Design of Information Systems

Railway Planner

AG Datenbanksysteme
Fachbereich 3
Universität Bremen

by

Merlin Burri
Marlon Flügge
Tilman Ihrig

Frank Hilken
Prof. Dr. Martin Gogolla

Handed in on
August 18, 2017
4.12 TrackSection::init() ........................................... 22
4.13 Route::init() ................................................. 23
4.14 Route::addStage() ........................................... 24
4.15 Route::removeStage() ......................................... 25
4.16 Route::overlaps() ............................................. 25
4.17 Route::getAvailableTrain() .................................. 26
4.18 Route::getAvailableDriver() ................................. 27
4.19 Route::getAvailableConductor() ............................. 27
4.20 Stage::init() .................................................. 28
4.21 Stage::temporallyOverlaps() ................................. 30
4.22 Stage::getAvailableTrackSection() ......................... 30
4.23 Driver::init() ................................................ 31
4.24 Driver::assignToRoute() ..................................... 32
4.25 Conductor::init() ............................................ 33
4.26 Conductor::assignToRoute() ................................. 33
4.27 Conductor::createRoute() ................................... 34

5 Scenarios ......................................................... 37
5.1 Invariants ...................................................... 37
5.1.1 Train, Driver and Conductor ............................... 39
5.1.2 Route ....................................................... 44
5.1.3 Stage ....................................................... 49
5.1.4 Platform ................................................... 57
5.1.5 Time ......................................................... 60
5.2 Operations ..................................................... 61

6 Queries .......................................................... 75
6.1 Resources ...................................................... 75
6.1.1 Workload .................................................... 76
6.1.2 Available resources for route ............................. 77
6.2 Route .......................................................... 78
6.2.1 Stops for route .............................................. 78
6.2.2 Routes for origin and destination ......................... 79
6.2.3 Routes for origin and destination with departure and arrival times 79
6.2.4 Routes for origin, destination, current time ............ 80
6.2.5 Routes for origin, destination, arrival time ............ 82
6.2.6 Routes for origin, destination, current time and train type 83
6.3 Miscellaneous ................................................. 84
6.3.1 Conductor's timetable .................................... 84
6.3.2 Reachable train stations from train station ............ 85

7 Outlook ........................................................... 87

A Code ............................................................. 89
1. Introduction

Author: Marlon Flügge

This paper is the result of our efforts to model a rudimentary railway planning system, as part of the course ‘Design of Information Systems’ in the summer semester of 2017. Our model was developed and evaluated using USE (UML Based Specification Environment), a tool to model information systems based on UML (Unified Modeling Language) and OCL (Object Constraint Language) developed at the University of Bremen by the Database Systems Group.

First we will describe our system on a high level basis, followed by presenting the system’s UML class diagram. Afterwards, we will discuss invariants represented as OCL expressions. What will be following is a description of the operations with their corresponding SOIL (Simple OCL-based Imperative Language) implementations. Finally, we will present a few example scenarios to test our invariants and operations and show a few exemplary OCL expressions used to query useful information.

1.1 System description

The system being modelled in this paper is a planning system for a fictitious railway company. Its purpose is to enable the planning of regular scheduled railway traffic. The finer details of the system are described below.

The system’s centerpiece are routes. In general, a route describes a train ride from one train station to another, more specifically it describes a complete journey of a train from its starting station to its final destination. Usually there are several train stations along the way. Each stage along the journey is defined in a separate object. A route contains all stages making up the complete route, an assigned train as well as a train driver and a conductor. The start and end stations as well as departure and arrival times are all contained within the individual stages.

A stage describes a direct train ride from one train station to another. It consists of a source platform as well as a destination platform, both having associated train stations. Also, each stage has a departure and arrival time. Additionally, every stage is assigned a track section that has to connect the source and destination. The time is specified by hours and minutes.

A train station has a unique name specifying its exact location. Moreover, a train station has multiple platforms that all have an ID that is unique for its assigned train station. A
track section is defined by the two connected train stations. Since there can be multiple track sections between two train stations and we want to assign specific tracks to specific stages, a track section also has a unique id.

Every train has a type and a number, the combination of which is unique for every train. Finally, there are two types of employees: Train drivers and conductors, both having unique employee ids.

There are several things that have to be considered when planning routes. For example, when a track sections is used for multiple stages at the same time, the destination of all stages needs to be the same (so that the trains don’t collide head-on) and there has to be a certain difference in both the departure times and arrival times so that all trains remain at a certain distance over the section. Other limitations include employees and trains not being able to service multiple routes with overlapping timeframes or platforms not being able to host multiple trains at the same time.

There are multiple ways in which the system can assist while planning railway traffic, for instance by asking the system to display all connections between two train stations. Moreover, giving a specific time, one could ask for the next connection between two stations. One could also ask for all visited stations on a route.

When looking for employees to assign to routes, one could also retrieve all employees that are available during a specified time period. The same can be done for trains. Also, one could specify a train station and a point in time and ask for all available platforms.
2. Model

Author: Marlon Flügge

Figure 2.1 shows the class diagram for our railway planner. This basically shows the specifications given in the system overview implemented in actual OCL. In the following we will briefly discuss the classes and their attributes included in your model. Operations will be discussed in a more in-depth manner in chapter 4.

2.1 Classes

First of all, all defined classes will be presented.

2.1.1 Train

This class represents a simple train. Its only attribute is type, which is used to model different classes of trains, like 'ICE' or 'RE'. The class has two operations: One is a simple initialization operation that sets the type parameter, the other one lets us assign the train to given route.

2.1.2 Route

As described in the system overview a route represents a train ride from an origin station to a destination, which can span across multiple train stations modeled using Stage objects. Origin and destination can be derived by looking at the first and last stage respectively. Route has 7 methods in total: An initialization function, operations to add or remove stages, the operation 'overlaps', which takes another Route object and checks if they temporally overlap and 3 utility methods to get an available train, conductor or driver for this route.

2.1.3 Employee, Conductor & Driver

Employee is an abstract class that models any kind of employee involved in the process. It is supposed to store all of the attributes and operations that are universal to employees regardless of their specific position. For now it only stores the attribute name, but is easily expandable. Since it is an abstract class, it does not have an initialization operation. We included two subclasses of Employee in our model, namely Driver and Conductor.

A driver only symbolically represents the train’s driver without actually having any explicit functionality inside our system. Conductor, however, does have a purpose, additional to
Figure 2.1: Railway Planner Class Diagram
functioning as a symbolical conductor for a given route. The conductor is responsible for creating new routes in our system, using the `createRoute` operation.

Both `Driver` and `Conductor` have basic initialization operations as well as operations to assign them to a given route.

### 2.1.4 Stage

`Stage` models an elementary part of a route. We call a train ride "elementary" if it is leading from one train station to another without crossing any other train station on the way. Stages form the central aspect of a route, determining where the route starts, end and what train stations are passed on the way. Apart from the ubiquitous `init`-operation, stages also have an operation that checks if they temporally overlap with another, given stage as well as an operation to query a track section that could be used for this stage.

### 2.1.5 TrainStation, TrainSection & Platform

`TrackSection` models a track section connecting two train stations. `TrainStation` models a train station inside our system. Every train station needs a name so the user can differentiate between them more easily. A `TrainStation` can be queried for an available platform. `Platform` represents a platform on a train station. There can’t be more trains in a train stations at the same time than there are platforms on a station. A `Platform` can be queried to check if it is available at a given time.

### 2.1.6 Time

When scheduling railway traffic time is obviously a central aspect. In order to properly model this we created the class `Time`, which represents different points in time that can be associated with destinations and arrivals at train stations. This class offers operations that can be used to enforce a lot of constraints and invariants. While a time object has `hours` and `minutes` attributes, seconds or dates are not modeled within our system, since we only try to model a daily schedule and railway traffic cannot be accurately scheduled to within seconds since there are several outside factors that can influence the length of a train ride.

The `Time`-class also offers several utility operations, e.g. comparing two times to check which one is later.

### 2.2 Associations

Using the multiplicities next to the associations we can see what is needed for an object of a given class to be valid. We also mention a few of the invariants that are thoroughly explained in chapter 3.

#### 2.2.1 TrainForRoute

This association connects the `Train` and `Route` classes. It models the prerequisite that every planned route in our system needs exactly one associated train that is doing the actual ride. Additionally it allows a train to be associated to multiple routes at once. Via the `TrainNotUsedSimultaneously` invariant we make sure that these associated routes do not overlap in time.

#### 2.2.2 DriverOf & ConductorOf

These associations exist between the `Route` class and the `Employee` subclasses. Every route needs exactly one of each as formalized in the associations’ multiplicities. Similar to the train in the `TrainForRoute` association drivers and conductors can be associated with multiple routes, as long as these routes do not temporally overlap.
2.2.3 StagesForRoute

At heart a route is just a collection of stages which is modeled using this association. As previously explained a stage is just an elementary connection between two train stations without any intermediate stops. If a route is planned to span across multiple train stations every elementary connection from train station to train station will be modeled using a separate stage. Thus, the StagesForRoute association makes sure every route consists of at least one stage but doesn’t specify an upper bound for the number of stages that can make up a route. A stage, however, can only ever be part of a single route. Having the same stage associated to multiple routes would not make sense, since that would mean that multiple trains would be using the same track section at the same time.

2.2.4 TrackForStage

This association links a track section to a stage. Every stage needs a track section that the train can use to go from one train station to another. A track section can be associated to multiple stages as long as these don’t temporally overlap, which enforced using the invariants TimeDifferenceSameDirection and NoOverlapsOppositeDirections.

2.2.5 DestinationOfStage & OriginOfStage

A stage always has to lead from one train station to another, which is modeled using this association. The origin and destinations points are instances of the class Platform which are linked to certain train stations via the PlatformInStation association. Using these links it is possible to set or later determine the start and end points of a stage. Again, platforms can be used by multiple stages as long as these do not overlap in time.

2.2.6 PlatformInStation

As described this association links platforms to certain train stations. While a platform can only belong to one train station, a train station can obviously have multiple platforms in order to hold multiple trains at the same time.

2.2.7 EndPoints

This association represents the fact that track sections have to begin an end somewhere. In our model these points can only be train station, which is why EndPoints is connecting track sections to train stations. Since beginning and end have to be specified, a track section needs two associated train stations while the latter may be connected to variable number of track sections.

2.2.8 Departure & Arrival

As previously mentioned the concept of time is obviously essential for any problem that contains some form of scheduling as it is for our system. Using the Departure and Arrival associations we can model departure and arrival times of different stages, which later on can be used for planning and invariant checks. Consequently, these associations connect a stage with a departure and arrival time, respectively. Since points in time are not unique a given Time object might be used by multiple stages.
3. Invariants

Author: Merlin Burri

In the following chapter, the invariants, i.e. formal constraints, for the model will be discussed. This includes verbal descriptions as well as the OCL representations for our constraints for all classes in the model. If there are no invariants for a specific class, the class is not mentioned in this chapter. The tests for all invariants can be found in chapter 5.1 which follows the same structure as this chapter. For every invariant, the constraint is first described and what follows is the OCL representation. Since our model almost exclusively contains attributes defined by associations, for the most part, we do not need invariants that guarantee attributes being defined. This is implicitly covered by the multiplicities of our associations.

3.1 Train

The only invariant for the Train class is TrainNotUsedSimultaneously. The constraint makes sure that Train objects are not assigned to multiple Route objects that contain some form of temporal overlap, ensuring that trains are not used simultaneously in multiple routes. For the OCL representation, a utility operation called overlaps of the Route class is used. The operation is further described in 4.16.

1 — Train is not assigned to multiple Routes at the same time
2 context Train inv TrainNotUsedSimultaneously:
3 self.route->forall(r1: Route, r2: Route | r1.overlaps(r2) implies r1 = r2

3.2 Driver

In accordance with the Train invariant, the only Driver class invariant is called DriverNotUsedSimultaneously and ensures that no Driver objects are assigned to multiple Route objects, providing that the Route objects overlap in time. Here, we again make use of the overlaps operation of the Route class.

1 — Driver is not assigned to multiple Routes at the same time
2 context Driver inv DriverNotUsedSimultaneously:
3 self.route->forall(r1: Route, r2: Route |
3. Invariants

3.3 Conductor

In accordance with the two previously described invariants, for the second Employee subclass Conductor we have defined only one invariant: ConductorNotUsedSimultaneously. By again using the overlaps utility operation, the constraint ensures that a Conductor object is only assigned to multiple Route objects if there is no temporal overlap between the routes.

```plaintext
context Conductor inv ConductorNotUsedSimultaneously:
  self.route->forall(r1: Route, r2: Route |
    r1.overlaps(r2) implies r1 = r2
  )
```

It should be noted that it is not sufficient to define one invariant for the Employee class because...

3.4 Route

The first invariant of the Route class is called DepartureAfterArrivalPreviousStage. It makes sure that for the ordered set of Stage objects that the Route is associated with, the departure time of the next stage is later than the arrival time of the previous stage. To determine the next (and previous) stage(s), we take advantage of the used data structure for the StagesForRoute association, which – as mentioned – is implemented as an ordered set. To check whether or not a specific time is later than another time, we use the utility operation isLater of the Time class which is further explained in chapter 4.6.

```plaintext
context Route inv DepartureAfterArrivalPreviousStage:
  self.stage->forall(s: Stage |
    let currentStageNumber: Integer = s.indexOf(stages)
    in if (currentStageNumber < stage->size()) then
      s.departureTime .isLater(stages.at(currentStageNumber + 1).departureTime)
    else
      true
    endif
  )
```

Secondly, we have defined an invariant that similarly ensures that the destination (i. e. the arrival Platform object) of the previous stage equals the origin (the departure Platform object) of the next stage in the route: DeparturePlatformPreviousPlatform. This guarantees that a train arriving to a platform will always depart from that same platform. Since Platform objects are uniquely associated with TrainStation objects, the constraint also makes sure that the train station the train is departing from always equals the train station that it has previously arrived to. We again make use of the StagesForRoute association’s implementation to determine next and previous stage(s).
3.5 Stage

For every Stage in the Route, the Platform that the Train is departing from has to be the Platform that the Train arrived on in the previous Stage. This also makes sure that the TrainStation the Train is departing from equals the TrainStation that it arrived on in the previous Stage.

context Route inv DeparturePlatformPreviousPlatform:

```
self.stage ->forall (s: Stage |
  let currentStageNumber : Integer = stage ->indexOf(s)
  in if (currentStageNumber < stage ->size()) then
    s.destination = stage ->at(currentStageNumber + 1).origin
  else
    true
  endif)
```

The last invariant NoCircles for the Route class forbids circles within routes. For this purpose, we check every stage in the route and ensure that two different stages in one route never arrive to or depart from the same TrainStation. Routes starting and ending in the same train station are allowed.

context Route inv NoCircles:

```
self.stage ->forall (s1, s2: Stage |
  (s1.origin.trainStation = s2.origin.trainStation
  or
  s1.destination.trainStation = s2.destination.trainStation)
  implies
  s1 = s2)
```

### 3.5 Stage

The first invariant for the class Stage that we defined is called ArrivalAfterDeparture and regulates the arrival and departure time of a stage. By using the isLater utility operation of the Time class, we make sure that the arrival time of a stage is later than the departure time, so a train always arrives after it has departed.

context Stage inv ArrivalAfterDeparture:

```
self.arrivalTime.isLater(self.departureTime)
```

The second invariant TrackSectionConnectOriginDestination concerns itself with the TrackSection objects associated with stages, i.e. with the TrackForStage association. The constraint ensures that the assigned track section does in fact connect the two train stations that the stage is departing from/arriving to, with the departing and arriving station being defined by the departing and arriving platform of the stage.

context Stage inv TrackSectionConnectOriginDestination:

```
self.trackSection.trainStation ->exists (s: TrainStation |
  s = self.destination.trainStation)
```
The \texttt{NoOverlapsOppositeDirections} invariant asserts that there are no trains using the same track section at the same time while going in opposite directions. To be more specific, the constraint checks for all possible \texttt{Stage} pairs if they have the same assigned \texttt{TrackSection} object and, via another utility operation called \texttt{temporallyOverlaps} provided by the \texttt{Stage} class, whether the stages overlap in time. If that is the case, we make sure that both stages have the same destination by using the \texttt{DestinationOfStage} association, which of course amounts to them going in the same direction. The \texttt{temporallyOverlaps} operation is further elaborated on in chapter \ref{sec:time-overlaps}.

Lastly, we defined an invariant called \texttt{TimeDifferenceSameDirection}, which ensures that trains using the same track section at the same time going in the same direction have a certain difference (we arbitrarily chose 10 minutes) in their arrival and departures times. We again check all possible \texttt{Stage} pairs for usage of the same \texttt{TrackSection} object and temporal overlap. If there is overlap, we have to differentiate between two cases: In the first case, the first stage’s departure time is before the second stage’s. We then have to make sure that both the departure and the arrival time of the first stage are 10 minutes earlier than the respective times of the second. Accordingly, if the second stage’s departure time is before the first’s, the departure and arrival time of the second stage have to be before the respective times of the first. To extract the temporal difference between two \texttt{Time} objects, we use a utility operation called \texttt{getDifference} provided by the \texttt{Time} class, which is further explained in chapter \ref{sec:time-diff}.

Here, we can not just check the difference between arrival and departure times because of the implementation of the \texttt{getDifference} operation. Doing so without checking which train departs first would cause a system state in which a train overlaps another train while using the same track section to be valid. All in all, in combination with our \texttt{NoOverlapsOppositeDirections} invariant, we make sure that trains using the same track section while overlapping in time have to go into the same direction and that there has to be a difference of more than 10 minutes in their arrival and departure times, while forbidding overtakings.
3.6 Platform

For our Platform class, we have defined one invariant: MaxOneTrainPerPlatform. The constraint asserts that at the same time, no platform is occupied by multiple trains. To be more precise, all Stage objects associated with a Platform object via the DestinationOfStage association are inspected. First of all, the constraint ensures that multiple trains do not arrive at a single platform at the same time. Secondly, it is made sure that if two trains do arrive on the same platform, one of the trains has to depart again before the second arrives by inspecting the Stage set of the Route object associated to the current Stage object.

context Platform inv MaxOneTrainPerPlatform:

self.arrivingStage->forAll(a1, a2 | a1 = a2 or (a2.arrivalTime.isLater(a1.arrivalTime) or a1.arrivalTime.isLater(a2.arrivalTime)) and (a2.arrivalTime.isLater(a1.arrivalTime) implies a2.arrivalTime.isLater(a1.route.stage->at((a1.route.stage->indexOf(a1))+1).departureTime))

3.7 Time

The first invariant is called MinutesInInterval. Since we want to model a common clock with minute values in the interval from 0 to 59 and hour values from 0 to 23, the constraint ensures that the value for the minutes attribute is in exactly that interval.

context Time inv MinutesInInterval:

Time.allInstances->forAll( t: Time | t.minutes >= 0 and t.minutes < 60 )

Accordingly, the HoursInInterval invariant makes sure that the value for the hours attribute is in the interval from 0 to 23.

context Time inv HoursInInterval:

Time.allInstances->forAll( t: Time | t.hours >= 0 and t.hours < 24 )
4. Operations

Author: Tilman Ihrig

In this chapter the operations for each class in our model are introduced. This includes both the specification using pre- and post-conditions and the SOIL-implementations adhering to these specifications. Throughout this chapter, self is used to refer to the object on which the respective operation is called.

4.1 Train::init()

Initializes a Train object by assigning its type.

Parameters:

pType (String) Gives the type of a Train, e.g. RE or ICE.

Return value:
The operation has no return value.

Preconditions:

freshInstance self must be a fresh instance, i.e. its type must be undefined.
typeNotEmpty The given pType must contain at least one character.

A case could be made to also check whether the given type adheres to a specific naming scheme (e.g. ’RE’, ’ICE’ etc.). We have decided against specifying such a scheme. As such, nonsensical types are possible. On the other hand, there is complete freedom in expanding the number of train-types, as the precondition does not need to be changed every time a new train-type is introduced.

Postconditions:

typeAssigned The given type must be assigned correctly.

Implementation:

Assigns the given pType to type.
4. Operations

Code:

```plaintext
init (pType: String)
begin
  self.type := pType
end
pre freshInstance: self.type.isUndefined()
pre typeNotEmpty: pType.size > 0
post typeAssigned: self.type = pType
```

4.2 Train::assignToRoute()

Assigns a Train-object to a given Route by creating a corresponding TrainForRoute-association. If the Route already has an assigned Train, that association is deleted.

Parameters:

- `r` (Route)  The route to which `self` shall be assigned.

Return value:

The operation has no return value.

Preconditions:

- `trainRouteDefined`  The given Route must be defined.

Postconditions:

- `isAssigned`  `self` must be the train of the given Route.

A postcondition to check whether the association to a previously assigned Train has been deleted is not necessary, since the number of assignable Trains is limited to 1 in TrainForRoute.

Implementation:

Deletes the association between `r` and its currently assigned Train, if it already has an assigned Train, then creates an association between `self` and the given Route in TrainForRoute.

Code:

```plaintext
assignToRoute(r: Route)
begin
  if r.train.isDefined() then
    delete (r.train, r) from TrainForRoute;
  end;
  insert (self, r) into TrainForRoute;
end
pre trainRouteDefined: r.isDefined()
post isAssigned: r.train = self
```

4.3 TrainStation::init()

Initializes a TrainStation by assigning its name.
Parameters:

\( pName \) (String)  Gives the name of a \texttt{TrainStation}, e.g. \texttt{Bremen Hbf}.

Return value:

The operation has no return value.

Preconditions:

\( freshInstance \)  \texttt{self} must be a fresh instance, i.e. its \texttt{name} must be undefined.
\( nameNotEmpty \)  The given \texttt{pName} must contain at least one character.

Postconditions:

\( nameAssigned \)  The given name must be assigned correctly.

Implementation:

Assigns the given \texttt{pName} to \texttt{name}.

Code:

1. \texttt{init (pName: String)}
2. \hspace{1em} begin
3. \hspace{2em} self.name := pName
4. \hspace{1em} end
5. \hspace{1em} \texttt{pre freshInstance: self.name.isUndefined ()}
6. \hspace{1em} \texttt{pre nameNotEmpty: pName.size > 0}
7. \hspace{1em} \texttt{post nameAssigned: self.name = pName}

4.4 \texttt{TrainStation::getAvailablePlatform()}

Returns a Platform that is not used by any trains at a given Time.

Parameters:

\( t \) (Time)  The Time at which the Platform needs to be available.

Return value:

The operation returns a Platform that is available at the given Time. If no Platform is available, \texttt{null} is returned.

Preconditions:

\( hasPlatforms \)  \texttt{self} needs to have at least one Platform.
\( timeDefined \)  The time for which to check the availability needs to be defined.

Postconditions:

The operation has no postconditions.

Implementation:

Selects a Platform from all those that are available. For the availability check, \texttt{Platform::isAvailable()} is used.
4. Operations

Code:

```plaintext
getAvailablePlatform(t : Time) : Platform =
  self.platform -> any(p : Platform | p.isAvailable(t))
pre hasPlatforms: self.platform -> size > 0
pre timeDefined: t.isDefined()
```

4.5 Time::init()

Initializes a Time-object by assigning it a point in time, given as hours and minutes.

Parameters:

- `pHours (Integer)` Specifies the hours of a point in time.
- `pMinutes (Integer)` Specifies the minutes of a point in time.

Return value:

The operation has no return value.

Preconditions:

- `freshInstance` self must be a fresh instance, i.e. the `hours` and `minutes` must be undefined.
- `hoursInCorrectInterval` The given `pHours` must be valid hours in the 24-hour-system, i.e. between inclusively 0 and 23.
- `minutesInCorrectInterval` The given `pMinutes` must be valid, i.e. between inclusively 0 and 59.

Postconditions:

- `timeAssigned` The given `pHours` and `pMinutes` must be assigned correctly.

Implementation:

Assigns the given `pHours` to `hours` and `pMinutes` to `minutes`.

Code:

```plaintext
init(pHours: Integer, pMinutes: Integer)
begin
  self.hours := pHours;
  self.minutes := pMinutes;
end
pre freshInstance: self.hours.isUndefined() and
  self.minutes.isUndefined()
pre hoursInCorrectInterval: pHours >= 0 and pHours < 24
pre minutesInCorrectInterval: pMinutes >= 0 and pMinutes < 60
post timeAssigned: self.hours = pHours and
  self.minutes = pMinutes
```

4.6 Time::isLater()

Checks whether a Time-object is later than a given Time.
Parameters:

t (Time)  The Time-object to which self shall be 'compared'.

Return value:
The operation returns a Boolean value: True if self is later than t and False otherwise.

Preconditions:
The operation does not have any preconditions.

Postconditions:
The operation does not have any postconditions.

Implementation:

In addition to the two cases where self is intuitively later than t (hours are later or hours are equal and minutes are later), there is also a third case that needs to be considered, since dates are not modeled, but only 24-hour schedules. If a train departs shortly before midnight but arrives after midnight, then the arrival would not be considered later than the departure in the context of the two intuitive cases. For this reason, there is another case in which self is considered later, which is when the hours of self are 0 and the hours of the given Time-object are 23.

For Stages which span more than 1 hour between departure and arrival, this would not be enough, but expanding this to more hours before/after midnight would probably lead to more incorrect results than keeping it like this.

All three cases are disjuncted so that only one of them needs to be true to return True.

Code:

```java
1     −− checks if the Time the method is called on is
2     −− after the given Time
3     isLater ( t : Time ) : Boolean =
4         ( self . hours > t . hours) or
5         (( self . hours = t . hours) and ( self . minutes > t . minutes)) or
6         ( self . hours = 0 and t . hours = 23);
```

4.7 Time::getDifference()

Calculates the difference in minutes between a Time-object and a given Time.

Parameters:

t (Time)  The Time to compute the difference to.

Return value:
The operation returns an Integer which is positive if t is later than self and negative if self is later than t.

Preconditions:
The operation does not have any preconditions.
Postconditions:

The operation does not have any postconditions.

Implementation:

Calculates the difference by subtracting the hours and minutes of \texttt{self} from those of \texttt{t}, multiplying the difference in hours by 60 to get the difference in minutes.

Code:

\begin{verbatim}
1       — returns the difference between the given Time and self
2       — in minutes. Only positive if the given Time is later
3    getDifference( t: Time) : Integer =
4         (( t.hours - self.hours) * 60 + (t.minutes - self.minutes))
\end{verbatim}

4.8 \texttt{Time::getNextDepartureTime()}

Creates a new \texttt{Time}-object that is a default staying length later than \texttt{self}. The default length is set to 2 minutes. Used to automatically create a route without knowing all the times.

Note: This operation does not account for the hours change to 00 when crossing midnight. This was noticed too late to change it.

Parameters:

The operation does not have any parameters.

Return value:

The operation returns a \texttt{Time}-object.

Preconditions:

\texttt{timeDefined} \texttt{self} must have a defined time, i.e. its \texttt{hours} and \texttt{minutes} must be defined.

Postconditions:

The operation does not have any postconditions.

Implementation:

Creates a new \texttt{Time}-object that is 2 minutes later than \texttt{self}. If \texttt{minutes} are 58 or higher, this means that the \texttt{hours} are increased by 1 and the \texttt{minutes} decreased by 58.
4.9 Time::getStageEndTime()

Creates a new Time-object that is a default driving length later than self. The default length is set to 30 minutes. Used to automatically create a route without knowing all the times.

Note: This operation does not account for the hours change to 00 when crossing midnight. This was noticed too late to change it.

Parameters:

The operation does not have any parameters.

Return value:

The operation returns a Time-object.

Preconditions:

\[\text{timeDefined} \quad \text{self must have a defined time, i.e. its hours and minutes must be defined.}\]

Postconditions:

The operation does not have any postconditions.

Implementation:

Creates a new Time-object that is 30 minutes later than self. If minutes are 30 or higher, this means that the hours are increased by 1 and the minutes decreased by 30.

Code:

```
1−−returns a default new departure time from a station with self
2−−as the arrival time at that station. Default staying time in
3−−a station is set at 2 minutes.
4getNextDepartureTime() : Time
5begin
6    declare newTime : Time;
7    newTime := new Time();
8    if (self.minutes < 58) then
9        newTime.init(self.hours, self.minutes + 2)
10        newTime.init(self.hours + 1, self.minutes - 58)
11    end;
12    result := newTime
13end
15    pre timeDefined: hours.isDefined() and minutes.isDefined()
```
4. Operations

Code:

1  --- returns a default ending time for a stage with self as the
2  --- starting time. Default stage length is 30 minutes.
3  getStageEndTime() : Time
4  begin
5     declare newTime : Time;
6     newTime := new Time();
7     if (self.minutes < 30) then
8         newTime.init(self.hours, self.minutes + 30)
9     else
10        newTime.init(self.hours + 1, self.minutes - 30)
11     end;
12     result := newTime
13  end
14  pre timeDefined: hours.isDefined() and minutes.isDefined()

4.10 Platform::init()

Initializes a Platform by assigning it a number and a TrainStation.

Parameters:

pNumber (Integer)  Gives the number of the Platform.
ts (TrainStation)  Gives the TrainStation in which the Platform is located.

Return value:

The operation has no return value.

Preconditions:

freshInstance self must be a fresh instance, i.e. its number must be undefined and it must not be associated with a TrainStation.

numberPositive The given pNumber must be positive.

stationDefined The given TrainStation must be defined.

platformNumberNotTaken The given TrainStation must not have a Platform with the same number as self.

Postconditions:

numberAssigned The given number must be assigned correctly.

platformAssigned self must be assigned to the given TrainStation.

Implementation:

Assigns the given pNumber to number and inserts an association between self and ts into PlatformInStation.
4.11 Platform::isAvailable()

Checks whether a Platform is available at a given Time.

Parameters:

\[ t (\text{Time}) \] The Time at which the Platform needs to be available.

Return value:

The operation returns a Boolean: True if self is free at the given Time and False otherwise.

Preconditions:

\[ \text{timeDefined} \] The time for which to check the availability needs to be defined.

Postconditions:

The operation has no postconditions.

Implementation:

A Platform is available at a Time if there is no Train currently on it (arrived with a previous Stage and didn’t depart) and no Train was on it 5 minutes prior or arrives on it until at least 5 minutes later. All arriving trains must therefore arrive at least 5 minutes after \( t \), which can be checked using \( \text{Time::getDifference} \) which only returns a positive value if the given Time is later then the one on which the operation is called, or depart again at least 5 minutes before \( t \), which can be checked using \( \text{Time::getDifference} \) again. A corresponding departing stage for an arriving stage is defined as a stage that uses the same train and departs after the arriving stage has arrived.
4. Operations

Code:

```java
/* checks whether a platform is available at a given time */
/* (no trains currently on that platform or arriving/departing */
/* within 5 minutes) */
isAvailable(t : Time) : Boolean =
  self.arrivingStage -> forall
    (aS: Stage |
     t.getDifference(aS.arrivalTime) > 5 or
     self.departingStage -> exists
       (dS: Stage |
        dS.route.train = aS.route.train and
        dS.departureTime.isLater(aS.arrivalTime) and
        (t.getDifference(dS.departureTime) < -5)
       )
    )
  pre timeDefined: t.isDefined()
```

4.12 TrackSection::init()

Initializes a TrackSection by assigning it two TrainStations as the two train stations this section connects.

Parameters:

- `endPoint1` (TrainStation) One end point of the TrackSection.
- `endPoint2` (TrainStation) The other end point of the TrackSection.

Return value:

The operation has no return value.

Preconditions:

- `freshInstance` self must be a fresh instance, i.e. it must not have any end points assigned to it yet.
- `endPointsDefined` The given `endPoint1` and `endPoint2` must be defined.

Postconditions:

- `typeAssigned` The given type must be assigned correctly.

Implementation:

Assigns the given `pType` to `type`.

Code:

```java
init(endPoint1: TrainStation, endPoint2: TrainStation)
begin
  insert(self, endPoint1) into EndPoints;
  insert(self, endPoint2) into EndPoints;
end
pre freshInstance: self.trainStation -> size() = 0
pre endPointsDefined: endPoint1.isDefined() and
  endPoint2.isDefined()
post sectionConnectedToStations: self.trainStation -> exists
  (s1, s2 |
   s1=endPoint1 and
   s2=endPoint2)
```
4.13 Route::init()

Initializes a Route by assigning it a Driver, a Conductor, a Train and a first Stage.

Parameters:

- **pDriver** (Driver): The driver of the train for this Route.
- **pConductor** (Conductor): The conductor of the train for this Route.
- **pTrain** (Train): The train to be assigned to this Route.
- **pFirstStage** (Stage): The first Stage of this Route.

Return value:

The operation has no return value.

Preconditions:

- **freshInstance**: self must be a fresh instance, i.e. it must not have any driver, conductor, train or stage.
- **driverDefined**: The given pDriver must be defined.
- **conductorDefined**: The given pConductor must be defined.
- **trainDefined**: The given pTrain must be defined.
- **stageDefined**: The given pFirstStage must be defined.

Postconditions:

- **driverAssigned**: The given Driver must be assigned correctly.
- **conductorAssigned**: The given Conductor must be assigned correctly.
- **trainAssigned**: The given Train must be assigned correctly.
- **firstStageAssigned**: The given first Stage must be assigned correctly.

Implementation:

The assignToRoute()-operations of Driver, Conductor and Train are used to assign the driver, conductor and train to self. Then pFirstStage is added to self by inserting the corresponding association into StagesForRoute. This is enough since self has no previous Stages, so it can only be the first Stage in the Route. Route::addStage() cannot be used since it requires the Route to have at least one Stage already.

Code:

```c
1    init (pDriver: Driver, pConductor: Conductor,
2           pTrain: Train, pFirstStage: Stage)
3    begin
4       pDriver.assignToRoute(self);
5       pConductor.assignToRoute(self);
6       pTrain.assignToRoute(self);
7       insert (pFirstStage, self) into StagesForRoute;
8    end
9    pre driverDefined: pDriver.isDefined()
10   pre conductorDefined: pConductor.isDefined()
11   pre trainDefined: pTrain.isDefined()
12   pre stageDefined: pFirstStage.isDefined()
13   pre freshInstance: self.driver.isUndefined() and
14       self.conductor.isUndefined() and
```
4 Operations

self.train.isUndefined() and
self.stage -> size() = 0
post driverAssigned: self.driver = pDriver
post conductorAssigned: self.conductor = pConductor
post trainAssigned: self.train = pTrain
post firstStageAssigned: self.stage->at(1) = pFirstStage

4.14 Route::addStage()

Adds a given Stage to the end of a Route.

Parameters:

pStage (Stage) A Stage to be added to self.

Return value:

The operation has no return value.

Preconditions:

stageDefined The given pStage must be defined.
stageComplete The given pStage must be complete, i.e. all its components must be defined.
stageStartEqualsPreviousEnd The given pStage must depart at the same Platform the currently last Stage arrives at. This also requires the Route to have at least one Stage already.
stageNotUsed The given pStage must not be used in a different Route because that would imply two trains sharing the same Platform at the same time.

Postconditions:

stageAdded The given pStage must now be the last Stage in self

Implementation:

An association between the given pStage and self is inserted into StagesForRoute. Since StagesForRoute is ordered, the added pStage is automatically the last Stage in self.

Code:

addStage(pStage: Stage)
begin
    insert(pStage, self) into StagesForRoute
end
pre stageDefined: pStage.isDefined()
pre stageComplete: pStage.departureTime.isDefined() and
    pStage.arrivalTime.isDefined() and
    pStage.origin.isDefined() and
    pStage.destination.isDefined() and
    pStage.trackSection.isDefined()
pre stageStartEqualsPreviousEnd:
    self.stage->at(1).destination = pStage.origin
4.15  Route::removeStage()

Removes a given Stage from a Route.

Parameters:

pStage (Stage) The Stage to be removed from self.

Return value:
The operation has no return value.

Preconditions:

stageDefined The given pStage must be defined.

stageRemovable The given pStage must be the first or last stage of self. Removing any other stage would result in the train arriving at a different platform than the one the next stage departs from.

Postconditions:

stageRemoved The given pStage must not be in self’s list of stages anymore.

Implementation:

Deletes the association between pStage and self from StagesForRoute.

Code:

1   removeStage(pStage: Stage)
2       begin
3       delete (pStage, self) from StagesForRoute;
4       end
5   pre stageDefined: pStage.isDefined()
6   — stages may only be removed if they are the first or last
7   — stage of the route so that the route will still be
8   — completeable
9   pre stageRemovable: self.stage -> last = pStage or
10      self.stage -> first = pStage
11   post stageRemoved: not(self.stage -> includes(pStage))

4.16  Route::overlaps()

Checks if a Route and a given Route have overlapping time frames.

Parameters:

r (Route) A Route for which to check if its time frame between departure and arrival overlaps with that of self.
Return value:
The operation returns a Boolean: True if self and r overlap, False otherwise.

Preconditions:
The operation has no preconditions.

Postconditions:
The operation has no postconditions.

Implementation:
A temporal overlap exists if the interval between departure time of the first Stage and arrival time of the last Stage in each Route is not completely disjunct. Those intervals are completely disjunct only if one Route’s departure time is after the other Route’s arrival time. Thus, this is checked for both possible orders and negated afterwards.

Code:

```java
1    -- checks if the time frames of the two given Route objects
2    -- overlap
3  overlaps( r: Route ) : Boolean =
4      not(
5        ( self.stage->first.departureTime.isLater
6          ( r.stage->last.arrivalTime)) or
7        ( r.stage->first.departureTime.isLater
8          ( self.stage->last.arrivalTime))
9  )
```

4.17 Route::getAvailableTrain()

Select a Train that could be used for this Route, i.e. a Train that is not assigned to a different Route in the time frame needed for self. Should only be used once all Stages needed for self are already added to it.

Parameters:
The operation does not have any parameters.

Return value:
The operation returns a Train that is available for this Route.

Preconditions:

hasStages self must have at least 1 Stage, so that a time frame for the Route can be discerned.

Postconditions:

foundAvailableTrain An available Train must be found, because this operation is used within Conductor::createRoute and no found Train would lead to errors later on.
**Implementation:**

Selects a `Train` from all `Train` instances that is not assigned to any `Route` which overlaps in time to `self`.

**Code:**

```java
getAvailableTrain() : Train =
Train.allInstances -> any
   (t: Train | t.route->forall
      (r: Route | not r.overlaps(self)))
pre hasStages: self.stage -> size > 0
post foundAvailableTrain: result.isDefined()
```

4.18 `Route::getAvailableDriver()`

Select a `Driver` that could be used for this `Route`, i.e. a `Driver` that is not assigned to a different `Route` in the time frame needed for `self`. Should only be used once all `Stages` needed for `self` are already added to it.

**Parameters:**

The operation does not have any parameters.

**Return value:**

The operation returns a `Driver` that is available for this `Route`.

**Preconditions:**

`hasStages` `self` must have at least 1 `Stage`, so that a time frame for the `Route` can be discerned.

**Postconditions:**

`foundAvailableDriver` An available `Driver` must be found, because this operation is used within `Conductor::createRoute` and no found `Driver` would lead to errors later on.

**Implementation:**

Selects a `Driver` from all `Driver` instances that is not assigned to any `Route` which overlaps in time to `self`.

**Code:**

```java
getAvailableDriver() : Driver =
Driver.allInstances -> any
   (d: Driver | d.route->forall
      (r: Route | not r.overlaps(self)))
pre hasStages: self.stage -> size > 0
post foundAvailableDriver: result.isDefined()
```

4.19 `Route::getAvailableConductor()`

Select a `Conductor` that could be used for this `Route`, i.e. a `Conductor` that is not assigned to a different `Route` in the time frame needed for `self`. Should only be used once all `Stages` needed for `self` are already added to it.
4. Operations

Parameters:
The operation does not have any parameters.

Return value:
The operation returns a Conductor that is available for this Route.

Preconditions:

hasStages self must have at least 1 Stage, so that a time frame for the Route can be discerned.

Postconditions:

foundAvailableConductor An available Conductor must be found, because this operation is used within Conductor::createRoute and no found Conductor would lead to errors later on.

Implementation:

Selects a Conductor from all Conductor instances that is not assigned to any Route which overlaps in time to self.

Code:

```
1  --returns a Conductor that is available for this Route
2  getAvailableConductor() : Conductor =
3      Conductor.allInstances -> any
4       (c: Conductor | c.route->forall
5          (r: Route | not r.overlaps(self))
6        )
7  pre hasStages: self.stage -> size > 0
8  post foundAvailableConductor: result.isDefined()
```

4.20 Stage::init()

Initializes a Stage by assigning it a departure and arrival Time, a departure and destination Platform and a TrackSection to use between the departure and arrival platform.

Parameters:

- pDepartureTime (Time) The Time at which this stage departs from its origin.
- pArrivalTime (Time) The Time at which this stage arrives at its destination.
- pOrigin (Platform) The Platform from which this stage departs.
- pDestination (Platform) The Platform at which this stage arrives.
- pTrackSection (TrackSection) The TrackSection this stage uses.

Return value:
The operation has no return value.
4.20. Stage::init()

Preconditions:

freshInstance

self must be a fresh instance, i.e. its departureTime, arrivalTime, origin, destination and trackSection must be undefined.

timesDefined

The given pDepartureTime and pArrivalTime must be defined.

platformsDefined

The given pOrigin and pDestination must be defined.

trackDefined

The given pTrackSection must be defined.

trackConnectsOriginAndDestination

The given pTrackSection must connect the TrainStations in which pOrigin and pDestination are located.

The preconditions for the times and platforms being defined are not split up further because the parameters for an operation being defined is a very trivial condition.

Postconditions:

departureTimeAssigned

The given pDepartureTime must be assigned correctly.

arrivalTimeAssigned

The given pArrivalTime must be assigned correctly.

originAssigned

The given pOrigin must be assigned correctly.

destinationAssigned

The given pDestination must be assigned correctly.

trackSectionAssigned

The given pTrackSection must be assigned correctly.

Implementation:

Inserts associations between self and the given parameters into Departure, Arrival, OriginOfStage, DestinationOfStage and TrackForStage.

Code:

1  // A stage needs an existing arrival- and departure-time
2  // as well as an existing origin- and destination-platform
3  // and an existing TrackSection
4  init (pDepartureTime: Time, pArrivalTime: Time,
5      pOrigin: Platform, pDestination: Platform,
6      pTrackSection: TrackSection)
7  begin
8      insert (pDepartureTime, self) into Departure;
9      insert (pArrivalTime, self) into Arrival;
10     insert (pOrigin, self) into OriginOfStage;
11     insert (pDestination, self) into DestinationOfStage;
12     insert (pTrackSection, self) into TrackForStage
13  end
14  pre freshInstance: departureTime.isUndefined() and
15     arrivalTime.isUndefined() and
16     origin.isUndefined() and
17     destination.isUndefined() and
18     trackSection.isUndefined()
19  pre timesDefined: pDepartureTime.isDefined() and
20     pArrivalTime.isDefined()
21  pre platformsDefined: pOrigin.isDefined() and
22     pDestination.isDefined()
4.21 Stage::temporallyOverlaps()

Checks if a Stage and a given Stage have overlapping time frames.

Parameters:

\( s : \text{Stage} \) A Stage for which to check if its time frame between departure and arrival overlaps with that of self.

Return value:

The operation returns a Boolean: True if self and s overlap, False otherwise.

Preconditions:

The operation has no preconditions.

Postconditions:

The operation has no postconditions.

Implementation:

The implementation checks for a temporal overlap in the same way as Route::overlaps().

Code:

```java
−− checks if two given Stage objects overlap temporally
temporallyOverlaps( s : Stage ) : Boolean =
not(
    ( self . departureTime . isLater ( s . arrivalTime ) ) or
    ( s . departureTime . isLater ( self . arrivalTime ) )
)
```

4.22 Stage::getAvailableTrackSection()

Returns a TrackSection that can be used for a Stage.

Parameters:

The operation has no parameters.
Return value:
The operation returns a TrackSection that connects origin and destination and is not yet used in the time frame of self.

Preconditions:
- timesDefined: The departure and arrival times of self must be defined to check for the availability in that time frame.
- stationsDefined: origin and destination of self must be defined to filter for matching TrackSections.

Postconditions:
- foundAvailableTrack: An available TrackSection must be found, because this operation is used within Conductor::createRoute and no found TrackSection would lead to errors later on.

Implementation:
Selects any TrackSection from all TrackSection-instances that connects origin and destination and is not yet used in the time frame needed for self.

Code:

```cpp
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
getAvailableTrackSection() : TrackSection
begin
    declare track : TrackSection;
    track := TrackSection.allInstances -> any
        (ts: TrackSection | (ts.stage -> forall (s: Stage |
            not(s.temporallyOverlaps(self)) ) and
            ts.trainStation ->
            includes(self.origin.trainStation) and
            ts.trainStation ->
            includes(self.destination.trainStation)
        );
    result := track;
end
pre timesDefined: self.departureTime.isDefined() and
    self.arrivalTime.isDefined()
pre stationsDefined: self.origin.isDefined() and
    self.destination.isDefined()
post foundAvailableTrack: result.isDefined()
```

4.23 Driver::init()

Initializes a Driver by assigning it a name.
Parameters:

\( pName \) (String) The name for the driver.

Return value:

The operation has no return value.

Preconditions:

- freshInstance self must be a fresh instance, i.e. it must not have a name yet.
- nameNotEmpty The given \( pName \) must contains at least one character.

Postconditions:

- nameIsInitialized The given \( pName \) must be assigned correctly.

Implementation:

Assigns the given \( pName \) to \( name \).

Code:

```object
1 init (pName: String)
2 begin
3 self.name := pName
4 end
5 pre freshInstance: name.undefined()
6 pre nameNotEmpty: pName.size > 0
7 post nameIsInitialized: self.name = pName
```

4.24 Driver::assignToRoute()

Assigns a Driver to a given Route by creating a corresponding DriverOfRoute-association. If the Route already has an assigned Driver, that association is deleted.

Parameters:

\( r \) (Route) The route to which \( self \) shall be assigned.

Return value:

The operation has no return value.

Preconditions:

- routeDefined The given Route must be defined.

Postconditions:

- isAssigned \( self \) must be the driver for the given Route.

A postcondition to check whether the association to a previously assigned Driver has been deleted is not necessary, since the number of assignable Drivers is limited to 1 in DriverOfRoute.
4.25 Conductor::init()

Initializes a Conductor by assigning it a name.

Parameters:

pName (String) The name for the conductor.

Return value:

The operation has no return value.

Preconditions:

freshInstance self must be a fresh instance, i.e. it must not have a name yet.
nameNotEmpty The given pName must contains at least one character.

Postconditions:

nameIsInitialized The given pName must be assigned correctly.

Implementation:

Assigns the given pName to name.

Code:

```c
init(pName: String)
begin
    self.name := pName
end
pre freshInstance: name.isUndefined()
pre nameNotEmpty: pName.size > 0
post nameIsInitialized: self.name = pName
```

4.26 Conductor::assignToRoute()

Assigns a Conductor to a given Route by creating a corresponding ConductorOfRoute-association. If the Route already has an assigned Conductor, that association is deleted.
4. Operations

Parameters:

\[ r \ (\text{Route}) \] The route to which \textit{self} shall be assigned.

Return value:

The operation has no return value.

Preconditions:

\[ \text{routeDefined} \] The given \textit{Route} must be defined.

Postconditions:

\[ \text{isAssigned} \] \textit{self} must be the conductor for the given \textit{Route}.

A postcondition to check whether the association to a previously assigned \textit{Conductor} has been deleted is not necessary, since the number of assignable \textit{Conductors} is limited to 1 in \textit{ConductorOfRoute}.

Implementation:

Deletes the association between \( r \) and its currently assigned \textit{Conductor}, if it already has an assigned \textit{Conductor}, then creates an association between \textit{self} and the given \textit{Route} in \textit{ConductorOfRoute}.

Code:

```
1  -- assigns this conductor to the given route
2  assignToRoute (r : Route) begin
3      if (r.conductor.isDefined()) then
4          delete (r.conductor, r) from ConductorOfRoute;
5      end;
6      insert (self, r) into ConductorOfRoute
7  end
8
9  pre routeDefined: r.isDefined()
10  post isAssigned: r.conductor = self
```

4.27 Conductor::createRoute()

Creates a \textit{Route} with a given starting \textit{TrainStation}, a given starting \textit{Time} and a sequence of following \textit{TrainStations}. The \textit{Platforms} for each \textit{Stage} are chosen depending on which \textit{Platforms} are available, the \textit{Time} interval for each \textit{Stage} is set to 30 minutes. The staying time in each \textit{TrainStation} is set to 2 minutes. The \textit{Train}, \textit{Driver} and \textit{Conductor} are assigned based on which ones are available for the created \textit{Route}.

Parameters:

\[ \text{startingStation} \ (\text{TrainStation}) \] The \textit{TrainStation} from which the \textit{Route} departs.

\[ \text{stations} \ (\text{Sequence(TrainStation)}) \] The \textit{TrainStations} that are serviced with this \textit{Route} (in order).

\[ \text{startTime} \ (\text{Time}) \] The starting \textit{Time} for this \textit{Route}. 
Return value:
The operation returns a newly created Route.

Preconditions:
- `startingStationDefined` The first TrainStation needs to be defined.
- `startTimeDefined` The given startTime must be defined.
- `enoughStations` There must be at least one more TrainStation in addition to the startingStation so that the created Route can have at least one Stage.

Postconditions:
- `driverAssigned` The created Route must have an assigned Driver.
- `conductorAssigned` The created Route must have an assigned Conductor.
- `trainAssigned` The created Route must have an assigned Train.
- `allStagesAdded` For every TrainStation in stations there must be a Stage in the newly created Route.
- `correctDepartingTime` The created Route must depart at the specified startTime

Note that to correctly define the operation, the stages in the created route would have to be checked for connecting the correct cities. This was noticed too late and thus could not be added as a postcondition anymore.

Implementation:
A new Route (newRoute) is created first, as well as a new Stage (currentStage) that departs from the startingStation. The startingStation and startTime are associated to the newly created Stage. Then, for each TrainStation in stations, an available Platform is searched using TrainStation::getAvailablePlatform(), the arrival Time is generated using Time::getStageEndTime() and a TrackSection is searched using Stage::getAvailableTrackSection(). The resulting arrival time, destination platform and used track section are associated to the currentStage.
After that, the departure time of the next Stage is generated using Time::getNextDepartureTime(). The next origin platform is the same as the previous destination. The next Stage becomes the new currentStage and the previous steps are repeated until the last TrainStation is reached. The last created currentStage is destroyed (along with its created associations) because it does not have a destination and should not be part of the Route.
Once all Stages are added, a Driver, Conductor and Train are searched using Route::getAvailableDriver(), Route::getAvailableConductor() and Route::getAvailableTrain() and then assigned to newRoute using Driver::assignToRoute(), Conductor::assignToRoute() and Train::assignToRoute(), respectively. The resulting Route is returned.

Note that if there are not enough resources (drivers, conductors, trains, available platforms etc.), the operation will fail midway due to violating postconditions of utility operations or operating on returned null. This is not a nice solution but it works and is intended to work like this for simplicity’s sake.

Code:

1. --- create a route using a list of train stations and a
2. --- start time. The time for each stage is set to 30 minutes.
3. --- To keep the code relatively simple, the departure time
4. --- is the same as the previous arrival time.
createRoute(startingStation : TrainStation ,
    stations : Sequence(TrainStation) ,
    startTime : Time) : Route
begin
    declare newRoute: Route ,
        currentStage : Stage ,
        currentTime : Time ;
    newRoute := new Route ();
    currentStage := new Stage ();
    insert(startTime , currentStage) into Departure ;
    insert(startingStation . getAvailablePlatform(startTime) ,
        currentStage) into OriginOfStage ;
    for station in stations do
        currentTime :=
            station . getAvailablePlatform(currentTime) ,
            currentStage) into DestinationOfStage ;
        insert(currentTime , currentStage) into Arrival ;
        insert(currentStage . getAvailableTrackSection () ,
            currentStage) into TrackForStage ;
        if(newRoute . stage -> size () = 0) then
            insert(currentStage , newRoute) into StagesForRoute ;
        else newRoute . addStage(currentStage) ;
        end ;
        currentStage := new Stage () ;
        insert(newRoute . stage -> last . destination ,
            currentStage) into OriginOfStage ;
        currentTime := currentTime . getNextDepartureTime () ;
        insert(currentTime , currentStage) into Departure ;
end ;
— remove last 'currentStage' and its associations
— as well as last 'currentTime'
destroy currentStage ;
destroy currentTime ;
    newRoute . getAvailableDriver (). assignToRoute(newRoute) ;
    newRoute . getAvailableConductor (). assignToRoute(newRoute) ;
    newRoute . getAvailableTrain (). assignToRoute(newRoute) ;
    result := newRoute ;
end
— there need to be at least 2 stations in a route
pre startingStationDefined : startingStation . isDefined ()
pre startTimeDefined : startTime . isDefined ()
pre enoughStations : stations -> size () > 0
post driverAssigned : result . driver . isDefined ()
post conductorAssigned : result . conductor . isDefined ()
post trainAssigned : result . train . isDefined ()
post allStagesAdded : result . stage -> size () =
    stations -> size ()
post correctDepartingTime : result . stage ->
    first . departureTime = startTime
5. Scenarios

In this chapter, various test cases checking the correctness and completeness of our operations and invariants will be presented. For each test case, we will start by giving a verbal explanation of the test case and its purpose. What will be following are the command sequences and the corresponding object diagrams as well as screenshots, for example showing violated constraints. In the first part of the chapter, our invariants are evaluated. Both negative (invariants violated) and positive scenarios are discussed. Generally, we refrain from providing screenshots of the constraint evaluation for positive test cases.

5.1 Invariants

Author: Merlin Burri

The command sequence for every test case can be found in the code/tests folder. For many invariants, the initial test case is a simple state that does not violate any constraints. This configuration is used multiple times and is constructed as follows: There is one Route object consisting of one stage, i.e. it is associated with a single Stage object. Associated with the route there is one Train, one Driver and one Conductor object. The stage is associated with two Platform objects, with each of them being associated with a TrainStation (bremen being the origin and rotenburg being the destination). The stage’s departure and arrival times are defined by a Time object, with the attribute values equating to 12:15 and 13:15 respectively. The command sequence (initial_state.cmd) used to create the initial state can be found next.

1  −− Bremen
2    !create bremen : TrainStation
3    !set bremen.name := 'Bremen Hauptbahnhof'
4
5  −− Rotenburg (Wümme)
6    !create rotenburg : TrainStation
7    !set rotenburg.name := 'Rotenburg (Wümme)'
8
9  −− Bremen
10    !create b1 : Platform
11    !set b1.number := 1
12
13  −− Rotenburg (Wümme)
14    !create r1 : Platform
15    !set r1.number := 2
Route 1 Bremen–Rotenburg

create br1 : Route

Bremen to Rotenburg (Wümme)

create brStage : Stage
create brRail1 : TrackSection

Train

create train1 : Train
set train1.type := 'ICE'

Employees

create driver1 : Driver
set driver1.name := 'John Lok'
create conductor1 : Conductor
set conductor1.name := 'Thomas'

Times

create departure : Time
set departure.hours := 12
set departure.minutes := 15

create arrival : Time
set arrival.hours := 13
set arrival.minutes := 15

Associations

PlatformInStation

insert (b1,bremen) into PlatformInStation
insert (r1,rotenburg) into PlatformInStation

Employee Associations

insert (driver1,br1) into DriverOfRoute
insert (conductor1,br1) into ConductorOfRoute

TrainForRoute

insert (train1,br1) into TrainForRoute

Stages StagesForRoute

insert (brStage,br1) into StagesForRoute

TrackForStage

insert (brRail1,brStage) into TrackForStage

OriginOfStage

insert (b1, brStage) into OriginOfStage
5.1. Invariants

5.1.1 Train, Driver and Conductor

Since all our invariants for the classes Train, Driver and Conductor simply forbid the simultaneous use of resources, we construct test cases to check these three invariants at once.

Test case 01 – TC_01_resources_NotUsedSimultaneously_0

This test case is the first positive scenario for all NotUsedSimultaneously invariants. Starting from the initial state, we add a second route br_route2 and assign the existing driver driver1, conductor conductor1 and train train1 to that second route. All three resources are now assigned to multiple routes. By setting the departure and arrival time of the new route (or more precisely, the new stage associated with the new route) to 14:35 and 15:35 respectively, we do not create temporal overlap between the two routes. Therefore, no NotUsedSimultaneously invariant is violated. Figure 5.1 shows the object diagram for this test case.

1 open initial_state.cmd
2
3 — create new platforms for new route/stage and associate them with
4 — train stations
5 !create r2 : Platform
6 !set r2.number := 2
7 !insert (r2, rotenburg) into PlatformInStation
8
9 !create b2 : Platform
10 !set b2.number := 2
11 !insert (b2, bremen) into PlatformInStation
12
13 — create route 2 Rotenburg – Bremen
14 !create br_route2 : Route
15
16 — create stage from Rotenburg to Bremen
17 !create brStage : Stage
18 !insert (brStage, br_route2) into StagesForRoute
19
20 — create arrival and departure time
21 — no temporal overlap with the first route
22 !create departure2 : Time
23 !set departure2.hours := 14
24 !set departure2.minutes := 35
Test case 02 – TC_02_ressources_NotUsedSimultaneously_0

For our second positive test case for the `NotUsedSimultaneously` invariants, we again start from the initial state. Like in test case 02, a second route from Bremen to Rotenburg is added. This time however, we set the departure and arrival time to 12:35 and 13:35.
5.1. Invariants

respectively, causing temporal overlap between the two routes. We introduce new Driver, Conductor and Train objects and assign them to that second route. Since no single resource is assigned to multiple routes with temporal overlap, no NotUsedSimultaneously is violated. Figure 5.2 shows the object diagram for this test case.

```plaintext
open initial_state.cmd

— create new platforms for new stage and associate them with
— train stations
!create r2 : Platform
!set r2.number := 2
!insert (r2, rotenburg) into PlatformInStation

!create b2 : Platform
!set b2.number := 2
!insert (b2, bremen) into PlatformInStation

— create route 2 Rotenburg – Bremen
!create br_route2_tempOverlap : Route

— create stage from Rotenburg to Bremen
!create rbStage : Stage
!insert (rbStage, br_route2_tempOverlap) into StagesForRoute

— create arrival and departure time
— temporal overlap with the first route
!create departure2 : Time
!set departure2.hours := 12
!set departure2.minutes := 35

!create arrival2 : Time
!set arrival2.hours := 13
!set arrival2.minutes := 35

— associate new stage with track section, platforms and times
!insert (brRail1, rbStage) into TrackForStage
!insert (b2, rbStage) into OriginOfStage
!insert (r2, rbStage) into DestinationOfStage
!insert (departure2, rbStage) into Departure
!insert (arrival2, rbStage) into Arrival

— different train, conductor and driver for the new, overlapping route
!create driver2 : Driver
!set driver2.name := 'John Lok II'
!create conductor2 : Conductor
!set conductor2.name := 'Thomas II'
!create train2 : Train
!set train2.type := 'RE'

!insert (driver2, br_route2_tempOverlap) into DriverOfRoute
!insert (conductor2, br_route2_tempOverlap) into ConductorOfRoute
!insert (train2, br_route2_tempOverlap) into TrainForRoute
```
Test case 03 – TC_03_ressources_NotUsedSimultaneously_1

We now introduce a negative test case for the NotUsedSimultaneously invariants. In accordance to test case 02, we start with the initial state and add a route that overlaps in time with the first route included in the initial state. In contrast to test case 02, we do not add new objects but assign the existing Driver, Conductor and Train objects to this new overlapping route. All three NotUsedSimultaneously are therefore violated, which is shown in figure 5.4. The object diagram is shown in figure 5.3.

```plaintext
1  open initial_state.cmd
2
3  −− create new platforms for new stage and associate them with
4  −− train stations
5  !create r2 : Platform
6  !set r2.number := 2
7  !insert (r2, rotenburg) into PlatformInStation
8
9  !create b2 : Platform
10  !set b2.number := 2
11  !insert (b2, bremen) into PlatformInStation
12
13  −− create route 2 Rotenburg – Bremen
14  !create br_route2_tempOverlap : Route
15
16  −− create stage from Rotenburg to Bremen
17  !create rbStage : Stage
18  !insert (rbStage, br_route2_tempOverlap) into StagesForRoute
```
5.1. Invariants

Figure 5.3: Object diagram for test case 03 – NotUsedSimultaneously constraints violated

— create arrival and departure time
21 — temporal overlap with the first route
22 !create departure2 : Time
23 !set departure2.hours := 12
24 !set departure2.minutes := 35
25
26 !create arrival2 : Time
27 !set arrival2.hours := 13
28 !set arrival2.minutes := 35
29
— associate new stage with track section, platforms and times
31 !insert (brRail1, rbStage) into TrackForStage
32 !insert (b2, rbStage) into OriginOfStage
33 !insert (r2, rbStage) into DestinationOfStage
34 !insert (departure2, rbStage) into Departure
35 !insert (arrival2, rbStage) into Arrival
36
— violate all 'NotUsedSimultaneously' constraints
38 !insert (driver1, br_route2_tempOverlap) into DriverOfRoute
39 !insert (conductor1, br_route2_tempOverlap) into ConductorOfRoute
40 !insert (train1, br_route2_tempOverlap) into TrainForRoute
5.1.2 Route

To show the correctness of our Route class constraints, we introduce an extension of our initial state. It is very similar to the previously presented initial state, with one major addition: The route is associated with two stages, so Rotenburg is now an intermediate stop and the last stop is Hamburg.

Test case 04 – TC_04_route_0

This initial state also serves as the positive scenario for all three Route class constraints. As mentioned, we use two stages for the route, brStage being the first and rhStage the second. Platform r1 of Rotenburg is both the destination of the first stage and the second stage’s origin. Therefore, DeparturePlatformPreviousPlatform is not violated. Also, the first stage’s arrival time is 13:15 and the second stage’s departure time is 13:16, so the train departs after it has arrived and not before. Thus, DepartureAfterArrivalPreviousStage is also satisfied. As the route also contains no circles, NoCircles is not violated either. Figure 5.5 illustrates the described valid system state.

```latex
1 open initial_state.cmd
2 
3 — Hamburg
4 ! create hamburg : TrainStation
5 ! set hamburg.name := 'Hamburg Hauptbahnhof'
6 
7 ! create h1 : Platform
8 ! set h1.number := 1
9 
10 — Rotenburg (Wümme) to Hamburg
11 ! create rhStage : Stage
12 ! create rhRail1 : TrackSection
13 
14 —
15 — Associations
16 —
17 
18 — PlatformInStation
19 ! insert (h1,hamburg) into PlatformInStation
20 
21 — Stages StagesForRoute
22 ! insert (rhStage,br1) into StagesForRoute
```

Figure 5.4: Violated constraints for test case 03
5.1. Invariants

![Diagram](image)

Figure 5.5: Object diagram for test case 04 – no violations

---

```
23 — TrackForStage
24 !insert (rhRail1, rhStage) into TrackForStage
25 — DestinationOfStage
26 !insert (h1, rhStage) into DestinationOfStage
27 — EndPoints
28 !insert (rhRail1, rotenburg) into EndPoints
29 !insert (rhRail1, hamburg) into EndPoints
30 — Arrival time in hamburg (not relevant for this test case)
31 !create hamburgArrival : Time
32 !set hamburgArrival.hours := 13
33 !set hamburgArrival.minutes := 40
34 !insert (hamburgArrival, rhStage) into Arrival
35 — not violating "DeparturePlatformPreviousPlatform"
36 !insert (r1, rhStage) into OriginOfStage
37 — —
38 — not violating "DepartureAfterArrivalPreviousStage"
39 — Departure time after arrival time of previous Stage
40 —
41 !create rotDeparture : Time
42 !set rotDeparture.hours := 13
43 !set rotDeparture.minutes := 16
44 !insert (rotDeparture, rhStage) into Departure
```

Test case 05 – TC_05_route_DepartureAfterArrivalPreviousStage_1

In our negative scenario for the DepartureAfterArrivalPreviousStage invariant, we simply set the arrival time of the first stage to 13:16. If we consider the departure time of the next stage, which is 13:16 as well, the constraint has been violated. The resulting object diagram can be seen in figure 5.6 and the resulting constraint evaluation window in
Test case 06 – TC_06_route_DeparturePlatformPreviousPlatform_1

The negative scenario for the DeparturePlatformPreviousPlatform works similarly. Instead of changing the arrival time of the first stage like in test case 05, we instead change the arriving platform. To achieve that, we introduce a new platform r2 associated with Rotenburg and associate the destination platform of the stage connecting Bremen and Rotenburg with that same platform. Since the next stage in the route, Rotenburg to Hamburg, departs from r1 and not r2, the constraint is violated. The resulting object diagram is displayed in figure 5.8 and the violated constraints in figure 5.9.

```plaintext
 1 open TC_04_route_0.cmd
 2
 3 −− create new platform for Rotenburg station
 4 !create r2 : Platform
 5 !set r2.number := 2
```
Test case 07 – TC_07_route_NoCircles_1

There is no need to explicitly provide a positive scenario for the NoCircles invariant, since basically every previous test case is just that. Test case 17 provides a positive example that is a bit more advanced, because there are multiple stages assigned to one route. To now provide a negative test case, we take the system state introduced by test case 04 and remove TrainStation hamburg, Platform h1 and TrackSection rhRail1. We then assign rhStage to brRail1 and b1 as destination. We then add a stage equal the brStage with different arrival and departure times. Now, br1 first goes from bremen to rotenburg, then back and afterwards again to rotenburg. We have a circle and the constraint is thus violated, which is shown in figure 5.11. Figure 5.10 shows the object diagram.

open TC_04_route_0.cmd
3 — destroy Hamburg and track section
4 !destroy hamburg
5 !destroy h1
6 !destroy rhRail1
7
8 — make rhStage go from Rotenburg to Bremen
9 !insert (brRail1, rhStage) into TrackForStage
10 !insert (b1, rhStage) into DestinationOfStage
11
12 — add new stage equal to brStage with different times
13 !create brStage2 : Stage
14
15 !create departure2 : Time
16 !set departure2.hours := 13
17 !set departure2.minutes := 45
18
19 !create arrival2 : Time
20 !set arrival2.hours := 14
21 !set arrival2.minutes := 45
22
23 !insert (brStage2, br1) into StagesForRoute
24 !insert (brRail1, brStage2) into TrackForStage
25 !insert (b1, brStage2) into OriginOfStage
26 !insert (r1, brStage2) into DestinationOfStage
27 !insert (departure2, brStage2) into Departure
28 !insert (arrival2, brStage2) into Arrival

Figure 5.10: Object diagram for test case 07 – NoCircles violated
5.1.3 Stage

Positive test cases for the **ArrivalAfterDeparture** invariant of the **Stage** class are implicitly given by all previous test cases, which contain (multiple) **Stage** objects with assigned arrival times being after the departure time and **ArrivalAfterDeparture** not being violated.

**Test case 08 – TC_08Stage_ArrivalAfterDeparture_1**

To create a negative test case for the **ArrivalAfterDeparture** invariant, we simply take the initial state introduced in the introduction of section 5.1 and change the arrival time of the only existing stage to 12:14. Since the stage’s departure time is set to 12:15, **ArrivalAfterDeparture** is violated. Figure 5.12 shows the resulting object diagram and figure 5.13 the violated constraints. We prefer to present an otherwise valid system state as opposed to a minimal system state that for example only contains a stage and time objects, so only the constraint, which we want to present a negative scenario for, is actually violated.

```plaintext
1 open initial_state.cmd
2
3   -- change arrival time of Stage to 12:14
4   -- since departure time is 12:15
5   -- ArrivalAfterDeparture is violated
6   !set arrival.hours := 12
7   !set arrival.minutes := 14
```

**Test case 09 – TC_09Stage_TrackSectionConnectOriginDestination_1**

Again, we do not provide a specific positive test case for the **TrackSectionConnectOriginDestination** invariant. All previous test cases show examples of system states rightfully and evidently not violating the constraint. For instance, in the initial state, we can see that the **TrackSection brRail1** assigned to the single stage **brStage** is associated with the two train stations that the origin and destination platform of **brStage** are associated with, resulting in no violation of **TrackSectionConnectOriginDestination**.

To provide a negative example, we take the initial state specified in the introduction of the section 5.1 and add a new **TrainStation hamburg** and associate **brRail1** to that train station instead of **bremen**. Thus, the track section assigned to **brStage** is no longer connecting the two train stations assigned to the origin and destination platform. In figure 5.14, the resulting system state is illustrated and figure 5.15 shows the violated constraints.

```plaintext
1 open initial_state.cmd
```
Figure 5.12: Object diagram for test case 08 – **ArrivalAfterDeparture** violated

Figure 5.13: Violated constraints for test case 08
2
3 — add new train station hamburg and change
4 — assignment of track section brRail1 to hamburg
5 — thus violating the constraint
6 ! create hamburg : TrainStation
7 ! delete (brRail1, bremen) from EndPoints
8 ! insert (brRail1, hamburg) into EndPoints

Test case 10 – TC_10_stage_NoOverlapsOppositeDirections_0

In the first positive test case for NoOverlapsOppositeDirections, we want to show that two stages using the same track section without any temporal overlap does not violate our constraint, even if the trains go in opposite directions. We therefore take the system state introduced for test case 02 and change the departure and arrival time of the our Stage rbStage to 13:35 and 14:35 respectively, since we do not want temporal overlap. To then make the two trains go in opposite directions, we switch Platform objects assigned to our Stage rbStage, causing r2 to be the origin and b2 the destination platform. Since there is no temporal overlap, NoOverlapsOppositeDirections is not violated. The resulting object diagram is shown in figure 5.16.
Figure 5.15: Violated constraints for test case 09

1 open TC_02_ressources_NotUsedSimultaneously_0.cmd
2
3 — change times so there is no temporal overlap
4 !set departure2.hours := 13
5 !set departure2.minutes := 35
6
7 !set arrival2.hours := 14
8 !set arrival2.minutes := 35
9
10 — switch destination and platform assignments
11 — of rbStage
12 !delete (b2, rbStage) from OriginOfStage
13 !delete (r2, rbStage) from DestinationOfStage
14
15 !insert (r2, rbStage) into OriginOfStage
16 !insert (b2, rbStage) into DestinationOfStage

Test case 11 – TC_11_stage_NoOverlapsOppositeDirections_0

For the second positive test case, we want to show that the constraint is not violated if two trains use differing track sections at the same time, while driving in opposite directions. We start with the system state introduced for test case 02 and again swap the origin and destination platform for Stage rbStage. We create a new TrackSection brRail2 and, after associating it with the two existing train stations, assign it to rbStage. Now, there is temporal overlap between our two stages and the trains go in opposite directions. Since different track sections are used, NoOverlapsOppositeDirections is not violated. Figure 5.17 shows the resulting object diagram.

1 open TC_02_ressources_NotUsedSimultaneously_0.cmd
2
3 — switch destination and platform assignments
4 — of rbStage
5 !delete (b2, rbStage) from OriginOfStage
6 !delete (r2, rbStage) from DestinationOfStage
7
8 !insert (r2, rbStage) into OriginOfStage
9 !insert (b2, rbStage) into DestinationOfStage
10
11 — introduce new track section between
12 — Bremen and Rotenburg
Test case 12 – TC_12_stage_NoOverlapsOppositeDirections_1

To provide a negative test case for NoOverlapsOppositeDirections, we construct a system state similar to the one introduced for test case 11. This time, we do not create a new track section. Thus, we have two trains using the same track section going in opposite directions while overlapping in time and NoOverlapsOppositeDirections is violated, which is shown in figure 5.19. The resulting object diagram is shown in figure 5.18.

13  !create brRail2 : TrackSection
14  !insert (brRail2, bremen) into EndPoints
15  !insert (brRail2, rotenburg) into EndPoints
16
17  -- assign new track section to rbStage
18  !delete (brRail1, rbStage) from TrackForStage
19  !insert (brRail2, rbStage) into TrackForStage

Test case 12 – TC_12_stage_NoOverlapsOppositeDirections_1

To provide a negative test case for NoOverlapsOppositeDirections, we construct a system state similar to the one introduced for test case 11. This time, we do not create a new track section. Thus, we have two trains using the same track section going in opposite directions while overlapping in time and NoOverlapsOppositeDirections is violated, which is shown in figure 5.19. The resulting object diagram is shown in figure 5.18.

1 open TC_02_ressources_NotUsedSimultaneously_0.cmd
2
3  -- switch destination and platform assignments
4  -- of rbStage causing trains going in opposite directions
5  -- violating NoOverlapsSameDirection due to temporal overlap
6  !delete (b2, rbStage) from OriginOfStage
7  !delete (r2, rbStage) from DestinationOfStage
8
9  !insert (r2, rbStage) into OriginOfStage
10  !insert (b2, rbStage) into DestinationOfStage
5. Scenarios

Figure 5.17: Object diagram for test case 11 – no violations

Figure 5.18: Object diagram for test case 12 – NoOverlapsOppositeDirections violated
A positive test case for the `TimeDifferenceSameDirection` invariant is given by test case 02. It shows that when two trains do in fact use the same track section while going in the same direction and overlapping in time, the constraint is not violated if the difference in departure and arrival time does exceed 10 minutes respectively. The first train is set to depart at 12:15 and arrive at 13:15, while the second train departs at 12:35 and arrives at 13:35. For both stages, the `TrackSection brRail1` is used. The difference is of course 20 minutes for both the arrival and departure times so the constraint is not violated.

**Test case 13 – TC_13_stage_TimeDifferenceSameDirection_1**

With our first negative test case for `TimeDifferenceSameDirection`, we want to make sure that the constraint is violated if one train overtakes the other while using the same track section, i.e. if one train arrives before the other, even though it departs later. The initial state is given by the system state used in test case 02. The departure time of the `Stage brStage` is then set to 12:50. Now, both the respective arrival and the departure times are still more than 10 minutes apart, but since the departure of `brStage` is earlier than the departure of `rbStage`, the same has to hold true for the arrival times. As that is not the case, the constraint is violated. Figure 5.20 shows the violated constraints. We refrain from providing an object diagram for this test case, since it basically equals the one shown in figure 5.2 with one small difference in the `minutes` attribute in the `Time` object `departure`.

```
1 open TC_02_ressources_NotUsedSimultaneously_0.cmd
2
3 — setting departure of first stage to 12:50
4 — thus violating the constraint, as the train
5 — arrives before the other one which is departing earlier
6 ! set departure.minutes := 50
```

**Test case 14 – TC_14_stage_TimeDifferenceSameDirection_1**

The second negative test case for `TimeDifferenceSameDirection` ensures that the constraint is violated, if the differences in arrival and departure times do not exceed 10 minutes. We again use the system state presented in test case 2 and manipulate the departure and arrival time for `Stage brStage`. The departure time is set to 12:30 and the arrival time to 13:30. Since the difference is now 5 minutes each, the constraint is violated, which is shown in figure 5.21. As in the previous test case, the resulting object diagram is not provided due to the differences being only marginal.

```
1 open TC_02_ressources_NotUsedSimultaneously_0.cmd
2
```
5. Scenarios

Test case 15 – TC_15_stage_TimeDifferenceSameDirection_1

For our next negative test case for the TimeDifferenceSameDirection invariant, we want to show that the constraint is also violated if only the difference in the departure times of trains using the same track section while going in the same direction does not exceed 10 minutes. Corresponding to test case 14, we set the departure time of Stage brStage to 12:30 without manipulating the arrival time. The constraint is violated. We again refrain from providing the resulting object diagram. The violated constraints are shown in figure 5.22.

Test case 16 – TC_16_stage_TimeDifferenceSameDirection_1

Corresponding to test case 15, with our last negative test case for TimeDifferenceSameDirection we want to show that the constraint is violated if only the difference in the arrival time of trains using the same track section does not exceed 10 minutes. This
5.1. Invariants

Figure 5.22: Violated constraints for test case 15

Figure 5.23: Violated constraints for test case 16

time, we manipulate the arrival time of \textit{brStage} to 13:30, thus violating the constraint which is shown in figure 5.23.

1 open TC\_02\_ressourses\_NotUsedSimultaneously\_0\_cmd

2 — setting arrival of first stage to 13:30 thus violating

3 — the constraint, as the train arrives only 5 minutes earlier

4 — than the other one

5 ! set arrival\_minutes := 30

5.1.4 Platform

To check the correctness of the \texttt{MaxOneTrainPerPlatform} invariant, we need to construct a system state that contains multiple stages arriving at the same platform. We therefore take the system state introduced for test case 2 and set the destination of \texttt{Stage \textit{rbStage}} to \texttt{Platform r1}.

Test case 17 – TC\_17\_platform\_MaxOneTrainPerPlatform\_0

To create a positive scenario for our invariant, we also add a new \texttt{TrainStaiton} object \textit{hamburg} with one associated platform \textit{h1}. We then create a new \texttt{Stage \textit{rhStage}} for our route \textit{br1} going from \textit{rotenburg} to \textit{hamburg}. By setting the departure time of \texttt{rhStage} to 13:30, we make sure that the train departs from platform \textit{r1} before the train assigned to route \texttt{br\_route2\_tempOverlap} arrives. Thus, no constraint is violated. The object diagram can be seen in figure 5.24.

1 open TC\_02\_ressourses\_NotUsedSimultaneously\_0\_cmd
3 — set destination platform of rbStage to
4 — r1 and remove r2
5 !destroy r2
6 !insert (r1, rbStage) into DestinationOfStage
7
8 — add Hamburg train station and platform/track section
9 !create hamburg : TrainStation
10 !set hamburg.name := 'Hamburg Hauptbahnhof'
11
12 !create h1 : Platform
13 !set h1.number := 1
14 !insert (h1, hamburg) into PlatformInStation
15
16 — add stage between Rotenburg and Hamburg
17 !create rhStage : Stage
18 !insert (rhStage, br1) into StagesForRoute
19 !create rhRail1 : TrackSection
20 !insert (rhRail1, rotenburg) into EndPoints
21 !insert (rhRail1, hamburg) into EndPoints
22 !insert (rhRail1, rhStage) into TrackForStage
23 !insert (h1, rhStage) into DestinationOfStage
24 !insert (r1, rhStage) into OriginOfStage
25
26 — setting departure from platform r1 to 13:30
27 !create rotDeparture : Time
28 !set rotDeparture.hours := 13
29 !set rotDeparture.minutes := 30
30 !insert (rotDeparture, rhStage) into Departure
31
32 !create hamburgArrival : Time
33 !set hamburgArrival.hours := 13
34 !set hamburgArrival.minutes := 50
35 !insert (hamburgArrival, rhStage) into Arrival

Test case 18 – TC_18_platform_MaxOneTrainPerPlatform_1

For our first negative test case, we want to show that if one train does not depart from a platform at all and another one arrives, the constraint is violated. We take the previous system state and simply refrain from adding a second stage to our route br1. As a result we have two trains arriving at platform r1 without one of them departing at all. The first train is blocking the platform and consequently, the constraint is violated, which is shown in figure 5.26. The resulting object diagram can be found in figure 5.25.

open TC_02_ressources_NotUsedSimultaneously_0.cmd

Test case 19 – TC_19_platform_MaxOneTrainPerPlatform_1
Figure 5.24: Object diagram for test case 17 – no violations

Figure 5.25: Object diagram for test case 18 – MaxOneTrainPerPlatform violated
To provide another negative test case, we use the system state introduced for test case 17. This time around, we set the departure time of \textit{rhStage} to 13:36, which causes the train of the first route to block the platform \textit{r1} when the train of the second route is set to arrive at 13:35. The constraint is thus violated. We again refrain from providing the resulting object diagram since it equals the one shown in figure 5.24, with one difference in the value for the \textit{minutes} attribute in the \textit{Time} object \textit{rotDeparture}. The violated constraints are shown in figure 5.27.

5.1.5 Time

Most of the previous test cases can serve as positive test cases that show that our \textit{InInterval} invariants are not violated when the \textit{minutes} and \textit{hours} values are within the correct interval. In the following, we therefore only explicitly present negative test cases.

\textbf{Test case 20 – TC\_20\_time\_HoursInInterval\_1}

The used system state for our negative scenario for the \textit{HoursInInterval} invariant consists one single \textit{Time} object. The value for the \textit{hours} attribute is set to 24. The value being not in the interval from 0 to 23, \textit{HoursInInterval} is violated, which is shown in figure 5.29. The object diagram is shown in figure 5.28.

```
1 !create hoursTooHigh : Time
2 !set hoursTooHigh.hours := 24
3 !set hoursTooHigh.minutes := 59
```
5.2. Operations

Test case 21 – TC_21_time_MinutesInInterval_1

For the MinutesInInterval invariant, we provide a negative scenario by constructing a system state consisting of one Time object and setting the value for the minutes attribute to -2. Since the value is not in the interval from 0 to 59, the constraint is violated. The violated constraints are shown in 5.31 and the object diagram can be found in figure 5.30.

1 ! create minutesTooLow : Time
2 ! set minutesTooLow. hours := 13
3 ! set minutesTooLow. minutes := -2

Test case 22 – TC_22_init_0

Since all classes have an init()-operation that can be used to easily assign attributes or associations, the first test checks whether all those operations work properly with valid inputs. For this, new objects of each class are created and initialized using their respective
Figure 5.32: Object diagram for test case 22

`init()`-operations. The object diagram 5.32 shows that all attributes are set correctly and all associations have been created correctly.

1. open initial_state.cmd
2. — tests `init()`-operations for all classes
3. — create objects to be initialized
4. — names contain 'new' so that they can be easily found in object diagram
5. ! create `newHamburgArrival`: Time
6. ! create `newHamburg`: TrainStation
7. ! create `newHamburg1`: Platform
8. ! create `newRotHamTrack`: TrackSection
9. ! create `newDriver`: Driver
10. ! create `newConductor`: Conductor
11. ! create `newTrain`: Train
12. ! create `newRotHamStage`: Stage
13. ! create `newRotHamRoute`: Route
14. ! create `newHamburgArrival`. `init` (13, 55)
15. ! create `newHamburg`. `init` (‘Hamburg Hbf’)  
16. ! create `newHamburg1`. `init` (1, `newHamburg`)  
17. ! create `newRotHamTrack`. `init` (rotenburg, `newHamburg`)  
18. ! create `newDriver`. `init` (‘Lukas’)  
19. ! create `newConductor`. `init` (‘Jim Knopf’)  
20. ! create `newTrain`. `init` (‘RE1234’)  
21. — use previous arrival in rotenburg as departure time
22. ! create `newRotHamStage`. `init` (arrival, `newHamburgArrival`, r1, `newHamburg1`, `newRotHamTrack`)  
23. ! create `newRotHamRoute`. `init` (`newDriver`, `newConductor`, `newTrain`, `newRotHamStage`)  

Test case 23 – TC_23_addStage_0

Next, we want to test whether `addStage` works as expected if it receives a valid stage. As can be seen in the object diagram 5.34, the created valid stage `newRotHamStage` has been successfully added to `br1`. 

1. open initial_state.cmd
To check that addStage can also fail to work we create a test case that violates a precondition, specifically stageStartEqualsPreviousEnd. For this, we let the next Stage start on a different platform in Rotenburg than the one it arrived in. In the object diagram 5.36 you can see that the created newRotHamStage is not linked to the route br1, because addStage failed. Consequently, addStage does not show up in the sequence diagram 5.37.
Figure 5.34: Object diagram for test case 23

Figure 5.35: Sequence diagram for test case 23
5.2. Operations

Figure 5.36: Object diagram for test case 24

Figure 5.37: Sequence diagram for test case 24

8 !create newRotHamTrack: TrackSection
9 !create newRotHamStage: Stage
10
11 !newRotenburgDeparture.init(13, 16)
12 !newHamburgArrival.init(13, 55)
13 !newHamburg.init('Hamburg Hbf')
14 !newHamburg1.init(1, newHamburg)
15 !newRot2.init(3, rotenburg)
16 !newRotHamTrack.init(rotenburg, newHamburg)
17 — let stage start in bremen instead of rotenburg
18 !newRotHamStage.init(newRotenburgDeparture, newHamburgArrival, newRot2, newHam1)
19 — add new Stage to br1—Route
20 !br1.addStage(newRotHamStage)
**Test case 25 – TC_25_removeStage_0**

We also want to test if a Stage can be removed using `removeStage` if its conditions are fulfilled. For this, we use Test case 23 and remove `newRotHamStage` from `br1` again. As can be seen in the object diagram 5.38, `newRotHamStage` is no longer associated with `br1`, so the operation works as expected.

```
1 open TC_23_addStage_0.cmd
2
3 ! br1.removeStage(newRotHamStage)
```

**Test case 26 – TC_26_removeStage_1**

To assert that you can’t just remove any stage from a route, we will now try to remove a stage that is in the middle of a route. For that, we add another stage back from Hamburg to Bremen to the state created in test case 23. Then we try to remove the middle stage from Rotenburg to Hamburg. In the object diagram 5.40 you can see that `newRotHamStage`, the stage from Rotenburg to Hamburg, is still associated with `br1`, because the precondition `stageRemovable` does not hold. Consequently, `removeStage()` does not show up in the sequence diagram 5.41.

```
1 open TC_23_addStage_0.cmd
2
3 ! create additional stage back from hamburg to bremenArrival
4 ! create hamburgDeparture: Time
5 ! create bremenArrival: Time
6 ! create bremHamTrack: TrackSection
7 ! create bremHamStage: Stage
8
9 ! hamburgDeparture.init(13, 56)
10 ! bremenArrival.init(14, 55)
11 ! bremHamTrack.init(newHamburg, bremen)
12 ! bremHamStage.init(hamburgDeparture, bremenArrival, newHamburg1, b1, bremHamTrack)
13
14 —add new Stage to br1-Route
```
Test case 27 – TC_27_assignToRoute_0

In Test case 22, assignToRoute() was already tested for a newly created Route, because it is used in Route::init() for assigning the driver, conductor and train. Now we want to check whether assigning a driver, conductor and train also works as expected if the route already has those assigned. For this, a new driver, conductor and train are created and assigned to br1. As can be seen in the object diagram 5.42, the old driver, conductor and train are no longer associated with br1, while the new ones are. The operation has worked as expected.

15  br1.addStage(bremHamStage)
16
17       -- try to remove middle stage: should not be possible
18  br1.removeStage(newRotHamStage)
Test case 28 – TC_28_createRoute_0

Lastly we want to check if creating a route using \texttt{Conductor::createRoute()} works as intended. Because this operation uses so many utility operations this test case also tests \texttt{TrainStation::getAvailablePlatform()}, \texttt{Time::getStageEndTime()}, \texttt{Stage::getAvailableTrackSection()}, \texttt{Time::getNextDepartureTime()} on valid system states.

To create a system state where a new \texttt{Route} can be created automatically, a new \texttt{TrainStation} \texttt{hamburg} along with a \texttt{Platform} and a \texttt{Tracksection} connecting it to \texttt{rotenburg} are created. The new \texttt{Route} shall start in Bremen and end in Hamburg via Rotenburg. Since the only platform in Rotenburg is taken by \texttt{train1}, a new platform is created there as well. Lastly, a new driver, conductor and train for the new route are created. The departure time is set to 18:18, so that it will not collide temporally with the existing route \texttt{br1}.

The object diagram [5.44] shows that a new Route has been created successfully with Stages from Bremen to Rotenburg and Rotenburg to Hamburg. No invariants are violated and no unnecessary objects have been created.

The sequence diagram [5.45] in this case is limited to just the \texttt{Conductor::createRoute()} operation and its operation calls. It shows the non-query operations used in \texttt{createRoute()} in the process of creating the new \texttt{Route}.

1 open initial_state.cmd
2
3 — tests automatically creating a Route
4 — if all necessary resources are available
5
6 !create newRouteDeparture: Time
Figure 5.42: Object diagram for test case 27
Figure 5.43: Sequence diagram for test case 27
7 ! create hamburg: TrainStation
8 ! create hamburg1: Platform
9 ‒ create new Rotenburg platform because r1 is
10 ‒ blocked by train1
11 ! create roten9: Platform
12 ! create rotHamTrack: TrackSection
13 ! create newDriver: Driver
14 ! create newConductor: Conductor
15 ! create newTrain: Train
16
17 ! newRouteDeparture. init (18, 18)
18 ! hamburg. init (’Hamburg Hbf’)
19 ! hamburg1. init (1, hamburg)
20 ! roten9. init (9, rotenburg)
21 ! rotHamTrack. init (rotenburg, hamburg)
22 ! newDriver. init (’Lukas’)
23 ! newConductor. init (’Jim Knopf’)
24 ! newTrain. init (’RE1234’)
25
26 ! newConductor. createRoute(bremen, Sequence{rotenburg, hamburg}, newRouteDeparture)

Test case 29 – TC_29_createRoute_1

We also want to check at least one case where creating a route with Conductor::createRoute() does not work. For this, we take test case 28, but don’t create an additional platform in Rotenburg, so that there is no available platform there. As expected, the operation fails because TrainStation::getAvailablePlatform() returns null. In the sequence diagram 5.47 you can see that the createRoute()-operation does not finish successfully. The object diagram 5.46 only contains the objects created before calling the Conductor::createRoute()-operation, so the system behaves as expected.
Figure 5.45: Sequence diagram for test case 28
Figure 5.46: Object diagram for test case 29

```
    Figure 5.47: Sequence diagram for test case 29
```

```plaintext
9 !create rotHamTrack: TrackSection
10 !create newDriver : Driver
11 !create newConductor: Conductor
12 !create newTrain: Train
13
14 !newRouteDeparture . init (18, 18)
15 !hamburg . init ('Hamburg Hbf')
16 !hamburg1 . init (1, hamburg)
17 !rotHamTrack . init (rotenburg, hamburg)
18 !newDriver . init ('Lukas')
19 !newConductor . init ('Jim Knopf')
20 !newTrain . init ('RE1234')
21
22 !newConductor . createRoute (bremen, Sequence{rotenburg, hamburg}, newRouteDeparture)
```
6. Queries

Author: Merlin Burri

In the following chapter, possible queries, i.e. OCL expressions that can query useful information contained in our model, will be discussed. For each query, we will start by giving a verbal explanation. Afterwards, the query itself will be presented and, if necessary, further explained. Lastly, the query will be evaluated, which mostly equates to stating the result of the query. The code for every query can also be found in the query_code.txt file. In order for the queries to return the expected results, the query_initial.cmd state has to be loaded and, if the corresponding query section demands it, the listed additional commands have to be executed.

To illustrate the queries, we will be using calls with exemplary parameters. Our initial state will be the same for all queries, unless otherwise stated, and is constructed as follows. We will start with the state presented in section 5.1 for test case 16. Another TrainStation object munich with no associations is created. We also introduce another track section between bremen and rotenburg. Furthermore, we set the arrival time of the Stage object rbStage to 13:40. The corresponding command sequence can be found in the following, the resulting object diagram in figure 6.1.

```plaintext
1 open tests/TC_16_platform_MaxOneTrainPerPlatform_0.cmd

3 — add new train station not connected to any other train station
4 !create munich : TrainStation
5 !set munich.name := 'Muenchen'

7 — add new track section between bremen and rotenburg
8 !create brRail2 : TrackSection
9 !insert (brRail2, bremen) into EndPoints
10 !insert (brRail2, rotenburg) into EndPoints

12 — set arrival time of second stage to 13:40
13 !set arrival2.minutes := 40
```

6.1 Ressources

In this section, queries regarding our normal resources, i.e. Train, Driver and Conductor will be discussed.
6.1.1 Workload

The next query allows us to determine the total working time for a resource, for example the total time a driver is assigned to a route. Pauses in between stages are included in that total working time, so long the stages are associated to the same route. To make the evaluation of this query (for Driver) a bit more interesting, starting from the initial query state, we set the departure and arrival time of Stage rbStage to 15:35 and 16:40, respectively, and assign driver1 to the second route br_route2_tempOverlap so he is assigned to multiple routes, while removing driver2 from the same route. The corresponding command sequence is listed next, the resulting object diagram is shown in figure 6.2.

```plaintext
drive 1 set departure2.hours := 15
2 set arrival2.hours := 16
3 delete (driver2, br_route2_tempOverlap) from DriverOfRoute
4 insert (driver1, br_route2_tempOverlap) into DriverOfRoute
```

The only parameter is the Driver object that the information is to be retrieved for. For our example query, we use driver1.

```plaintext
let
2 theDriver : Driver = @driver1
3 in
4 theDriver.route->collect ( r : Route |
5 r.stage->first ().departureTime
6 .getDifference (r.stage->last ().arrivalTime)
7 )->sum()
```

As a result, we obtain the total workload in minutes. For driver1, this amounts to 160 minutes, as the first route br1 starts at 12:15 and ends at 13:50 (95 minutes), while the second route br_route2_tempOverlap does so at 15:35 and 16:40 (65 minutes), respectively.

```plaintext
1 160 : Integer
```
If we query the same information for *driver2*, 0 is returned, since *driver2* is not associated to any route.

1 0 : Integer

Similarly, the same query can of course be constructed for *Constructor* and *Train* objects by simply replacing all occurrences of 'Driver' in the query with 'Conductor' or 'Train', respectively, and passing objects with the corresponding types as the argument.

### 6.1.2 Available resources for route

This type of query allows the extraction of resources that are available for a given route. For all objects of the respective resource type, all assigned routes are checked for temporal overlap with the new given route. If there is no temporal overlap for all assigned routes, the resource is added to the given list. If the given route is already associated to an object of the respective type, that object is also returned.

To check for the availability of *Driver* objects, the query trying to determine available drivers for *Route* *br1* would look as follows:

1 let
2   theRoute : Route = @br1
3 in
4   Driver.allInstances()->select( d : Driver |
5      d.route->forall( r : Route |
6         not r.overlaps(theRoute)
7       )
8       or
9       theRoute.driver.isDefined() and theRoute.driver = d
10 )

Since *driver1* is already assigned to *br1*, it is returned. Driver *driver2* is assigned to route *br_route2_tempOverlap*, which overlaps in time with *br1*. Therefore, *driver2* is not returned:
1 Set{driver1} : Set(Driver)

If we change the departure and arrival time for the stage associated with route \texttt{br\_route2\_temp\_Overlap} so there is no temporal overlap, the query will also return \texttt{driver2}. The corresponding command sequence can be found in the following, the result of the same query used previously after that.

1 ! set departure2.hours := 15
2 ! set arrival2.hours := 16

1 Set{driver1, driver2} : Set(Driver)

Similarly to the query in section 6.1.1, we can easily adapt this query to extract Conductor or Train instead of Driver objects by replacing all occurrences of 'Driver' and 'driver' with 'Conductor' and 'conductor' or 'Train' and 'train', respectively.

If we wanted to sort the returned resource objects by their total working time (lowest to highest), for example to determine the available resource with the lowest current workload, we can sort the output of this query by using the query presented in section 6.1.1:

1 let
2 newRoute : Route = @br1
3 in
4 Driver.allInstances() \rightarrow select (d : Driver |
5 d.route \rightarrow forAll (r : Route |
6 not r.overlaps(newRoute)
7 )
8 ) \rightarrow sortedBy(d | d.route \rightarrow collect (r : Route |
9 r.stage \rightarrow first().departureTime
10 ) \rightarrow getDifference(r.stage \rightarrow last().arrivalTime)
11 ) \rightarrow sum()
12 )

Querying the state previously created, one would get the following result, which makes sense, since \texttt{driver1} is assigned to one route for a total of 95 minutes and \texttt{driver2} to one for a total workload of 65 minutes:

1 Sequence{driver2, driver1} : Sequence(Driver)

6.2 Route

In the following section, we will discuss several queries concerning routes, not in the sense of the object Route, but in a more general one. Most of these queries could for example be used for a railway traffic application where users can look up specific routes.

6.2.1 Stops for route

The first query can be used to extract all stops for a route. All stages assigned to the route are considered and all destination stations are extracted, together with the respective arrival times. The start station of the route is not included. As the single parameter, a route is required. We choose \texttt{br1} for our exemplary query.
6.2. Route

1 let
2   theRoute : Route = @br1
3 in
4   theRoute.stage->collect ( s : Stage |
5       Tuple{
6         stop : s.destination.trainStation ,
7         hours : s.arrivalTime.hours ,
8         minutes : s.arrivalTime.minutes
9       }
10   )

The result of the query is as follows:
1 Sequence{Tuple{stop=rotenburg , hours=13,minutes=15},
2 Tuple{stop=hamburg, hours=13,minutes=50}}
3 : Sequence(Tuple(stop : TrainStation , hours : Integer , minutes : Integer ))

6.2.2 Routes for origin and destination

With our second route specific query, one can determine all routes that go from one given
train station to another. To also account for routes that contain the origin and destination
as intermediate stations, we not only check the first and the last stage of every route, but
every stage. Since we assume that within a route trains never go in circles, we simply
check every route for stages that contain the given origin train station as source and the
given destination as the target train station. If there is one stage that fulfills that criteria,
the assigned route is considered. The same holds true if there are two stages, where one
contains the origin as source and one the destination as target. The input parameters are
two TrainStation objects, the origin and the destination. In our example query, we want
to look up all routes going from bremen and rotenburg.

1 let
2   origin : TrainStation = @bremen,
3   destination : TrainStation = @rotenburg
4 in
5   Route.allInstances()->select( r : Route |
6       r.stage->exists( s : Stage |
7       s.origin.trainStation = origin
8       )
9       and r.stage->exists( s : Stage |
10       s.destination.trainStation = destination
11     )
12  )

The result of the query would be the following:
1 Set{br1, br_route2_tempOverlap} : Set(Route)

6.2.3 Routes for origin and destination with departure and arrival times

Our next query is an extension of the previous one. We now want to retrieve the departure
time at the given origin station (which is not necessarily the first station in the route) and
the arrival time at the given destination station (which is not necessarily the last station
in the route). From the selected routes, next to the route itself we collect the departure
and the arrival time of the stages assigned to the origin and destination, respectively. The
parameters are equal to the one described in section 6.2.2, the same goes for the example parameters. Since we assume circle free routes, when creating the tuple, we can safely select the first element of the selected collection of stages, since it can only contain one single object. If there were multiple stages in one route going to a single train station (or departing from one), there would be a circle.

```java
1 let
2    origin : TrainStation = @bremen,
3    destination : TrainStation = @rotenburg
4 in
5    Route.allInstances() -> select ( r : Route |
6        r.stage -> exists ( s : Stage |
7            s.origin.trainStation = origin
8          )
9    )
10   and
11    r.stage -> exists ( s : Stage |
12        s.destination.trainStation = destination
13    ) -> collect ( r : Route |
14        let
15        departureStage : Stage = r.stage -> select ( s : Stage |
16            s.origin.trainStation = origin
17              ) -> first ()
18        arrivalStage : Stage = r.stage -> select ( s : Stage |
19            s.destination.trainStation = destination
20              ) -> first ()
21        in
22        Tuple{
23            route : r,
24            dHours: departureStage.departureTime.hours,
25            dMinutes: departureStage.departureTime.minutes,
26            aHours: arrivalStage.arrivalTime.hours,
27            aMinutes: arrivalStage.arrivalTime.minutes
28        }
29 )
```

As a result, we obtain the following bag, where \(dHours\) and \(dMinutes\) determine the departure time and \(aHours\) and \(aMinutes\) the arrival time at the specified stations:

```java
1 Bag{ Tuple{route=br1 , dHours=12, dMinutes=15, aHours=13, aMinutes=15},
2 Tuple{route=br_route2_tempOverlap , dHours=12, dMinutes=35, aHours=13,
3 aMinutes=40}}
```

### 6.2.4 Routes for origin, destination, current time

We now introduce a query that additionally takes the current time as a parameter and returns all routes departing later than specified by the given time from the given origin and later arriving at the given destination. We adapt the query presented in section 6.2.3 and finally, sort the returned tuple by the difference of current time and departure time.

As additional parameters, we have the current time specified by two \(\text{Integers}\) \(\text{hours}\) and \(\text{minutes}\), which we initially set to 12:00. We again use \textit{bremen} as the given origin and
rotenburg as the destination. In addition to the routes themselves, we again return the departure and arrival times.

```sql
let
  origin : TrainStation = @bremen,
  destination : TrainStation = @rotenburg,
  hours : Integer = 12,
  minutes : Integer = 00
in
  Route.allInstances() > select ( r : Route |
    r.stage -> exists ( s : Stage |
      s.origin.trainStation = origin
      and
      ( s.departureTime.hours > hours ) or
      ( ( s.departureTime.hours = hours )
        and ( s.departureTime.minutes > minutes ) ) or
      ( s.departureTime.hours = 0 and hours = 23 )
    )
    and
    r.stage -> exists ( s : Stage |
      s.destination.trainStation = destination
    )
  ) -> collect ( r : Route |
    let
      departureStage : Stage = r.stage -> select ( s : Stage |
        s.origin.trainStation = origin
      ) -> first ()
      ,
      arrivalStage : Stage = r.stage -> select ( s : Stage |
        s.destination.trainStation = destination
      ) -> first ()
    in
tuple{
      route : r,
      dHours: departureStage.departureTime.hours,
      dMinutes: departureStage.departureTime.minutes,
      aHours: arrivalStage.arrivalTime.hours,
      aMinutes: arrivalStage.arrivalTime.minutes
    }
  ) -> sortedBy ( t | ( t.dHours - hours ) * 60 + ( t.dMinutes - minutes )

The result is the following sequence. As expected, route *br1* is in front of the second route, since the departure times are 12:15 and 12:35, respectively.

```
6.2.5 Routes for origin, destination, arrival time

Instead of looking for routes that depart after a certain point in time, one might look for trains arriving before a specific time. To create a query that can extract exactly that, we take the query of the previous section 6.2.4 and, as opposed to checking the departure time of the stage in the route that departs from the given origin, now check the arrival time of the stage arriving to the given destination. We sort the extracted routes by the difference between the arrival time and the desired arrival time. Like previously, we also return departure and arrival times for the routes in question.

For the origin and destination, we use *bremen* and *rotenburg*, respectively. We set the desired arrival time to 13:45.

```haskell
let origin : TrainStation = @bremen,
    destination : TrainStation = @rotenburg,
    arrivalHours : Integer = 13,
    arrivalMinutes : Integer = 45
in Route. allInstances() -> select ( r : Route |
    r.stage -> exists ( s : Stage |
        s.origin.trainStation = origin
    )
) and
    r.stage -> exists ( s : Stage |
        s.destination.trainStation = destination
    )
    and
    (arrivalHours > s.arrivalTime.hours) or
    ((arrivalHours = s.arrivalTime.hours) and (arrivalMinutes > s.arrivalTime.minutes)) or
    (arrivalHours = 0 and s.arrivalTime.hours = 23)
) -> collect ( r : Route |
    let
        departureStage : Stage = r.stage -> select ( s : Stage |
            s.origin.trainStation = origin
        ) -> first () ,
        arrivalStage : Stage = r.stage -> select ( s : Stage |
            s.destination.trainStation = destination
        ) -> first ()
    in
        Tuple{
            route : r ,
            dHours : departureStage.departureTime.hours ,
            dMinutes : departureStage.departureTime.minutes ,
            aHours : arrivalStage.arrivalTime.hours ,
            aMinutes : arrivalStage.arrivalTime.minutes
        }
```
6.2. Route

36 )–>sortBy(t | ((arrivalHours − t.aHours) * 60
37 + (arrivalMinutes − t.aMinutes))
38 )

Corresponding to our expectations, the following sequence is returned:

1 Sequence{Tuple{route=br_route2_tempOverlap ,dHours=12,dMinutes=35,
2 aHours=13,aMinutes=40},
3 Tuple{route=br1 ,dHours=12,dMinutes=15,aHours=13,aMinutes=15}}
4 : Sequence(Tuple(route :Route ,dHours: Integer ,dMinutes : Integer ,
5 aHours: Integer ,aMinutes: Integer ))

If we now set the desired arrival time to 13:35, the second route br_route2_tempOverlap
will no longer be included in the returned sequence, since the assigned train arrives at
13:40:

1 Sequence{Tuple{route=br1 ,dHours=12,dMinutes=15,aHours=13,aMinutes=15}}
2 : Sequence(Tuple(route :Route ,dHours: Integer ,dMinutes : Integer ,
3 aHours: Integer ,aMinutes: Integer ))

6.2.6 Routes for origin, destination, current time and train type

Again extending the query presented in section 6.2.4, we add another parameter, the
train type. In a possible scenario, one might only want to retrieve routes with a certain
train type because of the restrictions of his ticket. To filter the train type, we modify the
select-expression used to extract viable routes. As parameters, we therefore have the origin,
the destination, the current time and the train type as a String.

For our example query, we choose bremen as origin, rotenburg as destination, 12:00 as the
current time and 'RE' as the desired train type.

1 let
2 origin : TrainStation = @bremen ,
3 destination : TrainStation = @rotenburg ,
4 hours : Integer = 12 ,
5 minutes : Integer = 00 ,
6 trainType : String = 'RE'
7 in
8 Route.allInstances()–>select ( r : Route |
9 r.train.type = trainType
10 and
11 r.stage->exists ( s : Stage |
12 s.origin.trainStation = origin
13 and
14 (s.departureTime.hours > hours) or
15 ((s.departureTime.hours = hours)
16 and (s.departureTime.minutes > minutes)) or
17 (s.departureTime.hours = 0 and hours = 23)
18 )
19 and
20 r.stage->exists ( s : Stage |
21 s.destination.trainStation = destination
22 )
23 )–>collect ( r : Route |
24 let

Queries

\[
\begin{align*}
\text{departureStage} : \text{Stage} = r.\text{stage} & \rightarrow \text{select} \ (s : \text{Stage} | \ s.\text{origin}.\text{trainStation} = \text{origin} ) \rightarrow \text{first} () , \\
\text{arrivalStage} : \text{Stage} = r.\text{stage} & \rightarrow \text{select} \ (s : \text{Stage} | \ s.\text{destination}.\text{trainStation} = \text{destination} ) \rightarrow \text{first} () \\
\end{align*}
\]

\[
\text{in}
\]

\[
\text{Tuple}\{
\begin{align*}
\text{route} : r , \\
\text{dHours} : \text{departureStage}.\text{departureTime}.\text{hours} , \\
\text{dMinutes} : \text{departureStage}.\text{departureTime}.\text{minutes} , \\
\text{aHours} : \text{arrivalStage}.\text{arrivalTime}.\text{hours} , \\
\text{aMinutes} : \text{arrivalStage}.\text{arrivalTime}.\text{minutes}
\end{align*}
\}
\)

\[
\rightarrow \text{sortedBy} (t | ((t.\text{dHours} - \text{hours}) \times 60 + (t.\text{dMinutes} - \text{minutes}))
\]

In contrast to the first result of the query presented in section 6.2.4, our result sequence does not include \textit{br1}, because the assigned train is of the type ‘ICE’:

\[
\begin{align*}
\text{Sequence}&\{\text{Tuple}\{\text{route}=\text{br}\_\text{route2}\_\text{tempOverlap} , \text{dHours}=12 , \text{dMinutes}=35 , \\
\text{aHours}=13 , \text{aMinutes}=40\}\}
\end{align*}
\]

6.3 Miscellaneous

Author: Marlon Flügge

In the following section we will be presenting a few more miscellaneous queries.

6.3.1 Conductor’s timetable

This query returns the routes and corresponding time intervals a conductor is currently assigned to. It could be used by conductors themselves to check when and where they have to work but also by people planning the routes in order to quickly visualize availability for certain timeslots.

The query takes a parameter \textit{conductorSearch}, which is a search term that is subsequently used to only generate timetables for people of interest. Timetables are created separately for any conductor whose name contains the search term, each timetable represented inside its own \textit{Tuple}. After identifying relevant conductors all the routes for each conductor are collected. For each route an identifier is generated using the names of the origin and destination stations. Additionally a textual representation of the time interval reserved for the specific route is created using the departure and arrival times of the first and last stage respectively. These two strings are bundled inside a \textit{Tuple} and symbolically represent a timeslot inside the conductor’s timetable.

In this example we chose ‘Thomas’ as search term, which returns timeslots for both Thomas and Thomas II.

\[
\begin{align*}
\text{let}
\begin{align*}
\text{conductorSearch} : \text{String} = '\text{Thomas}'
\end{align*}
\)

In the following section we will be presenting a few more miscellaneous queries.
Conductor.allInstances()->select(con : Conductor |
con.name.indexOf(conductorSearch) > 0)
)-->collect(c : Conductor |
let
condRoutes = Route.allInstances()->select(r : Route |
r.conductor = c)
in
Tuple{
  conductor : c.name,
  routes : condRoutes.collect(r : Route |
  let
    origName = r.stage->first().origin.trainStation.name,
    destName = r.stage->last().destination.trainStation.name,
    departure = r.stage->first().departureTime,
    arrival = r.stage->last().arrivalTime
  in
    Tuple{
      routeName : origName.concat(' to ').concat(destName),
      interval : 'From '.concat(departure.hours.toString())
        .concat(':')
        .concat(departure.minutes.toString())
        .concat(' until ')
        .concat(arrival.hours.toString())
        .concat(':')
        .concat(arrival.minutes.toString())
    }
  )
}
)

The result of the query is the following:
Bag{Tuple{conductor='Thomas', routes=Bag{Tuple{routeName=
'Bremen Hauptbahnhof to Hamburg Hauptbahnhof',
interval='From 12:15 until 13:50'}}}, Tuple{conductor='Thomas II',
routes=Bag{Tuple{routeName='Bremen Hauptbahnhof to Rotenburg (Wuerm)',
interval='From 12:35 until 13:40'}}} : Bag(Tuple{conductor:String,
routes:Bag(Tuple{routeName:String, interval:String})})

### 6.3.2 Reachable train stations from train station

For our last query, we want to determine all train stations that are reachable from a given train station. To achieve that, we first go through all train stations directly connected via track sections. We do the same for all these train stations, and so forth, by using the closure operation. All these train stations are added to a list and transformed into a set to remove duplicate entries. The input parameter is a `TrainStation` object and in our example, we want to get all connected train stations for `bremen`.

let theStation : TrainStation = @bremen
in
theStation.trackSection.trainStation->closure(t : TrainStation |
Queries

The result of the query is the following:

Set\{bremen, hamburg, rotenburg\} : Set(TrainStation)

TrainStation munich is not included, since it is not connected to any train station. If we were to call the query on munich, the returned set would be empty. We now add a track section in between munich and hamburg:

create muRail : TrackSection
insert (muRail, munich) into EndPoints
insert (muRail, hamburg) into EndPoints

The query will now return munich as well:
Set\{bremen, hamburg, munich, rotenburg\} : Set(TrainStation)
7. Outlook

Author: Marlon Flügge

In this paper we presented a system to model the scheduling of daily railroad traffic. The system does not model actual railway traffic completely, however. There are a number of ways in which the system overly simplifies the problem because of a limitation in man-hours. General simplifications include:

- 7-day week: Only daily railway traffic is scheduled. In real life, the railway schedule may differ on different days, e.g. on weekends. Going even further, holidays may also impact the schedule, so that even a 7-day week would not be enough.

- TrackSections usable from whole TrainStations: In the model, TrackSections only connect TrainStations, implying that every track laid between two stations can be reached from every platform in each of those stations. In real life, this is usually not the case, limiting the connections to a number of platforms per TrainStation.

More specific simplifications and resulting problems include:

- Stages overlapping: `Stage::temporallyOverlaps()` assumes two Stages to be overlapping when their time intervals are not completely disjunct. This is then used to determine whether a TrackSection is available for a Stage at a given time. If two stages both head in the same direction, a small temporal overlap is not a problem, though, making the system in its current state inefficient in the utilization of available track sections. This could be remedied by introducing additional cases where there is no temporal overlap if two Stages go in the same direction with a minimum difference in departure and arrival times (e.g. 5 minutes).

- Teleporting resources: `Route::getAvailableTrain()` and its driver- and conductor-counterparts do not check for the current location of those resources. A train is considered available in Bremen if it just arrived in Hamburg, as long as it finished its Route and has no other Route planned in the near future. The transportation to the new station as well as the needed time are not considered, leading to potentially practically impossible schedules. This could be fixed by only making those resources available if their last serviced Stage ended in the same station as the one the querying Route departs from. Also, a corresponding invariant should be added.
• Midnight troubles: A train departing at 23:30 and arriving at 00:15 obviously arrives later than it departs, yet its time is lower. This is partially considered in `Time::isLater()`, but not every possible case can be covered. We also forgot to incorporate the midnight changing into `Time::getNextDepartureTime()` and `Time::getStageEndTime()`. Because of a limited amount of operations test cases, there could be more instances where we forgot to account for this that we haven’t noticed yet.

The only 'real' fix for this would be to create a full schedule with a complete calendar, i.e. also including day, month and year, which was not the intended goal of this system.
A. Code

```plaintext
1 model RailwayPlanner
2
3   -- classes
4
5 class Train
6   attributes
7       type : String;
8
9   operations
10      init(pType: String)
11         begin
12          self.type := pType
13         end
14         pre freshInstance: self.type.isUndefined()
15         pre typeNotEmpty: pType.size > 0
16         post typeAssigned: self.type = pType
17
18   -- assigns the train to the given route
19   assignToRoute(r: Route)
20         begin
21           if r.train.isDefined()
22               then
23                 delete (r.train, r) from TrainForRoute;
24                 end;
25               insert (self, r) into TrainForRoute;
26             end
27         pre trainRouteDefined: r.isDefined()
28         post isAssigned: r.train = self
29         end
30
31 class TrainStation
32   attributes
33       name : String;
34
35   operations
```
init(pName: String)
begin
  self.name := pName
end
pre freshInstance: self.name.isUndefined()
pre nameNotEmpty: pName.size > 0
post nameAssigned: self.name = pName

-- returns a platform that is available at the given time
getAvailablePlatform(t: Time): Platform =
  self.platform -> any(p: Platform | p.isAvailable(t))
pre hasPlatforms: self.platform -> size > 0
pre timeDefined: t.isDefined()
end

— only models hours and minutes because this is for scheduled daily traffic
class Time
attributes
  hours: Integer;
  minutes: Integer;

operations
init(pHours: Integer, pMinutes: Integer)
begin
  self.hours := pHours;
  self.minutes := pMinutes;
end
pre freshInstance: self.hours.isUndefined() and
  self.minutes.isUndefined()
pre hoursInCorrectInterval: pHours >= 0 and pHours < 24
pre minutesInCorrectInterval: pMinutes >= 0 and pMinutes < 60
post timeAssigned: self.hours = pHours and
  self.minutes = pMinutes

— checks if the Time the method is called on is
— after the given Time
isLater(t: Time): Boolean =
  (self.hours > t.hours) or
  ((self.hours = t.hours) and (self.minutes > t.minutes)) or
  (self.hours = 0 and t.hours = 23);

— returns the difference between the given Time and self
— in minutes. Only positive if the given Time is later
getDifference(t: Time): Integer =
  ((t.hours - self.hours) * 60 + (t.minutes - self.minutes))

— returns a default new departure time from a station with self
— as the arrival time at that station. Default staying time in
— a station is set at 2 minutes.
getNextDepartureTime(): Time
begin
  declare newTime: Time;
newTime := new Time();
if (self.minutes < 58) then
    newTime.init(self.hours, self.minutes + 2)
else
    newTime.init(self.hours + 1, self.minutes - 58)
end;
result := newTime
end
pre timeDefined: hours.isDefined() and minutes.isDefined()

— returns a default ending time for a stage with self as the
— starting time. Default stage length is 30 minutes.
getStageEndTime() : Time
begin
declare newTime : Time;
newTime := new Time();
if (self.minutes < 30) then
    newTime.init(self.hours, self.minutes + 30)
else
    newTime.init(self.hours + 1, self.minutes - 30)
end;
result := newTime
end
pre timeDefined: hours.isDefined() and minutes.isDefined()

14 class Platform
15 attributes
16    number : Integer;
17 operations
18    — A platform needs an existing trainstation and can’t change
19    — to a different TrainStation.
20    init(pNumber: Integer, ts: TrainStation)
21    begin
22        self.number := pNumber;
23        insert(self, ts) into PlatformInStation
24    end
25    pre freshInstance: self.number.isDefined() and
26        self.trainStation.isDefined()
27    pre numberPositive: pNumber > 0
28    pre stationDefined: ts.isDefined()
29    pre platformNumberNotTaken: not(ts.platform->exists(p | p
30        .number = pNumber))
31    post numberAssigned: self.number = pNumber
32    post platformAssigned: ts.platform->exists(p | p = self)
33
34    — checks whether a platform is available at a given time
35    — (no trains currently on that platform or arriving/departing
36    — within 5 minutes)
37    isAvailable(t: Time) : Boolean =
38        self.arrivingStage -> forAll
39            (aS: Stage |
t.getDifference(aS.arrivalTime) > 5 or
self.departingStage -> exists
  (dS: Stage |
   dS.route.train = aS.route.train and
   dS.departureTime.isLater(aS.arrivalTime) and
   (t.getDifference(dS.departureTime) < -5)
  )
)
pre timeDefined : t.isDefined()
end

class TrackSection
attributes
operations
init (endPoint1 : TrainStation, endPoint2 : TrainStation)
begin
  insert(self, endPoint1) into EndPoints;
  insert(self, endPoint2) into EndPoints;
end
pre freshInstance : self.trainStation -> size() = 0
pre endPointsDefined : endPoint1.isDefined() and
  endPoint2.isDefined()
post sectionConnectedToStations : self.trainStation -> exists
  (s1, s2 |
   s1=endPoint1 and
   s2=endPoint2)
end

class Route
operations
init (pDriver : Driver, pConductor: Conductor,
  pTrain : Train, pFirstStage : Stage)
begin
  pDriver.assignToRoute(self);
  pConductor.assignToRoute(self);
  pTrain.assignToRoute(self);
  insert(pFirstStage, self) into StagesForRoute;
end
pre driverDefined : pDriver.isDefined()
pre conductorDefined : pConductor.isDefined()
pre trainDefined : pTrain.isDefined()
pre stageDefined : pFirstStage.isDefined()
pre freshInstance : self.driver.isUndefined() and
  self.conductor.isUndefined() and
  self.train.isUndefined() and
  self.stage -> size() = 0
post driverAssigned : self.driver = pDriver
post conductorAssigned : self.conductor = pConductor
post trainAssigned : self.train = pTrain
post firstStageAssigned : self.stage->at(1) = pFirstStage
addStage(pStage : Stage)
begin
insert (pStage, self) into StagesForRoute
end

pre stageDefined: pStage.isDefined()
pre stageComplete: pStage.departureTime.isDefined() and
        pStage.arrivalTime.isDefined() and
        pStage.origin.isDefined() and
        pStage.destination.isDefined() and
        pStage.trackSection.isDefined()
pre stageStartEqualsPreviousEnd:
        self.stage->last.destination = pStage.origin
        −− stage should not be part of another route
pre stageNotUsed: Route.allInstances -> forAll
        (r: Route |
        not (r.stage -> includes(pStage))
        )
post stageAdded: self.stage -> last = pStage

removeStage(pStage: Stage)
begin
delete (pStage, self) from StagesForRoute;
end
pre stageDefined: pStage.isDefined()
        −− stages may only be removed if they are the first or last
        −− stage of the route so that the route will still be
        −− completeable
pre stageRemovable: self.stage -> last = pStage or
        self.stage -> first = pStage
post stageRemoved: not (self.stage -> includes(pStage))

−− checks if the time frames of the two given Route objects
−− overlap
overlaps (r: Route): Boolean =
        not (self.stage -> first.departureTime.isLater
        (r.stage -> last.arrivalTime)) or
        (r.stage -> first.departureTime.isLater
        (self.stage -> last.arrivalTime))

−− returns a Train that is available for this Route
getAvailableTrain(): Train =
Train.allInstances -> any
        (t: Train | t.route->forAll
        (r: Route | not r.overlaps(self))
        )
pre hasStages: self.stage -> size > 0
post foundAvailableTrain: result.isDefined()

−− returns a Driver that is available for this Route
getAvailableDriver(): Driver =
Driver.allInstances -> any
(d: Driver | d.route->forAll
(r: Route | not r.overlaps(self))
)
pre hasStages: self.stage -> size > 0
post foundAvailableDriver: result.isDefined()

returns a Conductor that is available for this Route
getAvailableConductor(): Conductor =
  Conductor.allInstances -> any
  (c: Conductor | c.route->forAll
    (r: Route | not r.overlaps(self))
  )
pre hasStages: self.stage -> size > 0
post foundAvailableConductor: result.isDefined()
end

class Stage
operations
  -- A stage needs an existing arrival- and departure-time
  -- as well as an existing origin- and destination-platform
  -- and an existing TrackSection
init(pDepartureTime: Time, pArrivalTime: Time,
    pOrigin: Platform, pDestination: Platform,
    pTrackSection: TrackSection)
begin
  insert(pDepartureTime, self) into Departure;
  insert(pArrivalTime, self) into Arrival;
  insert(pOrigin, self) into OriginOfStage;
  insert(pDestination, self) into DestinationOfStage;
  insert(pTrackSection, self) into TrackForStage
end
pre freshInstance: departureTime.undefined() and
  arrivalTime.undefined() and
  origin.undefined() and
  destination.undefined() and
  trackSection.undefined()
pre timesDefined: pDepartureTime.isDefined() and
  pArrivalTime.isDefined()
pre platformsDefined: pOrigin.isDefined() and
  pDestination.isDefined()
pre trackDefined: pTrackSection.isDefined()
pre trackConnectsOriginAndDestination:
  pTrackSection.trainStation->exists
  (s: TrainStation | s = pDestination.trainStation) and
  pTrackSection.trainStation->exists
  (s: TrainStation | s = pOrigin.trainStation)
post departureTimeAssigned: self.departureTime =
  pDepartureTime
post arrivalTimeAssigned: self.arrivalTime = pArrivalTime
post originAssigned: self.origin = pOrigin
post destinationAssigned: self.destination = pDestination
post trackSectionAssigned: self.trackSection = pTrackSection
— checks if two given Stage objects overlap temporally

temporallyOverlaps ( s : Stage ) : Boolean =
not ( self . departureTime . isLater ( s . arrivalTime )) or
s . departureTime . isLater ( self . arrivalTime ))

— returns a TrackSection that can be used for this stage, if there is any, i.e. a TrackSection that is not yet used in the time frame of this stage and connects origin and destination

getAvailableTrackSection () : TrackSection
begin
declare track : TrackSection ;
track := TrackSection . allInstances -> any ( ts : TrackSection | ts . stage -> forAll ( s : Stage | not ( s . temporallyOverlaps ( self ))
 )
) and
ts . trainStation -> includes ( self . origin . trainStation ) and
ts . trainStation -> includes ( self . destination . trainStation )
);
result := track ;
end
pre timesDefined : self . departureTime . isDefined () and
self . arrivalTime . isDefined ()
pre stationsDefined : self . origin . isDefined () and
self . destination . isDefined ()
post foundAvailableTrack : result . isDefined ()
end

abstract class Employee
attributes
name : String ;
end

class Driver < Employee
operations
init ( pName : String )
begin
    self . name := pName
end
pre freshInstance : name . isUndefined ()
pre nameNotEmpty : pName . size > 0
post nameIsInitialized : self . name = pName
— assigns this driver to the given route
assignToRoute (r : Route)
begin
    if (r.driver.isDefined()) then
        delete (r.driver, r) from DriverOfRoute;
    end;
    insert (self, r) into DriverOfRoute
end
pre routeDefined : r.isDefined()
post isAssigned : r.driver = self

class Conductor < Employee
operations
    init (pName: String)
    begin
        self.name := pName
    end
    pre freshInstance: name.undefined()
    pre nameNotEmpty: pName.size > 0
    post nameIsInitialized: self.name = pName

    assignToRoute (r : Route)
    begin
        if (r.conductor.isDefined()) then
            delete (r.conductor, r) from ConductorOfRoute;
        end;
        insert (self, r) into ConductorOfRoute
    end
    pre routeDefined: r.isDefined()
    post isAssigned: r.conductor = self

createRoute( startingStation : TrainStation ,
    stations : Sequence(TrainStation),
    startTime : Time) : Route
begin
    declare newRoute: Route,
    currentStage: Stage,
    currentTime: Time;
    newRoute := new Route();
    currentStage := new Stage();
    insert (startTime, currentStage) into Departure;
    insert (startingStation.getAvailablePlatform(startTime),
        currentStage) into OriginOfStage;
    for station in stations do
        currentTime :=
            currentStage.departureTime.getStageEndTime();
insert (station.getAvailablePlatform(currentTime),
currentStage) into DestinationOfStage;
insert (currentTime, currentStage) into Arrival;
insert (currentStage.getAvailableTrackSection(),
currentStage) into TrackForStage;
if (newRoute.stage -> size() = 0) then
  insert (currentStage, newRoute) into StagesForRoute;
else newRoute.addStage(currentStage);
end;
currentStage := new Stage();
insert (newRoute.stage -> last.destination,
currentStage) into OriginOfStage;
currentTime := currentTime.getNextDepartureTime();
insert (currentTime, currentStage) into Departure;
end;
−− remove last 'currentStage' and its associations
−− as well as last 'currentTime'
destroy currentStage;
destroy currentTime;
newRoute.getAvailableDriver().assignToRoute(newRoute);
newRoute.getAvailableConductor().assignToRoute(newRoute);
newRoute.getAvailableTrain().assignToRoute(newRoute);
result := newRoute;
end
−− there need to be at least 2 stations in a route
pre startingStationDefined: startingStation.isDefined()
pre startTimeDefined: startTime.isDefined()
pre enoughStations: stations -> size() > 0
post driverAssigned: result.driver.isDefined()
post conductorAssigned: result.conductor.isDefined()
post trainAssigned: result.train.isDefined()
post allStagesAdded: result.stage -> size() =
  stations -> size()
post correctDepartingTime: result.stage ->
  first.departureTime = startTime
end
−− associations
association PlatformInStation between
  Platform[*];
  TrainStation[1];
end
association DriverOfRoute between
  Driver[1];
  Route[*];
end
association ConductorOfRoute between
Conductor [1];
Route [*];
end

association TrainForRoute between
Train [1];
Route [*];
end

association StagesForRoute between
Stage [*] ordered;
Route [1];
end

association TrackForStage between
TrackSection [1];
Stage [*];
end

association OriginOfStage between
Platform [1] role origin;
Stage [*] role departingStage;
end

association DestinationOfStage between
Platform [1] role destination;
Stage [*] role arrivingStage;
end

association Departure between
Time [1] role departureTime;
Stage [*];
end

association Arrival between
Time [1] role arrivalTime;
Stage [*] role routePart;
end

association EndPoints between
TrackSection [*];
TrainStation [2];
end

constraints
—invariants for definedness of attributes
Section: The following Constraints apply to the class Train

Train is not assigned to multiple Routes at the same time

context Train inv TrainNotUsedSimultaneously:
  self.route->forall(r1: Route, r2: Route |
    r1.overlaps(r2) implies r1 = r2
  )

Section: The following Constraints apply to the class Employee and its subclasses (Conductor and Driver)

Driver is not assigned to multiple Routes at the same time

context Driver inv DriverNotUsedSimultaneously:
  self.route->forall(r1: Route, r2: Route |
    r1.overlaps(r2) implies r1 = r2
  )

Conductor is not assigned to multiple Routes at the same time

context Conductor inv ConductorNotUsedSimultaneously:
  self.route->forall(r1: Route, r2: Route |
    r1.overlaps(r2) implies r1 = r2
  )

Section: The following Constraints apply to the class Route

For every Stage in the Route, the Departure Time has to be after the Arrival Time of the previous Stage

context Route inv DepartureAfterArrivalPreviousStage:
  self.stage->forall(s: Stage |
    let currentStageNumber: Integer = stage->indexOf(s)
    in if (currentStageNumber < stage->size()) then
      stage->at(currentStageNumber + 1).departureTime
        .isLater(s.arrivalTime)
      else
        true
      endif
    )

For every Stage in the Route, the Platform that the Train is departing from has to be the platform that the Train arrived on in the previous Stage. This also makes sure that the TrainStation the Train is departing from equals the TrainStation that it arrived on in the previous Stage.

context Route inv DeparturePlatformPreviousPlatform:
  self.stage->forall(s: Stage |
    let currentStageNumber: Integer = stage->indexOf(s)
    in if (currentStageNumber < stage->size()) then
      stage->at(currentStageNumber + 1).departureTime
        .isLater(s.arrivalTime)
      else
        true
      endif
    )
in if (currentStageNumber < stage->size()) then
  s.destination = stage->at(currentStageNumber + 1).origin
else
  true
endif
)

Routes do not contain circles, which equates to every Stage in the Route
having differing source and destination TrainStations
context Route inv NoCircles:
  self.stage->forAll(s1, s2 : Stage |
    (s1.origin.trainStation = s2.origin.trainStation
     or
     s1.destination.trainStation = s2.destination.trainStation)
  implies
  s1 = s2
)

Section: The following Constraints apply to the class Stage

Departure time has to be before arrival time
context Stage inv ArrivalAfterDeparture:
  self.arrivalTime.isLater(self.departureTime)

the used TrackSection has to connect the origin and the
destination of the stage
context Stage inv TrackSectionConnectOriginDestination:
  self.trackSection.trainStation->exists(s : TrainStation |
    s = self.destination.trainStation
  )
  and self.trackSection.trainStation->exists(s : TrainStation |
    s = self.origin.trainStation
  )

No stages using the same sections at overlapping time frames
—going in opposite directions.
—Same used TrackSection and temporal overlap imply same direction
context s1, s2 : Stage inv NoOverlapsOppositeDirections:
  not (s1 = s2) and s1.trackSection = s2.trackSection
  and s1.temporallyOverlaps(s2) implies
  s1.destination.trainStation = s2.destination.trainStation

Same used TrackSection and temporal overlap imply a certain
difference in arrival and departure times
context s1, s2 : Stage inv TimeDifferenceSameDirection:
  not (s1 = s2) and s1.trackSection = s2.trackSection
  and s1.temporallyOverlaps(s2) implies
  if s2.departureTime.isLater(s1.departureTime) then
    s1.departureTime.getDifference(s2.departureTime) > 10 and
    s1.arrivalTime.getDifference(s2.arrivalTime) > 10
if s1.departureTime <= s2.departureTime and s2.arrivalTime <= s1.arrivalTime
   then
      s2.departureTime.getDifference(s1.departureTime) > 10 and
      s2.arrivalTime.getDifference(s1.arrivalTime) > 10
   endif

---

Section: The following Constraints apply to the class TrackSection

---

Section: The following Constraints apply to the class TrainStation

---

Section: The following Constraints apply to the class Platform

---

The next train may only arrive after the previous train has departed
Thus, each platform may host at most one train at a time
context Platform inv MaxOneTrainPerPlatform:

self.arrivingStage->forall(a1, a2 |
   a1 = a2 or
   a2.arrivalTime.isLater(a1.arrivalTime) or a1.arrivalTime
   .isLater(a2.arrivalTime))
   and
   every stopping train needs to depart before the next one arrives
   (a2.arrivalTime.isLater(a1.arrivalTime) implies
   a2.arrivalTime.isLater(a1.route.stage
   ->at((a1.route.stage->indexOf(a1))+1).departureTime))

---

Section: The following Constraints apply to the class Time

---

The value for the minutes attribute has to be in the interval [0, 59]
context Time inv MinutesInInterval:

Time.allInstances->forall( t: Time |
   t.minutes >= 0 and t.minutes < 60
)

---

The value for the hours attribute has to be in the interval [0, 23]
context Time inv HoursInInterval:

Time.allInstances->forall( t: Time |
   t.hours >= 0 and t.hours < 24
)