A Theory of Distributing Train Rescheduling

Chris George

United Nations University International Institute for Software Technology
(UNU/IIST), Macau

Abstract. We outline the formal modelling of a software system to support the scheduling and rescheduling of trains. The current (prototype) system supports only centralized rescheduling, but in practice rescheduling is done on an area basis, and a distributed system is required. Developing a distributed system involves the notions of *delegability* of functions to adjust schedules and *distributability* of functions to analyse them for conformance to regulations. We formalize these notions in terms of a more abstract, generic specification and then instantiate this to generate a specification of the distributed system.

1 Introduction

The PRaCoSy (Peoples Republic of China Railway Computing System) project [2] is a collaborative project between the Chinese Ministry of Railways and UNU/IIST. The main purpose is to produce software to assist in the dispatching of trains. Efficient dispatching, especially rescheduling to take account of disturbances to the timetable such as train delays, is critical to optimize the use of the railway resources — track, rolling stock and staff.

In the first phase of the project a formal model of the railway network and of a railway timetable [7, 8] were produced in the RAISE Specification Language (RSL [9]). These were developed (by hand) into a prototype *running map* tool [6] that allows train dispatchers to enter a timetable, graphically display it (figure 1) and graphically or textually edit it. This software is currently under evaluation in China.

To illustrate the problem of rescheduling, note in figure 1 that train Y1 overtakes the slower, stopping train K542 in Danyan. On this route, as is common, trains can only overtake at stations, where there are additional tracks. Now, suppose train Y1 leaves Nanjing 10 minutes late. If we adjust Y1's display accordingly it will be scheduled to leave Danyan at the same time as K542. The tool will report this conflict to the dispatcher, who then has to further adjust the timetable to remove the conflict. One option is to delay K542 in Danyan until Y1 has passed through; another is to delay Y1 a little more and let it overtake K542 in the next station, Changzhou.

In China, dispatching of trains is handled at several levels (figure 2). A *dispatch centre* is in charge of the trains over a large area. Beneath that are several *dispatch units*; each deals with part of the area and is in charge of a number of *stations* that actually dispatch trains according to instructions from their

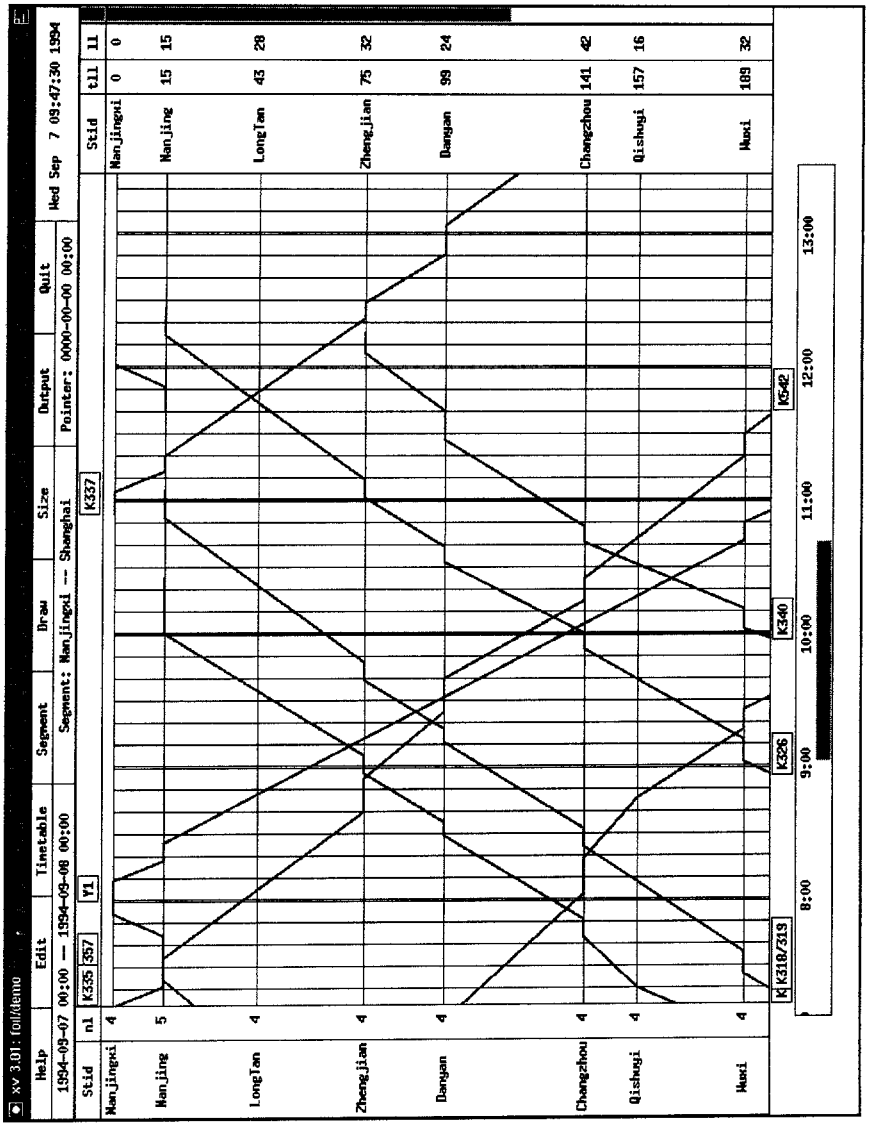

Fig. 1. Running map tool display

dispatch units. The term "station" includes both passenger stations and marshalling yards for the handling of freight trains.[1]

Dispatching is more complicated when there is more than one dispatch unit involved. The discussion earlier about whether train Y1 should overtake K542 in Danyan or Changzhou would be more complicated if these two stations were

[1] In this paper we will for simplicity use only the terminology relevant to passenger trains and passenger stations.

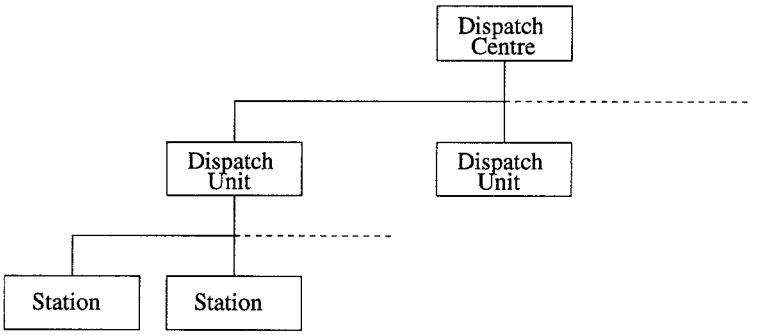

Fig. 2. Dispatch hierarchy

under different dispatch units. The next step in the project is to create a distributed version of the running map tool that can support concurrent rescheduling by dispatch units.

Distribution is largely a problem of dispatch units only having partial knowledge of the current situation and of consistency between the views of the separate dispatch units. The problem is exacerbated in this case since communication between dispatch units may be slow and unreliable and the processing power available may be limited. We are therefore interested in minimizing communication and in efficient algorithms.

There are a large number of different kinds of adjustments we can make to a timetable, ranging from adding or deleting a whole train to changing the platform a train will use at a station. There are also a number of analyses we need to do to check a timetable for consistency and adherence to rules (like the check we used to discover that after the first adjustment to Y1 it was scheduled to leave Danyan at the same time as K542). There are three particular questions we need to answer:

1. Can the distribution between dispatch units (even when their partial timetables are consistent) cause analysis either to fail, i.e. to not detect a violation of some rule, or to detect spurious violations?
2. When does an adjustment affect other dispatch units, i.e. when do we need to communicate to achieve consistency between dispatch units?
3. What analyses need to be redone after an adjustment?

If we can formalize these questions we have a theory of distributing adjustment and analysis functions that we can then apply to the actual ones to be used. We can then be sure that distributed analysis is effective, that after adjustments we communicate sufficient data between dispatch units to achieve consistency, and that we minimise the time taken to analyse adjustments. We achieve this by producing a theory of an abstraction of a timetable and its distribution.

The structure of the rest of this paper is as follows. Section 2 presents the formal model of the railway network and timetables, and the adjustment and analysis functions on timetables needed for dispatching. In particular, it shows how timetables are modelled as maps. Section 3 presents a generic theory of distributing maps and defines a number of functions that model criteria for distributing adjustment and analysis functions over maps. In section 4 the distributed timetable is then defined by instantiating the generic map. Section 5 draws some general conclusions and indicates directions for further work.

2 The railway network and timetables

In this section we present a formal model in RSL of the railway network and timetables, plus the adjustment and analysis functions needed to support the activities of scheduling and rescheduling.

2.1 The railway network

The network consists essentially of stations and lines connecting them. Stations have tracks (usually referred to as platforms in the case of passenger stations). A line goes from one station to another. For any track, there is a set of lines from which it is possible to get to the track, and a set of lines it is possible to reach from the track.

type
  Network, Line, Station, Track,
  Line_type == *up* | *down* | *both*
value
  lines : *Network* → *Line*-**set**,
  stations : *Network* → *Station*-**set**,
  tracks : *Station* → *Track*-**set**,
  from_station : *Line* → *Station*,
  to_station : *Line* → *Station*,
  lines_in : *Track* → *Line*-**set**,
  lines_out : *Track* → *Line*-**set**

The first four types defined here are *sorts*, abstract types. The functions defined (by signature) in the **value** declaration are all *observers* or attributes of these sorts. This style allows us to add later, if we need to, other attributes, such as the length of a track or the possible routes (through switches) between a track and each line it connects to.

The complete model includes more details. The rails are modelled in terms of *units*, the smallest portion of rails controlled by signals. Units may be linear or may model various kinds of switches (points) or crossovers. Lines and tracks are sequences of connected linear units; they may be connected to each other by connected sequences of possibly non-linear units.

We need to add to these type definitions and function signatures a number of constraints to state, for example, that lines do connect actual (different) stations. These are written as axioms, such as:

axiom
[*lines_connect_stations*]
$\forall$ *net* : *Network*, *line* : *Line* •
line $\in$ *lines(net)* $\Rightarrow$
from_station(line) $\neq$ *to_station(line)* $\wedge$
{*from_station(line)*, *to_station(line)*} $\subseteq$ *stations(net)*

Similarly, if a track in a station is reachable from a line (in *lines_in(track)*) then that line must be a line to the track's station (in the right direction if the line is *up* or *down*). And so on.

2.2 Timetables

Timetables are usually presented to passengers as tables indexed vertically by station and horizontally by train with entries giving the times of stops at stations. Timetables needed by railway staff will have more information: the time of trains passing through stations, both arrival and departure times for trains that stop, the tracks trains will use, the lines trains will use, etc.

In addition (and usually implicitly) passenger timetables indicate possible *connections* between trains: possibilities for passengers to change from a train to another one. For railway staff, connections are needed to allow the possibility also of transfer of staff, freight or rolling stock between trains.

Connections are related to particular stations, and we will model the information needed about a train's visit to (or passage through) a station as a *visit*. A map seems a natural data structure for a timetable, but there are several possibilities:

1. The passenger or train driver's view might be

   type TT = (*Train* $\xrightarrow[m]{}$ *Station* $\xrightarrow[m]{}$ *Visit*) $\times$ (*Station* $\xrightarrow[m]{}$ *Connection*-**set**)

2. The station staff's view might be

   type TT = *Station* $\xrightarrow[m]{}$ (*Train* $\xrightarrow[m]{}$ *Visit*) $\times$ *Connection*-**set**

3. Or we can flatten the nested maps in the first two possibilities:

   type TT = (*Station* $\times$ *Train* $\xrightarrow[m]{}$ *Visit*) $\times$ (*Station* $\xrightarrow[m]{}$ *Connection*-**set**)

In the above definitions note that $\times$ binds more tightly than $\xrightarrow[m]{}$ and that $\xrightarrow[m]{}$ associates to the right. How do we choose between these three? The usual criteria are:

- reducing any *invariants* needed to express consistency
- simplifying the functions to read or change the data structure.[2]

The consistency between the connections and visits (a connection may only mention stopping visits) is needed in all three; only the second avoids the need to express consistency between the domains of two maps. For ease of specifying functions on timetables, however, it turns out that the first and third are more convenient.

But we will also need to consider the distribution of timetables. This distribution, as we shall see, is by partitioning the stations according to what dispatch unit they belong to, and the first is not very suitable for this. The second is convenient for distribution but the depth of nesting makes the justifications very complicated. On balance the third version is best.

2.3 Scheduling and rescheduling

Before we can finalize the data structures of timetables we need to consider what functions are required by users, in particular dispatch staff. They have two essential tasks: scheduling and rescheduling.

Scheduling is the construction of a timetable for a particular shift (part of a day). Its inputs are the basic timetable (perhaps the summer timetable for a Saturday) plus additional information relevant to that day: extra trains required, trains affected by shortages or wrong positioning of rolling stock or staff, repair work affecting the network etc. It is done by dispatch centres. Essentially the task is to create and disseminate to dispatch units (and from them to stations) a timetable that is *feasible*. A feasible timetable is one that

- corresponds to the network in using existing and connected lines and tracks that are actually available
- corresponds to timing restrictions like allowing trains sufficient time to travel between stations
- corresponds to physical restrictions like not putting two trains in the same track or in opposite directions on lines usable in *both* directions in overlapping time intervals, or not putting more trains in the same direction on a line in any time interval than the signaling equipment available allows that line to contain
- does not violate various *regulations* like the minimum time allowed between arrivals at the same station.

There is also a requirement that a feasible timetable should generate minimal *disruption* when compared to the basic timetable plus any additional trains. Disruptions are the late arrival of trains and the breaking of connections.

[2] Here simplicity does not mean efficiency when implemented as code. We are concerned with specification, not execution. So simplicity means simplicity and clarity of expression (so that correctness may be checked effectively manually) and giving ease of proof (where correctness is checked formally). The data structure used in the final implemented code may be different.

Rescheduling is the changing of a timetable for a shift during that shift. Its inputs are the current timetable for the shift and actual events like arrivals and departures of trains. Such events may be normal (according to the timetable) or abnormal but anticipated, such as late arrivals or departures, breakdowns, etc. They do not include the abnormal and unanticipated, because physically impossible, like the sudden appearance of a train from nowhere, or the overtaking of one train by another on the same line, or the departure of a train that has not arrived.

It is apparent that scheduling and rescheduling are essentially the same: they start with inputs which include a timetable that may or may not be currently feasible and try to construct a timetable that is feasible with minimal disruption. The time constraints may be tighter for rescheduling, but scheduling is also time constrained: it has to be constructed before the shift starts. It would therefore be convenient if we could use the same data structure and as far as possible the same functions for both tasks; then we can use the same tools for both. This also means that the timetable is the output from both; we do not produce some other "schedule" type. This leads to two requirements for timetables:

1. Rescheduling requires that we can distinguish actual (occurred) events (like arrivals and departures) from scheduled events. Only scheduled events can be subject to rescheduling.
2. Rescheduling requires rapid responses (part of the purpose of introducing computer support) and it must be possible to make decisions about scheduled events in the immediate future while deferring decisions about later ones, even though after making the first change the resulting timetable may be infeasible. This may be unwise on some occasions, but the tools should allow dispatchers to make judgements about what they can sort out later.

The second requirement means that we must not constrain the type TT to be always feasible. So we cannot use the style we did with the network of imposing constraints as axioms (which must always hold). Instead we define some analysis functions that can check if a timetable is feasible and, if not, inform the user why. For example, we must be able to record in the timetable the arrival of a train even if it is later than the (scheduled) departure time. We see that the timetable (and its visual tool the running map) is a means of both (re)scheduling the movement of trains and also analysing the evolving state of the movements for potential problems.

3 Distribution

In this section we present a general theory of distribution of a map. Then we show how this may be applied to a map representing a timetable in section 4.

3.1 Distributing a map

We have already used the notation for maps. A type *Map* mapping elements from a domain type D to a range type R, both sorts, is defined by

type
D, *R*,
Map = *D* $\overrightarrow{m}$ *R*

The main operations available for the type *Map* are the prefix operator **dom** (giving the set of elements in the domain of a map) and application (giving the range value corresponding to a domain value). Application is written like function application; it is only (necessarily) defined for a value in the set returned by **dom**.

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

We can partition the domain type *D* by means of a total function *p_of* from *D* to some "partitioning" type (another sort) *P*:

type
P,
Dist_Map = *P* $\overrightarrow{m}$ *Map*

value
p_of : *D* → *P*,

distribute : *Map* → *Dist_Map*
distribute(*m*) ≡ [*p* ↦ *m* / *partition*(*p*) | *p* : *P*],

partition : *P* → *D*-**set**
partition(*p*) ≡ { *d* | *d* : *D* • *p_of*(*d*) = *p* }

The operator "/" is the "restriction to" operator. "*m* / *s*" is the restriction of the map *m* to those domain elements in the set *s*.

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* → *Map*
merge(*dm*) ≡
  [*d* ↦ *dm*(*p_of*(*d*))(*d*) | *d* : *D* • *p_of*(*d*) ∈ **dom** *dm* ∧ *d* ∈ **dom** *dm*(*p_of*(*d*))]

and it is then simple to justify[3] the theorem that *merge* is the (left) inverse of *distribute*:

∀ *m* : *Map* • *merge*(*distribute*(*m*)) = *m* *(1)*

[3] This and the other theorems mentioned in this paper have been justified using the RAISE justification editor [4].

3.2 Delegability

We now consider functions on a distributed map. Suppose we have some function f that can be applied to a global map (and generating a new 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 diagram in figure 3 "commutes".

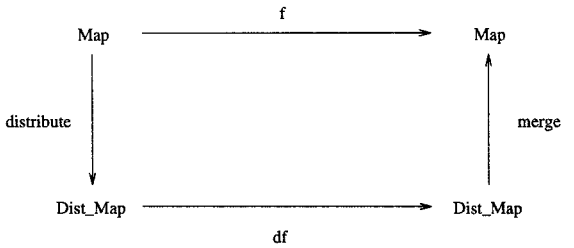

Fig. 3. Correctness of distributed function

That is, we require that

$$\forall\, m : Map \bullet merge(df(distribute(m))) = f(m) \qquad (2)$$

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

$$\forall\, m : Map \bullet distribute(f(m)) = df(distribute(m)) \qquad (3)$$

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

If there is no means of identifying a single value of P from the arguments of the function f, then distributing f must in general involve applying it to all the component maps of the distributed map. Consider, for example, prefixing all telephone numbers 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 in effect applied to the complete map by being applied to just one component map "delegable" since it is possible to "delegate" the responsibility for applying it to just one component.

"Lookup" functions will usually be delegable. Such a function will typically have type

$$D \times Map \rightarrow T$$

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 complete map. That is, we can define a test:

$$
\begin{array}{l}
\mathit{is\_delegable\_lookup} : (D \times \mathit{Map} \to T) \to \mathbf{Bool} \\
\mathit{is\_delegable\_lookup}(f) \equiv \\
\quad (\forall\ d : D,\ m : \mathit{Map} \bullet f(d,m) = f(d,\ \mathit{distribute}(m)(\mathit{p\_of}(d))))
\end{array}
$$

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

$$\mathbf{type}\ \mathit{Fun} = D \times \mathit{Map} \xrightarrow{\sim} \mathit{Map}$$

The symbol "$\xrightarrow{\sim}$" indicates that *Fun* is the type of partial functions, i.e. functions for which application may not be defined for some arguments. A function in *Fun* will typically be associated with a precondition which we can express as a function with type

$$\mathbf{type}\ \mathit{Pre\_Fun} = D \times \mathit{Map} \to \mathbf{Bool}$$

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

- truth of the precondition for the global map
- that distributing and applying the function only to one component gives the same map as applying the function to the global map and then distributing.

The first condition ensures that if we can apply a function locally we could have applied it globally; the second (condition (3) above) says we get the same result from local or global application. Here are the formal definitions:

$$
\begin{array}{l}
\mathit{delegate} : \mathit{Fun} \to D \times \mathit{Dist\_Map} \xrightarrow{\sim} \mathit{Dist\_Map} \\
\mathit{delegate}(f)(d,\ \mathit{dm}) \equiv \mathbf{let}\ p = \mathit{p\_of}(d)\ \mathbf{in}\ \mathit{dm} \dagger [p \mapsto f(d,\ \mathit{dm}(p))]\ \mathbf{end},
\end{array}
$$

$$
\begin{array}{l}
\mathit{delegate} : \mathit{Pre\_Fun} \to D \times \mathit{Dist\_Map} \to \mathbf{Bool} \\
\mathit{delegate}(\mathit{pre\_f})(d,\ \mathit{dm}) \equiv \mathbf{let}\ p = \mathit{p\_of}(d)\ \mathbf{in}\ \mathit{pre\_f}(d,\ \mathit{dm}(p))\ \mathbf{end},
\end{array}
$$

$$
\begin{array}{l}
\mathit{weakly\_delegable} : \mathit{Fun} \times \mathit{Pre\_Fun} \to \mathbf{Bool} \\
\mathit{weakly\_delegable}(f,\ \mathit{pre\_f}) \equiv \\
\quad \mathbf{let}\ \mathit{df} = \mathit{delegate}(f),\ \mathit{pre\_df} = \mathit{delegate}(\mathit{pre\_f})\ \mathbf{in} \\
\quad\quad \forall\ d : D,\ m : \mathit{Map} \bullet \\
\quad\quad\quad \mathit{pre\_df}(d,\ \mathit{distribute}(m)) \Rightarrow \\
\quad\quad\quad \mathit{pre\_f}(d,\ m) \wedge \mathit{df}(d,\ \mathit{distribute}(m)) = \mathit{distribute}(f(d,\ m)) \\
\quad \mathbf{end}
\end{array}
$$

(Note that we have overloaded "delegate".) We use the description "weakly" delegable because it may be that the pre-condition calculated for only one component map is stronger than that calculated for the global map. That is, it might be possible to apply the function to the global map but not to one component

only. If this is not so, i.e. truth of the precondition for the global map implies truth for the component (and the function is weakly delegable) then we say the function is "strongly" delegable:

strongly_delegable : *Fun* × *Pre_Fun* → **Bool**
strongly_delegable(f, pre_f) ≡
  weakly_delegable(f, pre_f) ∧
  let *pre_df* = *delegate(pre_f)* **in**
    ∀ *d* : *D*, *m* : *Map* • *pre_f(d, m)* ⇒ *pre_df(d, distribute(m))*
  end

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

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

Since 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)

The expression *f(d, m)* **post true** above says that *f(d, m)* is (uniquely) defined.

Having formulated this it is possible to justify the expected theorem

∀ *f* : *Fun*, *pre_f* : *Pre_Fun* •
  weakly_delegable1(f, pre_f) ⇒ *weakly_delegable(f, pre_f)* *(4)*

weakly_delegable1 provides us with a more convenient test than *weakly_delegable* for checking that a function is weakly delegable (and for strongly delegable we have only an additional relation between preconditions to justify). We can see theorem (4) as providing a partial decomposition of any proof of a function being delegable.

3.3 Analysis functions

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 component maps and merge the messages that result into the same set of messages as we would get if we analysed the global map. The same set of messages implies that when checking each component separately

- no messages are lost
- no "spurious" messages are generated.

We have another commuting diagram, figure 4.

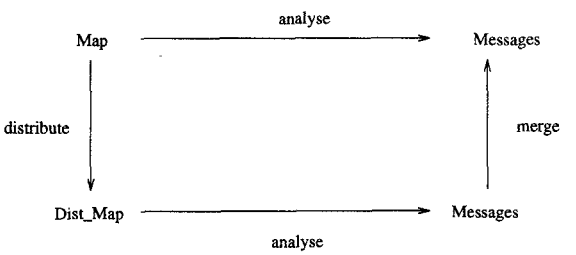

Fig. 4. Distributed analysis

We provide a function *analyse_distributable* to check this requirement:

```
type
  Message,
  Analyse = Map → Message-set
value
  analyse_distributable : Analyse → Bool
  analyse_distributable(analyse) ≡
    (∀ m : Map • analyse(m) = merge({ analyse(distribute(m)(p)) | p : P }))
```

$$
\begin{array}{l}
\mathit{merge} : (\mathit{Message}\textbf{-set})\textbf{-set} \to \mathit{Message}\textbf{-set} \\
\mathit{merge}(\mathit{mss}) \equiv \\
\quad \{\ m \mid m : \mathit{Message} \bullet \exists\ \mathit{ms} : \mathit{Message}\textbf{-set} \bullet \mathit{ms} \in \mathit{mss} \wedge m \in \mathit{ms}\ \}
\end{array}
$$

Suppose we have a current map and a distributable analysis function. We apply a delegable function to the appropriate component map 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

$$
\begin{array}{l}
\forall\ f : \mathit{Fun}, \mathit{pre\_f} : \mathit{Pre\_Fun}, \mathit{analyse} :\ \mathit{Analyse} \bullet \\
\quad (\forall\ d : D, m : \mathit{Map} \bullet \mathit{pre\_f}(d, m) \Rightarrow (f(d, m)\ \textbf{post true})) \wedge \\
\quad \mathit{weakly\_delegable}(f, \mathit{pre\_f}) \wedge \mathit{analyse\_distributable}(\mathit{analyse}) \Rightarrow \\
\quad (\forall\ d : D, m : \mathit{Map} \bullet \\
\qquad \textbf{let}\ \mathit{pre\_df} = \mathit{delegate}(\mathit{pre\_f})\ \textbf{in} \\
\qquad\quad \mathit{pre\_df}(d, \mathit{distribute}(m)) \Rightarrow \\
\qquad\quad \textbf{let} \\
\qquad\qquad \mathit{md} = \mathit{distribute}(m)(\mathit{p\_of}(d)), \\
\qquad\qquad \mathit{msgs} = \mathit{analyse}(\mathit{md}), \\
\qquad\qquad \mathit{md}' = f(d, \mathit{md}), \\
\qquad\qquad \mathit{msgs}' = \mathit{analyse}(\mathit{md}') \\
\qquad\quad \textbf{in} \\
\qquad\qquad (\forall\ \mathit{msg} : \mathit{Message} \bullet \\
\qquad\qquad\quad \mathit{msg} \in \mathit{msgs} \wedge \mathit{msg} \notin \mathit{msgs}' \Rightarrow \\
\qquad\qquad\quad \mathit{msg} \notin \mathit{analyse}(m \setminus \textbf{dom}\ \mathit{md}) \\
\qquad\qquad) \Rightarrow \\
\qquad\qquad \mathit{analyse}(f(d, m)) = \mathit{analyse}(m) \setminus \mathit{msgs} \cup \mathit{msgs}' \\
\qquad\quad \textbf{end} \\
\qquad \textbf{end})
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{analyse\_disjoint} : \mathit{Analyse} \to \textbf{Bool} \\
\mathit{analyse\_disjoint}(\mathit{analyse}) \equiv \\
\quad (\forall\ m : \mathit{Map}, p1, p2 : P \bullet p1 \neq p2 \Rightarrow \\
\qquad \mathit{analyse}(\mathit{distribute}(m)(p1)) \cap \mathit{analyse}(\mathit{distribute}(m)(p2)) = \{\})
\end{array}
$$

Then we know that if a function f is delegable and we apply f to a component map only, then we may apply a distributable and disjoint function *analyse* by applying *analyse* to the component only.

3.4 Partial analysis

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.

In section 1 we posed three questions for distributed timetables:

1. Can the distribution between dispatch units (even when their partial timetables are consistent) cause analysis either to fail, i.e. to not detect a violation of some rule, or to detect spurious violations?
2. When does an adjustment affect other dispatch units, i.e. when do we need to communicate to achieve consistency between dispatch units?
3. What analyses need to be redone after an adjustment?

It should be apparent that the notions of distributability of analysis functions, of delegability of change functions, and of adequacy of partial analysis are formalizations of these questions applied to maps. We have also, implicitly, stated what we mean by a consistent distributed map — one that is a distribution of a global map.

Now we need to instantiate the theory for timetables.

4 Distributing timetables

We earlier decided to use a pair of maps to model a timetable, one for visits and one for connections. We distribute these separately, by instantiating the generic distributed map twice. This generic distributed map is defined in an RSL module *DIST_MAP* which is parameterized by the types D, R, P and *Message*, plus the function *p_of*. For each instantiation we need to provide a parameter object defining these four types and one value.

```
type
  Dispatch_Unit, Message,
  TT :: visits : Visits  connections : Connections,
  Visits = Station × Train ~m~> Visit,
  Connections = Station ~m~> Connection-set
value
  du_of : Station → Dispatch_Unit
object
  A :
    class
      type
        D = Station × Train, R = Visit, P = Dispatch_Unit, M = Message
      value
        p_of : D → P
        p_of((st, tn)) ≡ du_of(st)
    end,
  V : DIST_MAP(A),
  B :
    class
      type
        D = Station, R = Connection-set, P = Dispatch_Unit, M = Message
      value
        p_of : D → P = du_of
    end,
  R : DIST_MAP(B)
type Dist_TT = V.Dist_Map × R.Dist_Map
```

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

We need a theory of which adjustment functions are delegable. Adjustment functions on timetables will apply to one or both of the *visits* and *connections* parts, and we can define the predicate *weakly_delegable* as follows:

```
value
  weakly_delegable :
    (Station × Train × TT ~→ TT) × (Station × Train × TT → Bool) → Bool
  weakly_delegable(f, pre_f) ≡ true
  pre (∃ vf : V.Fun, rf : R.Fun, pre_vf : V.Pre_Fun, pre_rf : R.Pre_Fun •
```

$\forall$ *st : Station, tn : Train, tt : TT* •
pre_f(st, tn, tt) =
(pre_vf((st, tn), visits(tt)) $\wedge$ *pre_rf(st, connections(tt)))* $\wedge$
(pre_f(st, tn, tt) $\Rightarrow$
f((st, tn, tt)) = *mk_TT(vf((st, tn), visits(tt)), rf(st, connections(tt))))* $\wedge$
V.weakly_delegable(vf, pre_vf) $\wedge$ *R.weakly_delegable(rf, pre_rf))*

strongly_delegable is defined similarly.

Note that this definition is only partial; the precondition gives a sufficient but not necessary condition for delegability when the adjustment function is defined in terms of two functions applied separately to the *visits* and *connections* of a timetable. This is true for all the functions currently defined on timetables,[4] but we allow for the possibility of functions not defined in this manner, when we would define the appropriate *delegable* function from first principles in a manner analogous to the definition in *DIST_MAP*.

Some adjustment functions change only the *visits* or *connections* of a timetable. We can consider such functions as defined in terms of a function on each part, one of which is the identity function (with precondition true). It is easy to justify the theorem for *DIST_MAP* (and hence for both the *visits* and the *connections* of the distributed timetable) that an identity function is strongly delegable.

For analysis functions to be distributable and disjoint we proceed similarly:

analyse_distributable : (TT $\rightarrow$ *Message*-**set**) $\rightarrow$ **Bool**
analyse_distributable(analyse) $\equiv$ **true**
pre ($\exists$ *vanalyse : V.Analyse, ranalyse : R.Analyse* •
$\forall$ *tt : TT* •
analyse(tt) = *vanalyse(visits(tt))* $\cup$ *ranalyse(connections(tt))* $\wedge$
V.analyse_distributable(vanalyse) $\wedge$ *R.analyse_distributable(ranalyse)),*

analyse_disjoint : (TT $\rightarrow$ *Message*-**set**) $\rightarrow$ **Bool**
analyse_disjoint(analyse) $\equiv$ **true**
pre ($\exists$ *vanalyse : V.Analyse, ranalyse : R.Analyse* •
$\forall$ *tt : TT* •
analyse(tt) = *vanalyse(visits(tt))* $\cup$ *ranalyse(connections(tt))* $\wedge$
vanalyse(visits(tt)) $\cap$ *ranalyse(connections(tt))* = {} $\wedge$
V.analyse_disjoint(vanalyse) $\wedge$ *R.analyse_disjoint(ranalyse))*

Again the analysis may be calculated from one part only, and we take the other to be the constant function returning the empty set. It is easy to justify that this constant function is both distributable and disjoint.

The need for sufficient but not necessary conditions for distributability and disjointness for analysis functions is more apparent than for adjustment functions. We will, for example, have an analysis function to report inconsistencies between connections and visits. It will report if either of the trains mentioned in a connection for a station does not visit the station, or if the arrival and

[4] This is one of the things that made us choose the pair of maps model.

departure times of the trains do not satisfy the time requirement of the connection. We cannot define this analysis function in terms of separate analyses of the *visits* and the *connections*.[5] We can instead define it for the distributed timetable as the union of the messages for the corresponding function applied to the local timetables. The analysis function is then immediately distributable by definition. It will also be disjoint if each message mentions the station involved.

Examples Consider first some examples of adjustment functions. One function we will need is *report_arrival* to record an actual arrival of a train at a station. This function is delegable, since it only affects one visit and it is possible to find the local map corresponding to the station involved. In fact it is strongly delegable; the precondition will be that the train does in fact visit the station and if this is true globally it is true locally. The precondition is expressed by the function *includes_visit*, so to justify the assertion that *report_arrival* is strongly delegable we assert and then justify the theorem

$$
\begin{array}{l}
\forall\ t : T.Time \bullet \\
\quad strongly\_delegable(\\
\qquad \lambda\ (st,\ tn,\ tt) : Station \times Train \times TT \bullet report\_arrival(st,\ tn,\ t,\ tt), \\
\qquad \lambda\ (st,\ tn,\ tt) : Station \times Train \times TT \bullet includes\_visit(st,\ tn,\ visits(tt)))
\end{array}
$$

Now consider an adjustment function *append_visit* to append a visit to a train journey. This involves adding the new visit and also changing the previous visit to show that the train is scheduled to leave at some time on the appropriate line. Therefore its precondition implies that both stations are in the domain of the map. This function is delegable, but only weakly: the two stations may be in the domain of the global map but not the local one.

An example of an analysis function that is distributable is one that reports inconsistencies between station visits and the network, such as tracks not being in the relevant station, arrival or departure lines not leading to or from the station, or not being connected to the track. It will also be disjoint if the messages mention the station involved.

An example of an analysis function that is not distributable is one that reports if there is no following visit for a non-final visit. If the two stations involved belong to different dispatch units then no message can be generated for the distributed analysis but one might be generated by analysing a global timetable.

5 Conclusions

At the point where this piece of the PRaCoSy project began we had a prototype running map tool to support train dispatchers, and hence had specified and implemented a model of railway network and timetables, together with a

[5] This is a cost of the pair of maps model.

large number of adjustment and analysis functions for timetables. The complete specification, excluding theorems and proofs, is some 1400 lines in 15 modules of RSL, all produced using the RAISE tools [1]. The running map tool does not support the distribution of dispatching across dispatch units, and developing this distribution capacity is the aim of the next part of the project.

The requirements analysis for distribution led to some requirements on the adjustment and analysis functions. We could have formalized these requirement in terms of the full specification, but this would be complicated and tend to obscure the issues in the detail. We chose instead to formalize the requirements in terms of an abstraction, a map, and then to re-specify the timetable as an instantiation of the abstraction. The notions of "delegability", "distributability" and "adequacy of partial analysis" were therefore simply and clearly defined for maps and immediately available for timetables, as they instantiate maps. In particular, we have predicates to check these properties, and are thus able to justify whether a particular timetable adjustment or analysis function satisfies them or not. The full details may be found in [3].

The further steps towards a distributed software system to support scheduling and rescheduling are listed in the following sections.

5.1 Validation of (re)scheduling functions

The adjustment and analysis functions defined for timetables need to be validated against the requirements of train dispatchers and amended or augmented as necessary. Much of this was done in the earlier stages of the PRaCoSy project (by engineers from the Chinese Railways who have worked or are working on the project). We have a number of data flow diagrams showing the communications between dispatch units, dispatch centres and stations; we have definitions of terms that we use (translated into Chinese); etc. And the current prototype has been installed in the first target area for the final system to give us feedback. But it is always essential to review specifications with the customers and, if possible, with future users to validate that what is being produced is what is wanted. Few of these users will be able to read RSL in sufficient detail (though those working at UNU/IIST certainly can). For others we need to "replay" the specification, by writing natural language documents describing them (a kind of "reverse engineering" of requirements) and/or by constructing prototypes.

5.2 Delegable and distributable functions

All adjustment functions need to be checked for the conditions (if any) for which they are delegable and the analysis functions checked for the conditions (if any) for which they are distributable and disjoint. This involves writing the appropriate theorems and then justifying them with the tools. The results need to be compared with the current railway practices and suitable protocols devised for communicating between dispatch centres, and dispatch units when functions are applied locally but do not meet the requirements for local application.

5.3 Concurrent system

The specification can then be developed into a concurrent system with server processes for each dispatch unit and a means, by incorporating the protocols devised in section 5.2, of communication between dispatch units, dispatch centres and stations. An architecture for a concurrent version of the generic distributed map has already been produced [5] using the RAISE method [10] and is expected to be instantiated for the complete system just as the distributed map was instantiated for the timetable.

6 Acknowledgements

This work relies heavily on the earlier work done by UNU/IIST staff Dines Bjørner and Søren Prehn and the Fellows from the Chinese Railways: Dong Yulin, Jin Danhua, Liu Xin, Ma Chao and Sun Guoqin. Hong Mei also contributed during her brief Fellowship.

Useful comments on an earlier draft of this paper were made by Liu Liansuo, Yang Dong and Tomasz Janowski.

References

1. Peter Michael Bruun et al: RAISE tools user guide. Technical report LACOS/-CRI/DOC/4, CRI A/S (1995)
2. Dong Yulin and Dines Bjørner: PRaCoSy: Document deliverables. Technical report dyl/deliv/1, UNU/IIST (1994)
3. Chris George: Distributed train rescheduling. Technical report 42, UNU/IIST (1995)
4. Chris George and Søren Prehn: The RAISE Justification Handbook. Technical report LACOS/CRI/DOC/7, CRI A/S (1995)
5. Hong Mei: Distributed concurrent architecture for rescheduling. Technical report hm/arch/1, UNU/IIST (1995)
6. Liu Xin: A simple running map display tool. Technical report lx/tool/01, UNU/IIST (1994)
7. Søren Prehn: A formal model of the railway application domain system. Technical report sp/5, UNU/IIST (1994)
8. Søren Prehn: A railway running map design. Technical report sp/12, UNU/IIST (1994)
9. The RAISE Language Group: The RAISE Specification Language. Prentice Hall BCS Practitioners Series (1992)
10. The RAISE Method Group: The RAISE Development Method. Prentice Hall BCS Practitioners Series (1995)