültigkeitsperiode gleichzeitig beziehen kann.

RailML-Dateien ohne Gültigkeitsperiode haben entweder kein Element timetablePeriod oder nur Elemente timetablePeriod ohne startDate+endDate. Wenn keine Gültigkeitsperiode definiert ist, dürfen die Elemente operatingPeriod ebenfalls nicht über die Attribute startDate, endDate und bitMask und auch nicht über Unterelemente specialService verfügen.

StartDate und endDate dürfen nur gemeinsam angegeben werden, nicht nur eines der beiden, d. h. es darf keine „offenen“ Perioden geben. Die Differenz der Attribute startDate und endDate definiert die Länge der Bitmasken (Attribut bitMask) der Elemente operatingPeriod.

Abbildung spezieller Verkehrstageregelungen ohne Datumsbezug

Verkehrstageregelungen (Saisonierungen) werden im RailML-Element *timetable.operatingPeriods* codiert. In einer Verkehrstageregelung (*operatingPeriod*) sind die einzelnen Tage auf zwei redundante Arten codiert: Einmal im Attribut *bitMask* über die gesamte Periode sowie in den Unterelementen *operatingDay* und *specialService* aufgeschlüsselt nach konkreter Eingabe in Regel- und Sonderverkehrstage. Der Grund für diese Dopplung ist, dass sich die Bitmaske zwar technisch relativ gut weiterverarbeiten, jedoch nicht eindeutig in eine Nutzereingabe rückverwandeln lässt. (D. h. es gibt mehrere Eingaben, die zur gleichen Bitmaske führen.) Für einige Anwendungen ist es wichtig, die Angaben genau so zu übermitteln, wie der Anwender sie eingegeben hat. Im Übrigen kann es relativ aufwändig sein, aus einer Bitmaske eine Trennung in Regel- und Sonderverkehrstage vorzunehmen (es kann z. B. eine Korrelationsanalyse erfordern, die Eingabe „W[Sa], nicht 24., 31.12.; auch 17.11.“ aus einer Bitmaske wiederherzustellen).

Der Beschreibung mit *operatingDay*-Elementen kommt besondere Bedeutung zu, wenn z. B. bei „strategischen“ (langfristigen) Fahrplänen keine Gültigkeitsperiode definiert ist (keine *timetablePeriod* bzw. ohne *startDate*+*endDate*; s. a. vorheriges Kapitel).

Das Element *operatingDay* enthält ein Unterelement **operatingDayDeviance**, dessen Bedeutung an den folgenden Beispielen erläutert werden soll:

Im Allgemeinen gibt das Element *operatingDayDeviance* an, das die Verkehrstageregelung abweichend von *operatingCode* verkehrt oder nicht verkehrt, wenn ein Wochentag in der unter *holidayOffset* angegebenen Relation zu einem Feiertag steht. Das Attribut *ranking* definiert, welches *operatingDayDeviance*-Element gilt, falls es mehrere gleichzeitig zutreffende gibt. Beispielsweise könnte ein Wochentag gleichzeitig *Feiertag* und *ein Tag vor einem Feiertag* sein (in Deutschland z. B. der 25.12.).

Beispiel 1
Die Verkehrstageregelung W[Sa] = Mo-Fr[S] = „Montag bis Freitag, jedoch nicht an Feiertagen“ wird wie folgt abgebildet:

```
<operatingPeriod id='…' name='W[Sa]' …>
  <operatingDay operatingCode='1111100'>
    <operatingDayDeviance operatingCode='0000000' holidayOffset='0'/>
  </operatingDay>
</operatingPeriod>
```

Die Angabe `operatingCode='1111100'` bedeutet, dass die Regelung an den ersten fünf Wochentagen (=Montag bis Freitag) verkehrt. Die Angabe `operatingCode='0000000' holidayOffset='0'` bedeutet, dass die Regelung abweichend davon *nicht* verkehrt (=0000000), wenn der jeweilige Tag auf einen Feiertag fällt (holidayOffset=0 - „null Tage Versatz zu einem Feiertag“ hat).

Beispiel 2
Die Verkehrstageregelung S = „Sonntag und alle Wochenfeiertage“ wird wie folgt abgebildet:

```
<operatingPeriod id='…' name='S' …>
  <operatingDay operatingCode='0000001'>
    <operatingDayDeviance operatingCode='1111111' holidayOffset='0'/>
  </operatingDay>
</operatingPeriod>
```

Die Angabe `operatingCode='0000001'` bedeutet, dass die Regelung nur an Sonntagen verkehrt. Die Angabe `operatingCode='1111111' holidayOffset='0'` bedeutet, dass die Regelung abweichend davon an allen Tagen verkehrt, die auf einen Feiertag fallen.

Beispiel 3

Die Verkehrstageregelung vS = „Werktage vor Sonn- und Feiertagen“ wird wie folgt abgebildet:

```
<operatingPeriod id='…' name='vS' …>
  <operatingDay operatingCode='0000010'>
    <operatingDayDeviance operatingCode='1111110' holidayOffset='-1' ranking='2'/>
    <operatingDayDeviance operatingCode='0000000' holidayOffset='0' ranking='1'/>
  </operatingDay>
</operatingPeriod>
```

Die Angabe `operatingCode='0000010'` bedeutet, dass die Regelung nur an Sonnabenden verkehrt (=vor Sonntagen). Die Angabe `operatingCode='1111110' holidayOffset='-1'` bedeutet, dass die Regelung auch an Montagen bis Sonnabenden verkehrt, sofern sie einen Tag vor einem Feiertagen liegen (holidayOffset=-1). Die Angabe `operatingCode='0000000' holidayOffset='0'` bedeutet, dass die Regelung nicht an Feiertagen verkehrt. Falls ein Tag gleichzeitig Feiertag und ein Tag vor einem Feiertag ist (in Deutschland z. B. 25.12.), verkehrt die Regelung nicht, denn es gilt dann die holidayOffset=0-Angabe. Entsprechend RailML-Dokumentation (http://wiki.railml.org/index.php?title=TT:operatingDayDeviance) gilt im Zweifelsfall die Angabe mit niedrigerem *ranking*-Wert.

Insgesamt sind die drei Zeilen (von oben nach unten) in etwa wie folgt zu interpretieren:

Zeile 1:	„verkehrt an Samstagen,...
Zeile 2:	... sowie an Montagen bis Samstagen wenn vor einem Feiertag,...
Zeile 3:	...jedoch nicht an Feiertagen“

Beispiel 4

Ein Zug beginnt seine Fahrt an Sa+S = „Samstagen sowie Sonn- und Feiertagen“ und verkehrt im Laufe der Fahrt über Mitternacht. Die Rückfahrt nach Mitternacht muss zwangsläufig an „Folgetagen von Sa+S“ stattfinden. In RailML würde Sa+S wie folgt abgebildet:

```
<operatingPeriod id='…' name='Sa+S' …>
  <operatingDay operatingCode='0000011'>
    <operatingDayDeviance operatingCode='1111111' holidayOffset='0'/>
  </operatingDay>
</operatingPeriod>
```

Die Folgetage von Sa+S sind demzufolge (alle Bitmasken um ein Bit nach rechts rotiert):

```
<operatingPeriod id='…' name='Folgetage von Sa+S' …>
  <operatingDay operatingCode='1000001'>
    <operatingDayDeviance operatingCode='1111111' holidayOffset='+1'/>
  </operatingDay>
</operatingPeriod>
```

Für diese Regelung gibt es im deutschsprachigen Raum keine standardisierte Kurzbezeichnung. Sie ist nicht identisch mit So+nS, denn letztere würde abgebildet durch:

```
<operatingPeriod id='…' name='So+nS' …>
  <operatingDay operatingCode='1000001'>
    <operatingDayDeviance operatingCode='1111110' holidayOffset='+1' ranking='2'/>
    <operatingDayDeviance operatingCode='0000000' holidayOffset='0' ranking='1'/>
  </operatingDay>
</operatingPeriod>
```

Mit RailML lassen sich auch Folge- und Vortage „sauber“ abbilden, was allein mit deutschen Kurzbezeichnungen so nicht möglich ist. Solange eine Fahrplanperiode definiert ist, ließen sich solche Folge- und Vortagesregelungen durch Verwendung von Sonderverkehrstagen (*specialService* in RailML) umschreiben. Wenn jedoch keine Fahrplanperiode zur Verfügung steht, ist die Lösung mit *operatingDayDeviance* die einzig mögliche. Dies betrifft insbesondere die bei Wettbewerbsverfahren (Ausschreibungsfahrplänen ohne konkrete Fahrplanperiode) im Vor- und Nachlauf um Wochenenden recht häufig vorkommenden Regelungen „Arbeitstage vor Ⓐ v Ⓒ (Arbeitstage vor arbeitsfreien Tagen), Ⓐ n Ⓒ (Arbeitstage nach arbeitsfreien Tagen), Ⓒ v Ⓐ (arbeitsfreie Tage vor Arbeitstagen) und Ⓒ n Ⓐ (arbeitsfreie Tage nach Arbeitstagen).

Abbildung spezieller Verkehrstageregelungen mit Datumsbezug

Hierbei geht es um solche Fälle wie „verkehrt nicht am 24. und 31.12." oder „nur vom 01.07.-31.08.". Voraussetzung hierfür ist, dass auch eine zugehörige Gültigkeitsperiode (*timetablePeriod* mit angegebenem *startDate+endDate*) definiert ist.

Für diese Fälle können folgende zusätzlichen Elemente und Attribute verwendet werden:
- Attribute *startDate+endDate* des Elements *operatingDay,*
- Unterelemente *specialService* des Elements operatingPeriod.

Die Angabe der Attribute **startDate** und **endDate** in *operatingDay* beschränkt die Anwendung der Bildungsregel des operatingDay-Elements auf diesen Datumsbereich. Beide sind nur gemeinsam zu benutzen und dürfen nur Tage innerhalb der Gültigkeitsperiode der *timetablePeriod* referenzieren.

Bei **specialService** handelt es sich um eine Aufzählung von **Ausnahmeverkehrstagen**, an denen von den durch die *operatingDay*-Elemente definierten Bildungsregeln abgewichen wird. Dabei steht das Attribut type mit Ausprägung
=*include* für Zusatzverkehrstage,
=*exclude* für Ausfalltage.
Die Attribute *startDate* und *endDate* definieren den ersten und letzten Gültigkeitstag der Ausnahme, sind nur gemeinsam zu benutzen und dürfen nur Tage innerhalb der Gültigkeitsperiode der *timetablePeriod* referenzieren. Das Attribut *singleDate* ist im Fall *startDate=endDate* anstatt dieser beiden zu verwenden.

Beispiel 1: verkehrt nur vom 14.12. bis 20.12.

```
<operatingPeriod id='…' name='nur 14.12.-28.12.' timetablePeriodRef='ttp_2020_21'
        bitMask='0111111111111111100000…000'>
    <operatingDay operatingCode='1111111' startDate='2020-12-14' endDate='2020-12-28'/>
</operatingPeriod>
```

Beispiel 2: verkehrt nicht am 25.12. und nicht am 01.01.

```
<operatingPeriods>
  <operatingPeriod id='…' name='täglich; nicht 25.12.; 1.1.'
       timetablePeriodRef='ttp_2020_21'
       bitMask='11111111111101111110111…111'>
    <operatingDay operatingCode='1111111' startDate='2020-12-13' endDate='2021-12-11'/>
    <specialService type='exclude' singleDate='2020-12-25'/>
    <specialService type='exclude' singleDate='2021-01-01'/>
  </operatingPeriod>
</operatingPeriods>
```

Beispiel 3: verkehrt vom 13.12. bis 31.01. nur Sa und vom 01.07.-31.08. täglich; auch am 25.12. und 01.01, nicht am 15.08."

(Verzicht auf Darstellung von name, bitMask usw.)

```
<operatingPeriod id='…' timetablePeriodRef='ttp_2020_21'>
  <operatingDay operatingCode='0000010' startDate='2020-12-13' endDate='2021-01-31'/>
  <operatingDay operatingCode='1111111' startDate='2020-07-01' endDate='2021-08-31'/>
  <specialService type='include' singleDate='2020-12-25'/>
  <specialService type='include' singleDate='2021-01-01'/>
  <specialService type='exclude' singleDate='2021-08-15'/>
</operatingPeriod>
```

Falls mehrere *operatingDay*-Elemente in einem *operatingPeriod*-Element vorkommen, müssen diese durch ihre *startDate-endDate*-Bereiche oder ihre *operatingCode*-Bitmaske immer disjunkt sein.

Mitternachtsübergänge

Es gibt mehrere Möglichkeiten, Mitternachtsübergänge in RailML korrekt abzubilden:

1. Mit Hilfe der Attribute **arrivalDay** und **departureDay**.
2. Durch Aufteilen des Zuglaufs in Zugteile vor und nach Mitternacht sowie Wechsel der *operatingPeriod*-Referenz zwischen diesen Zugteilen
   a) und Verwenden des Attributs **dayOffset** der nach Mitternacht gültigen *operatingPeriod*[4],
   b) und durch um einen Tag verschobene Angabe der Verkehrstage der nach Mitternacht gültigen *operatingPeriod*.

Möglichkeit 1 ist die regulär vorgesehene für Mitternachtsübergänge innerhalb eines in RailML abgebildeten Zuglaufs. Es ist ausdrücklich nicht vorgesehen, einen Zuglauf *nur wegen des Mitternachtsübergangs* in Zugteile aufzuteilen. Möglichkeit 2a kommt dennoch in besonderen des Mitternachtsübergangs vor einem (hier betrachteten) Zuglauf in Betracht. Möglichkeit 2b soll vermieden werden.

Mitternachtsübergänge innerhalb des abgebildeten Zuglaufs

Für Mitternachtsübergänge *innerhalb eines abgebildeten Zuglaufs* sind in RailML die Attribute **arrivalDay** und **departureDay** vorgesehen. Sie stellen eine Zählung der Anzahl Mitternachtsübergänge relativ zu einem Bezugspunkt (z. B. Abfahrtsort; s. u.) dar. Diese Attribute sind optional mit Default-Wert 0, d. h. sie müssen solange nicht angegeben werden, solange ein Zug noch nicht über Mitternacht gefahren ist. Nach dem ersten Mitternachtsübergang, d. h. sobald ihr Wert >0 ist, sind sie jedoch anzugeben:

Beispiel 1: Mitternachtsübergang während eines Zwischenaufenthalts

hier Mitternachtsübergang

```
<ocpTT ocpRef='ocp_DOLB' ocpType='stop'>
  <times scope='scheduled' arrival='23:59:49' departure='00:00:19' departureDay='1'/>
  <sectionTT section='DOLB-DN_E' lineRef='ln_80_6212' trackRef='tr_80_6212_2' trackInfo='1'>
    <runTimes minimalTime='PT48S' operationalReserve='PT1S'/>
  </sectionTT>
  <stopDescription commercial='true' stopOnRequest='false'>
    <stopTimes minimalTime='PT30S'/>
  </stopDescription>
</ocpTT>
```

Alle weiteren Ankunfts-, Abfahrts- und Durchfahrtszeiten des Zuges bis zu seinem Laufwegende sind mit *arrivalDay=1* und *departureDay=1* gekennzeichnet.

Beispiel 2: Mitternachtsübergang während der Fahrt

hier Mitternachtsübergang

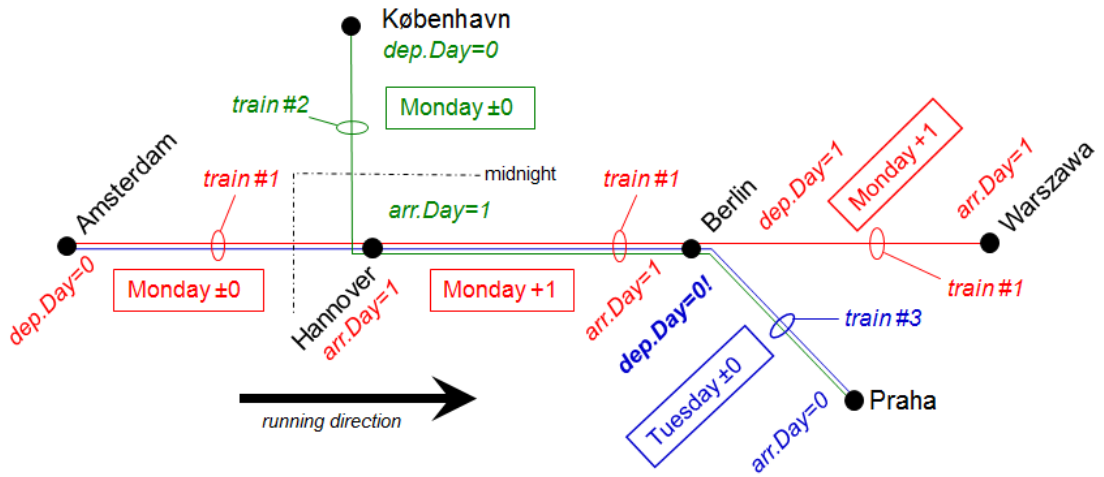
```
<ocpTT ocpRef='ocp_DNKW' ocpType='pass'>
  <times scope='scheduled' departure='23:55:00'/>
</ocpTT>
<ocpTT ocpRef='ocp_DNKW_A' ocpType='pass'>
  <times scope='scheduled' departure='23:55:35'/>
</ocpTT>
<ocpTT ocpRef='ocp_DNKO' ocpType='stop'>
  <times scope='scheduled' arrival='23:57:53' departure='23:58:23'/>
</ocpTT>
<ocpTT ocpRef='ocp_DWT_N' ocpType='pass'>
  <times scope='scheduled' departure='00:01:25' departureDay='1'/>
</ocpTT>
<ocpTT ocpRef='ocp_DWT' ocpType='stop'>
  <times scope='scheduled' arrival='00:02:17' arrivalDay='1' departure='00:03:00'
                                                              departureDay='1'/>
</ocpTT>
```

Alle weiteren Ankunfts-, Abfahrts- und Durchfahrtszeiten des Zuges bis zu seinem Laufwegende sind mit *arrivalDay=1* und *departureDay=1* gekennzeichnet.

[4] Das Attribut *dayOffset* wird mit RailML 2.2 eingeführt.

Referenzort der Tages-/Mitternachtszählung

- Es ist vorgesehen, dass die Tages- bzw. Mitternachtszählung (arrival/departureDay) **an der ersten Abfahrt eines Zuges mit 0 beginnt**.
- Demzufolge kann der Wert -1 vorkommen bei der ersten Ankunft im seltenen Fall, dass der Zug am ersten Bahnhof über Mitternacht steht und zuvor „von außerhalb“ kommt.
- Wenn ein Zug aus mehreren (zu aufeinanderfolgenden Abschnitten verketteten) Zugteilen besteht, bezieht sich die Tageszählung i. d. R. auf den ersten Abfahrtsort **des Gesamtzuglaufs**. Bei einzelnen Zugteilen kann es also vorkommen, dass diese bereits mit Tageszählung >0 beginnen.
- Wenn Zugläufe (hier in RailML: *commercialTrains*) von einem Zug auf einen anderen übergehen, kann es zu einem „Rückschritt“ der Tageszählung kommen. (Dies bedeutet, dass der Zuglauf von einem Zug, der bereits über Mitternacht gefahren ist, übergeht auf einen Zug, der noch nicht über Mitternacht gefahren ist.) Auch dies ist kein Fehler.



In diesem Bild gibt es drei Züge (rot, grün, blau), die wegen der RailML-internen Syntax in mind. neun Zugteile unterteilt sein müssen (an den schwarz dargestellten Stationen). Im Zuglauf der Zugteile grün und blau kommt es zum „Rücksprung“ des Tagesindex‘ in Berlin (arrivalDay=1 auf departureDay=0). Der Rücksprung geht i. d. R.* mit einem scheinbaren Verkehrstagewechsel einher (operatingPeriodRef wechselt: Montag auf Dienstag). Beide Effekte zusammen führen zu ‘Monday +1’ auf ‘Tuesday +0’.

* Nur „in der Regel“, denn wenn die Züge täglich fahren würden, gäbe es keinen scheinbaren Verkehrstagewechsel.

Warum gibt es keinen Rücksprung auf dem Warschauer Abschnitt? In diesem Beispiel wurde unterstellt, dass es einen durchgehenden (betrieblichen) Zug Amsterdam – Warschau gibt. Da dieser in Amsterdam vor Mitternacht beginnt, beziehen sich seine arrival/departureDay-Angaben auf die Amsterdamer Abfahrt, und er kommt in Warschau mit arrivalDay=1 an, aber ohne Rücksprung. Im Gegensatz dazu ist für den Prager Zweig unterstellt, dass in Berlin ein neuer (betrieblicher) Zug beginnt. Dessen Abfahrt mit departureDay=0 liegt bereits nach Mitternacht, daher der Rücksprung.

Bedeutung der Tages-/Mitternachtszählung

- Die Verkehrstage-Angabe (*operatingPeriod*) ist allein nicht interpretierbar, sondern immer nur im Zusammenhang mit dem jeweiligen Tagesindex (*arrival/departureDay*). Um festzustellen, wann ein Zug tatsächlich fährt, muss man z. B. die Bitmaske um die Anzahl Stellen aus (arrival/departureDay + dayIndex) verschieben.

- Ob ein effektiver Wechsel der Verkehrstage innerhalb des Zuglaufs stattfindet, kann nur durch Vergleichen der verschobenen Bitmasken herausgefunden werden. Ein Wechsel der *operatingPeriod* allein ist nicht zwingend ein effektiver Verkehrstagewechsel.
- Es kann beliebig viele Kombinationen aus Verkehrstage-Angabe (*operatingPeriod*) und Tagesindex (*arrival/departureDay*) geben, die inhaltlich das gleiche bedeuten.
- Das schreibende Programm ist frei in der Wahl der Kombination, diese kann auch „unterwegs“ wechseln.

Mitternachtsübergänge außerhalb des abgebildeten Zuglaufs

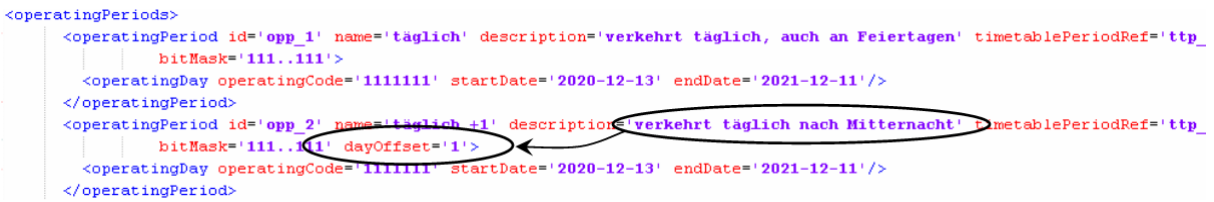
I. d. R. ist es beabsichtigt – bei einigen Anwendungen auch von außen her erforderlich – dass ein Zug mit Tageszählung =0 beginnt, d. h. dass sich die Verkehrstage, mit denen er identifiziert wird, auf den ersten *abgebildeten* Abfahrtsbahnhof beziehen.

Sollte der Zug bereits vor dem ersten *abgebildeten* Abfahrtsbahnhof – quasi „im Ausland“ – über Mitternacht gefahren sein, besteht damit nicht mehr die Möglichkeit, mit Tagesindex (*arrival/departureDay*) >0 am ersten abgebildeten Bahnhof zu beginnen.

Damit bliebe dann nur noch die Möglichkeit (eingangs dieses Kapitels mit 2b bezeichnet), die Verkehrstage um die Anzahl Tage versetzt anzugeben, wie oft der Zug zuvor über Mitternacht gefahren ist. Diese Lösung ist jedoch ausdrücklich nicht gewollt u. a. aus folgenden Gründen:

- Fahrplanperioden sind i. d. R. nicht geschlossen, d. h. auf den letzten Tag einer Fahrplanperiode folgt nicht wieder der erste Tag der gleichen Periode. Ein Zug, der in einer Fahrplanperiode *täglich* beginnt und über Mitternacht fährt, fährt danach genau genommen nicht mehr täglich (nämlich nicht am ersten Tag der Fahrplanperiode, dafür aber zusätzlich am ersten Tag der nächsten Fahrplanperiode). Die Bitmaske seiner Verkehrstage (*operatingPeriod.bitMask*) ist gegenüber vor Mitternacht um ein Bit in Richtung aufsteigenden Datums verschoben, d. h. selbst die zuvor nur aus Einsen bestandene Täglich-Bitmaske beginnt danach mit einer Null.
- Das um Tage verschobene Angeben der Verkehrstageregelungen (*operatingPeriods*) ist ausdrücklich zu vermeiden, da wie bereits im Kapitel *Abbildung spezieller Verkehrstageregelungen ohne Datumsbezug* erwähnt, für viele praktisch vorkommende Verkehrstageregelungen keine exakte Folgetagsentsprechung existiert (z. B. nicht für die in Deutschland recht verbreiteten „werktags außer Samstags“ und „Sonn- und Feiertags“).

Aus diesen Gründen wird mit RailML 2.2 neu das Attribut **dayOffset** im Element *operatingPeriod* eingeführt:



Die Bitmaske der mit *dayOffset≠0* anzugebenden operatingPeriods ist nicht verschoben darzustellen, d. h. die Bitmaske ist identisch wie beim Fall dayOffset=0.

Grundsätzlich besteht die an sich gleichwertige Möglichkeit, in den betreffenden Fällen anstatt *dayOffset>0* weiterhin mit *arrival/departureDay>0* auch am ersten Bahnhof zu beginnen. Ein lesendes Programm sollte beide Varianten verarbeiten können. Die Variante mit *dayOffset>0* ist vorgesehen für Fälle, in denen *departureDay=0* am ersten Bahnhof aus externen Gründen erzwungen werden muss. Letzteres ist allerdings auch die Empfehlung des RailML-Konsortiums.

Primärschlüssel von Zügen, Mehrfachzugläufe, Ergänzungsfahrpläne

Aus verschiedenen Gründen ist in der Eisenbahnwelt ein Mehrfachverwenden von Zugnummern üblich. In einigen Fällen ist es z. B. organisatorisch nicht gewünscht, z. B. bei geringen Differenzen zweier Lagen eines Zuges zwei unterschiedliche Zugnummern zu verwenden.

Gleichzeitig müssen mit zunehmender Verbreitung und konsequenterem Einsatz von Rechentechnik früher intuitiv gehandhabte Zusammenhänge heute „informatorisch sauber“ abgebildet werden – Züge benötigen einen rechentechnisch eindeutigen **Primärschlüssel**.

Innerhalb einer RailML-Datei enthält das Attribut id einen eindeutigen Primärschlüssel, welcher aber ausdrücklich nicht zur allgemeinen Verwendung außerhalb der RailML-Datei vorgesehen ist. Die Zugnummer ist jedoch aus o. g. Gründen ebensowenig als Primärschlüssel geeignet. Der externe Primärschlüssel muss daher i. d. R. aus mehreren Feldern zusammengefasst werden. Um dies zu erleichtern, unterstützt RailML die Attribute scope und additionalTrainNumber. Das Wertetripel aus trainNumber, scope und additionalTrainNumber muss immer eindeutig sein, d. h. ist grundsätzlich als externer Primärschlüssel geeignet.

Durch das Attribut **scope** werden mehrere Züge mit gleicher Zugnummer (die disjunkte Verkehrstage haben müssen) unterschieden. Man kann diese auch als alternative zeitliche Lagen oder „Varianten“ ein und desselben Zuges auffassen. Das Attribut **additionalTrainNumber** dient der Unterscheidung von Einträgen mit gleicher Zugnummer und gleichem scope-Attribut. In Anwendungsfällen, in denen die Zugnummer selbst immer eindeutig ist, haben scope und additionalTrainNumber keine Bedeutung; hier ist immer scope=*primary*.

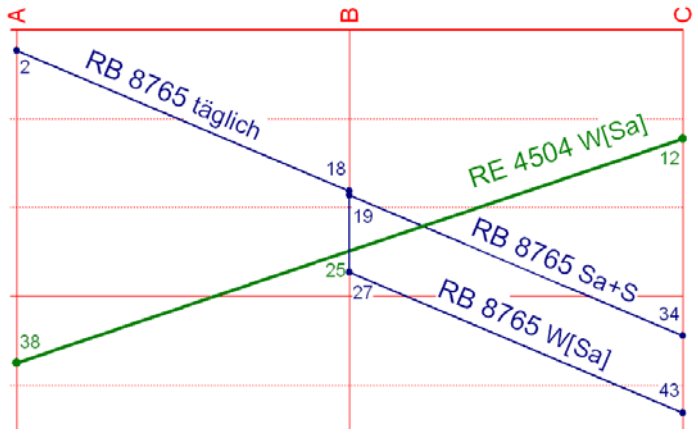
Den terminologischen Zusammenhang zwischen RailML-Bezeichnungen, FBS-Bezeichnungen und Bezeichnungen der DB Netz AG gibt folgende Tabelle wieder:

in RailML	in FBS	bei DB Netz
train.scope	Mehrfachzuglauf	„Ergänzungsfahrplan“
train.scope=primary	Hauptlauf (mit Index=1)	„Stammfahrplan" oder „Stammzug“
	Hauptlauf (mit Index>1)	*- nicht möglich -*
train.scope=secondaryStart	Vornebenlauf	„Startflügel“
train.scope=secondaryEnd	Nachnebenlauf	„Zielflügel“
train.scope=secondaryInner	Zwischennebenlauf	„Doppelfahrplan“

Im Allgemeinen gilt, dass es an einem Verkehrstag jede Zugnummer nur einmal geben darf. Daher müssen die Verkehrstage von Mehrfachzugläufen disjunkt sein. Weiterhin gilt praktisch in den meisten Fällen die Einschränkung, dass ein Zug-Primärschlüssel (o. g. Wertetripel) an einem Verkehrstag und einer Betriebsstelle nur einmal vorkommen darf. Daher muss ein Zuglauf i. d. R. auf Mehrfachzugläufe aufgeteilt werden, sobald er an einer Betriebsstelle mehrfach vorbeikommt. Dies ist jedoch eine praktische Forderung z. B. aus sicherungstechnischen Gründen und derzeit keine RailML-Anforderung.

Anmerkung:
In den folgenden Beispielen sind vereinfachend nur zeitliche Abweichungen der Mehrfachzugläufe bei gleicher Strecke gezeigt. Ebenso sind jedoch auch räumliche Abweichungen (andere Laufwege / Strecken) möglich.

typisches Beispiel für „Nachnebenlauf“:



Abbildungsmöglichkeiten:

```
<train id='tro_8765P' type='operational' trainNumber='8765' scope='primary'>
<train id='tro_8765E' type='operational' trainNumber='8765' scope='secondaryEnd'>
```

RB 8765 **Hauptlauf**:
- verkehrt täglich im Abschnitt A-B
- verkehrt Sa+S im Abschnitt B-C

RB 8765 **Nachnebenlauf**:
- verkehrt nur im Abschnitt B-C und nur an W[Sa]

oder:

RB 8765 **Hauptlauf**:
- verkehrt täglich im Abschnitt A-B
- verkehrt W[Sa] im Abschnitt B-C

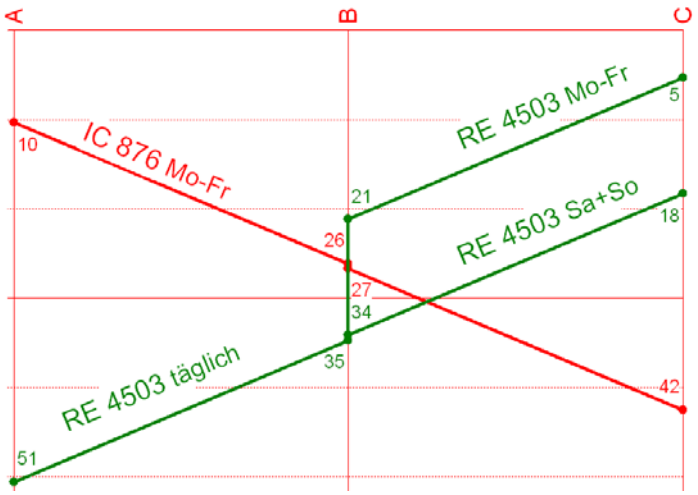
RB 8765 **Nachnebenlauf**:
- verkehrt nur im Abschnitt B-C und nur an Sa+S

oder:

```
<train id='tro_8765P1' type='operational' trainNumber='8765' scope='primary' additionalTrainNumber='1'>
<train id='tro_8765P2' type='operational' trainNumber='8765' scope='primary' additionalTrainNumber='2'>
```

RB 8765 **1. Hauptlauf** verkehrt W[Sa] A-B-C
RB 8765 **2. Hauptlauf** verkehrt Sa+S A-B-C
- beide Hauptläufe überlagern sich im Abschnitt A-B

typisches Beispiel für „Vornebenlauf“:



Abbildungsmöglichkeiten:

```
<train id='tro_4503P' type='operational' trainNumber='4503' scope='primary'>
<train id='tro_4503S' type='operational' trainNumber='4503' scope='secondaryStart'>
```

RE 4503 **Hauptlauf**:
- verkehrt Mo-Fr im Abschnitt C-B
- verkehrt täglich im Abschnitt B-A

RE 4503 **Vornebenlauf**:
- verkehrt nur im Abschnitt C-B und nur an Sa+So

oder:

RE 4503 **Hauptlauf**:
- verkehrt Sa+So im Abschnitt C-B
- verkehrt täglich im Abschnitt B-A

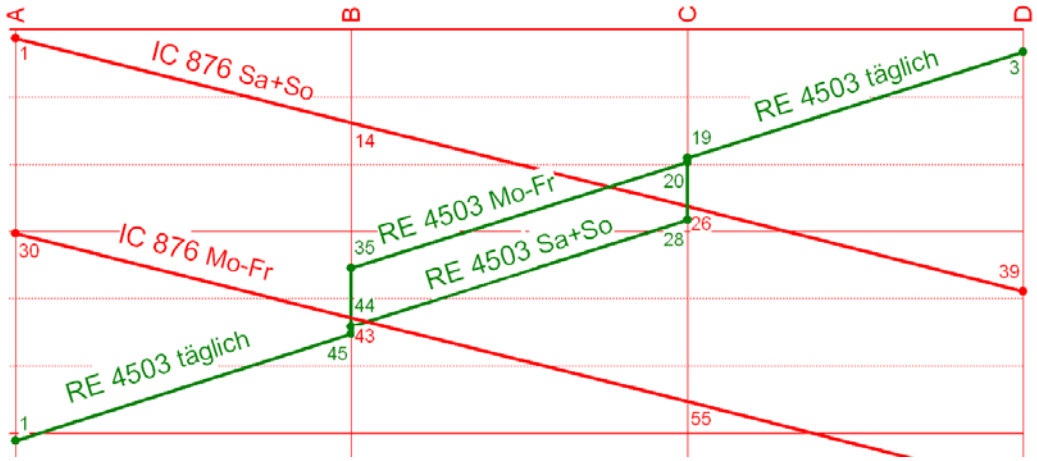
RE 4503 **Vornebenlauf**:
- verkehrt nur im Abschnitt C-B und nur an Mo-Fr

oder:

```
<train id='tro_4503P1' type='operational' trainNumber='4503' scope='primary' additionalTrainNumber='1'>
<train id='tro_4503P2' type='operational' trainNumber='4503' scope='primary' additionalTrainNumber='2'>
```

RE 4503 **1. Hauptlauf** verkehrt Mo-Fr C-B-A
RE 4503 **2. Hauptlauf** verkehrt Sa+So C-B-A
- beide Hauptläufe überlagern sich im Abschnitt B-A

typisches Beispiel für „Zwischennebenlauf“:



Abbildungsmöglichkeiten:

```
<train id='tro_4503P' type='operational' trainNumber='4503' scope='primary'>
<train id='tro_4503I' type='operational' trainNumber='4503' scope='secondaryInner'>
```

RE 4503 **Hauptlauf**:
- verkehrt täglich im Abschnitt D-C
- verkehrt Mo-Fr im Abschnitt C-B
- verkehrt täglich im Abschnitt B-A

RE 4503 **Zwischennebenlauf**:
- verkehrt nur im Abschnitt C-B und nur an Sa+So

oder:

wie oben, jedoch Hauptlauf und Zwischennebenlauf im Abschnitt C-B getauscht

oder:

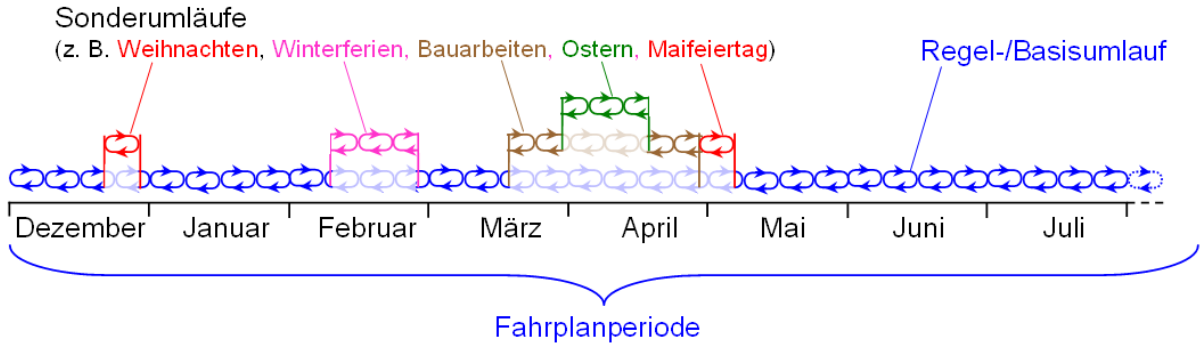
```
<train id='tro_4503P1' type='operational' trainNumber='4503' scope='primary' additionalTrainNumber='1'>
<train id='tro_4503P2' type='operational' trainNumber='4503' scope='primary' additionalTrainNumber='2'>
```

RE 4503 **1. Hauptlauf** verkehrt Mo-Fr D-A
RE 4503 **2. Hauptlauf** verkehrt Sa+So D-A
- beide Hauptläufe überlagern sich in den Abschnitten D-C und B-A

Umlaufpläne

In RailML sind **offene** und **geschlossene Umläufe** möglich. Geschlossene Umläufe können (theoretisch beliebig oft) wiederholt werden, da auf die letzte Fahrt im Umlauf wieder die erste Fahrt folgt. Offene Umläufe hingegen müssen nicht wiederholt werden können - bei ihnen folgt auf die letzte Fahrt entweder ein anderer Umlaufplan oder das Ende der Fahrplanperiode. Typischer Weise sind geschlossene Umläufe solche in einem Grund- oder Regelumlaufplan und offene Umläufe solche in einem Sonderumlaufplan. Im Allgemeinen kann man davon ausgehen, dass geschlossene Umläufe über einen längeren Zeitraum gelten (daher die Wiederholung, um nicht jeden Tag des längeren Zeitraums individuell abzubilden), während offene Umläufe über einen kürzeren Zeitraum gelten, in dem keine Wiederholungen möglich oder notwendig sind. Eine typische Zeitgrenze ist eine Woche, denn spätestens nach einer Woche sind Umlaufwiederholungen üblich. Offene Umläufe sind daher meist (fast immer) kürzer als eine Woche.

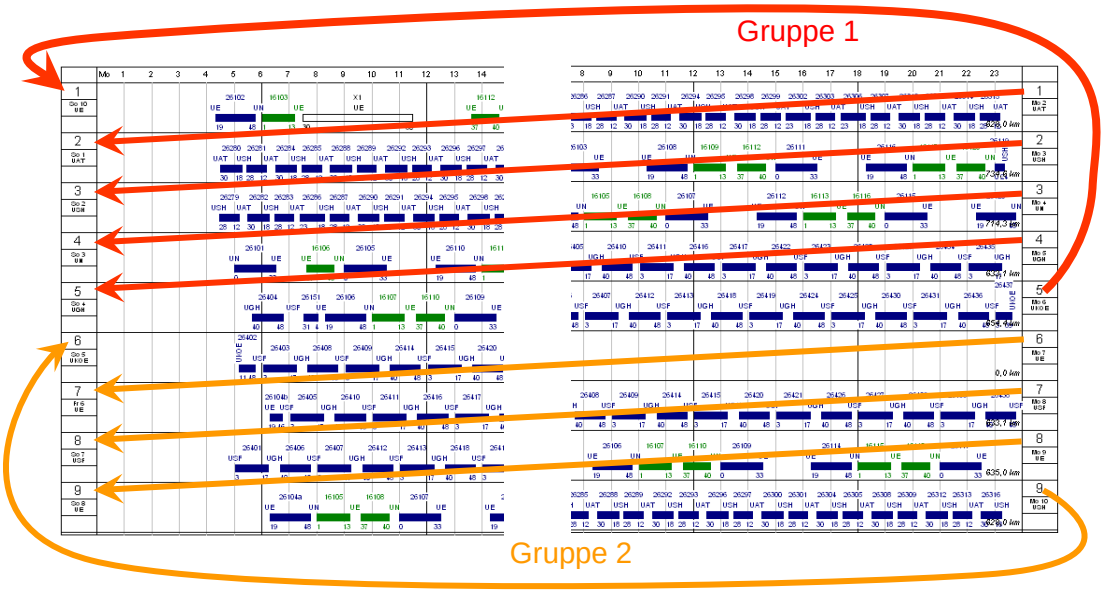
Wichtig für RailML ist, dass auf Grund der Möglichkeit offener Umläufe zu bestimmten Fahrten *keine* nachfolgende Fahrten angegeben sind. Zur begrifflichen Eindeutigkeit werden im Folgenden als *offene* Umlaufpläne solche bezeichnet, bei denen nicht an allen Fahrten eine Folgefahrt angegeben ist. *Geschlossene* Umläufe sind demzufolge solche, bei denen an allen Fahrten eine Folgefahrt angegeben ist - ungeachtet ihrer Gültigkeitsdauer und der Frage, ob sie tatsächlich wiederholt werden sollen.



Beispiel für offene und geschlossene Umläufe: Der durch blaue Zyklen symbolisierte Regel- oder Basisumlauf ist geschlossen, d. h. wiederholt sich theoretisch recht häufig. Er wird gelegentlich durch Sonderumläufe ersetzt/überschrieben, die teilweise geschlossen sind (mit Wiederholungen: hier Winterferien, Bauarbeiten, Ostern) und teilweise offen (ohne Wiederholung: hier Weihnachten, Maifeiertag).

Ein Umlaufplan kann mehrere **Umlaufgruppen** enthalten. Umlaufgruppen sind eine Eigenart insbesondere von Eisenbahn-Umlaufplänen: Jede Umlaufgruppe stellt für sich einen eigentlichen Umlauf dar; daher kann es sein, dass in anderen terminologischen Zusammenhängen das als *Umlauf* bezeichnet wird, was in der (deutschsprachigen) Eisenbahn-Fachwelt nur eine *Umlaufgruppe* ist. Jede Umlaufgruppe könnte theoretisch in einem eigenen Umlaufplan abgebildet werden. Dies wird in der Praxis aber oft so nicht vorgenommen, weil man u. U. bemüht ist, *alle* Fahrzeuge einer Baureihe in *einem* Umlaufplan darzustellen.

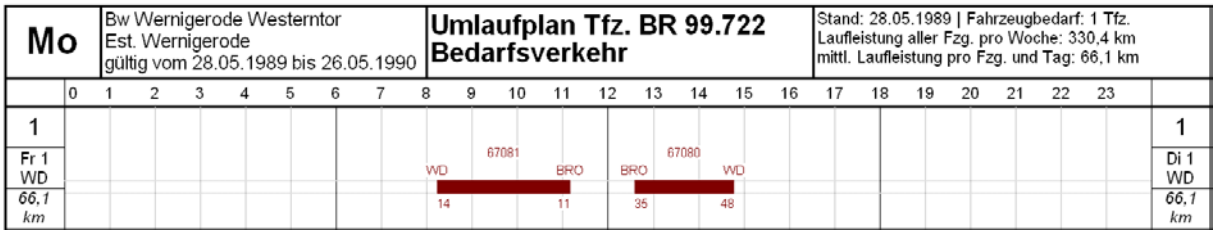
Beispiel: Auf einer einfachen Strecke pendeln zwei Triebwagen. Sie fahren immer an beiden Streckenenden gleichzeitig ab, begegnen sich in der Mitte und kommen gleichzeitig am anderen Ende an. Die Fahrtenanzahl je Richtung ist gerade, also jeder Triebwagen fährt 10x hin und 10x zurück. Damit kommt jeder am Abend wieder dort an, wo er früh begonnen hat. Macht man jetzt einen Umlaufplan für die Triebwagen, bekommt man eigentlich zwangsläufig zwei Umläufe (für jeden Triebwagen einen – sie können ja nie die rollen tauschen). Der klassische Eisenbahner sagt aber trotzdem „Umlauf Baureihe soundso“ dazu und behauptet, dass das ein Umlauf wäre, obwohl es informatorisch eher zwei sind.



Umlaufplan-Beispiel 1

Der einfachste *offene* Umlauf könnte theoretisch aus nur einer Fahrt bestehen, wobei sich dann allerdings die zumindest philosophische Frage stellt, ob das wirklich ein *Um*lauf ist. Der einfachste *geschlossene* Umlauf sollte mindestens zwei Fahrten haben, sofern wir Züge, die im Kreis fahren, einmal ausschließen wollen.

Umlaufpläne von Eisenbahnen sind oft komplex; so ist es auch ihre Darstellung in RailML. Daher ist es gar nicht so einfach, ein *einfaches* Beispiel und übersichtlich darstellbares zu finden... Um also zunächst mit dem nahezu einfachsten vorstellbaren Umlaufplan beginnen zu können, gehen wir in der Geschichte noch einmal etwas zurück:



Der folgende RailML-Auszug ist vollständig; er wird nur zum Kommentieren unterbrochen.

Das RailML-Element **rostering** definiert einen Umlaufplan für eine Fahrzeugbaureihe (welche über *vehicleRef* oder *formationRef* referenziert wird).

```
<rostering id='rost_99.722' name='Bedarfsverkehr' description='Umlaufplan Tfz. BR 99.722'
                                                  vehicleRef='veh_DR.99.722'
                                                  defaultPreProcessingTime='PT2M0S'
                                                  defaultPostProcessingTime='PT2M0S'>
```

Das RailML-Element **blockPart** enthält einen Einsatz (eine Fahrt oder ein sonstiger Einsatz – z. B. Tanken, Wartung, Rangieren, Vorheizen usw.).

```
  <blockParts>
    <blockPart id='bp_67081_WD' begin='08:14:18' startOcpRef='ocp_WD' mission='timetable'
                                trainPartRef='tp_67081_WD-DRW' end='09:33:23'
                                endOcpRef='ocp_DRW' runLength='14.060'/>
    <blockPart id='bp_67081_DRW' begin='09:50:18' startOcpRef='ocp_DRW' mission='timetable'
                                trainPartRef='tp_67081_DRW-BRO' end='11:10:38'
                                endOcpRef='ocp_BRO' runLength='18.980'/>
    <blockPart id='bp_67080_BRO' begin='12:35:18' startOcpRef='ocp_BRO' mission='timetable'
                                trainPartRef='tp_67080' end='14:47:44'
                                endOcpRef='ocp_WD' runLength='33.040'/>
  </blockParts>
```

Es sind folgende Ausprägungen von *blockPart* vorgesehen:

- **Fahrt als Referenz auf einen Zugteil**: *mission=timetable; trainPartRef* ist anzugeben; die übrigen Attribute (*startOcpRef*, *endOcpRef*, *begin*, *end*, *runLength*) sind redundant zu den gleichbedeutenden Eigenschaften der Zugteile bei deren Definition in der RailML-Datei. Das Wiederholen dieser Attribute ist optional; sofern sie wiederholt werden, müssen sie widerspruchsfrei sein.
- **Fahrt ohne Referenz auf einen Zugteil**: *mission=fullRun* oder *emptyRun*; *trainPartRef* darf nicht angegeben sein, *startOcpRef*, *endOcpRef*, *begin* und *end* müssen angegeben sein.
- **Dienst** (der keine Fahrt ist): mission=*shunting* (Rangierdienst)
  =*maintenance* (Wartungsdienst)
  =*standBy* (Bereitschafts- und Reservedienst)
  =*preheating* (Vorheizdienst)
  =*refuel* (Auffüllen von Vorräten, Tanken usw.)
  trainPartRef darf nicht angegeben sein, *startOcpRef*, *endOcpRef*, *begin* und *end* müssen angegeben sein, *startOcpRef= endOcpRef*.

Im obigen Beispiel bilden die ersten beiden *blockParts* den Zug 67081; die Trennung beider ist in der Umlauf-Grafik nicht sichtbar. Der dritte *blockPart* bildet den Zug 67080. Dienste sind in diesem Beispiel nicht enthalten.

Das RailML-Element **block** verkettet *blockParts* zu (willkürlichen) Abfolgen, die z. B. den üblicherweise in den Umlaufgrafiken dargestellten Blöcken entsprechen. Hier wird ersichtlich, dass der Zug(=Block) 67081 aus zwei *blockParts* zusammengesetzt ist, der Zug 67080 nur aus einem:

```
  <blocks>
    <block id='bl_67081' name='N 67081 B W[Sa]'>
      <blockPartSequence sequence='1'>
        <blockPartRef ref='bp_67081_WD'/>
      </blockPartSequence>
      <blockPartSequence sequence='2'>
        <blockPartRef ref='bp_67081_DRW'/>
      </blockPartSequence>
    </block>
    <block id='bl_67080' name='N 67080 B W[Sa]'>
      <blockPartSequence sequence='1'>
        <blockPartRef ref='bp_67080_BRO'/>
      </blockPartSequence>
    </block>
  </blocks>
```

Das RailML-Element **circulaiton** definiert die Reihenfolge der „Blöcke“ in Bezug auf Verkehrstage über operatingPeriodRef. Es ist je ein *circulation*-Element für jeden Block an jedem Umlauftag enthalten. Durch die Attribute *nextBlockRef* und *nextOperatingPeriodRef* jedes *circulation*-Elements ergibt sich die Verkettung der Blöcke und damit die eigentliche Umlaufabfolge. Das Attribut *operatingPeriodRef* jedes *circulation*-Elements gibt über den Inhalt der referenzierten Verkehrstageregelung (indirekt) an, ob und wie oft ein Umlauftag wiederholt werden muss, um die gesamte Gültigkeit des Umlaufplans abzudecken.

```
  <circulations>
    <circulation blockRef='bl_67081' operatingPeriodRef='opp_9'
                 nextBlockRef='bl_67080' nextOperatingPeriodRef='opp_9'/>
    <circulation blockRef='bl_67080' operatingPeriodRef='opp_9'
                 nextBlockRef='bl_67081' nextOperatingPeriodRef='opp_9'/>
  </circulations>
</rostering>
```

Das erste circulation-Element (bl_67081/opp_9) verweist als Nachfolger auf das zweite Element bl_67080/opp_9. Die Wertepaarung **blockRef/operatingPeriodRef** stellt einen Primärschlüssel innerhalb der circulations-Liste dar, so dass über den Verweis **nextblockRef/nextoperatingPeriodRef** immer eindeutig ein Ziel gefunden werden kann.

Da das Verweisziel des zweiten Elements (bl_67080/opp_9) wieder das erste ist, ist der Umlauf damit bereits vollständig und geschlossen.

Würde das letzte circulation-Element keinen Nachfolger definieren (nextblockRef/nextoperatingPeriodRef sind nicht angegeben), würde es sich um einen offenen Umlauf handeln.

Damit die beiden Züge **täglich** diesen Umlaufplan fahren, müsste in diesem Beispiel die *operatingPeriod* opp_9 als täglich definiert sein. Falls opp_9 nur einen einzigen Tag definiert, würde der Umlauf auch nur an einem Tag gelten. Im konkreten Beispiel zeigte opp_9 auf die Verkehrstageregelung „W[Sa]“ (=Mo-Fr, außer Feiertags) bei Bedarf, woraus folgte, dass das Fahrzeug an Sa+S zumindest durch diesen Umlauf nicht beansprucht wurde.

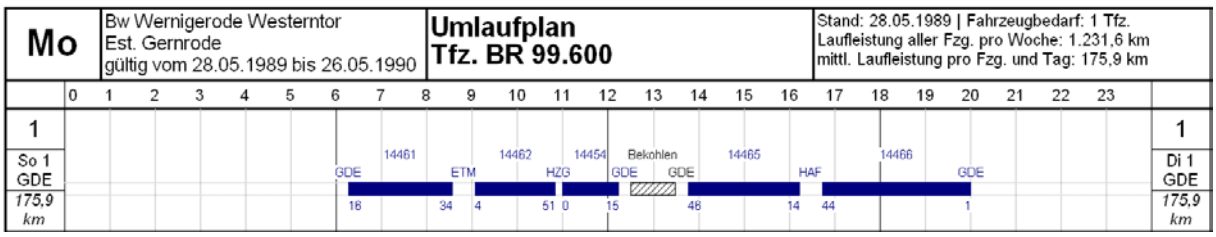
Die Umlauf-Abfolge von z. B. fünf Wochentagen Mo-Fr ließe sich auch durch Aufzählen der fünf Tage abbilden, was gleichbedeutend mit obigem Beispiel wäre:

```
  <circulations>
    <circulation blockRef='bl_67081' operatingPeriodRef='opp_Mo'
                 nextBlockRef='bl_67080' nextOperatingPeriodRef='opp_Mo'/>
    <circulation blockRef='bl_67080' operatingPeriodRef='opp_Mo'
                 nextBlockRef='bl_67081' nextOperatingPeriodRef='opp_Di'/>
    <circulation blockRef='bl_67081' operatingPeriodRef='opp_Di'
                 nextBlockRef='bl_67080' nextOperatingPeriodRef='opp_Di'/>
    <circulation blockRef='bl_67080' operatingPeriodRef='opp_Di'
                 nextBlockRef='bl_67081' nextOperatingPeriodRef='opp_Mi'/>
    <circulation blockRef='bl_67081' operatingPeriodRef='opp_Mi'
                 nextBlockRef='bl_67080' nextOperatingPeriodRef='opp_Mi'/>
    <circulation blockRef='bl_67080' operatingPeriodRef='opp_Mi'
                 nextBlockRef='bl_67081' nextOperatingPeriodRef='opp_Do'/>
    <circulation blockRef='bl_67081' operatingPeriodRef='opp_Do'
                 nextBlockRef='bl_67080' nextOperatingPeriodRef='opp_Do'/>
    <circulation blockRef='bl_67080' operatingPeriodRef='opp_Do'
                 nextBlockRef='bl_67081' nextOperatingPeriodRef='opp_Fr'/>
    <circulation blockRef='bl_67081' operatingPeriodRef='opp_Fr'
                 nextBlockRef='bl_67080' nextOperatingPeriodRef='opp_Fr'/>
    <circulation blockRef='bl_67080' operatingPeriodRef='opp_Fr'
                 nextBlockRef='bl_67081' nextOperatingPeriodRef='opp_Mo'/>
  </circulations>
</rostering>
```

Sofern mehr als ein *circulation*-Element für einen Block existiert, müssen die *circulation*-Elemente für diesen Block bezüglich der Verkehrstage disjunkt sein.

Umlaufplan-Beispiel 2

Hier noch ein zusammenhängendes Beispiel mit ein paar mehr Fahrten und mit eingeschobenem Dienst (wobei der Umlauf immernoch „Züge“ von Trivialität hat, wenn es von der „Baureihe“ nur ein einziges Exemplar gibt):



```
<rostering id='rost_99.600' name='Tfz. BR 99.600' vehicleRef='veh_DR.99.600'
                    defaultPreProcessingTime='PT2M0S' defaultPostProcessingTime='PT2M0S'>
  <blockParts>
    <blockPart id='bp_14461GDE' begin='06:16:18' startOcpRef='ocp_GDE' mission='timetable'
              trainPartRef='tp_14461' end='08:33:02' endOcpRef='ocp_ETM' runLength='44.461'/>
    <blockPart id='bp_14462ETM' begin='09:04:18' startOcpRef='ocp_ETM' mission='timetable
              trainPartRef='tp_14462' end='10:50:06' endOcpRef='ocp_HZG' runLength='32.841'/>
    <blockPart id='bp_14454HZG' begin='11:00:18' startOcpRef='ocp_HZG' mission='timetable'
              trainPartRef='tp_14454' end='12:14:08' endOcpRef='ocp_GDE' runLength='17.480'/>
    <blockPart id='bp_Bekohlen' begin='12:30:00' end='13:30:00' startOcpRef='ocp_GDE
              endOcpRef='ocp_GDE' operatingPeriodRef='opp_9' mission='maintenance'/>
    <blockPart id='bp_14465GDE' begin='13:46:18' startOcpRef='ocp_GDE' mission='timetable'
              trainPartRef='tp_14465' end='16:13:59' endOcpRef='ocp_HAF' runLength='40.580'/>
    <blockPart id='bp_14466HAF' begin='16:44:18' startOcpRef='ocp_HAF' mission='timetable'
              trainPartRef='tp_14466' end='20:00:52' endOcpRef='ocp_GDE' runLength='40.580'/>
  </blockParts>
  <blocks>
    <block id='bl_14461' name='P 14461'>
      <blockPartSequence sequence='1'>
        <blockPartRef ref='bp_14461GDE'/>
      </blockPartSequence>
    </block>
    <block id='bl_14462' name='P 14462'>
      <blockPartSequence sequence='1'>
        <blockPartRef ref='bp_14462ETM'/>
      </blockPartSequence>
    </block>
    <block id='bl_14454' name='P 14454'>
      <blockPartSequence sequence='1'>
        <blockPartRef ref='bp_14454HZG'/>
      </blockPartSequence>
    </block>
    <block id='bl_Bekohlen' name='Bekohlen'>
      <blockPartSequence sequence='1' preProcessingTime='PT0S' postProcessingTime='PT0S'>
        <blockPartRef ref='bp_Bekohlen'/>
      </blockPartSequence>
    </block>
    <block id='bl_14465' name='P 14465'>
      <blockPartSequence sequence='1'>
        <blockPartRef ref='bp_14465GDE'/>
      </blockPartSequence>
    </block>
    <block id='bl_14466' name='P 14466'>
      <blockPartSequence sequence='1'>
        <blockPartRef ref='bp_14466HAF'/>
      </blockPartSequence>
    </block>
  </blocks>
  <circulations>
    <circulation blockRef='bl_14461' operatingPeriodRef='opp_9' vehicleIdx='1'
                    nextBlockRef='bl_14462' nextOperatingPeriodRef='opp_9'/>
    <circulation blockRef='bl_14462' operatingPeriodRef='opp_9' vehicleIdx='1'
                    nextBlockRef='bl_14454' nextOperatingPeriodRef='opp_9'/>
    <circulation blockRef='bl_14454' operatingPeriodRef='opp_9' vehicleIdx='1'
                    nextBlockRef='bl_Bekohlen' nextOperatingPeriodRef='opp_9'/>
    <circulation blockRef='bl_Bekohlen' operatingPeriodRef='opp_9' vehicleIdx='1'
                    nextBlockRef='bl_14465' nextOperatingPeriodRef='opp_9'/>
    <circulation blockRef='bl_14465' operatingPeriodRef='opp_9' vehicleIdx='1'
                    nextBlockRef='bl_14466' nextOperatingPeriodRef='opp_9'/>
    <circulation blockRef='bl_14466' operatingPeriodRef='opp_9' vehicleIdx='1'
                    nextBlockRef='bl_14461' nextOperatingPeriodRef='opp_9'/>
  </circulations>
</rostering>
```

Kopfinformationen (Dublin Core Metadata Element Set)

Die folgende Beschreibung enthält eine Empfehlung für die Verwendung des Dublin Core (DC) Metadata Element Set aus **rail:metadata** (am Anfang der RailML-Datei). Momentan ist es nicht zwingend erforderlich, die DC-Elemente in der hier beschriebenen Art zu verwenden. Da es aber dem Datenaustausch nicht dienlich ist, gleiche Attribute mit verschiedenen Bedeutungen zu verwenden, wird die hier beschriebene Anwendung dringend empfohlen.

```
<railml version='2.0' … >
  <metadata>
    <dc:format>2.0.3</dc:format>
    <dc:identifier>1</dc:identifier>
    <dc:language>1252  (ANSI - Lateinisch I)</dc:language>
    <dc:source>iPLAN.exe V1.2.0.528 NtzIntf_RailML2.dll V2.0.4.23</dc:source>
    <!--created with FBS (www.irfp.de) iPLAN.exe V1.2.0.528 NtzIntf_RailML2.dll V2.0.4.23-->
    <dc:date>2012-03-01T11:14:59</dc:date>
    <dc:creator>iRFP</dc:creator>
  </metadata>
```

Das Attribut **railml.version** soll die „Nenn-Version“ enthalten, unter der die RailML-Schemendateien veröffentlicht wurden. Typische Werte sind derzeit **2.0**, **2.1** und **2.2**.

Im **Dublin Core Metadata Element Set** (Namensraum „dc:“, Struktur railml.metadata) lassen sich implementierungsabhängige Versionsnummern und andere Kopfinformationen unterbringen.

Das Attribut **metadata.format** enthält die interne Versionsnummer der Schemen-Ausprägung (auch RailML-Profil genannt). Diese Versionsnummer ändert sich dann, wenn sich die Interpretation oder die Vollständigkeit der Umsetzung des RailML-Schemas durch die schreibende Software ändert. Ein lesendes Programm sollte prüfen, dass diese Versionsnummer nicht niedriger ist als die Version, mit der das Programm frühestens getestet wurde. Insbesondere kann hiermit vom lesenden Programm einfach geprüft werden, ob bestimmte Daten vorhanden sein werden, die

- notwendig sind für das lesende Programm,
- in RailML als optional gekennzeichnet sind,
- in der konkreten Schemeninstanz obligatorisch sind.

Es ist Sache der Programmierung der schreibenden Software, die Werte für metadata.format festzulegen. Diese Werte sind daher nur im Zusammenhang mit metadata.source interpretierbar. Es wird empfohlen, sich dabei an der zugrunde liegenden RailML-Schemenversion zu orientieren. (Im obigen Beispiel: Instanz Nr. 3 von RailML 2.0.) Außerdem ist es Aufgabe der schreibenden Programmierung, nur eindeutige Werte von metadata.format innerhalb eines Wertes metadata.source zuzulassen.

Das Attribut **metadata.identifier** enthält eine Kompatibilitätsnummer als einfachen Integer-Wert. Diese Nummer wird nur dann geändert, wenn ein bestehender Datenwert (Attribut oder Element) nachträglich uminterpretiert wird (z. B. in Folge einer Fehlerkorrektur). Ein lesendes Programm sollte dieses Attribut auf exakt den erwarteten Wert prüfen – sonst läuft es Gefahr, dass die auszuwertenden Datenfelder nicht mehr den erwarteten Inhalt enthalten. Beispielsweise könnte ein Geschwindigkeitsfeld bis zu einem bestimmten Zeitpunkt die Geschwindigkeit in km/h enthalten. Es sei angenommen, dass sich das nachträglich als nicht RailML-konform herausstellte. Um die RailML-Konformität wiederherzustellen, müsste des Geschwindigkeitsfeld nachträglich – ohne umbenannt zu werden – auf die Einheit m/s umgestellt werden. In diesem Falle würde metadata.identifier um eins weitergezählt werden, um lesende, nicht aktualisierte Programme davon abzuhalten, den neuen Wert als km/h einzulesen.

Das Attribut metadata.identifier wird nicht aktualisiert, wenn neue Daten hinzukommen, was es von metadata.format unterscheidet. Es wird erwartet, dass sich metadata.identifier im Gegensatz zu metadata.format nur sehr selten ändert.

Das Attribut **metadata.source** enthält eine Zeichenkette, die eindeutig das schreibende Programm identifiziert. Optionale Versionsnummern können der Fehleranalyse dienen.

Das Attribut **metadata.language** enthält Nummer und (optional) Namen des Zeichensatzes, in dem die Daten im schreibenden Programm vorliegen. Dieser Wert kann z. B. von Bedeutung sein, wenn die in der RailML-Datei enthaltenen UTF-8-Zeichenketten (Bahnhofsnamen usw.) vom lesenden Programm in eine Nicht-Unicode-Zeichenkette umgewandelt werden müssen.

Dieser Wert ist nicht zu verwechseln mit `<?xml ... encoding='UTF-8' ?>`, welcher die Codierung der RailML-Datei und damit aller enthaltener Zeichen definiert. Da eine RailML-Datei normalerweise im Zeichensatz UTF-8 codiert ist, ist der Wert metadata.language zum reinen Auslesen der RailML-Datei nicht notwendig. Er ist nur von Bedeutung, wenn die Eigennamen aus der RailML-Datei im Zielprogramm in Nicht-Unicode-Zeichensätze umgewandelt werden müssen. Ein lesendes Programm soll hier nicht erst alle Eigennamen auf eventuell vorhandene Umlaute „scannen“ müssen, um sich auf eher empirische Weise für einen Zeichensatz zu entscheiden. Vielmehr soll das schreibende Programm – wannimmer möglich – quasi die Herkunft oder Zugehörigkeit der Namen zu einer Sprache angeben.

Beispielsweise wäre im Falle von `<dc:language>1253 (ANSI - Greek)</dc:language>` für ein lesendes Programm ohne diese Angabe erst durch Feststellen eigenartig hoher Unicode-(UTF-8-)Werte in den Eigennamen feststellbar, dass es sich offensichtlich um griechische Namen handelt. Sofern ein schreibendes Programm die geografische Zuordnung der Namen nicht feststellen kann, soll es metadata.language auslassen.

metadata.language soll als erste Zeichen – bis zu einem trennenden Leerzeichen – die Code-Seite (Codepage) des Zeichensatzes als dezimalen Zahlenwert enthalten. Danach kann optional der Name der Codepage angegeben werden.

Das Attribut **metadata.date** enthält optional Datum und Uhrzeit des Exports (des Erzeugens der RailML-Datei) im xs:dateTime-Format.

Das Attribut **metadata.creator** enthält optional den Anwender- oder Lizenznamen des Anwenders, der die Datei erzeugt hat (z. B. Anmeldename am Betriebssystem oder Firmenname, auf den die Programmlizenz ausgestellt ist.)