



The United Nations
University

UNU/IIST

International Institute for
Software Technology

Distributed Train Rescheduling

Chris George

25 April 1995

Draft

UNU/IIST Report No. 42

\mathcal{T}

UNU/IIST

UNU/IIST serves developing countries in attaining self-reliance in software technology: (i) own development of high integrity computing systems, (ii) highest level post-graduate university teaching, (iii) international level research, and, through the above, (iv) use of as sophisticated software as reasonable.

UNU/IIST contributes through: (a) advanced, joint industry university advanced development projects in which rigorous techniques supported by semantics based tools are case-study applied to large scale software developments, (b) own and joint university and academy institute research in which new techniques for application domain and computing platform modeling, requirements capture, software engineering and programming are being investigated, (c) advanced, post-graduate and post-doctoral level courses which typically teach design calculi oriented software development techniques, (d) events [like panels, task forces, workshops and symposia], and (e) dissemination.

Application-wise, the advanced development projects presently focus on software to support large-scale infrastructure systems such as railways, manufacturing industries, health care systems, etc., and are thus aligned with UN and Intl. Aid System concerns. The research projects parallel and support the advanced development projects.

Technically speaking, the focus of UNU/IIST, in all of the above, at present, is on applying, teaching, researching, and disseminating Design Calculi oriented techniques and tools for trustworthy software development. UNU/IIST currently emphasizes techniques that permit proper development steps and interfaces. UNU/IIST endeavours to also promulgate sound project and product management principles.

UNU/IIST's dissemination strategy is to act as a clearing house for reports from primarily industrial country research and technology centres to primarily developing country industry, university and academy centres. At present more than 175 institutions worldwide contribute to UNU/IIST's report collection while UNU/IIST at the same time subscribes to more than 125 international scientific and technical journals. Awareness of reports received (and produced) as well as of journal articles is to be disseminated regularly to developing country centres — who are then free to order a reasonable number of report and article copies from UNU/IIST.

Dines Bjørner, Director

UNU/IIST Reports are either *R*esearch, *T*echnical, *C*ompedia or *A*dministrative reports:

\mathcal{R} Research Report • \mathcal{T} Technical Report • \mathcal{C} Compendium • \mathcal{A} Administrative Report



The United Nations
University

UNU/IIST

**International Institute for
Software Technology**

P.O. Box 3058
Macau

Distributed Train Rescheduling

Chris George

Abstract

This document builds on the basis of previous work, in particular PRaCoSy documents SP/12 and SP/13. It aims to deal with two problems: (i) how to reschedule as well as schedule trains (ii) how to distribute these tasks.

Draft

Chris George (CWG) is a Visiting Senior Research Fellow at UNU/IIST, 1 September 1994 — 31 August 1995, seconded from CRI Inc., a leading Danish IT Systems House. CWG is one of the main contributors to RAISE, CWG got his MA (English, Cambridge University) 1968, Post Graduate Certificate in Education (Bristol University) 1969, BA (Mathematics, Open University) 1980, MSc (Computer Science, Essex University) 1984. Subsequently taught English and Mathematics 1969 – 79, STC Technology Ltd., Harlow, UK 1979 – 90 and CRI Inc., Denmark since 1990 — from where he is on a one year leave of absence. Work assignments included: 1979 – 1982: Developed support environment for production of telephony software, 1983 – 1985: Wrote simulator and other semantic tools for hardware description language, 1985 – 1990: Technical leader on RAISE ESPRIT project, primarily on method, 1982 – 1994: Helped develop and present courses on discrete mathematics, VDM and RAISE, 1990–1994: Project coordinator of LaCoS ESPRIT project.

Contents

1	Introduction	1
2	Changes to previous descriptions	1
2.1	New features	1
2.2	Omitted features	1
2.3	Generalizations	2
3	Timetables to support rescheduling	2
4	Distribution	3
4.1	Distributing a map	4
4.2	Distributing timetables	9
5	Specifications	11
5.1	First specification	11
5.2	Second specification	26
6	Work to be done	34
6.1	Validation of (re)scheduling functions	34
6.2	Analysis functions	34
6.3	Delegable and distributable functions	34
6.4	Concurrent system	34

1 Introduction

This document builds on the basis of previous work, in particular PRaCoSy documents SP/12 and SP/13. It aims to deal with two problems: (i) how to reschedule as well as schedule trains (ii) how to distribute these tasks.

We start in section 2 by noting changes from the previous approach. In section 3 we discuss how to use a generalised notion of timetable to support rescheduling. Section 4 presents an approach to the distribution of timetables and the operations on them. In section 5 we present full specifications at an initial and second level of abstraction (and show that the development from the first to the second is correct). Finally section 6 describes the work still to be done.

2 Changes to previous descriptions

Compared in particular with SP/12, there are some new features, some omitted features and some more generalised approaches.

2.1 New features

We have added the notion of *Relation*. A relation is one of

connection : a condition that a set of trains must all be simultaneously stopped in a station for at least some interval

disconnection : a condition that any stops in a station of trains in a set must be separated by at least some interval

dependency : a condition that one train must arrive and stop at a station at last some interval before another, having stopped, departs

We believe the notion of a dependency is essential to the planning of train movements because it allows the possibility of the movement of passengers, staff, goods or rolling stock between trains. A connection is a generalization of a dependency. Disconnections allow the passengers or goods on different trains to be prevented from mixing.

2.2 Omitted features

Some features have been omitted as details that are either not very interesting or not much developed in previous models. It is considered that they can easily be inserted. In particular we do not include track types or lengths or train lengths, and neither do we include the details of conversion between times in different formats.

2.3 Generalizations

We use times rather than lengths and speeds (which seems more natural). We also distinguish between the minimum time needed for a train to traverse a line, which is a property of the line, and the expected time for a particular train to do so. This effectively allows for the differences in speeds between different kinds of train to be modelled.

The notion of segment has disappeared, and journeys are no longer modeled as sequences of station visits. This was done to avoid imposing possibly over-restrictive conditions relating journeys and dispatch units. Instead a visit may include details of the next station and the previous one (effectively allowing the sequences of stations to be generated).

We use subtypes very rarely. We have predicates describing when a station visit, say, is well-formed, but we do not impose the restriction that a visit must be well-formed. This allows us to store information that requires attention and is the key to a notion of timetable that supports rescheduling, as we shall see in section 3.

Finally we have divided the well-formedness conditions into “dynamic” conditions that are independent of the physical network (and almost all of which relate to times) and the “static” conditions that relate a timetable to a network of lines and stations. This is mainly a matter of convenience, but may also have some affect on the implementation. The network will generally change only rarely, and if the network is unchanged and only certain kinds of change are made to timetables (e.g. changing times but not tracks or lines), static well-formedness is guaranteed to be maintained (and hence need not be checked).

3 Timetables to support rescheduling

PAR/TERMS/09 defines scheduling as “an operation of planning and adjusting some line segments and stations to trains”. It involves resolving access conflicts and respecting train priorities. Rescheduling is not defined there.

To distinguish the two we shall distinguish between the initial construction of a timetable and its subsequent modification during operations. We shall say that creating a timetable and modifying it before the timetable starts to operate is scheduling; modifying a timetable after it has started to operate is rescheduling.

We then note that the aim of rescheduling is to create a timetable that is *feasible*, i.e. allows all the trains required to make their journeys while avoiding any access conflicts and fulfilling any relations and operational rules of the railway (like maximum speeds and minimum times between events). We can take the required set of journeys and relations in a period to be a requirement for a timetable for that period. When we do scheduling there are three possibilities:

1. a feasible timetable is created containing all the train journeys and also conforming to their required arrival and departure times and to the required relations
2. a feasible timetable is created containing all the train journeys but for which some intended arrival and departure times differ from the requirement and/or some relations are not met
3. no feasible timetable can be created containing all the train journeys

If we have 1. then scheduling is complete. If 2. then we may decide to accept the modified times and remove or weaken the unsatisfied relations, or we may decide to change some requirements and reiterate. If 3. then we need to change the requirements and reiterate¹.

In judging whether to accept situation 2. or reiterate we would probably want some measure of the *disruption* of a timetable compared to another (feasible) one. Measuring disruption would involve late (or cancelled) arrivals, early (or cancelled) departures, unfulfilled relations. There would also need to be *negative* disruptions such as extra stops by a train to help compensate for another train's cancellation. That is, disruption must be measured according to the effect on passenger or goods journeys rather than on train journeys.

Scheduling results in a timetable that is feasible. During operation trains may be delayed, or may break down, or even more catastrophic events may occur. The aim of rescheduling is to adapt a timetable so that

1. it is consistent with occurred events
2. it is feasible
3. it has minimum disruption compared with the originally scheduled timetable

It should now be apparent that scheduling and rescheduling are very similar activities.

Note we have assumed that the notion of timetable includes basic timetables, shift plans and stage plans. These only differ in their interpretation of time and in the range of times they cover.

We make a design decision that since we find much of the activities of scheduling and rescheduling are the same we shall use one data structure (and hence as far as possible one set of functions on that data structure) for both activities. This has two consequences:

1. We must be able to record in a timetable whether an event has occurred or not.
2. We must be careful about restricting the information we can record in a timetable. In particular we must not restrict ourselves to feasible timetables. Otherwise we will not be able to record in them the occurrence of events that make them infeasible. This is why, as we mentioned earlier, we generally do not use subtypes in the specification.

4 Distribution

We first present in section 4.1 a general theory of distribution of a map. Then we show how this may be applied to a timetable in section 4.2

¹It does not matter whether we have situation 3. because we can prove the requirements are not achievable or because we cannot find a solution; the effective problem is the same. We might, however, be concerned with providing tools that give us some assurance that if we have 3. then there is probably no possible solution

4.1 Distributing a map

We take the idea of distributing a map to mean partitioning the (possible) domain into a set of disjoint subsets and then forming a submap for each subset. Thus a map is partitioned into disjoint maps.

Suppose our map is of type

type Map = D \xrightarrow{m} R

We can partition the domain type D by means of a total function from D to some “partitioning” type P :

type Dist_Map = P \xrightarrow{m} Map

value

p_of : D \rightarrow P,

distribute : Map \rightarrow Dist_Map

distribute(m) \equiv [p \mapsto m / partition(p) | p : P],

partition : P \rightarrow D-set

partition(p) \equiv { d | d : D • p_of(d) = p }

For example, we can see the telephone directories (maps from names to telephone numbers) for different areas of a country as a distribution of a national directory partitioned by “Area” according to the function “lives in”. For a division into “white” and “yellow” pages the partition type would contain the two elements “private” and “business”.

There is an obvious “inverse” to *distribute*, *merge*:

merge : Dist_Map \rightarrow Map

merge(dm) \equiv [d \mapsto dm(p_of(d))(d) | d : D • p_of(d) \in **dom** dm \wedge d \in **dom** dm(p_of(d))]

and it is then simple to prove the theorem that *merge* is the (left) inverse of *distribute*:

$$\forall m : \text{Map} \bullet \text{merge}(\text{distribute}(m)) = m \quad (1)$$

We now consider functions on a distributed map. Suppose we have some function f that can be applied to a whole map, and propose to apply instead a function df to the distributed map. We can define a notion of df being a “correct” version of f if the disagram in figure 1 “commutes”.

That is, we require that

$$\forall m : \text{Map} \bullet \text{merge}(df(\text{distribute}(m))) = f(m) \quad (2)$$

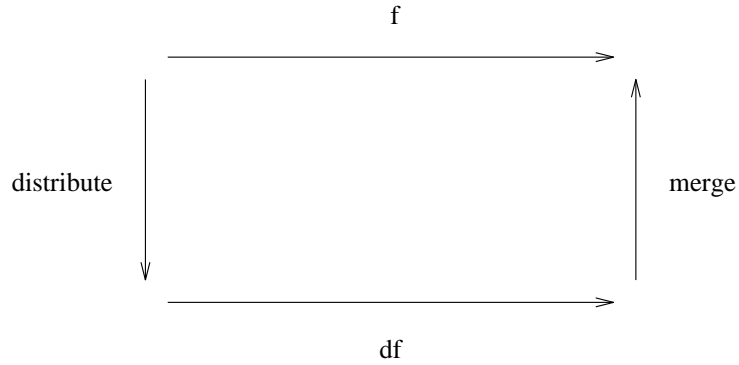


Figure 1: Correctness of distributed function

From (1) we can see that (2) will hold if

$$\forall m : \text{Map} \bullet \text{distribute}(f(m)) = \text{df}(\text{distribute}(m)) \quad (3)$$

and it is condition (3) that we shall employ in the following discussion of “delegable” functions.

A function for which there is no means of identifying a single value of P from its arguments must in general be applied to all components. Consider, for example, prefixing all telephone numbers of people with a digit. But to change the number of a particular subscriber we can apply *p_of* to the subscriber’s name to get the area and then change just the appropriate area directory.

We call a function that can be applied to the (notional) complete map by being applied to just one component “delegable” since it is possible to “delegate” the responsibility for applying it to just one component.

“Lookup” functions will typically be delegable. Such a function will typically have type

$$D \times \text{Map} \rightarrow T$$

where T is the result type of the lookup (and does not involve *Map*). Such a function is delegable if we get the same result when we calculate a P value from the first argument and apply the lookup to the component of the map as we would if we applied the lookup to the (notional) complete map. That is, we can define a test:

$$\begin{aligned} \text{is_delegable_lookup} &: (D \times \text{Map} \rightarrow T) \rightarrow \mathbf{Bool} \\ \text{is_delegable_lookup}(f) &\equiv f(d,m) = f(d, \text{distribute}(m)(\text{p_of}(d))) \end{aligned}$$

We shall return to lookup functions (in a particular general form) later. We first consider functions that change the map. Such a function will typically have a type

type

$$\text{Fun} = D \times \text{Map} \xrightarrow{\sim} \text{Map}$$

and be associated with a precondition which we can express as a function with type

type

$$\text{Pre_Fun} = D \times \text{Map} \rightarrow \mathbf{Bool}$$

We describe such a function as “weakly” delegable if truth of the precondition applied to one component implies

- truth of the precondition for the whole map, and
- distributing and applying the function to one component only gives the same map as applying the function to the whole map and then distributing

The first condition ensures that if we can apply locally we could have applied globally; the second (condition (3) above) says we get the right result. Here are the formal definitions:

$$\begin{aligned} \text{delegate} &: \text{Fun} \rightarrow D \times \text{Dist_Map} \xrightarrow{\sim} \text{Dist_Map} \\ \text{delegate}(f)(d, dm) &\equiv \mathbf{let} \ p = p_of(d) \ \mathbf{in} \ dm \ \dagger \ [p \mapsto f(d, dm(p))] \ \mathbf{end}, \end{aligned}$$

$$\begin{aligned} \text{delegate} &: \text{Pre_Fun} \rightarrow D \times \text{Dist_Map} \rightarrow \mathbf{Bool} \\ \text{delegate}(\text{pre_}f)(d, dm) &\equiv \mathbf{let} \ p = p_of(d) \ \mathbf{in} \ \text{pre_}f(d, dm(p)) \ \mathbf{end}, \end{aligned}$$

$$\begin{aligned} \text{weakly_delegable} &: \text{Fun} \times \text{Pre_Fun} \rightarrow \mathbf{Bool} \\ \text{weakly_delegable}(f, \text{pre_}f) &\equiv \\ &\mathbf{let} \ df = \text{delegate}(f), \ \text{pre_}df = \text{delegate}(\text{pre_}f) \ \mathbf{in} \\ &\quad \forall d : D, m : \text{Map} \bullet \\ &\quad \quad \text{pre_}df(d, \text{distribute}(m)) \Rightarrow \\ &\quad \quad \text{pre_}f(d, m) \wedge df(d, \text{distribute}(m)) = \text{distribute}(f(d, m)) \\ &\mathbf{end} \end{aligned}$$

Such a function is called weakly delegable because it may be that the pre-condition calculated for only one component is stronger than that calculated for the whole map. That is, it might be possible to apply the function to the whole map but not to one component only. If this is not so, i.e. truth of the precondition for the whole map implies truth for the component (and the function is weakly delegable) then we say the function is “strongly” delegable:

$$\begin{aligned} \text{strongly_delegable} &: \text{Fun} \times \text{Pre_Fun} \rightarrow \mathbf{Bool} \\ \text{strongly_delegable}(f, \text{pre_}f) &\equiv \\ &\quad \text{weakly_delegable}(f, \text{pre_}f) \wedge \\ &\quad \mathbf{let} \ \text{pre_}df = \text{delegate}(\text{pre_}f) \ \mathbf{in} \\ &\quad \quad \forall d : D, m : \text{Map} \bullet \text{pre_}f(d, m) \Rightarrow \text{pre_}df(d, \text{distribute}(m)) \\ &\mathbf{end} \end{aligned}$$

It should be apparent that (ignoring preconditions for the present) a function should be delegable if

- the changed relevant component is the same as the extraction of the component after applying the function to the whole map
- all other components are unchanged if extracted after applying the function to the whole map

Remembering that two maps are equal if they have equal domains and give the same results on application to domain values, this allows us to define a function (in the weak case) that expresses this more algorithmic view of the check for a function being delegable:

```

weakly_delegable1 : Fun × Pre_Fun → Bool
weakly_delegable1(f, pre_f) ≡
  (∀ d : D, m : Map • pre_f(d, m) ⇒ (f(d, m) post true)) ∧
  (∀ d : D, m : Map •
    let ds = partition(p_of(d)) in
      pre_f(d, m / ds) ⇒
      pre_f(d, m) ∧
      let (m1, m2) = (f(d, m / ds), f(d, m)) in
        dom m1 = dom m2 ∩ ds ∧
        (∀ d' : D • d' ∈ dom m1 ⇒ m1(d') = m2(d')) ∧
        (∀ p : P • p ≠ p_of(d) ⇒
          dom m ∩ partition(p) = dom m2 ∩ partition(p)
        ) ∧
        (∀ d' : D • d' ∈ dom m ∧ d' ∈ dom m2 ∧ d' ∉ ds ⇒
          m(d') = m2(d'))
    end
  end

```

Having formulated this it is possible to prove the expected theorem

```

∀ f : Fun, pre_f : Pre_Fun •
  weakly_delegable1(f, pre_f) ⇒ weakly_delegable(f, pre_f)

```

This provides us with a more suitable function for checking that a function is weakly delegable (and for strongly delegable we have only an additional relation between preconditions to prove). We can see this theorem as providing a partial decomposition of any proof of a function being delegable.

As mentioned above, lookup functions are generally fairly easy to deal with. Of rather more interest in our case will be functions that “analyse” a map (in particular functions that check for well-formedness).

An analysis function may be one that simply returns **true** or **false**, but of rather more usefulness is one that returns some information, say a set of “messages”. The question is whether we can analyse the components and merge the messages that result into the same set of messages as we would get if we analysed the whole map. The same set of messages implies that when checking each component separately

- no messages are lost
- no “spurious” messages are generated
- we will not object to getting the same message from two or more components (and we might even think this added some robustness).

We provide a function to check this requirement:

type

Analyse = Map \rightarrow Message-**set**

value

analyse_distributable : Analyse \rightarrow **Bool**

analyse_distributable(analyse) \equiv

$(\forall m : \text{Map} \bullet \text{analyse}(m) = \text{dunion}(\{ \text{analyse}(\text{distribute}(m)(p)) \mid p : P \}))$,

dunion : (Message-**set**)-**set** \rightarrow Message-**set**

dunion(mss) \equiv

$\{ m \mid m : \text{Message} \bullet \exists ms : \text{Message-}\mathbf{set} \bullet ms \in mss \wedge m \in ms \}$

Suppose we have a current map and a distributable analysis function. Suppose we apply a distributable function to the appropriate component of the distributed map. Under what circumstances can we update our set of global messages by applying the analysis function only to the changed component?

It should be apparent that if the analysis applied to the changed component does not cause any messages to be deleted that were also produced by another component then we will get the correct global change in messages by analysing only the changed component. That is, we have a theorem

```

forall f : Fun, pre_f : Pre_Fun, analyse : Analyse  $\bullet$ 
  ( $\forall d : D, m : \text{Map} \bullet \text{pre\_f}(d, m) \Rightarrow (\text{f}(d, m) \text{ post true})$ )  $\wedge$ 
  weakly_delegable(f, pre_f)  $\wedge$  analyse_distributable(analyse)  $\Rightarrow$ 
  ( $\forall d : D, m : \text{Map} \bullet$ 
    let pre_df = delegate(pre_f) in
      pre_df(d, distribute(m))  $\Rightarrow$ 
      let
        md = distribute(m)(p_of(d)),
        msgs = analyse(md),
        md' = f(d, md),
        msgs' = analyse(md')
      in
        ( $\forall \text{msg} : \text{Message} \bullet$ 
          msg  $\in$  msgs  $\wedge$  msg  $\notin$  msgs'  $\Rightarrow$ 
          msg  $\notin$  analyse(m  $\setminus$  dom md)
        )  $\Rightarrow$ 
        analyse(f(d, m)) = analyse(m)  $\setminus$  msgs  $\cup$  msgs'
      end
    end
  )

```

In particular, an analysis function may be *analysis_disjoint*, i.e. it cannot generate the same message from two different components:

```

analyse_disjoint : Analyse → Bool
analyse_disjoint(analyse) ≡
  (∀ m : Map, p1, p2 : P • p1 ≠ p2 ⇒
    analyse(distribute(m)(p1)) ∩ analyse(distribute(m)(p2)) = {})

```

Then we know that if a function is distributable and we apply it to a component then we may perform a distributable and disjoint analysis by applying it only to the component.

When we apply a function to a map it is likely only to change part of it. It may then be possible to “update” an analysis by repeating only a part of it. We can formalize this notion:

```

is_adequate_partial_analysis : Analyse × Fun × Pre_Fun × Analyse → Bool
is_adequate_partial_analysis(analyse, f, pre_f, part_analyse) ≡
  (∀ d : D, m : Map • pre_f(d, m) ⇒
    let
      msgs = analyse(m),
      old_msgs = part_analyse(m),
      m' = f(d, m),
      new_msgs = part_analyse(m')
    in
      analyse(m') = msgs \ old_msgs ∪ new_msgs
    end)

```

Note that there is some interplay here between the precondition *pre_f* and the partial analysis function *part_analyse*. Suppose, for example, that *f* is inserting some new data into a map. Then by strengthening *pre_f* (in particular by checking the new data to be inserted) it will typically be possible to do less checking in *part_analyse*. This suggests one extreme possibility: for some functions it may be possible to define *pre_f* so that analysing the resulting map generates exactly the same messages as before the function was applied. In this case we can take *part_analyse* as the constant function returning the empty set of messages. In practice we need to balance the amount of computation in *pre_f* (and also the resulting restrictiveness of possible applications of *f*) against the amount of computation in *part_analyse*. Then applying *is_adequate_partial_analysis* is a means of checking we have not missed anything.

We now have a number of analytical techniques we can use to decide when change functions and analysis functions may safely only be applied locally, and for deciding if a partial re-analysis is adequate.

4.2 Distributing timetables

There are three fairly natural possibilities for the data structure of timetables.

1. The first choice is close to the model in SP/12:

```

type TT = (Train  $\xrightarrow{m}$  Station  $\xrightarrow{m}$  Visit) × (Station  $\xrightarrow{m}$  Relation-set)

```

This model is quite convenient for most of the functions we want to apply to create, change and analyse timetables. However, when we come to distribute them we want to distribute by dispatch

units, which are essentially sets of stations. We can consider this as a pair of maps defining visits and relations respectively; the second distributes naturally but the first rather less so.

2. The second choice is to make Station the domain of both maps and hence merge them, which looks like a better data structure:

```
type TT = Station  $\overrightarrow{m}$  (Train  $\overrightarrow{m}$  Visit)  $\times$  Relation-set
```

This model distributes conveniently but some of the functions for creating and changing timetables are not so natural and the triple depth of nesting of maps for the distributed version makes the proofs complicated.

3. The third version flattens the map nesting:

```
type TT = ((Station  $\times$  Train)  $\overrightarrow{m}$  Visit)  $\times$  (Station  $\overrightarrow{m}$  Relation-set)
```

This does not look so well structured as the second (because of the apparent need for an invariant relating the domains of the two maps) but does give reasonably convenient functions on timetables and distribution, and is the one finally selected.

The third, chosen model has two maps and we distribute them separately. We do this by instantiating the generic distributed map *DIST_MAP* twice. For each instantiation we need to provide a parameter object defining the types *D*, *R*, *P* and *Message*, plus the function *p_of*:

```
type
  Dispatch_Unit, Message,
  TT :: visits : Visits relations : Relations,
  Visits = Station  $\times$  Train  $\overrightarrow{m}$  Visit,
  Relations = Station  $\overrightarrow{m}$  Relation-set
value
  du_of : Station  $\rightarrow$  Dispatch_Unit
object
  A :
    class
      type
        D = Station  $\times$  Train, R = Visit, P = Dispatch_Unit, M = Message
      value
        p_of : D  $\rightarrow$  P
        p_of((st, tn))  $\equiv$  du_of(st)
    end,
  V : DIST_MAP(A),
  B :
    class
      type
        D = Station, R = Relation-set, P = Dispatch_Unit, M = Message
      value
        p_of : D  $\rightarrow$  P = du_of
    end,
  R : DIST_MAP(B)
type Dist_TT = Dispatch_Unit  $\overrightarrow{m}$  TT
```

(We have used a short record for the type TT of timetables rather than the (isomorphic) cartesian product discussed above.)

Functions on timetables will in general apply to both components, and we need theories of which functions are delegable, distributable and disjoint. The definitions are fairly obvious:

value

```

weakly_delegable :
  (Station × Train × TT  $\xrightarrow{\sim}$  TT) × (Station × Train × TT → Bool) → Bool
weakly_delegable(f, pre_f)  $\equiv$ 
  (∃ vf : V.Fun, rf : R.Fun, pre_vf : V.Pre_Fun, pre_rf : R.Pre_Fun •
    ∀ st : Station, tn : Train, tt : TT •
      pre_f(st, tn, tt) =
        (pre_vf((st, tn), visits(tt)) ∧ pre_rf(st, relations(tt))) ∧
        (pre_f(st, tn, tt) ⇒
          f((st, tn, tt) = mk_TT(vf((st, tn), visits(tt)), rf(st, relations(tt)))
        ) ∧
        V.weakly_delegable(vf, pre_vf) ∧ R.weakly_delegable(rf, pre_rf)),

```

```

analyse_distributable : (TT → Message-set) → Bool
analyse_distributable(analyse)  $\equiv$ 
  (∃ vanalyse : V.Analyse, ranalyse : R.Analyse •
    ∀ tt : TT •
      analyse(tt) = vanalyse(visits(tt)) ∪ ranalyse(relations(tt)) ∧
      V.analyse_distributable(vanalyse) ∧ R.analyse_distributable(ranalyse)),

```

```

analyse_disjoint : (TT → Message-set) → Bool
analyse_disjoint(analyse)  $\equiv$ 
  (∃ vanalyse : V.Analyse, ranalyse : R.Analyse •
    ∀ tt : TT •
      analyse(tt) = vanalyse(visits(tt)) ∪ ranalyse(relations(tt)) ∧
      vanalyse(visits(tt)) ∩ ranalyse(relations(tt)) = {} ∧
      V.analyse_disjoint(vanalyse) ∧ R.analyse_disjoint(ranalyse))

```

strongly_delegable is defined just like *weakly_delegable*.

5 Specifications

This section presents the specification in a number of development levels. Each is presented top-down.

5.1 First specification

At this first level we concentrate on the general ideas of distribution and in the “dynamic” well-formedness checks. So the notion of a network is very rudimentary. *Stations*, *Tracks* and *Lines* are sorts with just a few functions (*time*, *capacity* and *is_dual*) introduced for lines.

```

scheme
PROJECT0 =
  extend DIST_SCHEDULE0 with
  class
  value
    project_by_stations : N.Station-set × TT → TT
    project_by_stations(sts, tt) ≡
      mk_TT(project_by_stations(sts, visits(tt)), project_by_stations(sts, relations(tt))),

    project_by_stations : N.Station-set × Visits → Visits
    project_by_stations(sts, vs) ≡
      [ (st, tn) ↦ vs(st, tn) | (st, tn) : N.Station × Train • (st, tn) ∈ dom vs ∧ st ∈ sts ],

    project_by_stations : N.Station-set × Relations → Relations
    project_by_stations(sts, relns) ≡ relns / sts,

    /* any visit overlapping with the period must have a corresponding
       visit;
       any relation referring to a visit overlapping with the period must
       be included and all the visits referred to by included relations
       must have corresponding visits;
       visits and relations included in the result must be in the original
    */
    /* the post condition does not specify that the result is the
       smallest such. This is assumed to be achieved in the
       implementation
    */
    project_by_period : (T.Time × T.Time) × TT → TT
    project_by_period((start, finish), tt) as tt'
    post
      let vs = visits(tt), relns = relations(tt), vs' = visits(tt'), relns' = relations(tt') in
        ∀ st : N.Station, tn : Train •
          (
            includes_visit(st, tn, vs) ⇒
              let
                v = get_visit(st, tn, vs),
                ivl = (T.mk_Sched_Time(start, T.pending), T.mk_Sched_Time(finish, T.pending))
              in
                T.have_minimum_overlap((arr(v), dep(v)), ivl, 0) ⇒ includes_visit(st, tn, vs')
            end
          ) ∧
          (
            ∀ reln : Relation •
              st ∈ dom relns ∧ reln ∈ relns(st) ∧ tn ∈ trains(reln) ∧ includes_visit(st, tn, vs') ⇒
                st ∈ dom relns' ∧
                reln ∈ relns'(st) ∧
                (
                  ∀ tn' : Train •
                    tn' ∈ trains(reln) ∧ includes_visit(st, tn', vs) ⇒ includes_visit(st, tn', vs')
                )
          )
        ) ∧

```

```

      (
        includes_visit(st, tn, vs')  $\Rightarrow$ 
        includes_visit(st, tn, vs)  $\wedge$  get_visit(st, tn, vs') = get_visit(st, tn, vs)
      )  $\wedge$ 
      (st  $\in$  dom relns'  $\Rightarrow$  st  $\in$  dom relns  $\wedge$  relns'(st)  $\subseteq$  relns(st))
    end
  end

```

scheme

DIST_SCHEDULE0 =

extend SCHEDULE0 **with**

class

type Dispatch_Unit, Message

value

du_of : N.Station \rightarrow Dispatch_Unit

object

A :

class

type D = N.Station \times Train, R = Visit, P = Dispatch_Unit, M = Message

value

p_of : D \rightarrow P

p_of((st, tn)) \equiv du_of(st)

end,

V : DIST_MAP(A),

B :

class

type D = N.Station, R = Relation-set, P = Dispatch_Unit, M = Message

value

p_of : D \rightarrow P = du_of

end,

R : DIST_MAP(B)

type Dist_TT = Dispatch_Unit \overline{m} TT

value

weakly_delegable :

(N.Station \times Train \times TT $\xrightarrow{\sim}$ TT) \times (N.Station \times Train \times TT \rightarrow **Bool**) \rightarrow **Bool**

weakly_delegable(f, pre_f) \equiv

(

\exists vf : V.Fun, rf : R.Fun, pre_vf : V.Pre_Fun, pre_rf : R.Pre_Fun \bullet

\forall st : N.Station, tn : Train, tt : TT \bullet

pre_f(st, tn, tt) = (pre_vf((st, tn), visits(tt)) \wedge pre_rf(st, relations(tt))) \wedge

(

pre_f(st, tn, tt) \Rightarrow

f((st, tn, tt)) = mk_TT(vf((st, tn), visits(tt)), rf(st, relations(tt)))

) \wedge

V.weakly_delegable(vf, pre_vf) \wedge R.weakly_delegable(rf, pre_rf)

```

    ),
strongly_delegable :
  (N.Station × Train × TT  $\xrightarrow{\sim}$  TT) × (N.Station × Train × TT → Bool) → Bool
strongly_delegable(f, pre-f)  $\equiv$ 
  (
     $\exists$  vf : V.Fun, rf : R.Fun, pre_vf : V.Pre_Fun, pre_rf : R.Pre_Fun •
     $\forall$  st : N.Station, tn : Train, tt : TT •
    pre_f(st, tn, tt) = (pre_vf((st, tn), visits(tt))  $\wedge$  pre_rf(st, relations(tt)))  $\wedge$ 
    (pre_f(st, tn, tt)  $\Rightarrow$  f(st, tn, tt) = mk_TT(vf((st, tn), visits(tt)), rf(st, relations(tt))))  $\wedge$ 
    V.strongly_delegable(vf, pre_vf)  $\wedge$  R.strongly_delegable(rf, pre_rf)
  ),
analyse_distributable : (TT → Message-set) → Bool
analyse_distributable(analyse)  $\equiv$ 
  (
     $\exists$  vanalyse : V.Analyse, ranalyse : R.Analyse •
     $\forall$  tt : TT •
    analyse(tt) = vanalyse(visits(tt))  $\cup$  ranalyse(relations(tt))  $\wedge$ 
    V.analyse_distributable(vanalyse)  $\wedge$  R.analyse_distributable(ranalyse)
  ),
analyse_disjoint : (TT → Message-set) → Bool
analyse_disjoint(analyse)  $\equiv$ 
  (
     $\exists$  vanalyse : V.Analyse, ranalyse : R.Analyse •
     $\forall$  tt : TT •
    analyse(tt) = vanalyse(visits(tt))  $\cup$  ranalyse(relations(tt))  $\wedge$ 
    vanalyse(visits(tt))  $\cap$  ranalyse(relations(tt)) = {}  $\wedge$ 
    V.analyse_disjoint(vanalyse)  $\wedge$  R.analyse_disjoint(ranalyse)
  )
end

```

scheme

DIST_MAP(X : DIST_MAP_PARM) =

class**type**

D = X.D,
 R = X.R,
 P = X.P,
 Map = D \xrightarrow{m} R,
 Dist_Map = P \xrightarrow{m} Map,
 Message = X.M,
 Fun = D × Map $\xrightarrow{\sim}$ Map,
 Pre_Fun = D × Map → **Bool**,
 Analyse = Map → Message-set

value

p_of : D → P = X.p_of,

distribute : Map → Dist_Map

distribute(m) \equiv [p \mapsto m / partition(p) | p : P],

merge : Dist_Map \rightarrow Map

merge(dm) \equiv [d \mapsto dm(p_of(d))(d) | d : D • p_of(d) \in **dom** dm \wedge d \in **dom** dm(p_of(d))],

partition : P \rightarrow D-set

partition(p) \equiv { d | d : D • p_of(d) = p },

delegate : Fun \rightarrow D \times Dist_Map $\xrightarrow{\sim}$ Dist_Map

delegate(f)(d, dm) \equiv **let** p = p_of(d) **in** dm \dagger [p \mapsto f(d, dm(p))] **end**,

delegate : Pre_Fun \rightarrow D \times Dist_Map \rightarrow **Bool**

delegate(pre_f)(d, dm) \equiv **let** p = p_of(d) **in** pre_f(d, dm(p)) **end**,

weakly_delegable : Fun \times Pre_Fun \rightarrow **Bool**

weakly_delegable(f, pre_f) \equiv

let df = delegate(f), pre_df = delegate(pre_f) **in**

\forall d : D, m : Map •

pre_df(d, distribute(m)) \Rightarrow pre_f(d, m) \wedge df(d, distribute(m)) = distribute(f(d, m))

end,

strongly_delegable : Fun \times Pre_Fun \rightarrow **Bool**

strongly_delegable(f, pre_f) \equiv

weakly_delegable(f, pre_f) \wedge

let pre_df = delegate(pre_f) **in**

\forall d : D, m : Map • pre_f(d, m) \Rightarrow pre_df(d, distribute(m))

end,

weakly_delegable1 : Fun \times Pre_Fun \rightarrow **Bool**

weakly_delegable1(f, pre_f) \equiv

(\forall d : D, m : Map • pre_f(d, m) \Rightarrow (f(d, m) **post true**)) \wedge

(

\forall d : D, m : Map •

let ds = partition(p_of(d)) **in**

pre_f(d, m / ds) \Rightarrow

pre_f(d, m) \wedge

let (m1, m2) = (f(d, m / ds), f(d, m)) **in**

dom m1 = **dom** m2 \cap ds \wedge

(\forall d' : D • d' \in **dom** m1 \Rightarrow m1(d') = m2(d')) \wedge

(\forall p : P • p \neq p_of(d) \Rightarrow **dom** m \cap partition(p) = **dom** m2 \cap partition(p)) \wedge

(\forall d' : D • d' \in **dom** m \wedge d' \in **dom** m2 \wedge d' \notin ds \Rightarrow m(d') = m2(d'))

end

end

),

analyse_distributable : Analyse \rightarrow **Bool**

analyse_distributable(analyse) \equiv

(\forall m : Map • analyse(m) = dunion({ analyse(distribute(m)(p)) | p : P })),

analyse_disjoint : Analyse \rightarrow **Bool**

analyse_disjoint(analyse) \equiv

```

(
  ∀ m : Map, p1, p2 : P •
    p1 ≠ p2 ⇒ analyse(distribute(m)(p1)) ∩ analyse(distribute(m)(p2)) = {}
),
)

dunion : (Message-set)-set → Message-set
dunion(mss) ≡ { m | m : Message • ∃ ms : Message-set • ms ∈ mss ∧ m ∈ ms },

is_adequate_partial_analysis : Analyse × Fun × Pre_Fun × Analyse → Bool
is_adequate_partial_analysis(analyse, f, pre_f, part_analyse) ≡
(
  ∀ d : D, m : Map •
    pre_f(d, m) ⇒
    let
      msgs = analyse(m),
      old_msgs = part_analyse(m),
      m' = f(d, m),
      new_msgs = part_analyse(m')
    in
      analyse(m') = msgs \ old_msgs ∪ new_msgs
    end
)
end

```

scheme

```
DIST_MAP_PARM = class type D, R, P, M value p_of : D → P end
```

scheme

```

SCHEDULE0 =
  extend extend class object N : NETWORK0 end with TIMETABLE0(N) with
  class
    type Journey = N.Station  $\overrightarrow{m}$  Visit

  value
    delete_train : Train × TT → TT
    delete_train(tn, tt) ≡ mk_TT(delete_train(tn, visits(tt)), delete_train(tn, relations(tt))),

    delete_train : Train × Visits → Visits
    delete_train(tn, vs) ≡ vs \ { (st, tn) | st : N.Station },

    delete_train : Train × Relations → Relations
    delete_train(tn, relns) ≡ [ st ↦ delete_train(tn, relns(st)) | st : N.Station • st ∈ dom relns ],

    delete_train : Train × Relation-set → Relation-set
    delete_train(tn, relns) ≡
      { delete_train(tn, reln) | reln : Relation • reln ∈ relns ∧ card (trains(reln) \ {tn}) ≥ 2 },

    delete_train : Train × Relation  $\overset{\sim}{\rightarrow}$  Relation
    delete_train(tn, reln) ≡
      case reln of
        mk_Connection(tns, ivl) → mk_Connection(tns \ {tn}, ivl),

```

```

    mk_Disconnection(tns, ivl) → mk_Disconnection(tns \ {tn}, ivl),
   $\overline{\quad}$  → reln
end
pre can_delete_train(tn, reln),

can_delete_train : Train × Relation  $\xrightarrow{\sim}$  Bool
can_delete_train(tn, reln)  $\equiv$  card (trains(reln) \ {tn})  $\geq$  2,

add_train : Train × Journey × TT → TT
add_train(tn, j, tt)  $\equiv$ 
  replace_visits(visits(tt) † [ (st, tn) ↦ j(st) | st : N.Station • st ∈ dom j ], tt),

delete_first_visit : N.Station × Train × TT  $\xrightarrow{\sim}$  TT
delete_first_visit(st, tn, tt)  $\equiv$ 
  mk_TT(delete_first_visit(st, tn, visits(tt)), delete_visit(st, tn, relations(tt)))
pre can_delete_first_visit(st, tn, visits(tt)),

delete_first_visit : N.Station × Train × Visits  $\xrightarrow{\sim}$  Visits
delete_first_visit(st, tn, vs)  $\equiv$ 
  let
    st' = station(dep_info(get_visit(st, tn, vs))),
    v' = replace_arr_info(nil, get_visit(st', tn, vs))
  in
    vs \ {(st, tn)} † [(st', tn) ↦ v']
  end
pre can_delete_first_visit(st, tn, vs),

can_delete_first_visit : N.Station × Train × Visits → Bool
can_delete_first_visit(st, tn, vs)  $\equiv$ 
  includes_visit(st, tn, vs) ∧ is_first_visit(tn, get_visit(st, tn, vs), vs),

delete_last_visit : N.Station × Train × TT  $\xrightarrow{\sim}$  TT
delete_last_visit(st, tn, tt)  $\equiv$ 
  mk_TT(delete_last_visit(st, tn, visits(tt)), delete_visit(st, tn, relations(tt)))
pre can_delete_last_visit(st, tn, visits(tt)),

delete_last_visit : N.Station × Train × Visits  $\xrightarrow{\sim}$  Visits
delete_last_visit(st, tn, vs)  $\equiv$ 
  let
    st' = station(arr_info(get_visit(st, tn, vs))),
    v' = replace_dep_info(nil, get_visit(st, tn, vs))
  in
    vs \ {(st, tn)} † [(st', tn) ↦ v']
  end
pre can_delete_last_visit(st, tn, vs),

can_delete_last_visit : N.Station × Train × Visits → Bool
can_delete_last_visit(st, tn, vs)  $\equiv$ 
  includes_visit(st, tn, vs) ∧ is_last_visit(tn, get_visit(st, tn, vs), vs),

delete_visit : N.Station × Train × Relations → Relations

```

```

delete_visit(st, tn, relns)  $\equiv$ 
  if st  $\in$  dom relns then relns  $\dagger$  [st  $\mapsto$  delete_train(tn, relns(st))] else relns end,

```

```

prepend_visit : N.Station  $\times$  Train  $\times$  Visit  $\times$  TT  $\xrightarrow{\sim}$  TT
prepend_visit(st, tn, v, tt)  $\equiv$ 
  replace_visits(prepend_visit(st, tn, v, visits(tt)), tt)
  pre can_prepend_visit(st, tn, v, visits(tt)),

```

```

prepend_visit : N.Station  $\times$  Train  $\times$  Visit  $\times$  Visits  $\xrightarrow{\sim}$  Visits
prepend_visit(st, tn, v, vs)  $\equiv$ 
  let
    st' = station(dep_info(v)),
    v' = get_visit(st', tn, vs),
    arr_info' = info(st, line(dep_info(v))),
    v'' = replace_arr_info(arr_info', v')
  in
    vs  $\dagger$  [(st, tn)  $\mapsto$  v, (st', tn)  $\mapsto$  v'']
  end
  pre can_prepend_visit(st, tn, v, vs),

```

```

can_prepend_visit : N.Station  $\times$  Train  $\times$  Visit  $\times$  Visits  $\rightarrow$  Bool
can_prepend_visit(st, tn, v, vs)  $\equiv$ 
   $\sim$  includes_visit(st, tn, vs)  $\wedge$ 
  dep_info(v)  $\neq$  nil  $\wedge$ 
  let st' = station(dep_info(v)) in
    includes_visit(st', tn, vs)  $\wedge$  arr_info(get_visit(st', tn, vs)) = nil
  end,

```

```

append_visit : N.Station  $\times$  Train  $\times$  Visit  $\times$  T.Time  $\times$  TT  $\xrightarrow{\sim}$  TT
append_visit(st, tn, v, t, tt)  $\equiv$ 
  replace_visits(append_visit(st, tn, v, t, visits(tt)), tt)
  pre can_append_visit(st, tn, v, visits(tt)),

```

```

append_visit : N.Station  $\times$  Train  $\times$  Visit  $\times$  T.Time  $\times$  Visits  $\xrightarrow{\sim}$  Visits
append_visit(st, tn, v, t, vs)  $\equiv$ 
  let
    st' = station(arr_info(v)),
    v' = get_visit(st', tn, vs),
    dep_info' = info(st, line(dep_info(v)), t),
    v'' = replace_dep_info(dep_info', v')
  in
    vs  $\dagger$  [(st, tn)  $\mapsto$  v, (st', tn)  $\mapsto$  v'']
  end
  pre can_append_visit(st, tn, v, vs),

```

```

can_append_visit : N.Station  $\times$  Train  $\times$  Visit  $\times$  Visits  $\rightarrow$  Bool
can_append_visit(st, tn, v, vs)  $\equiv$ 
   $\sim$  includes_visit(st, tn, vs)  $\wedge$ 
  arr_info(v)  $\neq$  nil  $\wedge$ 
  let st' = station(arr_info(v)) in
    includes_visit(st', tn, vs)  $\wedge$  dep_info(get_visit(st', tn, vs)) = nil

```

end,

$\text{amend_visit} : \text{N.Station} \times \text{Train} \times \text{Visit} \times \text{TT} \xrightarrow{\sim} \text{TT}$

$\text{amend_visit}(\text{st}, \text{tn}, \text{v}, \text{tt}) \equiv$
 $\text{replace_visits}(\text{amend_visit}(\text{st}, \text{tn}, \text{v}, \text{visits}(\text{tt})), \text{tt})$
pre $\text{can_amend_visit}(\text{st}, \text{tn}, \text{v}, \text{visits}(\text{tt})),$

$\text{amend_visit} : \text{N.Station} \times \text{Train} \times \text{Visit} \times \text{Visits} \xrightarrow{\sim} \text{Visits}$

$\text{amend_visit}(\text{st}, \text{tn}, \text{v}, \text{vs}) \equiv \text{vs} \uparrow [(\text{st}, \text{tn}) \mapsto \text{v}]$ **pre** $\text{can_amend_visit}(\text{st}, \text{tn}, \text{v}, \text{vs}),$

$\text{can_amend_visit} : \text{N.Station} \times \text{Train} \times \text{Visit} \times \text{Visits} \rightarrow \mathbf{Bool}$

$\text{can_amend_visit}(\text{st}, \text{tn}, \text{v}, \text{vs}) \equiv$
 $\text{includes_visit}(\text{st}, \text{tn}, \text{vs}) \wedge$
let $v' = \text{get_visit}(\text{st}, \text{tn}, \text{vs})$ **in**
if $\text{arr_info}(v') = \text{nil}$ **then**
 $\text{arr_info}(v) = \text{nil}$
else
 $\text{arr_info}(v) \neq \text{nil} \wedge \text{station}(\text{arr_info}(v)) = \text{station}(\text{arr_info}(v'))$
end \wedge
if $\text{dep_info}(v') = \text{nil}$ **then**
 $\text{dep_info}(v) = \text{nil}$
else
 $\text{dep_info}(v) \neq \text{nil} \wedge \text{station}(\text{dep_info}(v)) = \text{station}(\text{dep_info}(v'))$
end
end,

$\text{add_relation} : \text{N.Station} \times \text{Relation} \times \text{TT} \rightarrow \text{TT}$

$\text{add_relation}(\text{st}, \text{reln}, \text{tt}) \equiv \text{replace_relations}(\text{add_relation}(\text{st}, \text{reln}, \text{relations}(\text{tt})), \text{tt}),$

$\text{add_relation} : \text{N.Station} \times \text{Relation} \times \text{Relations} \rightarrow \text{Relations}$

$\text{add_relation}(\text{st}, \text{reln}, \text{relns}) \equiv$
if $\text{st} \in \text{dom } \text{relns}$ **then** $\text{relns} \uparrow [\text{st} \mapsto \text{relns}(\text{st}) \cup \{\text{reln}\}]$ **else** $\text{relns} \cup [\text{st} \mapsto \{\text{reln}\}]$ **end,**

$\text{delete_relation} : \text{N.Station} \times \text{Relation} \times \text{TT} \rightarrow \text{TT}$

$\text{delete_relation}(\text{st}, \text{reln}, \text{tt}) \equiv \text{replace_relations}(\text{delete_relation}(\text{st}, \text{reln}, \text{relations}(\text{tt})), \text{tt}),$

$\text{delete_relation} : \text{N.Station} \times \text{Relation} \times \text{Relations} \rightarrow \text{Relations}$

$\text{delete_relation}(\text{st}, \text{reln}, \text{relns}) \equiv$
if $\text{st} \in \text{dom } \text{relns}$ **then** $\text{relns} \uparrow [\text{st} \mapsto \text{relns}(\text{st}) \setminus \{\text{reln}\}]$ **else** relns **end,**

$\text{report_arrival} : \text{N.Station} \times \text{Train} \times \text{T.Time} \times \text{TT} \xrightarrow{\sim} \text{TT}$

$\text{report_arrival}(\text{st}, \text{tn}, \text{t}, \text{tt}) \equiv$
 $\text{replace_visits}(\text{report_arrival}(\text{st}, \text{tn}, \text{t}, \text{visits}(\text{tt})), \text{tt})$
pre $\text{includes_visit}(\text{st}, \text{tn}, \text{visits}(\text{tt})),$

$\text{report_arrival} : \text{N.Station} \times \text{Train} \times \text{T.Time} \times \text{Visits} \xrightarrow{\sim} \text{Visits}$

$\text{report_arrival}(\text{st}, \text{tn}, \text{t}, \text{vs}) \equiv$
let $v = \text{get_visit}(\text{st}, \text{tn}, \text{vs}), v' = \text{replace_arr}(\text{T.mk_Sched_Time}(\text{t}, \text{T.occurred}), v)$ **in**
 $\text{vs} \uparrow [(\text{st}, \text{tn}) \mapsto v']$
end
pre $\text{includes_visit}(\text{st}, \text{tn}, \text{vs}),$

```
report_departure : N.Station × Train × T.Time × TT → TT
```

```
report_departure(st, tn, t, tt) ≡
  replace_visits(report_departure(st, tn, t, visits(tt)), tt)
  pre includes_visit(st, tn, visits(tt)),
```

```
report_departure : N.Station × Train × T.Time × Visits  $\xrightarrow{\sim}$  Visits
```

```
report_departure(st, tn, t, vs) ≡
  let v = get_visit(st, tn, vs), v' = replace_dep(T.mk_Sched_Time(t, T.occurred), v) in
    vs † [(st, tn) ↦ v']
  end
  pre includes_visit(st, tn, vs),
```

```
report_passage : N.Station × Train × T.Time × TT → TT
```

```
report_passage(st, tn, t, tt) ≡
  replace_visits(report_passage(st, tn, t, visits(tt)), tt)
  pre includes_visit(st, tn, visits(tt)),
```

```
report_passage : N.Station × Train × T.Time × Visits  $\xrightarrow{\sim}$  Visits
```

```
report_passage(st, tn, t, vs) ≡
  let
    v = get_visit(st, tn, vs),
    v' =
      replace_dep
        (T.mk_Sched_Time(t, T.occurred), replace_arr(T.mk_Sched_Time(t, T.occurred), v))
  in
    vs † [(st, tn) ↦ v']
  end
  pre includes_visit(st, tn, vs)
```

```
end
```

```
scheme
```

```
TIMETABLE0(N : NETWORK0) =
```

```
  hide
```

```
    no_station_conflicts, movements_separated, no_track_conflicts, no_line_conflicts, pending_arrivals,
    no_opposing_occupancy, is_sat_Connection, is_sat_Disconnection, is_sat_Dependency
```

```
  in
```

```
    class
```

```
      type
```

```
        TT :: visits : Visits ↔ replace_visits relations : Relations ↔ replace_relations,
```

```
        Visits = N.Station × Train  $\xrightarrow{m}$  Visit,
```

```
        Relations = N.Station  $\xrightarrow{m}$  Relation-set,
```

```
        Visit ::
```

```
          track : N.Track ↔ replace_track
```

```
          arr : T.Sched_Time ↔ replace_arr
```

```
          dep : T.Sched_Time ↔ replace_dep
```

```
          arr_info : Opt_Arr_Info ↔ replace_arr_info
```

```
          dep_info : Opt_Dep_Info ↔ replace_dep_info,
```

```
        Opt_Arr_Info == nil | info(station : N.Station, line : N.Line),
```

```
        Opt_Dep_Info == nil | info(station : N.Station, line : N.Line, line_time : T.Time_Interval),
```

```
        Relation = Connection | Disconnection | Dependency,
```

```

Connection :: trains : Train_set2 overlap : T.Time_Interval,
Disconnection :: trains : Train_set2 separation : T.Time_Interval,
Dependency :: arriver : Train departer : Train interval : T.Time_Interval,
Train,
Train_set2 = { | tns : Train-set • card tns ≥ 2 | }

```

value

```

min_dep_sep, min_arr_sep, min_arr_dep_sep, min_line_sep, min_stop_time : T.Time_Interval,

```

```

/* no station or line conflicts;
   all visits well formed;
   for intermediate visits pending arrival at next station is line
   time after departure;
   connections, disconnections and dependencies satisfied
*/
is_dyn_wf_TT : TT → Bool
is_dyn_wf_TT(tt) ≡
  let vs = visits(tt) in
    no_station_conflicts(vs) ∧
    no_line_conflicts(vs) ∧
    (
      ∀ st : N.Station •
        (
          ∀ tn : Train •
            includes_visit(st, tn, vs) ⇒
              let v = get_visit(st, tn, vs) in
                is_dyn_wf_Visit(tn, v) ∧
                (
                  is_intermediate_visit(tn, v, vs) ⇒
                    let v' = next_visit(tn, v, vs) in
                      T.has_duration((dep(v), arr(v')), line_time(dep_info(v)))
                )
              end
            )
          )
        )
      ) ∧
    (
      st ∈ dom relations(tt) ⇒
        (
          ∀ reln : Relation •
            reln ∈ relations(tt)(st) ⇒
              case reln of
                mk_Connection(tns, ivl) → is_sat_Connection(st, tns, ivl, vs),
                mk_Disconnection(tns, ivl) → is_sat_Disconnection(st, tns, ivl, vs),
                mk_Dependency(arr, dep, ivl) → is_sat_Dependency(st, arr, dep, ivl, vs)
              end
            )
          )
        )
      )
    )
  end,

/* occurred departure implies occurred previous arrival;

```

```

    pending departure is either no stop or stop for at least minimum
    time
  */
  is_dyn_wf_Visit : Train × Visit → Bool
  is_dyn_wf_Visit(tn, v) ≡
    T.is_wf_Interval(dep(v), arr(v)) ∧
    (is_a_stop(v) ⇒ T.has_minimum_duration((arr(v), dep(v)), min_stop_time)),

  is_a_stop : Visit → Bool
  is_a_stop(v) ≡ T.time(dep(v)) > T.time(arr(v)),

  includes_visit : N.Station × Train × Visits → Bool
  includes_visit(st, tn, vs) ≡ (st, tn) ∈ dom vs,

  get_visit : N.Station × Train × Visits  $\xrightarrow{\sim}$  Visit
  get_visit(st, tn, vs) ≡ vs(st, tn) pre includes_visit(st, tn, vs),

  next_visit : Train × Visit × Visits  $\xrightarrow{\sim}$  Visit
  next_visit(tn, v, vs) ≡ vs(station(dep_info(v)), tn) pre is_intermediate_visit(tn, v, vs),

  is_intermediate_visit : Train × Visit × Visits → Bool
  is_intermediate_visit(tn, v, vs) ≡ dep_info(v) ≠ nil ∧ (station(dep_info(v)), tn) ∈ dom vs,

  /* has successor but no predecessor */
  is_first_visit : Train × Visit × Visits → Bool
  is_first_visit(tn, v, vs) ≡ is_intermediate_visit(tn, v, vs) ∧ arr_info(v) = nil,

  /* has predecessor but no successor */
  is_last_visit : Train × Visit × Visits → Bool
  is_last_visit(tn, v, vs) ≡
    dep_info(v) = nil ∧ arr_info(v) ≠ nil ∧ (station(arr_info(v)), tn) ∈ dom vs,

  /* different trains have separated movements in or out of the
  station;
  no track conflicts
  */
  no_station_conflicts : Visits → Bool
  no_station_conflicts(vs) ≡
    (
      ∀ st : N.Station, tn1, tn2 : Train •
        tn1 ≠ tn2 ∧ includes_visit(st, tn1, vs) ∧ includes_visit(st, tn2, vs) ⇒
        let v1 = get_visit(st, tn1, vs), v2 = get_visit(st, tn2, vs) in
        movements_separated(v1, v2) ∧ no_track_conflicts(v1, v2)
    )
    end
  ),

  /* arrivals and departures separated;
  departures on same line separated
  */
  movements_separated : Visit × Visit → Bool
  movements_separated(v1, v2) ≡

```

```

let a1 = arr(v1), a2 = arr(v2), d1 = dep(v1), d2 = dep(v2) in
  T.have_minimum_separation(a1, a2, min_arr_sep)  $\wedge$ 
  T.have_minimum_separation(d1, d2, min_dep_sep)  $\wedge$ 
  T.have_minimum_separation(a1, d2, min_arr_dep_sep)  $\wedge$ 
  T.have_minimum_separation(a2, d1, min_arr_dep_sep)  $\wedge$ 
  (
    dep_info(v1)  $\neq$  nil  $\wedge$  dep_info(v2)  $\neq$  nil  $\wedge$  line(dep_info(v1)) = line(dep_info(v2))  $\Rightarrow$ 
    T.have_minimum_separation(d1, d2, min_line_sep)
  )
end,

/* track occupancies do not overlap in time */
no_track_conflicts : Visit  $\times$  Visit  $\rightarrow$  Bool
no_track_conflicts(v1, v2)  $\equiv$ 
  track(v1) = track(v2)  $\Rightarrow$ 
  T.has_minimum_duration((dep(v1), arr(v2)), 1)  $\vee$ 
  T.has_minimum_duration((dep(v2), arr(v1)), 1),

/* if a train departure is pending:
   the number of trains currently on that line in the same
   direction is less than the line capacity;
   all such trains are scheduled to arrive earlier;
   any trains in the opposite direction arrive before this departs
   or depart after it arrives
*/
no_line_conflicts : Visits  $\rightarrow$  Bool
no_line_conflicts(vs)  $\equiv$ 
  (
     $\forall$  st : N.Station, tn : Train  $\bullet$ 
    includes_visit(st, tn, vs)  $\Rightarrow$ 
    let v = get_visit(st, tn, vs) in
      T.status(dep(v)) = T.pending  $\wedge$  dep_info(v)  $\neq$  nil  $\Rightarrow$ 
      let
        lin = line(dep_info(v)),
        pending = pending_arrivals(st, tn, lin, T.time(dep(v)), vs),
        arr = T.time(dep(v)) + line_time(dep_info(v))
      in
        (pending = {}  $\vee$  card pending < N.line_capacity(lin)  $\wedge$  T.max(pending) < arr)  $\wedge$ 
        (
          N.is_dual(lin)  $\Rightarrow$ 
          let st' = station(dep_info(v)) in
            no_opposing_occupancy(lin, st', T.time(dep(v)), arr, vs)
          end
        )
      )
    )
  )
end
end,

pending_arrivals : N.Station  $\times$  Train  $\times$  N.Line  $\times$  T.Time  $\times$  Visits  $\rightarrow$  T.Time-set
pending_arrivals(st, tn, lin, dep, vs)  $\equiv$ 
  { T.time(dep(v')) + line_time(dep_info(v')) |

```

```

    tn' : Train, v' : Visit •
    tn' ≠ tn ∧ includes_visit(st, tn', vs) ⇒
    v' = get_visit(st, tn', vs) ∧
    dep_info(v') ≠ nil ∧
    line(dep_info(v')) = lin ∧
    T.time(dep(v')) < dep ∧ T.time(dep(v')) + line_time(dep_info(v')) > dep
  },

/* any departure from station on line either arrives before start
   or departs after finish
*/
no_opposing_occupancy : N.Line × N.Station × T.Time × T.Time × Visits → Bool
no_opposing_occupancy(lin, st, start, finish, vs) ≡
(
  ∀ tn : Train •
  includes_visit(st, tn, vs) ⇒
  let v = get_visit(st, tn, vs) in
  is_intermediate_visit(tn, v, vs) ∧ dep_info(v) ≠ nil ∧ line(dep_info(v)) = lin ⇒
  T.time(arr(next_visit(tn, v, vs))) < start ∨ T.time(dep(v)) > finish
  end
),

/* connection is satisfied if:
   each train stops at the station;
   the stopped times of different trains overlap sufficiently
*/
is_sat_Connection : N.Station × Train_set2 × T.Time_Interval × Visits → Bool
is_sat_Connection(st, tns, ivl, vs) ≡
(∀ tn : Train • tn ∈ tns ⇒ includes_visit(st, tn, vs) ∧ is_a_stop(get_visit(st, tn, vs))) ∧
(
  ∀ tn1, tn2 : Train •
  {tn1, tn2} ⊆ tns ∧ tn1 ≠ tn2 ⇒
  let v1 = get_visit(st, tn1, vs), v2 = get_visit(st, tn2, vs) in
  T.have_minimum_overlap((arr(v1), dep(v1)), (arr(v2), dep(v2)), ivl)
  end
),

/* disconnection is satisfied if:
   each train visits the station;
   if two trains both stop, their stopped times are sufficiently
   separated
*/
is_sat_Disconnection : N.Station × Train_set2 × T.Time_Interval × Visits → Bool
is_sat_Disconnection(st, tns, ivl, vs) ≡
(∀ tn : Train • tn ∈ tns ⇒ includes_visit(st, tn, vs)) ∧
(
  ∀ tn1, tn2 : Train •
  {tn1, tn2} ⊆ tns ∧ tn1 ≠ tn2 ⇒
  let v1 = get_visit(st, tn1, vs), v2 = get_visit(st, tn2, vs) in
  ∼ is_a_stop(v1) ∨
  ∼ is_a_stop(v2) ∨

```

```

        T.have_minimum_separation((arr(v1), dep(v1)), (arr(v2), dep(v2)), ivl)
    end
),
/* dependency is satisfied if:
   both trains stop at the station;
   arriver arrives sufficiently before departer departs
*/
/* note: if arriver and departer are same a dependency can specify
   a longer than usual stopping time
*/
is_sat_Dependency : N.Station × Train × Train × T.Time_Interval × Visits → Bool
is_sat_Dependency(st, arriver, departer, ivl, vs) ≡
    includes_visit(st, arriver, vs) ∧
    includes_visit(st, departer, vs) ∧
let v1 = get_visit(st, arriver, vs), v2 = get_visit(st, departer, vs) in
    is_a_stop(v1) ∧ is_a_stop(v2) ∧ T.has_minimum_duration((dep(v2), arr(v1)), ivl)
end,

trains : Relation → Train-set
trains(reln) ≡
case reln of
    mk_Connection(tns, _) → tns,
    mk_Disconnection(tns, _) → tns,
    mk_Dependency(arr, dep, _) → {arr, dep}
end
end

scheme
NETWORK0 =
class
    type Station, Track, Line

    value
        line_time : Line → T.Time_Interval,
        line_capacity : Line → Nat,
        is_dual : Line → Bool
end

object
T : TIME

scheme
TIME =
class
    type
        Time = Nat,
        Time_Interval = Nat,
        Sched_Time :: time : Time status : Event_Status,
        Event_Status == pending | occurred,
        Interval = Sched_Time × Sched_Time

```

```

value
  /* end does not precede beginning;
     occurred end implies occurred beginning
  */
  is_wf_Interval : Interval → Bool
  is_wf_Interval(b, e) ≡ time(e) ≥ time(b) ∧ (status(e) = occurred ⇒ status(b) = occurred),

  /* end pending implies interval has at least minimum duration */
  has_minimum_duration : Interval × Time_Interval → Bool
  has_minimum_duration((b, e), min) ≡ status(e) = pending ⇒ time(e) - time(b) ≥ min,

  /* end pending implies interval has duration */
  has_duration : Interval × Time_Interval → Bool
  has_duration((b, e), dur) ≡ status(e) = pending ⇒ time(e) - time(b) = dur,

  /* either pending implies times separated by at least minimum */
  have_minimum_separation : Sched_Time × Sched_Time × Time_Interval → Bool
  have_minimum_separation(t1, t2, min) ≡
    status(t1) = pending ∨ status(t2) = pending ⇒ abs (time(t1) - time(t2)) ≥ min,

  /* end of either pending implies overlap by at least minimum */
  have_minimum_overlap : Interval × Interval × Time_Interval → Bool
  have_minimum_overlap((b1, e1), (b2, e2), min) ≡
    status(e1) = pending ∨ status(e2) = pending ⇒
    min({time(e1), time(e2)}) - max({time(b1), time(b2)}) ≥ min,

  /* end of either pending implies separation by at least minimum */
  have_minimum_separation : Interval × Interval × Time_Interval → Bool
  have_minimum_separation((b1, e1), (b2, e2), min) ≡
    status(e1) = pending ∨ status(e2) = pending ⇒
    max({time(b1), time(b2)}) - min({time(e1), time(e2)}) ≥ min,

  min : Time-set  $\xrightarrow{\sim}$  Time
  min(ts) as t post t ∈ ts ∧ (∀ t' : Time • t' ∈ ts ⇒ t' ≥ t) pre ts ≠ {},

  max : Time-set  $\xrightarrow{\sim}$  Time
  max(ts) as t post t ∈ ts ∧ (∀ t' : Time • t' ∈ ts ⇒ t ≥ t') pre ts ≠ {}
end

```

5.2 Second specification

In this level we add details of the physical network (right down to the level of units and connectors from which railway lines are composed). This allows us to add the “static” well-formedness checks on timetables.

PROJECT1, *DIST_SCHEDULE1* and *SCHEDULE1* are obtained trivially by changing the names of the extended schemes (replacing the final 0 with 1) and are not included here. *DIST_MAP*, *DIST_MAP_PARM*, *T* and *TIME* are unchanged.

```

scheme
TIMETABLE1(N : NETWORK1) =
  extend TIMETABLE0(N) with
  class
  value
    /* journeys and relations statically well-formed */
    is_stat_wf_TT : N.Network × TT → Bool
    is_stat_wf_TT(net, tt) ≡
      is_stat_wf_Visits(net, visits(tt)) ∧ is_stat_wf_Relations(visits(tt), relations(tt)),

    /* each visit statically well-formed;
       for each train, possible to form a complete sequence of stations
       that is statically well-formed
    */
    is_stat_wf_Visits : N.Network × Visits → Bool
    is_stat_wf_Visits(net, vs) ≡
      (
        ∀ tn : Train •
          (∀ st : N.Station • (st, tn) ∈ dom vs ⇒ is_stat_wf_Visit(net, st, vs(st, tn))) ∧
          let journey = [ st ↦ vs(st, tn) | st : N.Station • (st, tn) ∈ dom vs ] in
            is_stat_wf_journey(net, journey)
          end
      ),

    /* visit involves known station and track in that station;
       arrival line is a line from preceding station to the track;
       departure line is a line to next station from the track and
       time is at least time for line;
    */
    is_stat_wf_Visit : N.Network × N.Station × Visit → Bool
    is_stat_wf_Visit(net, st, v) ≡
      st ∈ N.stations(net) ∧
      track(v) ∈ N.tracks(st) ∧
      case arr_info(v) of
        nil → true, info(st', ln) → N.is_line_from(ln, st') ∧ N.is_line_to_track(ln, track(v))
      end ∧
      case dep_info(v) of
        nil → true,
        info(st', ln, ivl) →
          N.is_line_to(ln, st') ∧ N.is_line_from_track(ln, track(v)) ∧ ivl ≥ N.line_time(ln)
      end,

    /* stations exist in network:
       possible to form complete sequence of stations that is
       statically well-formed
    */
    is_stat_wf_journey : N.Network × (N.Station  $\xrightarrow{m}$  Visit) → Bool
    is_stat_wf_journey(net, j) ≡
      dom j ⊆ N.stations(net) ∧
      (
        ∃ stl : N.Station* •

```

```

    elems stl = dom j ∧ len stl = card dom j ∧ is_stat_wf_station_list(net, stl, j)
  ),

/* visit list is not empty;
   each visit is statically well-formed;
   final visit has either no departure line or departs outside
   possible stations;
   for each non-final visit, either:
   next station is in journey;
   departure line statically corresponds to arrival line of next
   visit, i.e. lines are same and stations match; or
   next station is not in journey and preceding station of next
   visit in this journey is not in the journey
*/
is_stat_wf_station_list : N.Network × N.Station* × (N.Station  $\xrightarrow{m}$  Visit)  $\xrightarrow{\sim}$  Bool
is_stat_wf_station_list(net, stl, j)  $\equiv$ 
  case stl of
    ⟨⟩ → false,
    ⟨st⟩ →
      let v = j(st) in
        is_stat_wf_Visit(net, st, v) ∧ (dep_info(v) = nil ∨ station(dep_info(v))  $\notin$  dom j)
      end,
    ⟨st1, st2⟩ ^ stl' →
      let v1 = j(st1), v2 = j(st2) in
        is_stat_wf_Visit(net, st1, v1) ∧
        dep_info(v1)  $\neq$  nil ∧
        arr_info(v2)  $\neq$  nil ∧
        if station(dep_info(v1)) ∈ dom j then
          line(dep_info(v1)) = line(arr_info(v2)) ∧
          station(dep_info(v1)) = st2 ∧ station(arr_info(v2)) = st1
        else
          station(arr_info(v2))  $\notin$  dom j
        end
      end ∧
      is_stat_wf_station_list(net, ⟨st2⟩ ^ stl', j)
    end
  pre elems stl  $\subseteq$  dom j,

/* each relation statically well-formed */
is_stat_wf_Relations : Visits × Relations → Bool
is_stat_wf_Relations(vs, relns)  $\equiv$ 
  (
    ∀ st : N.Station •
      st ∈ dom relns  $\Rightarrow$ 
      (∀ reln : Relation • reln ∈ relns(st)  $\Rightarrow$  is_stat_wf_Relation(vs, st, reln))
  ),

/* trains in relation visit station */
is_stat_wf_Relation : Visits × N.Station × Relation → Bool
is_stat_wf_Relation(vs, st, reln)  $\equiv$  (∀ tn : Train • tn ∈ trains(reln)  $\Rightarrow$  (st, tn) ∈ dom vs)
end

```

scheme

NETWORK1 =

hide track_route, line_route, lines_in, lines_out **in**
extend extend NETWORK0 **with** ROUTE **with**
class

type Network, Line_type == single | dual

value

stations : Network \rightarrow Station-set,
tracks : Station \rightarrow Track-set,
lines : Network \rightarrow Line-set,
/* tracks and lines are linear routes */
track_route : Track \rightarrow Linear_Route,
line_route : Line \rightarrow Linear_Route,
line_type : Line \rightarrow Line_type,
from_station : Line \rightarrow Station,
to_station : Line \rightarrow Station,
/* each track has a number of possible lines into it and from it;
each such line is associated with a route from the line to the
track or from the track to the line respectively. These routes
will be those, typically through one or more switches, that
station management needs to set to allow trains to move in and
out of stations. Such routes may be (partially) shared between
tracks and lines
*/
lines_in : Track \rightarrow (Route \times Line)-set,
lines_out : Track \rightarrow (Route \times Line)-set

axiom

[is_dual_def] $\forall ln : \text{Line} \bullet \text{is_dual}(ln) \equiv \text{line_type}(ln) = \text{dual}$,

/* different tracks share no units */
[disjoint_tracks]

$\forall net : \text{Network}, st, st' : \text{Station}, tr, tr' : \text{Track} \bullet$
 $\{st, st'\} \subseteq \text{stations}(net) \wedge tr \in \text{tracks}(st) \wedge tr' \in \text{tracks}(st') \wedge tr \neq tr' \Rightarrow$
 $\text{disjoint_routes}(\text{track_route}(tr), \text{track_route}(tr')),$

/* different lines share no units */
[disjoint_lines]

$\forall net : \text{Network}, ln, ln' : \text{Line} \bullet$
 $\{ln, ln'\} \subseteq \text{lines}(net) \wedge ln \neq ln' \Rightarrow \text{disjoint_routes}(\text{line_route}(ln), \text{line_route}(ln')),$

/* tracks and lines share no units */
[disjoint_tracks_and_lines]

$\forall net : \text{Network}, st : \text{Station}, tr : \text{Track}, ln : \text{Line} \bullet$
 $st \in \text{stations}(net) \wedge tr \in \text{tracks}(st) \wedge ln \in \text{lines}(net) \Rightarrow$
 $\text{disjoint_routes}(\text{track_route}(tr), \text{line_route}(ln)),$

/* tracks and routes between tracks and lines share no units */
[disjoint_tracks_and_routes]

$\forall net : \text{Network}, st, st' : \text{Station}, tr, tr' : \text{Track}, route : \text{Route}, ln : \text{Line} \bullet$

```

    {st, st'} ⊆ stations(net) ∧
    tr ∈ tracks(st) ∧ tr' ∈ tracks(st') ∧ (route, ln) ∈ lines_in(tr) ∪ lines_out(tr) ⇒
    disjoint_routes(track_route(tr'), route),

/* lines and routes between tracks and lines share no units */
[disjoint_lines_and_routes]
  ∀ net : Network, st : Station, tr, tr' : Track, route : Route, ln, ln' : Line •
  st ∈ stations(net) ∧
  tr ∈ tracks(st) ∧ (route, ln) ∈ lines_in(tr) ∪ lines_out(tr) ∧ ln' ∈ lines(net) ⇒
  disjoint_routes(line_route(ln'), route),

[lines_connect_stations]
  ∀ net : Network, ln : Line •
  ln ∈ lines(net) ⇒
  from_station(ln) ≠ to_station(ln) ∧ {from_station(ln), to_station(ln)} ⊆ stations(net),

/* routes between tracks and lines do connect those tracks and
   lines;
   note that only the routes have known directions
*/
[tracks_connected_to_lines]
  ∀ net : Network, st : Station, tr : Track, route : Route, ln : Line •
  st ∈ stations(net) ∧ tr ∈ tracks(st) ⇒
  (
    (route, ln) ∈ lines_in(tr) ⇒
    ln ∈ lines(net) ∧
    form_a_route(line_route(ln), route, track_route(tr)) ∧ is_line_to(ln, st)
  ) ∧
  (
    (route, ln) ∈ lines_out(tr) ⇒
    ln ∈ lines(net) ∧
    form_a_route(track_route(tr), route, line_route(ln)) ∧ is_line_from(ln, st)
  )
)

value
  is_line_to : Line × Station → Bool
  is_line_to(ln, st) ≡
    if is_dual(ln) then st ∈ {to_station(ln), from_station(ln)} else st = to_station(ln) end,

  is_line_from : Line × Station → Bool
  is_line_from(ln, st) ≡
    if is_dual(ln) then st ∈ {to_station(ln), from_station(ln)} else st = from_station(ln) end,

  is_line_from_track : Line × Track → Bool
  is_line_from_track(ln, tr) ≡ (∃ route : Route • (route, ln) ∈ lines_out(tr)),

  is_line_to_track : Line × Track → Bool
  is_line_to_track(ln, tr) ≡ (∃ route : Route • (route, ln) ∈ lines_in(tr))
end

scheme

```

```

ROUTE =
  hide
    is_Route, is_cyclic_Route, is_linear_Route, unit_paths_from_route, unit_in_route, first, last, rev
  in
    extend UNIT_STATES with
      class
        type
          Route = { | pl : P* • pl ≠ ⟨ ⟩ ∧ is_Route(pl) | },
          Linear_Route = { | route : Route • is_linear_Route(route) | }

        value
          /* paths involve units with possible states;
             paths are connected and non-cyclic
          */
          is_Route : P* → Bool
          is_Route(pl) ≡
            (
              ∀ i : Nat •
                (i ∈ inds pl ⇒ (∃ u : U, ps : P-set • pl(i) ∈ ps ∧ ps ∈ UΩ(u)) ∧
                 {i, i + 1} ⊆ inds pl ⇒ snd(pl(i)) = fst(pl(i + 1)))
            ) ∧
            ~ is_cyclic_Route(pl),

          is_cyclic_Route : P* → Bool
          is_cyclic_Route(pl) ≡ (∃ i, j : Nat • {i, j} ⊆ inds pl ∧ i < j ∧ fst(pl(i)) = snd(pl(j))),

          is_linear_Route : Route → Bool
          is_linear_Route(route) ≡
            let u = U_from_P(hd route) in is_linear_U(u) end ∧
            (tl route = ⟨ ⟩ ∨ is_linear_Route(tl route)),

          /* from a route calculates units involved and paths through them;
             hence generates required states of units
          */
          unit_paths_from_route : Route → (U × P)*
          unit_paths_from_route(route) ≡
            let p = hd route, t = tl route in
              ((U_from_P(p), p)) ^ if t = ⟨ ⟩ then ⟨ ⟩ else unit_paths_from_route(t) end
            end,

          unit_in_route : U × Route → Bool
          unit_in_route(u, route) ≡ (∃ p : P • (u, p) ∈ elems unit_paths_from_route(route)),

          disjoint_routes : Route × Route → Bool
          disjoint_routes(route, route') ≡
            ~ (∃ u : U • unit_in_route(u, route) ∧ unit_in_route(u, route')),

          /* first connector in a route */
          first : Route → C
          first(route) ≡ fst(hd route),

```

```

/* last connector in a route */
last : Route → C
last(route) ≡ snd(route(len route)),

/* r1 (possibly reversed), r2 and r3 (possibly reversed) can be
   connected into a route
*/
form_a_route : Route × Route × Route → Bool
form_a_route(r1, r2, r3) ≡
  (last(r1) = first(r2) ∨ first(r1) = first(r2) ∧ is_Route(rev(r1))) ∧
   (last(r2) = first(r3) ∨ last(r2) = last(r3) ∧ is_Route(rev(r3))),

rev : P* → P*
rev(pl) ≡ case pl of ⟨⟩ → ⟨⟩, ⟨(c, c')⟩ ^ t → rev(t) ^ ⟨(c', c)⟩ end
end

```

scheme

```

UNIT_STATES =
  extend PATH with
  class
  type Σ = P-set, Ω = Σ-set

  value
  /* possible states of a unit */
  UΩ : U → Ω

  axiom [feasible_states] ∀ u : U, ps : Σ • ps ∈ UΩ(u) ⇒ ps ⊆ UPs(u)
end

```

scheme

```

PATH =
  extend UNIT with
  class
  type P = { | (c, c') : C × C • c ≠ c' | }

  value
  /* paths through a unit */
  UPs : U → P-set

  axiom
  [paths_go_through_units]
  ∀ u : U •
    ∀ (c, c') : P •
      (c, c') ∈ UPs(u) ⇒
        let lcs = lUCs(u), rcs = rUCs(u) in c ∈ lcs ∧ c' ∈ rcs ∨ c' ∈ lcs ∧ c ∈ rcs end,

  [at_most_one_unit_per_path] ∀ u, u' : U, p : P • p ∈ UPs(u) ∧ p ∈ UPs(u') ⇒ u = u'

  value
  fst : P → C
  fst((c, c')) ≡ c,

```

```

    snd : P → C
    snd((c, c')) ≡ c'

    value
      U_from_P : P ≃ U
      U_from_P(p) as u post p ∈ UPs(u) pre (∃ u : U • p ∈ UPs(u))
    end

    scheme
      UNIT =
        class
          type U, C

          value
            /* left and right connectors of a unit */
            lUCs, rUCs : U → C-set,

            UCs : U → C-set
            UCs(u) ≡ lUCs(u) ∪ rUCs(u)

          axiom
            [two_ends] ∀ u : U • lUCs(u) ≠ {} ∧ rUCs(u) ≠ {},

            [ends_disjoint] ∀ u : U • lUCs(u) ∩ rUCs(u) = {},

            [two_unit_connectors] ∀ c : C • card { u | u : U • c ∈ UCs(u) } ≤ 2

          value
            is_linear_U : U → Bool
            is_linear_U(u) ≡ card UCs(u) = 2
          end
        end

```

Showing that the second specification implements the first is trivial:

- *NETWORK1* \preceq *NETWORK0* by construction (extension)
- *TIMETABLE1* does not implement *TIMETABLE0* (it has a stronger parameter) but

```
class object N : NETWORK1 end ⊢ TIMETABLE1(N)  $\preceq$  TIMETABLE0(N)
```

by construction (extension)

- *SCHEDULE1* \preceq *SCHEDULE0* by compositionality of implementation
- *DIST_SCHEDULE1* \preceq *DIST_SCHEDULE0* by compositionality of implementation
- *PROJECT1* \preceq *PROJECT0* by compositionality of implementation

6 Work to be done

There are several things to be done. The final aim is a system involving dispatch units (the nodes of the distributed timetable) for doing rescheduling and a dispatch centre for doing scheduling and distributing the initial timetable to the dispatch units.

6.1 Validation of (re)scheduling functions

The functions defined in *SCHEDULE* need to be validated against the requirements of train dispatchers and amended or augmented as necessary.

6.2 Analysis functions

All the *is_dyn_wf* and *is_stat_wf* functions need to be augmented with corresponding functions generating sets of messages explaining non-well-formedness. Then the original functions are defined as returning **true** when the corresponding functions return empty sets.

6.3 Delegable and distributable functions

The (re)scheduling functions from section 6.1 need to be checked for the conditions (if any) for which they are delegable and the analysis functions from section 6.2 checked for the conditions (if any) for which they are distributable and disjoint. The results need to be compared with the current railway practices and suitable protocols devised for communicating between dispatch units and/or dispatch centres when functions are applied locally but do not meet the requirements for local application.

6.4 Concurrent system

The specification can then be developed into a concurrent system with server processes for each dispatch unit and a dispatch centre, incorporating the protocols devised in section 6.3.