Dines Bjørner, Chris George and Søren Prehn **1**

Scheduling and rescheduling of trains

1.1 Introduction: the PRaCoSy project

The PRaCoSy (Peoples Republic of China Railway Computing System) project was a collaborative project between the Chinese Ministry of Railways and UNU/-IIST, the United Nations University International Institute for Software Technology in Macau. The first phase ran from September 1993 to December 1994 and the second from August 1995 to March 1996.

The project aimed to develop skills in software engineering for automation in the Chinese Railways. A specific goal was the automation of the system for monitoring the movement of trains and rescheduling their arrivals and departures to satisfy operational constraints.

1.1.1 The problem

Efficient use of railway resources involves good allocation of resources: railway track, rolling stock and staff. This project was concerned with allocation of track. There are two activities involved:

scheduling : the creation of a timetable for all the trains: passenger, freight, military, etc.

rescheduling : the modification of the timetable to take account of disturbances such as lateness of trains and breakdowns.

A computerized system to support these activities is called a *dispatch* system; *dispatchers* are people responsible for monitoring and coordinating the movements of trains, ensuring that they run as far as possible according to the timetable. They do this by communicating with stations the timetables and adjustments to them. Technically, a *station* is anywhere that there are switches (points) enabling trains to move from one line to another, or where different lines meet. In particular it includes both passenger stations and marshalling yards for freight trains.

In China dispatching is done by dispatchers working in *dispatch units* and communicating with stations and other dispatch units. They make decisions about how to make changes to arrivals and departures of trains in order to minimize the effects of disturbances. Currently the work is entirely manual (pencil and paper)

and slow. The slowness is due to both the manual methods and poor communications.

When dispatchers make decisions to dispatch, delay or reroute trains they need to check a number of things. Is a new route feasible? Are there enough platforms or tracks in each station to hold the trains intended to occupy it at any one time? Are there clashes over the occupations of tracks or of lines between stations? Are the rules in China about minimum separations between departures and arrivals being adhered to? Can the trains make the journeys in the times allowed to them, according the the normal and any special speed restrictions? Etcetera. Knowing and conforming to these rules is part of the skill of the dispatcher, but also something the computer can check more rapidly and more definitely, allowing the dispatcher to concentrate on the tactical decisions of what changes to try.

It might seem that dispatching is safety-critical in keeping trains safely separated. This is in fact not the case: there are other safety systems to prevent accidents. But a timetable that predicts an impossible future is of little use; it will simply cause more rescheduling to be necessary as the problems become manifest.

The area chosen for the initial stage of the project was the 600km line between Zhengzhou and Wuhan. This includes part of the main north-south line between Beijing and Guangzhou (Canton) and is one of the busiest railway areas in China. It is also a critical national infrastructural resource. The area includes 8 dispatch units and 90 stations. See Figure 1.1. It is not a toy example!

1.2 The running map tool

Figure 1.2 shows the prototype *running map* tool that was produced in phase 1 of the project.
This copies closely the large sheets of paper that are used currently by dispatchers. Stations are listed vertically, time passes horizontally, and the paths of trains are shown in the central area. Currently the display is showing part of a timetable for trains running between Nanjingxi and Shanghai (the southern portion of the line from Beijing to Shanghai). Consider train Y1, due to depart from Nanjing at 08:28. Note that it is due to overtake train K335 in Danyan. Now suppose Y1 leaves Nanjing a few minutes late. There are several possibilities:

- Y1 may be able to travel more quickly than timetabled and pass through Danyan on schedule;
- Y1 may pass through Danyan only a very short time before K335 is due to leave; K335 will then be delayed;
- Y1 may pass through Danyan just after K335 has left and (unless they are or can be put on different lines) Y1 will be delayed. The extent to which Y1 is delayed will depend on the signalling, and hence the train capacity, of the line between Danyan and Changzhou.

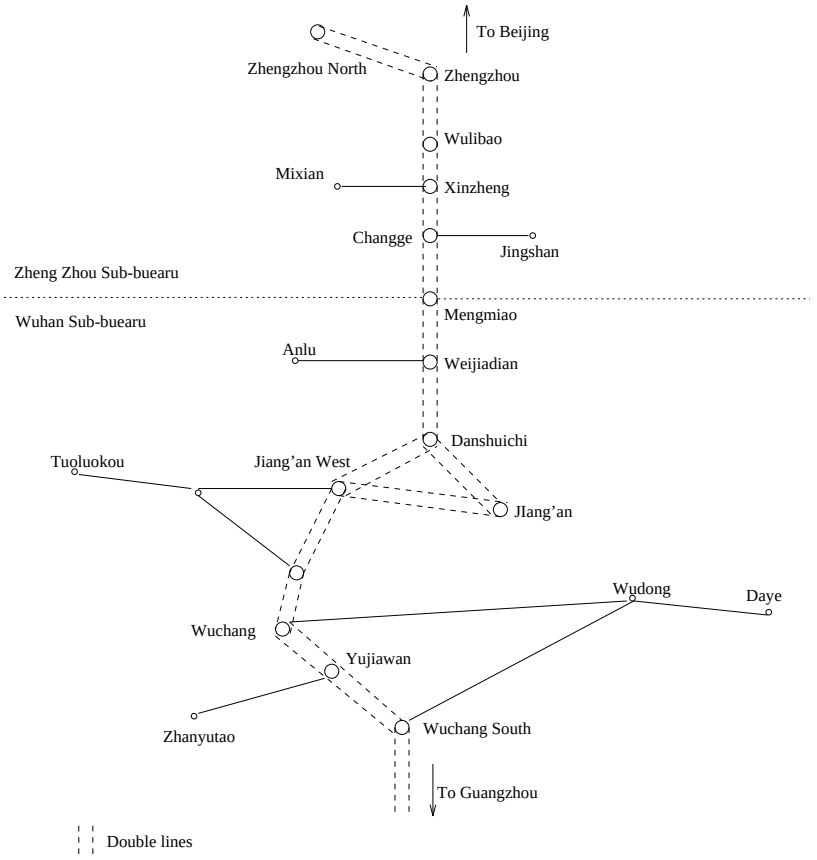

Figure 1.1: Zhengzhou — Wuhan line map

The dispatcher has to decide which of these will occur and react accordingly. If K335 is not a passenger train it might be possible to dispatch it from Danyan early, or even cancel its stop there, so that Y1 can overtake it it at Changzhou (which might also mean changing K335's track at Changzhou). Or K335 can be told to wait at Danyan until Y1 has passed and, perhaps, be delayed at Changzhou but be back on time at Wuxi since it seems to travel comparatively slowly from Changzhou to Wuxi.

In this case the dispatcher has perhaps nearly an hour to make his decision, work out the detailed adjustments and transmit them, but this is unusually long. The display shown here is also unrealistically sparse; it only shows a typical timetable for passenger trains.

The dispatcher needs information about how fast trains are allowed to travel on different lines, on the tracks available at stations, on the lines available (including the possibility of switching a train to the "opposite" line), on the capacity of lines,

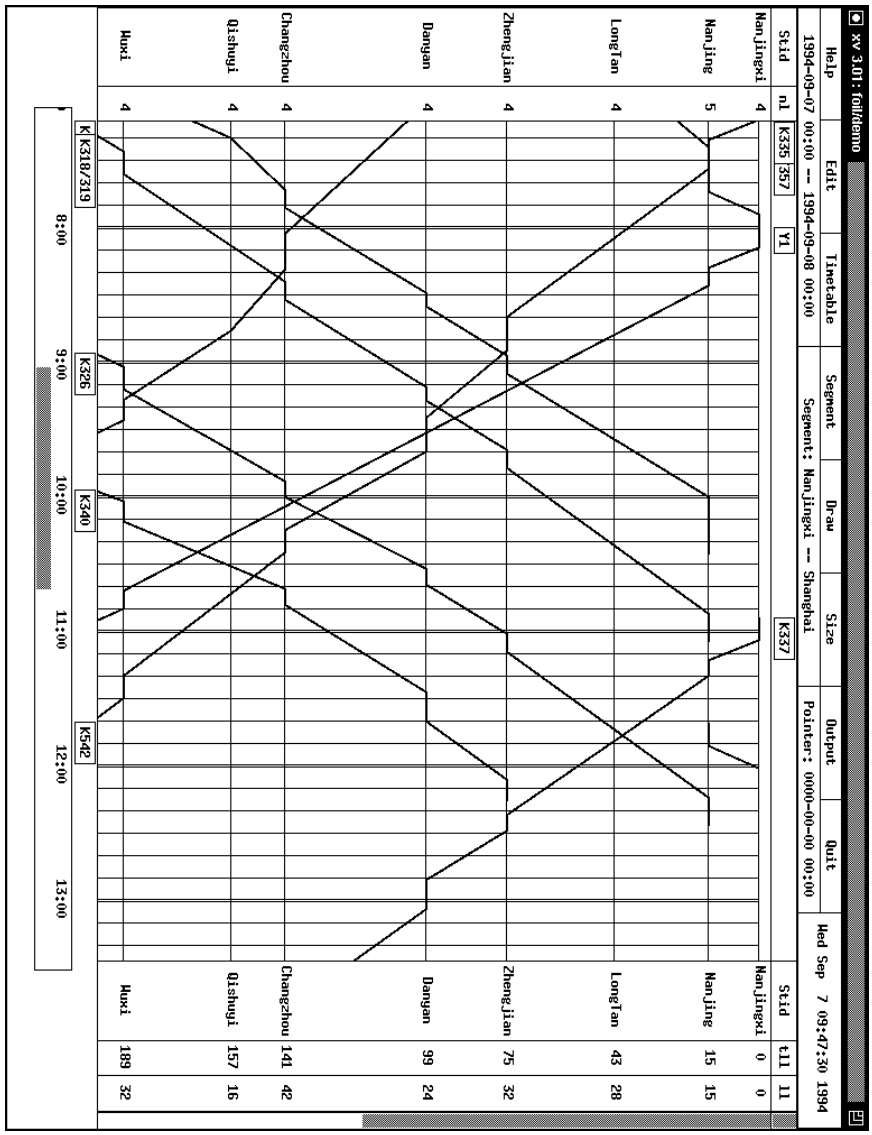

Figure 1.2: Running map tool display

on the relative priorities of trains, etc.

The mode of operation of the tool for rescheduling is that the dispatcher can make adjustments either graphically (by clicking and dragging on the display) or textually (by clicking on a train identifier icon to bring up a textual timetable for the train and editing it). If such an operation breaks one of the "rules" (see Section 1.4.5) then a pop-up window will appear describing the infraction. When the dispatcher is happy with the adjustments they can be "committed" and the new

timetable output.

The same tool can be used for scheduling — the creation of timetables. This is done by starting with the "basic" passenger timetable for 24 hours for the current period, such as Summer, and then adding all the freight and additional trains required for that day. This is then split into 12-hour *shift plans* and then again into 3-hour *stage plans*. A dispatcher is typically operating one stage plan and also reflecting "knock-on" effects into the next one.

The display shows one *segment*, which is a set of stations connected linearly by lines. The area covered by a dispatch unit will typically involve several segments. So, for example, train K340 terminates at Zhengjian but train K326 continues from Nanjing on a different segment. The textual display for the trains show this. It is possible to switch the display to different segments.

This prototype tool is only intended to aid planners and dispatchers. It does not try to find solutions to problems. It was felt that such a "passive" tool would be much more likely to find acceptance initially than one that tried to take over the existing dispatching job. It was also felt that it would take some time to understand the tactics used by actual dispatchers, and to create a tool whose proposals would be respected by them.

1.2.1 Methodological approach

PRaCoSy is one of UNU/IIST's *advanced development projects*. In these projects the work takes place at UNU/IIST in Macau and is done by "fellows" from one or more developing countries trained and supervised by UNU/IIST staff. In this project the fellows were (except for one) software engineers from the Chinese Railways.

The purpose of such projects is both to train the software engineers involved and also to improve the software capability of their institution. A main component in this capability is the development of *domain models* in their area of interest — railways in this case. So a major part of the project was devoted to domain analysis and modelling of railways.

Work done by UNU/IIST involving public funds is automatically public domain. So software production within such projects is limited to prototype, demonstrator software.

Three stages were therefore followed in phase 1 of the project:

Domain analysis : thorough understanding and documentation of the components of a railway

Requirements capture : documentation of the various functions required to support scheduling and rescheduling

Software development : production of a prototype tool

In the three Sections 1.4, 1.5 and 1.6 we describe these three stages.

Phase 2, involving new Fellows had two main tasks:

1. following evaluation of the prototype running map, to consider and specify the changes needed to make a tool that could actually be used by dispatchers;

2. to consider the problems of a distributed running map, allowing for dispatchers and dispatch units to operate concurrently while maintaining a consistent view of the schedule and adjustments to it.

Phase 2 is described in Section 1.7.

In Section 1.8 we draw some conclusions.

1.2.2 Levels of formality

Throughout the project the approach was *formal* using the RAISE specification language and method [RAI92, RAI95]. There is a variety of ways in which formality may be applied to software development:

- The "lightest" approach is to write an initial specification and then use it to produce programming language code.
- A "heavier" approach is to write the initial specification and then to *refine* it in one or more steps into a more concrete specification before producing a program from it. This approach gives the opportunity to:
  - *prove* the refinement steps are correct, the "heaviest" and most expensive approach, or to
  - *justify* the refinements wholly or partly informally, the "rigorous" approach [Jon80].

There are also other variations possible. A common tactic is to choose particularly critical properties of a system and prove only those for the initial specification and refinements. Safety-critical properties are often tackled in this way.

Common to all these approaches is the pivotal role of the initial specification. It is this that is meant to describe the domain and the system requirements, and it must be shown to be correct with respect to the client's or customer's notions of what they want. Showing this we call *validation*. Validation is inevitably informal (assuming your client is not able to present you with a formal specification) because their requirements are written in natural language, which is inevitably ambiguous and typically incomplete in some respects and inconsistent in others.

In this project we adopted the "light" approach, which we saw as most appropriate for a system that was not safety-critical. It therefore differs from examples like [DM94, Han94, OH95, Sim94]. (Although a description of a system to monitor rail freight [DPdB95] points out that systems whose failure is expensive may also be considered critical.) We also wanted to adopt a "rapid prototype" style of development. Such a style is particularly appropriate when you hope to provide

computer support for an activity previously done manually, and where user acceptance may be part of the problem. "The tool must be acceptable to users" is hard to formalize! A brief introduction to the RAISE Specification Language and the style adopted is given in Section 1.3.

We believe that much of the advantage of formality lies in creating the initial specification. It serves to isolate and clarify the important concepts, to make them amenable to formal or informal but still precise analysis, and also to clarify those aspects which are not or cannot be formalized, the "non-functional" requirements like acceptability. A development plan can then make sure that these are dealt with in some way. (A general discussion on non-functional requirements and how to deal with them is beyond the scope of this chapter.) Getting more out of formality, proving critical properties or doing refinement, gives further confidence in correctness but at substantial cost: there is a law of diminishing returns.

1.3 The RAISE Specification Language

The RAISE Specification Language (RSL) [RAI92] is a "wide spectrum" language: it is possible to describe applicative or imperative, sequential or concurrent systems. The normal style proposed in the RAISE method [RAI95] is to start with an abstract, applicative sequential specification and to develop this to a concrete specification, initially still applicative and then, usually, imperative and perhaps concurrent. Phase 1 of this project adopted the "light" approach referred to previously. It only used the applicative sequential style and was rather more concrete than a style one would adopt using a "heavier" approach. Phase 2 used a more abstract approach to analysing the distribution of the running map tool and also developed a concurrent specification to describe an architecture for it.

RSL specifications are collections of modules, usually **scheme**s which are named (and possibly parameterized) **class** expressions. We do not present here the complete specification; that for phase 1 can be found in [Pre94] and for phase 2 in [Don96]. Both are available via the UNU/IIST home page http://www.iist.unu.edu. Neither do we show here the division into modules. Most modules have a particular "type of interest" and provide functions to create, modify and observe values of this type. There are modules describing the type *Track*, then one using *Track* to define *Station*, one defining *Line*, then one using *Station* and *Line* to define *Network*, and so on.

A typical applicative class expression contains one or more **type** declarations, one or more **value** declarations for defining constants and functions and perhaps some **axiom** declarations containing axioms used to constrain the values.

1.3.1 Types

Type declarations may be "abstract" or "concrete". An abstract type, also termed a "sort", is just given a name. If we declare

type *Date*

then *Date* is a sort. We know nothing about its structure, about how dates will be represented. It might later be modelled as a natural number (the type **Nat**) interpreted as days since some base date, or a record of day, month and year.

If *Date* is defined as a sort it is still possible to constrain it to say, for example, that *Date* values are totally ordered. We might do this by introducing and operator and some axioms:

value
  $\leq$ *: Date* $\times$ *Date* $\rightarrow$ **Bool**
axiom
  [*reflexive*]
    $\forall$ *d : Date* $\cdot$ *d* $\leq$ *d,*
  [*transitive*]
    $\forall$ *d1,d2,d3 : Date* $\cdot$ *d1* $\leq$ *d2* $\wedge$ *d2* $\leq$ *d3* $\Rightarrow$ *d1* $\leq$ *d3,*
  ...

In the phase 1 specification only a few rather uninteresting types, like identifiers for tracks, stations, trains, etc. were defined as sorts, and no axioms were given. In this case all we have for values in the type are equality and inequality (with the standard congruence properties).

A "concrete" type is defined by being equal to some other type, or a type expression formed from other types. For example

type
  Year = **Nat**,
  Month = {| *n :* **Nat** $\cdot$ *n* $\geq$ *1* $\wedge$ *n* $\leq$ *12* |},
  Day = {| *n :* **Nat** $\cdot$ *n* $\geq$ *1* $\wedge$ *n* $\leq$ *31* |},
  Date = {| *d : Year* $\times$ *Month* $\times$ *Day* $\cdot$ *is_day(d)* |}

defines *Date* concretely in terms of tuples of *Year*, *Month* and *Day*. **Nat** is built in, together with literals like *0* and *1*, plus standard operators like $+$ and $\leq$. *Month*, *Day* and *Date* are all subtypes: *is_day* will be a function defined elsewhere (to deal with months of less than 31 days and with leap years). *is_day* has result type **Bool**, and so might be termed a predicate: predicates are not distinguished from functions.

$\times$ is a type constructor. Others used here are **-set** (finite power set), * (finite sequence), $\xrightarrow[m]{}$ (finite map), $\rightarrow$ (total function) and $\xrightarrow{\sim}$ (partial function). Each has a number of corresponding operators, such as $\in$ (membership) and $\subseteq$ (subset) for sets, **dom** (domain) and **rng** (range) for maps. Function, map and list (index) applications are written by enclosing arguments in brackets, as in *is_day(d)*.

We also commonly use "variant" and "record" types. Alternatives to some of those above are

type
Month == *Jan* | *Feb* | ... | *Dec*,
Date′ :: *year* : *Year* *month* : *Month* *day* : *Day*,
Date = {| *d* : *Date′* • *is_date(d)* |}

Here *Month* is a variant (in this case just like an enumerated type in other languages: more complicated variants, including recursive ones, are possible). The ellipsis ..., used here for convenience, is not valid RSL. *Date′* is a record. It is much like the tuple used earlier for *Date* but allows convenient extraction of components, by "destructors" like *year*. Destructors are functions, so if *d* is a *Date′*, *year(d)* is its year.

1.3.2 Constants and functions

Constants and functions are values, and must be given at least "signatures" — names and types:

value
start : *Date*,
next : *Date* → *Date*

In a concrete style we often give "explicit" definitions for values. So we might write

value
start : *Date* = *mk_Date′*(0, Jan, 1),
tomorrow : *Date* → *Date*
tomorrow(d) ≡
if *end_of_month(d)* **then** ...
else *mk_Date′*(*year(d)*, *month(d)*, *date(d)* + 1)
end

mk_X is the "constructor" for a record type *X*.

More implicit styles are also possible. For example, one might define *yesterday* as the left inverse of *tomorrow* by a postcondition:

value
yesterday : *Date* $\xrightarrow{\sim}$ *Date*
yesterday(d) **as** *d′*
post *tomorrow(d′)* = *d*
pre *d* ≠ *start*

Or we might use an axiom:

value
$yesterday : Date \xrightarrow{\sim} Date$
axiom
$\forall\ d : Date \cdot yesterday(tomorrow(d)) = d$

If we are doing more formal development we might start with the axiom or post-condition for *yesterday* and later devise the explicit algorithm and then prove it satisfies the earlier specification. This is a simple example of refinement.

1.3.3 Benefits

All this looks like rather like functional programming, and, apart from the axioms (which were rarely used in this "light" style), directly implementable. So why bother with a specification?

The built-in data structures are much more convenient to use than those available in a language like C. This allows very rapid modelling of the domains involved and convenient and terse expressions of the functions. These are easy to read, easy to discuss with others, and easy to modify. And it is also easy to express critical properties, either as theorems or as earlier specifications, and then prove them (which is not always so easy!). Some of this would also be true in a functional programming language, or in one having the built-in types as generic modules. But the ability to state and reason about critical properties (informally or with a proof tool) would still be lacking.

Second, there are functions even in this style which are not immediately executable but which are expressible as simple specifications. For example, suppose we want to state the property that in a timetable, if a train stops it does so for at least *min_stop* minutes. Assume the from a station visit *STV* we can extract or calculate an arrival time *arr* and a departure time *dep*. We can define a predicate on a timetable of type *TT*:

value
$min\_arr\_dep\_separation : TT \to \mathbf{Bool}$
$min\_arr\_dep\_separation(TT) \equiv$
$\quad (\forall\ stv : STV \cdot is\_in(stv, tt) \Rightarrow$
$\quad\quad dep(stv) = arr(stv) \vee$
$\quad\quad dep(stv) - arr(stv) \geq min\_stop)$

This specification is not executable because of the universal quantification. The specification does not say how we extract all the station visits from the timetable. Instead it is expressed at the level of the requirement. Hence it is easy to validate against the requirement. It is important that the initial specification has this property of being "at the appropriate level" for validation, for checking that it meets the requirements. If we are forced to write something executable, and give thought to the algorithm as well as the condition, this makes things much more difficult.

Neither does this specification say what we do if the timetable does not meet the requirement. Presumably we would like the final program to say something rather more informative than "false".

We will see in Section 1.6 how the problems of algorithm and message generation were dealt with when we translated to C.

We shall also see in Section 1.7.3 how the specification in terms of such predicates over timetables led to a completely different style of implementation, using constraint propagation.

We are not advocating such a style for all projects. Critical applications will need more statements of even higher level properties and hence a greater degree of abstraction and of proof. We are trying to show how a "light" approach can be extremely valuable in obtaining a precise definition of the problem domain and in capturing the main requirements.

A more abstract approach is also useful in tackling issues that we want to separate from the details of the example being dealt with. The work done in phase 2 on distributing maps, described in Section 1.7.2, is an example of this.

1.4 Domain analysis

The purpose of domain analysis is to understand and document the components of the system and its environment. So in our case we ask immediately *What is (re)scheduling?* This leads immediately to descriptions in terms of *running maps*, *timetables* and railway *regulations*, which in turn involve terms like *network* of *lines* and *stations* and *time intervals*. To understand these terms in the fullest sense we need assurance that what we think they mean corresponds to what our customers think they mean, and when our customers speak a different language and come from a different cultural tradition it is particularly critical. Even if we were experts in railways there is a significant, perhaps even enhanced, danger that the differences in railway cultures will cause problems. In phase 1 the fellows were not experienced in dispatching.

This problem was tackled from two directions: informal and formal.

1.4.1 Informal description

The informal domain descriptions involved several components. They were intended for domain experts, i.e. railway staff, especially dispatchers, rather than computing experts:

Synopsis : a summary of the domain;

Narrative : a more detailed explanation of the domain in terms of its *components* and *processes*;

Terminology : a list of technical terms and their definitions.

In fact the first document in this group was translated from Chinese; it was the first "statement of requirements". Then we produced a synopsis and narrative and effectively "replayed" our understanding of (currently relevant) parts of these requirements. At the same time the terminology document was written in English and then translated to Chinese so that it could more effectively be checked by people in China.

1.4.2 Formal description

At the same time as trying to capture the domain in natural language, a formal model was constructed. This formal model was be the basis for the requirements capture and software development, and hence the basis for its correctness (or otherwise). There were two components:

Domain specification : specifying the intrinsic domain notions in RSL [RAI92];

Data flow diagrams : recording basically the *organization* of the activities involved in (re)scheduling. Figure 1.3 is an example.

The labelled arrows were accompanied by descriptions of the data. E.g. E1 is "Station states; track states; time traces". Time traces are relevant events like train arrivals and departures, with times.

We did not claim any mathematical semantics for our data flow diagrams; they should perhaps be termed "semi-formal" in that they almost certainly could be formalized [LPT94] but we did not want to use them in a formal way. That is, we had no intention of *reasoning* that our system would be correct with respect to them.

1.4.3 Aspects of domains

In doing domain analysis we consider and document a number of aspects:

Intrinsics : The essential technical aspects of the domain — in our case such things as lines, stations, trains, timetables etc.

Support technology : The technical aspects of the domain that depend on a particular technology and which may change with time. Thus particular signalling and interlocking mechanisms would be in this category.

Regulations : The regulations that affect the system, i.e. that it must conform to or for which breaches must be detected and handled in a special manner. This is obviously important for railways, and would be important in any safety-critical system.

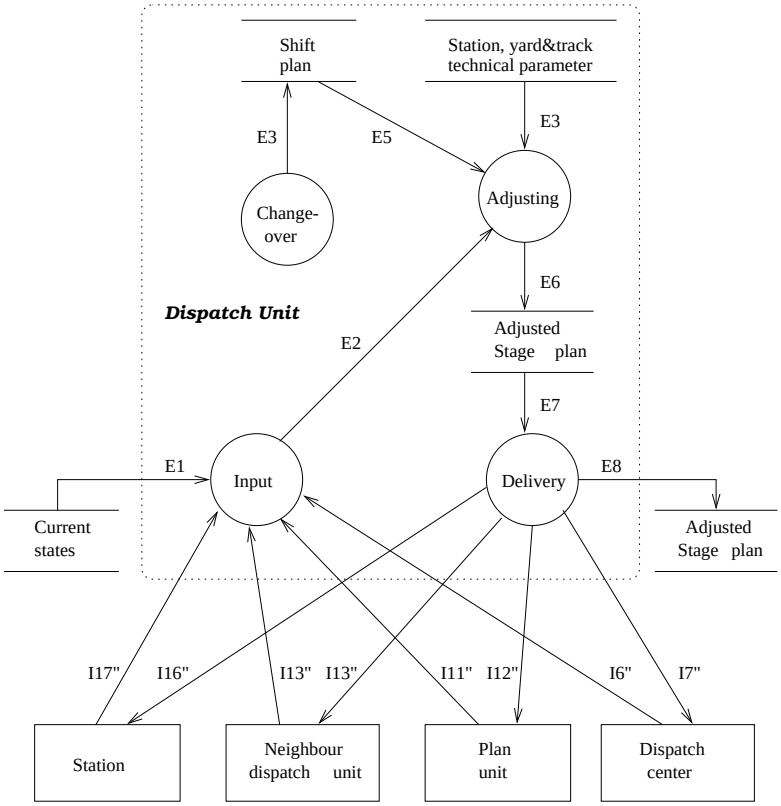

Figure 1.3: Data flow diagram for stage plan adjustment

Staff : The maintainers and (for safety-critical systems, for example) its inspectors or people who license it for use.

Clients : The users of the system.

Computing and communications platforms : Details of the hardware on which the system is to run or with which it is required to interact.

The intrinsics are documented by creating a formal description, a specification of the system. Support technology aspects may be captured by extensions of the intrinsics, but in general we try to avoid this, to make at least the initial specification independent of them. Regulations should be captured formally. Other aspects are typically mostly informal; they are (like support technology) likely to give direction later in the development but not be captured in the initial specification, which should be high level and abstract.

Why do we stress the idea of domain analysis prior to requirements capture? These notions are normally regarded as part of requirements capture. There are

two reasons:

- Domain analysis involves understanding the environment of the system as well as the system itself. That is, we look more widely than the boundaries of the system; we try to understand the world (including the human culture) within which it will operate. Only when we have thoroughly understood this do we attempt to define all the processes in this domain, the facilities our system will provide. Defining these facilities (again mostly formally) will be the requirements capture step.

- Domain analysis is often wider than the immediate concerns of the current system: it can be the basis of other systems in the same domain. For example, the model of lines and stations we developed for the running map tool provided the basis for work developing tools for station management [Yul95]. This has obvious reuse advantages for anyone hoping or planning to produce further software in the domain. Using the same underlying domain model also makes it much easier to ensure that systems will interoperate correctly.

  In fact the model used to describe station management needed more detail about switches and crossovers, about the detailed routes between lines and tracks. We defined a notion of a *unit* as a piece of track with *connectors*. Lines and tracks are constructed from linear (two-connector) units. Switches and crossovers have three or more connectors. Units have possible paths through them (so we can distinguish switched and non-switched crossovers, for example) and also states (so that we can model the changing of a switch). The notions of line and track used in the model presented here can be easily extended with information about their constituent units.

1.4.4 Specification of the railway network

The *network* consists of *stations* connected by *lines*. A station consists essentially of a number of *tracks*. Figure 1.4 shows a station with five tracks.

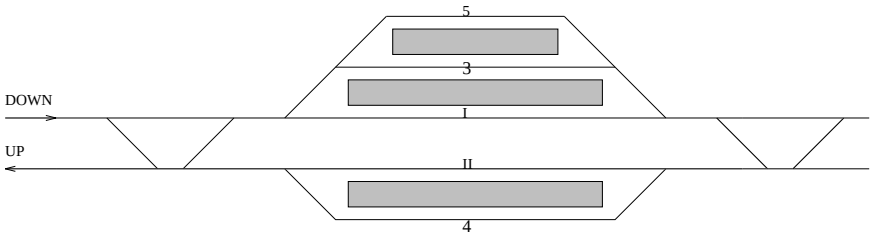

Figure 1.4: Example station layout

The Roman numbering of the "through" tracks used for non-stopping trains, and the even/odd numbering for up/down lines are Chinese conventions.

1.4.4.1 Tracks

For each track there are number of lines from which the track can be reached, and a number of lines that can be reached from the track. Tracks also have lengths and types. Stations, lines and tracks all have identifiers. We define the type *TR* to model tracks:

type
TR :: flns : LNid-**set** *tlns : LNid*-**set** *t : TRtype lng : LNG,*
TRtype == *LINE* | *SIDING* | *PLATFORM* | *FREIGHT,*
LNG = **Nat**,
LNid

This and the rest of the specification were written in RSL, and the RAISE tools [B$^+$95] were used to type check them and pretty print them.

TR is a record with four components: the possible incoming lines *flns*, outgoing lines *tlns*, type *t*, and length *lng*. *TRtype* is a variant type containing four constant values.

1.4.4.2 Stations

The station type *ST* is then a finite map (in general a many-one relation) of track identifiers to tracks:

type $ST = TRid \xrightarrow{m} TR$

1.4.4.3 Lines

Lines go from one station to another, have a type, a length and a maximum speed:

type
$LN' :: s1 : STid \;\; s2 : STid \;\; lt : Ltyp \;\; lng : LNG \;\; sp : SP,$
Ltyp == *UP* | *DOWN* | *BOTH,*
SP = **Nat**,
$LN = \{\!|\; ln : LN' \cdot s1(ln) \neq s2(ln) \;|\!\}$

We expressed the requirement that lines link different stations by use of a subtype definition for the type *LN*.

1.4.4.4 The network

The network consists of a collection of stations, modelled as a map, and a collection of lines, also modelled as a map. But there is a convention that for any two stations connected by one or more lines, one direction between them is *UP* and the other is *DOWN*. (The running map display also follows this convention in its orientation.) We want to observe this convention in the way we label lines by their

end stations: going *DOWN* from *s1* to *s2*. So if there is a line from one station to another then any lines between these two stations will have the stations in the same order, i.e. none will have them in the opposite order:

```
type
  ST_m = STid m→ ST,
  LN_m′ = LNid m→ LN,
  LN_m = {| lnm : LN_m′ • is_wf_LN_m(lnm) |}
value
  is_wf_LN_m : LN_m′ → Bool
  is_wf_LN_m(lnm) ≡
    (∀ ln, ln′ : LN •
      {ln,ln′} ⊆ rng lnm ⇒ (s1(ln),s2(ln)) ≠ (s2(ln′),s1(ln′)))
```

The network is the combination of the line map and the station map, modelled as a cartesian product (tuple), but with a number of well-formedness conditions:

```
type
  NW′ = ST_m × LN_m,
  NW = {| nw : NW′ • is_wf_NW(nw) |}
value
  is_wf_NW : NW′ → Bool
  is_wf_NW(stm, lnm) ≡
    (∀ ln : LN • ln ∈ rng lnm ⇒ {s1(ln),s2(ln)} ⊆ dom stm) ∧
    (∀ s : STid • s ∈ dom stm ⇒
      lnstoST(s,lnm) ∩ lnsfromST(s,lnm) = {} ∧
      (∀ tr : TR •
        tr ∈ rng stm(s) ⇒
          flns(tr) ⊆ lnstoST(s,lnm) ∧ tlns(tr) ⊆ lnsfromST(s,lnm)))
```

The well-formedness conditions for the network are:

- the stations at the ends of lines are in the station map, and
- for each station in the station map:
  - the lines into the station are disjoint from the lines out of the station, and
  - for each track in the station, the lines into it are a subset of the lines into the station and the lines from it are a subset of the lines from the station.

The auxiliary function *lnstoST* is defined as follows:

```
value
  lnstoST : STid × LN_m → LNid-set
```

lnstoST(s,lnm) ≡
  { *lnid* | *lnid : LNid* • *lnid* ∈ **dom** *lnm* ∧ *is_lnto(lnm(lnid),s)* },

is_lnto : LN × *Stid* → **Bool**
is_lnto(ln, s) ≡
  let *mk_LN′(s1,s2,lt,_,_) = ln* **in**
    (lt = DOWN ∧ *s2 = s)* ∨ *(lt = UP* ∧ *s1 = s)* ∨
    (lt = BOTH ∧ *s* ∈ {*s1,s2*}*)*
  end

lnstoST defines the lines into a station using a set comprehension: it returns those lines in the line map that go into the station. A line goes into a station *s* if it is a *DOWN* line and its second end station is *s* or if it is an *UP* line ...

lnsfromST is defined similarly.

1.4.5 Specification of timetables

The following concepts were defined:

Station visit : A record of station and track identifiers, arrival and departure times, optional departure line, arrival and departure lengths of trains. The well-formedness condition is that the arrival is not earlier than the departure (equality indicating passage through without stopping). Time is modelled by a type *T*, a subtype of **Nat** including those times in minutes that represent dates from the (the arbitrarily chosen) beginning of 1993 to the end of 2399:

type
  STV′ :: s : STid tr : TRid a : T d : T dln : OptLN al : LNG dl : LNG,
  STV = {| *stv : STV′* • *d(stv)* ≥ *a(stv)* |},
  optLN == nil | *mk_LN(l : LNid)*

Journey : A sequence of station visits. The well-formedness condition is that the sequence is non empty, departure at one station precedes arrival at the next, and only the last visit may have no departure line.

type
  J′ = *STV**,
  J = {| *j : J′* • *j* ≠ ⟨⟩ ∧ *is_wf_J(j)* |}
value
  is_wf_J : J′ → **Bool**
  is_wf_J(j) ≡
    (∀ *i :* **Nat** •
      {*i,i+1*} ⊆ **inds** *j* ⇒
        a(j(i+1)) > *d(j(i))* ∧ *(dln(j(i))* = *nil* ⇒ *i* = **len** *j))*

There is an additional function expressing the well-formedness of a journey with respect to a network. It requires that the stations exist, the track exists in the station, the departure and arrival lengths are not less than the track length, and (for visits apart from the last) the departure line is a line from the station to the next, goes from the track, leads to the next track, the departure length equals the next arrival length, and the travel time is consistent with the maximum speed and length of the line.

Segment : A non-empty sequence of stations. The well-formedness condition is that the adjacent stations are connected by at least one line, and the segment is listed in the *UP* order.

Timetable : A map from train identifiers to journeys. Train identifiers include a date as well as a (sort) identifier so that we can distinguish between the same train on different days.

type
TNid :: id : TNid_ dt : Date,
TNid_,
$TT = TNid \xrightarrow{m} J$

A "basic" 24-hour timetable, such as the Summer one, has all dates set to zero.

We do not use a subtype for well-formedness of timetables since we expect that during adjustment they will temporarily be ill-formed. Instead we specify a function that can be used to check them.

There are three kinds of well-formedness conditions for timetables:

1. Consistency between the timetable and the network, i.e. all journeys are well formed.
2. The physical constraint that trains on the same line cannot overtake.
3. A number of regulatory rules. There are minimal time separations between:
   - trains occupying the same track
   - two station entries on the same line
   - two station exits on the same line
   - occupations of a line in opposite directions.

Projection : A restriction of a timetable to a particular segment and a particular time period. A projection of a timetable gives the display of the running map tool.

Before discussing, in Section 1.5, the stage of requirements capture it is worth reflecting on what the domain analysis achieved. We had a formal and hence very specific semantic definition of a whole number of concepts that are commonly used in (Chinese) railway circles but were initially difficult for outsiders to grasp (apart from the translation problem in going from Chinese to English). And the specifications we generated (accompanied by commentary much like that used in this chapter) were examined and discussed by the Chinese fellows at UNU/IIST, so that we gained assurance that these particular definitions are the correct ones.

We had also defined a number of well-formedness conditions. Many of these are consistency requirements — stations at ends of lines exist, trains visit existing stations on existing tracks by existing and reachable lines, etc. Many others express *regulations* about the way that Chinese railways are operated. Hence the specification provides a *theory* of railway networks and of the operation of trains: we can formulate as theorems expected properties and justify them, either formally by theorem proving or informally. Doing so is an essential part of *validating* the specification, i.e. checking that it meets its requirements. It is an important feature of the model that it allows the easy and transparent statement of regulations as well-formedness functions, so that validation is simple. We can validate that a well-formed timetable complies with a regulation by pointing to the predicate expressing it. A close correspondence between requirements and the structure of the specification also provides a good basis for requirements tracking.

1.5 Requirements capture

Requirements capture extends the formal specification produced by the domain analysis with the definitions of the operational facilities that the system will provide to its users. In our case the domain analysis gave us in particular a definition of a timetable together with all the railway regulations that make a timetable well-formed. The task in requirements analysis is then to specify the running map tool. We already had the functions to project a timetable on to a particular period and a particular segment, hence defining the contents of the main window of the running map tool. We defined in addition:

- functions to input timetables to the tool — to support *scheduling*;
- functions to modify a timetable — to support *rescheduling*.

The first of these resulted eventually in a separate tool for timetable input, as the mainly graphical means of editing timetables in the running map tool turned out to be too clumsy for inputting large amounts of data. Then the normal mode of operating the running map tool was to start by loading a complete timetable. There are several kinds of timetable. The process starts with a *basic* timetable, which is not set to any particular day. From a basic timetable, 12-hour *shift plans* are generated by planning units, and from these typically 3-hour *stage plans* are generated and used for rescheduling by dispatch units. The ability to project over periods and segments allows all these activities to be supported.

Rescheduling is currently done manually, by drawing on large sheets of paper (prepared with the segment stations listed) lines indicating how the trains should move and stop. Rescheduling is done by erasure and redrawing. A computer tool to support this activity clearly needs to be mainly graphic, with the possibility to draw lines, move them, shift points on them, etc. But the display is also essentially an abstraction. It doesn't show which line between stations is being used by a train, or which track in a station. It is also hard to read precise times. (Hand drawn running maps do have more annotations to help with some of these problems.) So the tool has a number of tabular displays that can be shown by clicking on a train identifier, to show complete details of its journey, or on a station, to show complete details of all visits to that station. Changes can be made by editing either the tables or the graphic display.

We also attempted to formally specify the widgets generating the actual display of the running map tool, which is (X) window-based. This captured quite well the hierarchy of processes and the effects of events — button pushes, mouse clicks etc. — but says nothing about the appearance and is of little use in judging the usability of the tool. This specification was actually produced after the tool as part of its documentation. So, in fact, we started the development stage with a formal specification of the kernel of the running map tool, the part that checks for well-formedness of timetables plus a specification of the projection of a timetable that would form the basis of the centre of the display, but not of the buttons, pop-up menus, etc. that would form the user interface. Instead we drew the graphic design and checked informally that the functions required from the data flow diagrams could be supported.

It would also be possible to start requirements capture by making an *abstraction* of the specification from the domain analysis. It has been found elsewhere [DGPZ93] that the first specifications one writes, with the aim of understanding the problem, are often more concrete than is appropriate for starting development. There are standard techniques in RAISE for producing a more abstract specification from a concrete one [RAI95].

1.6 Software development

1.6.1 Design

It was decided to code the tool essentially from the initial specification rather than doing any formal development, at least in the sense of data structure development. Formalising the basic concepts by domain analysis and identifying and formalising all the functions needed to create and modify timetables had placed the project on a firm footing. So there were two design tasks to be done: the graphical user interface (which we had not specified formally) and the C data structures. The latter was part of translation.

The running map tool graphical interface was created using Athena widgets and a detailed design of the tool was done showing its intended appearance, the

widget tree and for each listing the widgets, their classes and attributes. The final tool (which precisely follows this design) was illustrated in Figure 1.2 on page 4.

The display separates stations (vertically) proportionately to their distances so that a train running at constant speed appears as a line of constant slope. The transformation to achieve this for any given segment was defined (in standard mathematical notation).

We wanted to be easily able to instantiate the tool for particular networks, and hence to be able to communicate these and to be easily able to inspect and edit them. So descriptors in, essentially, BNF were defined for them. Then corresponding C data structures were defined and the procedures to parse and unparse them written.

Timetables were handled similarly.

There was a change between the specification and the final code in that the functions to check for well-formedness were developed into functions that generated messages about any breach of well-formedness. These messages then appear in pop-up windows.

1.6.2 Translation

We though that for a demonstrator tool a fairly simple strategy for encoding the data structures would suffice. Of the data structures we had used, records, cartesian products and enumerated types either exist in C or could be coded immediately. For maps we used a simple strategy of encoding a map as a linked list, with the domain element added as an extra field to the range element. For example we have the following RSL type definitions and the corresponding C definitions:

type
$LN\_m' = LNid \xrightarrow[m]{} LN,$
$LN\_m = \{|\ lnm : LN\_m' \cdot is\_wf\_LN\_m(lnm)\ |\}$

```
typedef  struct LN_m_ * LN_m;
struct LN_m_ {
       LN     ln;           /* the line record */
       LN_m   next_p;       /* next_pointer */
};
```

type
$LN' :: s1 : STid\ ...,$
$LN = \{|\ ln : LN' \cdot s1(ln) \neq s2(ln)\ |\}$

```
typedef struct LN_ * LN
struct LN_ {
       LNid    lnid;
       STid    s1;
       ...
};
```

So what has happened to the subtypes? Functions were written for them in C (whether, as with *LN_m* the RSL used a separate function, or, as with *LN* it did not). For example, the C function `is_wf_LN`:

```
/* well_formed line check */
/* s1 must be different from s2.*/
bool is_wf_LN(LN ln)
{      if (Ident_eq(ln->s1, ln->s2))
         {error("Warning error:
                 expected a different station name
                 in line %s:\n", ln->lnid);
          return false;
        }
       return true;
};
```

As well as making the check the C code also includes the generation of a suitable error message. Such functions are then used as part of the parsing of data used to instantiate the tool; see Section 1.6.1. The well-formedness conditions on timetables are checked both on initial loading of a timetable and also after rescheduling changes; they produce messages in pop-up boxes.

Functions that involved existential or universal quantification were coded as iterative functions over the linked lists. Comprehended expressions were translated similarly. For example, the function *lnstoST*, for collecting the set of identifiers of lines into a station, was specified and then translated as follows:

value
  lnstoST : STid × *LN_m* → *LNid*-**set**
  lnstoST(s,lnm) ≡
    { *lnid* | *lnid : LNid* • *lnid* ∈ **dom** *lnm* ∧ *is_lnto(lnm(lnid),s)* }

```
/* apply a line map and a stid. return a lnid_set to stid.*/
IDS lnstoST(LN_m lnm, STid stid)
{   LN_m plns = lnm;
    IDS  ls   = IDS_NULL;
    while (plns != LN_m_NULL){
      if (is_lnto(lnm_get_ln(plns), stid))
        ls = ids_add(ls, ln_get_lnid(lnm_get_ln(plns)));
      plns = lnm_next(plns);
    }
    return ls;
  };
```

Hence the specification could be regarded as to a large extent “translatable”. At the time, mid-1994, the translator from RSL to C++ [B$^+$95] was still under development. If we used it now it could translate the map types since it has built in a standard translation for maps (not unlike the one we used). But the universally

and existentially quantified expressions would still need hand translation. All the functions had been specified explicitly, i.e. not using post conditions or axioms, and many could be translated directly (and could have been translated by a tool).

It is readily apparent that the RSL specification is much easier to read than the C code and hence much easier to validate against the requirements (and easier to change if found not to be correct). The separation of specification from coding supports a separation of concerns between conceptual correctness (is this the appropriate condition?) from algorithmic correctness and the appropriateness of the messages generated. In addition the separation of the two levels would make it possible to change the data structures used (to sorted linked lists or trees or hash tables rather than unsorted linked lists for some of the maps or sets, for example) if found necessary, without changing the specification.

The C function `lnstoST` uses a type `IDS` modelling sets of identifiers (again as linked lists) together with functions like `ids_add` (for adding an identifier to a set). There are functions for deletion, for set union, intersection etc. all collected into a separate module. This was typical of the general approach: each of the RSL modules defining tracks, stations, lines, network etc. were extended with functions to create such an object, add it, delete it, get each component of it, and check its well-formedness. This meant that the C code generated, as well as following the modularity of the specification, had a distinctly "object oriented" flavour to it, with each kind of entity accessed and manipulated by its own particular collection of functions.

1.7 Phase 2

The first group of fellows returned to China in December 1994. Two more fellows were supposed to come from Zhengzhou in January 1995 but there were problems and they did not come to Macau until August that year.

1.7.1 Improving the running map specification

The main comments they reported on the prototype running map was its inability to handle different kinds of train (passenger, freight, special and military) and locomotive (electric, diesel, and steam). It also lacked a number of special symbols used as annotations: to show new trains starting, terminating, coming from a neighbouring dispatch unit, going to a neighbouring dispatch unit, trains merging, trains splitting, temporary speed restrictions on lines, lines blocked by accidents or for repair, etc.

There were a number of standard intervals for two trains arriving at a station on the same line, departing on the same line, etc. In the original model these were assumed to be constant: in fact they depend on the station and the line. The times for travel on lines are also not in fact constant: they depend on the type of locomotive and whether the line is being used in the "opposite" mode (an *UP* train

on a *DOWN* line, or vice versa). There are also additions for acceleration and/or deceleration if the train is starting from rest and/or stopping. These were documented from the official Chinese manuals, with formulae and diagrams and then included in a new specification, cross referenced by comments to the formulae in the documentation.

These changes were quite straightforward to make. They appeared only in the record types for stations, trains and lines (each in separate RSL modules) and in the predicates used to check well-formedness of timetables. The overall structure of the specification and the types for the network and timetables were unaffected.

1.7.2 Distributing the specification

The prototype running map and its specification assumed a single timetable that could be projected into several segments. But in practice scheduling is done on an area basis by *planning units*, who pass schedules on to *dispatch centres* who partition them amongst *dispatch units*. These dispatch units do the actual rescheduling, communicating with stations, their dispatch centre and neighbouring dispatch units. We needed to work out how to distribute a timetable and the adjustments to it, and to analyse when adjustments to one component would affect others.

This was done by specifying a general theory of *distributing* a map according to a *partition* of its domain. This could be applied by representing a timetable as

type $TT = STid \times TNid \xrightarrow{m} STV$

and then partitioning this according to which dispatch unit (DUid) each station belongs to. The distributed timetable would have a type

type $DTT = DUid \xrightarrow{m} TT$

The theory of partitioning was developed generically and then instantiated in this way.

In the generic theory the notion of *delegability* was defined . A function f to change a map is *delegable* if the diagram in Figure 1.5 commutes:

Here df is f applied to just one component of the distributed map (and only exists if f has a domain value of the map as a parameter, allowing the component to be identified). *merge* is the inverse of *distribute*. Intuitively, a function to adjust a timetable is delegable if the change can be made by one dispatcher and the resulting timetable, formed by merging the distributed ones, would be the same as if the adjustment had been made to an undistributed timetable. An algorithm for checking delegability was defined and proved correct using the RAISE justification editor [B$^+$95] (the only time formal proof was used in this project).

A similar notion was applied to the concept of *analysing* a map and generating messages (to be used for the messages reporting infractions of the timetabling rules). We need to know when, after an adjustment, we will not create any spurious messages or lose any messages because of the distribution, i.e. when we analyse

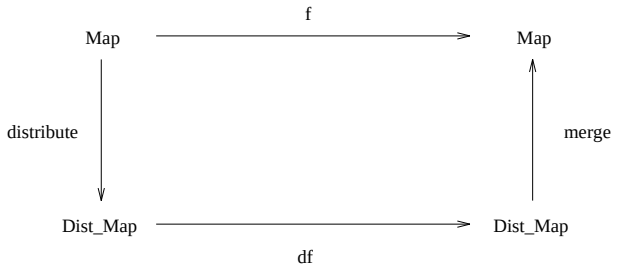

Figure 1.5: Correctness of distributed function

each component separately. This can be defined in terms of a similar commuting diagram.

Another notion, not directly related to distribution but expected to be of use, was that of *partial analysis*. For a particular adjustment function, what analyses need to be redone and which are guaranteed to generate the same messages? Knowing this will enable us to only perform some of the checks after changes, and to reliably improve the speed of the running map tool.

These ideas were formally specified and instantiated for timetables [Geo95]. This would have enabled all the functions for adjusting timetables and for checking for well-formedness to be checked for delegability and distributability, and for communication procedures with neighbouring dispatch units to be defined where necessary.

This is an example of a general method for defining distributed systems that has been found effective on several systems. First the complete system is specified as one entity. Then the division into components is done and the notion of correctness of the distributed system defined in terms of some equivalence with the original. This gives a theory about the communications and high level protocols needed in the distributed system. Trying to work "bottom up" from specifications of the distributed components makes things much more difficult because it lacks the notion of correctness.

It is also worth noticing that the analysis can be done without any need to specify the distributed system as a concurrent system: the analysis is all done on an applicative model, and can even, as here, be defined initially in terms of a parameterized abstraction of the original model, later instantiated to the actual one.

A development of the applicative distributed running map to a concurrent system was done [Mei95].

1.7.3 A constraint-based approach

Jimmy Ho Man Lee and Ho-Fung Leung of the Intelligent Real-Time Systems Laboratory at the Chinese University of Hong Kong set up a joint project with UNU/IIST during the academic year 1995–6 involving two final-year undergraduates and a PhD student. They wanted to take the running map tool further, into a tool that would apply constraint propagation techniques to the rescheduling problem, and generate proposed solutions.

They were given the RAISE specification from phase 1, and given a brief tutorial on how to read it — they had no previous experience with formal specification. They also had all the other documentation and the existing prototype. They were able, with little interaction, to reproduce the existing tool (on a different platform as well as with a different implementation technique). This is a striking example of the use of a formal description to precisely state requirements, and even to transmit them to people previously unacquainted with the notation after minimal training. It also shows the benefit of specifying the conditions to be checked rather than the algorithms to do so. The conditions can be validated against those in the existing documentation, because they are expressed in a similar mathematical style at the same level of abstraction, and also fairly easily communicated to others.

They were also able to devise and implement some strategies for rescheduling [CCL+96].

1.8 Conclusions

1.8.1 Achievements

Phase 1 took just over a year, involving most of the time of five fellows from the Chinese railways (though one also worked much of the time on his MSc thesis on station management [Yul95], and also as a system administrator) and the part time help of first the first and third of the authors of this chapter, then the first and second. The RSL specification for phase 1 was some 850 lines and the C code 15 000 lines (of which 5 000 is the non-specified graphical user interface). The modular structure of the C code follows closely that of the specification. The documentation runs to 600 pages. Many of the PRaCoSy documents are available on the World Wide Web, via the UNU/IIST home page http://www.iist.unu.edu.

This is rather more than a normal industrial development of such a system, but a lot of time was spent on domain analysis and on training the fellows not only in RAISE (and C) but also in a number of software engineering disciplines. It also included work to set up configuration management and version control systems. Last but not least, the fellows were working in a foreign language.

The quality of the resulting tool is very high for a prototype. No records were kept of errors found but few were discovered. The performance was initially poor, but after some tuning in terms of extra code to determine what checks needed to

be re-done after changes to the running map it became quite acceptable (running on a SUN sparc workstation).

Phase 2 lasted 8 months and involved 2 fellows from the Chinese railways. This phase was unfortunately curtailed when the two fellows were called back to Zhengzhou, for reasons that were never fully explained to us, but seemed to be their being needed for other projects. At the time they left we had incorporated the changes needed to the network and the timetables (while preserving the structure of the specification) and were close to translating to a new prototype.

1.8.2 Role of formal methods

Formal methods are often claimed to be expensive to introduce, difficult and expensive to use, to lack adequate tools, to be inapplicable to large examples, to be incomprehensible to customers, to be applicable if at all only to safety- or mission-critical systems [Hal90, BH95]. This project provides some evidence to counter these claims.

The fellows from China were "up to speed" with RAISE in quite a short time. The "light" use of RAISE as a means of describing the domain and software requirements clearly and providing a basis for the code to be written was, we believe, very effective and provided a development route for a non-critical system that was both fast and reliable. The system is not extremely large, but is certainly considerably more than an academic example. The RAISE tools are robust, fast, capable of supporting projects involving several people, and produce good documentation. The success of the separate group from Hong Kong in re-implementing and further developing the running map tool using a different technology, using a formal specification as their main input, is striking. The use of a formal method in a "rapid prototyping" style is unusual, at least in the literature, but proved effective, and we believe that phase 2 could have rapidly produced a second, distributed prototype involving all the extra details needed by dispatchers in a very short time. The substantial initial work analysing and describing the railway domain also proved effective in supporting one fellow's separate work on station management.

At the same time, we must point out the need for experts in training and assisting such a project. Using formal methods involves a different way of approach, in which analysing, understanding and defining the problem is the major task, and writing the code is deferred and done quite late. This involves skill and judgement, and industries are well advised to seek external help initially until they have developed their own experts. It also takes time to develop an appropriate culture, in which a project that has so far produced lots of specifications but no code is not automatically seen as in danger.

1.8.3 Further work

The project stopped very abruptly: not an unknown event but disappointing to those involved. Recently, however, interest has been expressed in India and in Russia in continuing the work.

For India, the running map tool was ported to Linux. This exposed several of the usual problems in the behaviour of the tool due to differences in the behaviour of C compilers (even though both were gcc) and in the behaviour of the widgets (again supposedly identical) but none (apparently, but without substantial testing) in the code for checking timetables.

1.9 Acknowledgements

The authors of this chapter trained, advised and assisted, but the actual work was done by others.

Most of the work was done by the fellows from the Chinese railways: Dong Yulin, Jin Danhua, Liu Xin, Ma Chao and Sun Guoqin in phase 1; Liu LianSuo and Yang Dong in phase 2.

Srinivasan Parthasarathy from Hyderabad, India worked on the project for 9 months in phase 1 as a visiting researcher, and Hong Mei from Fudan University, Shanghai, China worked on it as a fellow for 2 months between the two main phases.

References

[B+95] Peter Michael Bruun et al. RAISE Tools User Guide. Technical Report LACOS/CRI/DOC/4, CRI A/S, 1995.

[BH95] J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, July 1995.

[CCL+96] C.K. Chiu, C.M. Chou, J.H.M. Lee, H.F. Leung, and Y.W. Leung. A Constraint-Based Interactive Train Rescheduling Tool. In *Second International Conference on Principles and Practice of Constraint Programming*, volume 1118 of *Lecture Notes in Computer Science*, pages 104–118. Springer-Verlag, August 1996.

[DGPZ93] B. Dandanell, J. Gørtz, J. Storbank Pedersen, and E. Zierau. Experiences from Applications of RAISE. In *FME'93: Industrial Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[DM94] Babak Dehbonei and Fernando Mejia. Formal Methods in the Railways Signalling Industry. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME'94 : Industrial Benefits of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 26–34. Springer Verlag, 1994.

[Don96] Yang Dong. Basic Specification of PRaCoSy. Technical Report yd/define/1, UNU/IIST, P.O.Box 3058, Macau, March 1996.

[DPdB95] Eugène Dürr, Nico Plat, and Michiel de Boer. CombiCom: Tracking and Tracing Rail Traffic using VDM++. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Applications of Formal Methods*, pages 203–225. Prentice-Hall International, 1995.

[Geo95] Chris George. A Theory of Distributing Train Rescheduling. Research Report 51, UNU/IIST, P.O.Box 3058, Macau, December 1995. Published in: Marie-Claude Gaudel and James Woodcock (eds.), *FME'96: Industrial Benefit and Advances in Formal Methods*, LNCS 1051, Springer-Verlag, 1996, pp. 499–517.

[Hal90] J. A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

[Han94] Kirsten Mark Hansen. Validation of a Railway Interlocking Model. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME'94: Industrial Benefit of Formal Methods*, volume 873, pages 582–601. Springer-Verlag, October 1994.

[Jon80] Cliff B. Jones. *Software Development A Rigorous Approach*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1980.

[LPT94] Peter Gorm Larsen, Nico Plat, and Hans Toetenel. A Formal Semantics of Data Flow Diagrams. *Formal Aspects of Computing*, 6(6), December 1994.

[Mei95] Hong Mei. A Generic Concurrent Distributed Architecture. Technical note, UNU/IIST, P.O.Box 3058, Macau, June 1995.

[OH95] Takahiko Ogino and Yuji Hirao. Formal methods and their applications to safety-critical systems of railways. *QR of RTRI*, 36(4):198–203, December 1995.

[Pre94] Søren Prehn. A Railway Running Map Design. Technical Report SP/12/3, UNU/IIST, P.O.Box 3058, Macau, July 1994.

[RAI92] The RAISE Language Group. *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall International, 1992.

[RAI95] The RAISE Method Group. *The RAISE Development Method*. BCS Practitioner Series. Prentice Hall International, 1995.

[Sim94] Andrew Simpson. A Formal Specification of an Automatic Train Protection System. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME'94: Industrial Benefit of Formal Methods*, volume 873. Springer-Verlag, October 1994.

[Yul95] Dong Yulin. The Formal Development of a Railway Station Route Management System. M.Sc. Thesis, University of Macau, Taipa. Macau, May 1995. Available from UNU/IIST.