"UML–ising" Formal Techniques*

Dines Bjørner[a], Chris W. George[b], Anne E. Haxthausen[a],
Christian Krog Madsen[c], Steffen Holmslykke[a], and Martin Pěnička[d]

[a] Computer Science and Engineering Dept., Informatics and Mathematical Modelling Inst.,
Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark,
{db,ah}@imm.dtu.dk+steffen@holmslykke.com;
[b] Director ai., UNU/IIST, P.O.Box 3058, Macau SAR, China, cwg@iist.unu.edu;
[c] Rovsing A/S, Dyregårdsvej 2, DK–2740 Skovlunde, Denmark, christian@krog-madsen.dk;
[d] Faculty of Transportation, Czech Technical University, Na Florenci 25,
CZ-11000 Prague 1, The Czech Republic; penicka@fd.cvut.cz + martin@imm.dtu.dk.

Abstract. This invited paper presents a number of correlated specifications of example railway system problems. They use a variety of partially or fully integrated formal specification. The paper thus represents a mere repository of what we consider interesting case studies.

The existence of the Unified Modeling Language [10, 67, 36, 20] has caused, for one reason or another, the research community to try formalise one or another facet of UML. In this paper we report on another way to achieve what UML attempts to achieve: Broadness of application, convenience of notation, and multiplicity of views. Whether these different UML views are unified, integrated, correlated or merely co–located is for others to dispute. We also seek to support multiple views, but are also in no doubt that there must be sound, well defined relations between such views.

We thus report on ways and means of integrating formal techniques such as RAISE (RSL) [58, 59], Petri Nets [56, 62, 37, 61, 41], Message and Live Sequence Charts [42–44, 64, 13], Statecharts [23, 24, 26, 27], RAISE with Timing (TRSL) [18, 45, 46], and TRSL with Duration Calculus [79, 30]. In this way one achieves a firm foundation for combined uses of these formal development techniques, one that can be believably deployed for as wide a spectrum, or even a wider spectrum of software (and hardware) development, as, respectively than UML.

1 The Problem

1.1 The Issues

When we describe, in informal, yet reasonably precise natural (or at least domain specific professional) language the entities, the functions, the events and behaviours of an application domain, then we encounter, perhaps, little, if any problem. Our use of natural language is very flexible. Without hardly noticing it, we slip from one mode of description to another mode. (What these modes are will be apparent in the next paragraph.)

When, now, on the basis of the informal narrative, we wish to formalise this description, then we might very well encounter serious problems. We refer here to the current inability of any one formal specification language to cater for all kinds of modes: Functional, imperative (ie., with states being changed by assignments to variables), logical, temporal, and concurrency modes, the latter with events and behaviours. In particular we often slip, in natural language, from describing such *qualitative* aspects of timing as concurrent behaviours and their synchronisation and communication, to such *quantitative* aspects of timing as absolute and relative time: *"12:05 am"* to *"after 5 minutes and 30 seconds"* — without hardly noticing it.

* This invited paper is to be presented at the 3rd International Workshop on Integration of Specification Techniques for Applications in Engineering (INT-Workshop), 28th of March 2004, in Barcelona, Spain, as part of the German Research Foundation (DFG)'s priority research programme "Integration of Software Specification Techniques for Applications in Engineering". The main author's presentation at this ETAPS related event is also sponsored by DFG. The present paper is expected to also appear in a Springer–Verlag book which collects papers from the DG INT project.

Put differently: Some formal specification languages may cater, as does RSL, the specification language of RAISE [58, 59], for functional, imperative, logical, and parallel behaviours — but RSL does not cater, neither for "true" concurrency, nor for time. Also: The diagrammatic constructs of Petri Nets [56, 62, 37, 61], of Statecharts [23, 24], and of Live Sequence Charts [13] cater for the qualitative facets of concurrency and timing (as does RSL), but they do so diagrammatically, and as such they are indeed oftentimes more appealing to casual readers than "flat" texts (ie., RSL). Similarly RSL's "flat text" module structuring (schemes, classes and objects) are, to some, inferior in communicability to UML's Class Diagrams [10, 67, 36].

This therefore is the problem: To combine, to integrate, uses of two or more formalisms in one specification — such that we can still retain (most of) the virtues of any of the formal notations: For example abstraction, reasoning, and refinement.

1.2 Integrating Formal Techniques

No one formal specification language can reasonably be expected to cover all modes of descriptions, all kinds of universes of discourse.

There is, therefore, an effort going on, world–wide, in integrating, in combining, different specification paradigms, such as mentioned above. Notable efforts can be referenced:

Combining Statecharts and Z for the design of safety-critical control systems [75] (1996), Integrated Formal Methods [17] (1996), A combination of Object-Z and CSP [14] (1997), Specifying embedded systems with Statecharts and Z [19] (1998), An Operational Semantics for Timed RAISE, TRSL [18] (1999), Linking DC together with TRSL [30] (2000), Study of graphical and temporal specification techniques [49] (2003), Integration of Specification Techniques [48] (2003).

An underlying theme here is that of Unifying Theories of Programming [32] (1998), Unifying Theories of Parallel Programming [77] (2002), and Semantic Integration of Heterogeneous Software Specifications (2003) [65].

Many other references could be given to papers that seeks to provide answers to integration issues: [75, 17, 14, 32, 19, 11, 60, 77].

1.3 Structure of Paper

The paper is structured as follows: First (Sect. 2) we provide a setting, basically common to the whole paper, namely a specification, in RSL, of properties of the layout of railway nets. First we present it in a "flat" version of RSL, ie., without RSL's parameterised **scheme** and **class** facilities. Then we present "the same" specification with those modularising facilities (Sect. 3). From that, without much analysis, we present a UML Class Diagram (Sect. 4). Sect. 5 discuses relations between RSL and UML.

Then we "pick" another, albeit related, problem, one of timing, and show (Sect. 6.1) its specification in RSL extended with timing [18], and Timed RSL extended with durations [30], in the sense of the Duration Calculus [79] (Sect. 6.2).

Independently we show an example of combining a RSL specification with Petri Nets (Sect. 7), and finally a specification embodying Live Sequence Charts and Statecharts (Sect. 8)

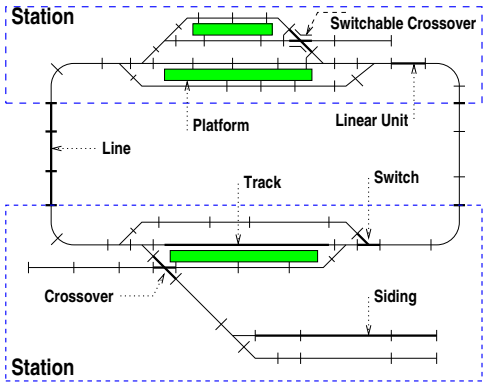
1.4 Prerequisites

Professional software engineers today are expected to be sufficiently versant in either of the notational systems and intentions of VDM, Z, RAISE, or B [15, 76, 59, 1]. Enough to understand this paper's use of RSL [58]. They are likewise expected to be sufficiently versant in UML's usage of Petri Nets, Message Sequence Charts (MSCs) and Statecharts (SCs) to likewise follow the paper's use of those mechanisms, including Live Sequence Charts. As for TRSL (Timed RSL) and DC (Duration Calculus) we do not expect the same insight – so, please consider this paper a good reason for "catching up" by reading the referenced TRSL paper [18] or PhD Thesis [78], respectively DC book [79] (or original paper [80]). There will be ample references, later, to books on UML, Petri Nets, MSCs, SCs, etc.

2 "Flat" RAISE (DB)

Our "running" example is taken from the domain of railways. First informally, as a rough sketch supported by a "snapshot" layout diagram. Then formally.

We constrain ourselves to the modeling of just the static aspects of the topology of a railway net. That is: Of net, lines and stations. And of units of lines and station (and hence of nets). And of connectors of units. Examples of a net, of two lines and two stations, of both lines consisting each of three linear units. Of the stations consisting of tracks (ie., platform tracks and sidings), and of otherwise also consisting of simple switch, simple crossover, and of switchable crossover units.



2.1 Informal Description

We narrate a precise, yet informal description:

We introduce the phenomena of railway nets, lines, stations, tracks, (rail) units, and connectors.

1. A railway net consists of one or more lines and two or more stations.
2. A railway net consists of rail units.
3. A line is a linear sequence of one or more linear rail units.
4. The rail units of a line must be rail units of the railway net of the line.
5. A station is a set of one or more rail units.
6. The rail units of a station must be rail units of the railway net of the station.
7. No two distinct lines and/or stations of a railway net share rail units.
8. A station consists of one or more tracks.
9. A track is a linear sequence of one or more linear rail units.
10. No two distinct tracks share rail units.
11. The rail units of a track must be rail units of the station (of that track).
12. A rail unit is either a linear, or is a switch, or a is simple crossover, or is a switchable crossover, etc., rail unit.
13. A rail unit has one or more connectors.
14. A linear rail unit has two distinct connectors, a switch rail unit has three distinct connectors, crossover rail units have four distinct connectors (whether simple or switchable), etc.
15. For every connector there are at most two rail units which have that connector in common.
16. Every line of a railway net is connected to exactly two, distinct stations of that railway net.
17. A linear sequence of (linear) rail units is a non-cyclic sequence of linear units such that neighbouring units share connectors.

The numbering of the text items is used as cross references in Sects. 2.2, 3 and 4.

2.2 Formal Description

And finally, in this introductory example, we formalise the previous informal narrative.

```
type
    N, L, S, Tr, U, C
value
1.  obs_Ls: N → L-set,
1.  obs_Ss: N → S-set
2.  obs_Us: N → U-set,
3.  obs_Us: L → U-set
5.  obs_Us: S → U-set,
8.  obs_Trs: S → Tr-set
12.  is_Linear: U → Bool,
12.  is_Switch: U → Bool
12.  is_Simple_Crossover: U → Bool,
12.  is_Switchable_Crossover: U → Bool
13.  obs_Cs: U → C-set

17. lin_seq: U-set → Bool
    lin_seq(us) ≡
     ∀ u:U • u ∈ us ⇒ is_Linear(u) ∧
     ∃ q:U* • len q = card us ∧ elems q = us ∧
         ∀ i:Nat • {i,i+1} ⊆ inds q ⇒ ∃ c:C •
             obs_Cs(q(i)) ∩ obs_Cs(q(i+1)) = {c} ∧
         len q > 1 ⇒ obs_Cs(q(i)) ∩ obs_Cs(q(len q)) = {}
```

Some formal axioms are now given, not all !

```
axiom
1. ∀ n:N • card obs_Ls(n) ≥ 1,
1. ∀ n:N • card obs_Ss(n) ≥ 2,
3. l:L • lin_seq(l)
4. ∀ n:N, l:L • l ∈ obs_Ls(n) ⇒ obs_Us(l) ⊆ obs_Us(n)
5. ∀ n:N, s:S • s ∈ obs_Ss(n) ⇒ card obs_Us(s) ≥ 1
6. ∀ s:S • obs_Us(s) ⊆ obs_Us(n)
7.  ∀ n:N,l,l′:L•{l,l′}⊆obs_Ls(n)∧l≠l′⇒obs_Us(l)∩ obs_Us(l′)={}
7.  ∀ n:N,l:L,s:S•l ∈ obs_Ls(n)∧s ∈ obs_Ss(n)⇒obs_Us(l)∩ obs_Us(s)={}
7.  ∀ n:N,s,s′:S•{s,s′}⊆obs_Ss(n)∧s≠s′⇒obs_Us(s)∩ obs_Us(s′)={}
8.  ∀ s:S • card obs_Trs(s) ≥ 1

9.  ∀ n:N, s:S, t:T • s ∈ obs_Ss(n) ∧ t ∈ obs_Trs(s) ⇒ lin_seq(t)
10.  ∀ n:N, s:S, t,t′:T •
    s ∈ obs_Ss(n) ∧ {t,t′} ⊆ obs_Trs(s) ∧ t≠t′
        ⇒ obs_Us(t) ∩ obs_Us(t′) = {}
15. ∀ n:N • ∀ c:C •
    c ∈ ∪ { obs_Cs(u) | u:U • u ∈ obs_Us(n) }
        ⇒ card{ u | u:U • u ∈ obs_Us(n) ∧ c ∈ obs_Cs(u) } ≤ 2
16. ∀ n:N,l:L •  l ∈ obs_Ls(n) ⇒
    ∃ s,s′:S • {s,s′} ⊆ obs_Ss(n) ∧ s≠s′ ⇒
        let sus = obs_Us(s), sus′ = obs_Us(s′), lus = obs_Us(l) in
        ∃ u:U • u ∈ sus, u′:U • u′ ∈ sus′, u″,u‴:U • {u″,u‴} ⊆ lus •
        let scs = obs_Cs(u), scs′ = obs_Cs(u′),
            lcs = obs_Cs(u″), lcs′ = obs_Cs(u‴) in
        ∃ ! c,c′:C • c ≠ c′ ∧ scs ∩ lcs = {c} ∧ scs′ ∩ lcs′ = {c′}
    end end
```

Elsewhere we have shown extensions of the above model into simple dynamics of unit switching [6], of principles of modeling such domains as railways [8], of possible relations between these kind of railway models and control theory [7], of using such models as that above for modeling train maintenance [57], train staff rostering [71], etc. Modeling the scheduling of trains, based on simpler models than the above, was shown in [9].

3 RAISE Model with Schemes (SH)

The previous specification was expressed in "flat" RSL. Next, and in preparation for the UML Class Diagram "rendition", we show a "structured" version of the above "flat" formulas. The structuring is afforded by RSL's **schema** and**class** mechanisms. Without much comments we present these schemes.[1]

The model presented in this section is somehow "equivalent", we claim, to the model just presented in section 2.2. The difference is in the use of parameterized schemes. Using schemes we can break the model into smaller modules. Each sort from the flat model is placed in a separate scheme and the functions and axioms which are associated with the sort are included with it. This should give an intuitive division of the flat model which may be more easily comprehended.

```
scheme Connectors = class type C end

scheme Units(connectors : Connectors) =
class
    type U
    value
        12 is_Linear: U→Bool,
        12 is_Switch: U→Bool,
        12 is_SimpleCrossover: U→Bool,
        12 is_SwitchableCrossover: U→Bool,
        13 obs_Cs: U→connectors.C-set,
    17 lin_seq: U-set→Bool
    lin_seq(us) ≡
        (∀ u:U•u ∈ us ⇒ is_Linear(u)∧
         (∃ q:U*•len q = card us∧elems q=us∧
            (∀ i:Nat•{i,i+1}⊆inds q ⇒
             (∃ c:connectors.C•
                 obs_Cs(q(i)) ∩ obs_Cs(q(i+1))={c}∧
                 len q>1 ⇒
                     obs_Cs(q(i)) ∩ obs_Cs(q(len q))={}))))
    end
```

We could single out each of the (so far mentioned) four disjoint kinds of *Units*, representing them as schemes. We show it only for the linear case:

```
scheme Linear(connectors : Connectors) =
extend Units(connectors) with
class
    type UL = U
    axiom
        ∀ l:UL•is_Linear(l)
                ∧∼is_Switch(l)
                ∧∼is_SimpleCrossover(l)
                ∧∼is_SwitchableCrossover(l),
        ∀ l:UL: card obs_Cs(l)=2
end
```

We go on:

```
scheme Sequence(
    connectors: Connectors,
    units: Units(connectors)) =
class
    type Seq
    value obs_Us: Seq→units.U-set
    axiom ∀ s: Seq•units.lin_seq(obs_Us(s))
end

scheme Lines(
    connectors: Connectors,
    units: Units(connectors)) =
extend Sequence(connectors,units) with
class
    type L
    value
     obs_Seq: L→Seq,
     obs_Us: L→units.U-set
     obs_Us(l) ≡ obs_Us(obs_Seq(l))
end

scheme Tracks(
    connectors: Connectors,
    units: Units(connectors)) =
extend Sequence(connectors,units) with
class
    type Tr
    value
     obs_Seq: Tr→Seq,
     obs_Us: Tr→units.U-set
     obs_Us(t) ≡ obs_Us(obs_Seq(t))
end

scheme Stations(
```

[1] The item numbers of some of the formulas of this section derive from Sect. 2.1.

```
    connectors: Connectors,
    units: Units(connectors),
    tracks: Tracks(connectors,units)) =
class
    type S
    value
     5 obs_Us: S→units.U-set,
     8 obs_Trs: S→tracks.Tr-set
    axiom
     5 ∀ s:S•card obs_Us(s)≥1,
     8 ∀ s:S•card obs_Trs(s)≥1,
     7 ∀ s,s′:S•s≠s′
                    ⇒ obs_Us(s) ∩ obs_Us(s′)={}
end

scheme Nets(
    connectors: Connectors,
    units: Units(connectors),
    lines: Lines(connectors,units),
    tracks: Tracks(connectors,units),
    stations: Stations(connectors,units,tracks)) =
class
    type N
    value
     1 obs_Ls: N→lines.L-set,
     1 obs_Ss: N→stations.S-set,
     2 obs_Us: N→units.U-set
    axiom
     1 ∀ n:N•card obs_Ls(n)≥1,
     1 ∀ n:N•card obs_Ss(n)≥2,
     4 ∀ n:N,l:lines.L•l ∈ obs_Ls(n)⇒
        lines.obs_Us(l)⊆obs_Us(n),
     6 ∀ n:N,s:stations.S•s ∈ obs_Ss(n)⇒
        stations.obs_Us(s)⊆obs_Us(n),
     7 ∀ n:N,l:lines.L,s:stations.S•
        l ∈ obs_Ls(n)∧s ∈ obs_Ss(n)
          ⇒lines.obs_Us(l) ∩ stations.obs_Us(s)={}
end
```

4 The UML Model (SH)

The two formal models of Sect. 2.2 and Sect. 3 were based on the informal description of railway nets (Sect. 2.1). The model in this section is expressed in `UML` but reflects the parameterised scheme model (of Sect. 3). This should of course amount to a model that is "equivalent" to the formal models. It is however known that the language of `Class Diagrams` is not as powerful an expression tool as is, for example, `RSL`. Properties expressible in, for example `RSL`, cannot be expressed by the `Object Constraint Language, OCL` [73, 74], of `UML`.

Notwithstanding, it is still a good idea to try express certain of the properties of the formal models in class diagrams. Our model is presented in figure 1.

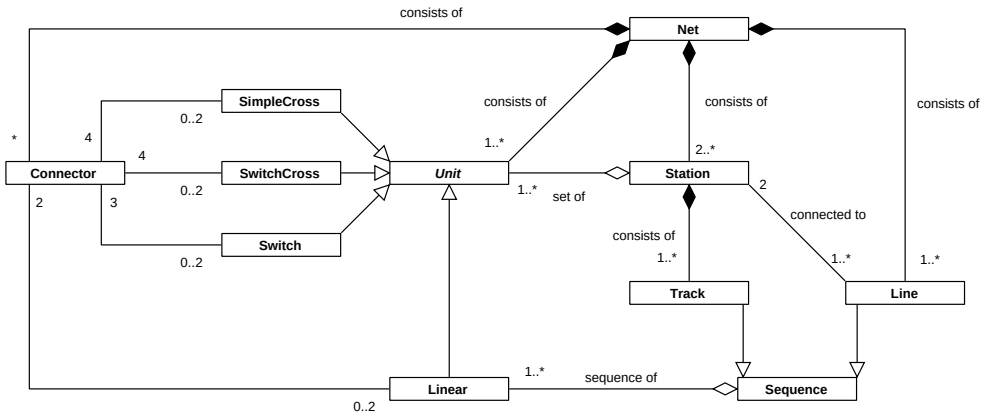


Fig. 1. UML Class Diagram for Rail Nets

In our class diagram for rail nets, the model has been divided into several "smaller" pieces which describe "smaller" parts. In this case the classes represent the phenomena introduced in the informal description and corresponding to the schemes of Sect. 3.

Items 1[2], 2, and 8 describe a *consist of* relationship between two phenomena. The latter item describes that a station consists of one or more tracks. This fits with the whole–part relationship that *composition* provides for in class diagrams. Here the station is the whole and it is not complete unless it has tracks and the tracks cannot exist without a station. As an example, item 8 is depicted, in the class diagram, as a solid line between the *Station* and *Track* classes; the first is marked with a filled diamond at the end of the line — indicating that it is the *whole*.

[2] The item numbers of this section derive from Sect. 2.1.

Items 3, 9, and 5 use respectively a *sequence of* and a *set of* to describe a relationship. This is again a whole–part relationship. The parts are, however, already part of the net. So to be able to maintain a reference to an existing part a *shareable* aggregation is used as a relation. As an example, item 3 is depicted, in the class diagram, as a solid line between the *Station* and the *Unit* classes; the first is marked with an hollow diamond at the end of the line — indicating that it is the *whole*.

In item 12 a unit is described as being either a *Linear*, *Switch*, *SimpleCross*, or *SwitchCross*. In the class diagram this is expressed by a *generalization* relationship where the *Unit* is an *abstract class*. Its class name is written in italics — so it cannot be instantiated.

Both the informal description in item 12 and the corresponding way it is modeled in the class diagram suggests that another axiom should be added. In the formal model four boolean functions are used to determine which type a given unit is. Here an axiom could be added, one which ensures that a unit only can be of one type. This is achieved in the class diagram since an object only can instantiate one class. The axiom could be as follows:

$$\forall\ u{:}U \bullet is\_Linear(u) \Rightarrow \sim(is\_Switch(u) \vee is\_SwitchableCrossover(u) \vee is\_SimpleCrossover(u))$$

Additional axioms should be added for each of the three other possible situations.

The two items 13 and 14 are overlapping. The latter expresses more properties. The latter explicitly describes the number of connectors which a given unit must have, while the former just states that a unit has at least one connector attached. If the latter is fulfilled then so is the former which makes it superfluous in this model. This was noticed while drawing the associations between the *Unit* class and its specializations. Here item 14 would in the class diagram amount to an *association* between each of the *specialized* classes of *Unit* and (and to) the *Connector* class. Item 13 would be an association between the abstract *Unit* class and (and to) the *Connector* class. If these were to be added to the class diagram, then it would mean that each of the specialisations, due to inheritance, also would have this relation (through generalisation), which is, however, not intended.

It is not possible to diagram items 4, 6, 7, 10, 11, 15, 16, and 17 in a class diagram, since they describe requirements to the instances of the static structure. As an example, item 4 is used and redisplayed for convenience: "The rail units of a line must be rail units of the railway net of the line." To be able to express this requirement we must be able to identify a particular unit and if it is part of a line then it must also be part of the net. This could, however, be achieved by using the `Object Constraint Language` [20, sec. 6]. We will not do so here.

5 Discussion: RSL and UML (SH)

During the creation of the `RSL` and the `UML` class diagram models, some observations have been made. These will be discussed in this section.

5.1 `UML` and `RSL` Relationship

While making the modular `RSL` model of Sect. 3, and the `UML` model of Sect. 4, it was intuitively decided which constructs to use in the languages. These choices are commented upon with regard to a more general relationship between the two languages.

Entity sets described in the informal description have in the `RSL` model been represented by sorts: Besides a few observer functions they are further unspecified. In the class diagram they are represented by classes which can be instantiated as objects. There is a resemblance here with `RSL` schemes since they also can be instantiated (as `RSL` objects). The style which have been chosen in the `RSL` model is applicative (ie., functional). There is perhaps a closer relationship between schemes and classes if an imperative modeling style had been used since the object in `RSL` would then contain a state.

One could argue that the models described are still in an initial phase and it is too early to determine what a state for a given phenomena should consist of. This is also apparent in the class diagram since none of the classes have any attributes nor operations which is also the reason for not including the compartments in the diagram.

The associations used in the class diagram are in the RSL specification described using observer functions on the sorts. Links, which in UML are instances of associations, are in UML models terms used to communicate messages; that is, invoke a method at the target object. As an alternative to the observer functions in RSL channels might be used as a representation.

The generalization relationship in UML and the **extend** construct in RSL seem quite similar since they both take respectively a class and a scheme and adds more information. A specialised class in UML can add attributes or operations to the ones already present in the generalised class. This is also possible with the **extend** construct of RSL. However before the generalisation versus extend relationship can be discussed it should be determined whether or not the UML class can be represented by schemes in RSL.

There are of course many more elements in UML but those used in the Fig. 1 are the most essential. Therefore this discussion will be constrained to those.

5.2 References

Although RSL has modules, it may be claimed not to be "a true" object oriented ("OO") language. This does not, however, mean that it is impossible to express object oriented models in RSL. The reason that RSL may be judged not to be "immediately" object oriented may be the claim that RSL does not provide for object references. But since objects of RSL can be grouped into **object arrays,** indexing can replace linking.

As an example the three schemes *Connectors*, *Units*, and *Lines* from section 3 can be used. The headers of the mentioned schemes are replicated below for convenience. The first scheme has no parameters since it does not use any sorts or functions from outside its own scheme. The *Units* scheme needs to know of the *Connectors* scheme since it uses its sort.

scheme Connectors = **class** ... **end**,
scheme Units(connectors:Connectors) = **class** ... **end**

The *Lines* scheme need only information from the *Units* scheme and not from the *Connectors* scheme. However to be able to instantiate the *Units* scheme an object instantiated from the *Connectors* scheme must be provided. It is not possible to pass an already instantiated object of units as the only parameter to the *Lines* scheme or formulated in another way it is not possible to pass an object by reference. This is a major difference between RSL and object oriented modeling. Thus it is necessary to give an object of type *Connectors* as parameter although it is not used by the *Lines* scheme.

scheme Lines(connectors:Connectors,units:Units(connectors)) = **class** ... **end**

It is, however, all a matter of how one approaches the modeling, the abstraction level and the refinement of models. Through a suitably chosen approach one may claim that RSL provides for all that "OO" provides.

5.3 Circularity

An association with composite aggregation in the class diagram which has the same class at each end introduces a recursive description. It is possible to define recursive structures in RSL using variants however it is not permitted to make a recursive type definition nor recursive modules. In this case a scheme is not a good choice for representing a UML class.

Recursive definitions have not been used in any of the RSL models nor in the UML models but was considered with respect to units and connectors. The question is whether a connector is an independent phenomena or a part of a unit. The latter seems to be best for describing railway nets. If a connector is part of a unit then aggregation should be used where a unit specialisation is the whole and the connector is the part. This would also mean that it is actually two connectors which is connected or, perhaps a better way to express it, is that when to units are connected then the connectors at the ends merge into one connector.

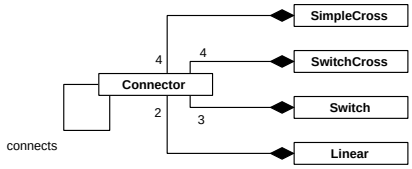


Fig. 2. Alternative model for connectors and specialized units.

The actual recursive solution was considered in the case where there still would be two connectors when two units are connected. That is the connectors are connected to each other. This would mean that a connector is part of a unit which would be modelled with aggregation and it would be connected to another connector which would be modelled with a ordinary association from the *Connector* class to itself; Hence the recursive definition.

5.4 Class Diagram Limitations

As mentioned in section 4 the class diagram could not contain all the information given in the (in)formal description(s) of railway nets. Particularly information that referred to the unique identity of an instance. Here it is necessary to use the `Object Constraint Language.`

It is possible to express some information in the diagram which in the `RSL` models are described using axioms. Examples are constraints on numbers, such as the minimum number of stations in a net. This is expressed in class diagrams using multiplicity.

• • •

In [16] a formal model has been presented, in `RSL`, of `UML`'s Class Diagram concept, together with a mechanism, a kind of "compiling algorithm" which translates `UML` Class Diagrams into `RSL`. Ongoing work at the first (DB) and second (CWG) co–authors' institutions, are carrying on this work of combining `RSL` with the graphics of `UML`'s Class Diagrams. The fifth co–author (SH) is involved in this work.

6 RAISE and Temporality (CWH+AH)

6.1 Timing and `RAISE: TRSL`

'Timed `RSL`', `TRSL`, was first treated in [18].

`RSL` originally had no built–in way to model time. Time could of course be modeled using `RSL`, but this is not in general very satisfactory. Without a built–in notion of time it would be impossible, for example, to specify basic components of timed systems such as "time out".

The extension of `RSL` to Timed `RSL` (`TRSL`) is minimal syntactically: there are just two additions. First is the type **Time**, just a synonym for the non–negative subtype of the existing type **Real**. Second is the new expression "**wait** *e*", where *e* is an expression of type **Time**.

The semantic changes are, of course, more considerable, but still largely confined to the constructs intended to specify communication and concurrency. The semantics is based on Wang Yi's work on Timed `ccs` [78], adapted to support value passing communication. It assumes that only the wait expression, input and output can consume time, adopts the principle of maximal progress, and includes time dependence. Time dependence enables a parallel expansion rule, but also adds expressiveness.

Methodologically, the intention is to develop specifications initially without regard to time, following the normal `RAISE` method, reaching an imperative concurrent specification: essentially a collection of communicating processes. At this point time is introduced in terms of wait expressions, and possibly extra choices for detections of time outs or other time dependent behaviour. There is more on the method in section 6.2.

We give here a few illustrative fragments. First, **wait** may just indicate a delay. Execution of the expression:

sensor_state := high ; **wait** δ ; sensor_state := low

will set and keep *sensor_state* high for precisely time δ, and then make it low.

A time out can be modeled by an external choice involving a **wait**. Suppose we need to take some special (abnormal) actions if a signal *normal* does not occur within time *t*. The expression:

normal? ; ...
□
wait t ; abnormal!()

will take the first choice provided an output on the channel *normal* occurs within time *t*. Otherwise, at time *t*, the wait terminates and the second choice becomes available. Provided there is some process waiting to handle the output *abnormal*, the principle of maximal progress will ensure the second choice occurs, and we would say the normal behaviour has timed out.

An example illustrating the use of time dependence will be given in section 6.2.

In [46, 45] denotational semantics of Timed RSL are given using Duration Calculus, to the combination of which we now turn.

6.2 TRSL and Duration Calculus

The Duration Calculi are covered in the seminal [79].

While TRSL is well–suited for timed design specifications, DC is well-suited for timed requirement specifications. This suggests the following development method [30] (illustrated in Fig. 3) for real-time systems integrating TRSL and DC specifications:

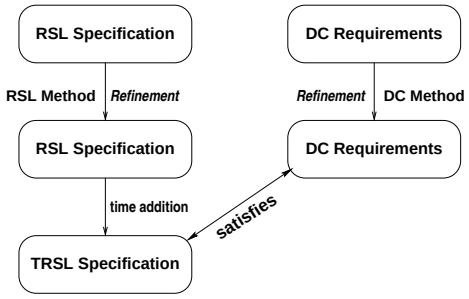


Fig. 3. A Development Method for Real-time Systems

1. The RAISE method [59] is used for stepwise developing a specification of the un–timed properties of the system, starting with an abstract, property–oriented RSL specification and ending with a concrete, implementation–oriented RSL specification.
2. In parallel with the RSL development of the un–timed system, a DC requirement specification of the real time properties of that system is developed. State variables in the DC specification are variables defined (at least) in the last RSL specification (and in the TRSL specification).
3. Timing information is added to the RSL specification achieving a TRSL specification of a real-time implementation.
4. It must be verified that the TRSL specification satisfies the DC specification.

Hence, there is no syntactic integration between the DC and TRSL specification, but only a consistency requirement that state variables used in the DC specification are variables defined in the TRSL specification. The integration is made in the form of a satisfaction (or refinement) relation. The approach for defining this relation has been to make an abstract interpretation within the DC formalism of TRSL

process definitions. Technically this is done by extending the operational semantics of TRSL [18] with behaviours which are DC formulas describing (parts of) the history of the observables of the system. The satisfaction relation between sentences in the two languages is then defined in terms of behaviours. The formal definition and proof rules can be found in [30].

Due to space limitations we just show a very simple example illustrating steps 2-4.

Problem description: Our goal is to specify those components of a railway control system that should perform train detection.

In the considered system sensors are used for train detection. When a train starts passing a sensor, the sensor should immediately become "high" and after a while it should fall back to "low". In order for the control system to be able to detect the high state the sensor must stay in the "high" state for a certain minimum of time, δ. Because of this requirement, trains should arrive at the sensor at least δ time apart. It may be safe to just record this as an assumption, because we know it is ensured by other parts of the system, or because δ (perhaps a fraction of a second for electronic equipment) is orders of magnitude less than an interval between trains could be. But sometimes such assumptions need to be checked at runtime, and that is what we assume here, as it gives us an opportunity to illustrate the use of time dependence. We assume that an error must be recorded if two trains arrive within δ of each other.

DC requirements: The requirement on the sensor is:

$$\Box((\lceil \text{sensor\_state=low} \rceil \bullet \lceil \text{sensor\_state=high} \rceil \bullet \lceil \text{sensor\_state=low} \rceil) \Rightarrow \ell \geq \delta)$$

This requirement says that any complete period with "high" state (i.e. one with a "low" state before and after) has a duration (ℓ) of at least δ.

TRSL Specification:

```
value δ : Time
type SensorState == low | high
channel detect_train, error, train_detected : Unit

value
  detect : Unit → in detect_train out train_detected, error Unit
  detect() ≡
    while true do
      let t = detect_train? in
        if t ≤ δ then error!()
        else train_detected!()
        end
      end
    end,

  sensor : Unit → in train_detected write sensor_state Unit
  sensor() ≡
    local variable sensor_state : SensorState:= low in
      while true do
        train_detected? ; sensor_state := high ; wait δ ; sensor_state := low
      end
    end
```

The channel *detect_train* represents the hardware train detection unit. We assume that every train enables an output on this channel.

The purpose of the process *detect* is to check that trains are at least time δ apart. Provided trains are sufficiently separated it signals their arrival to the *sensor* process; otherwise it signals an error. *detect*'s behaviour depends on the time t that it waits for input on the *detect_train* channel.[3] If t is too small an error is signaled. Otherwise the detection event is passed to the *sensor* process using another channel *train_detected*. If we had made the assumption that trains could not possibly arrive within time δ of each other, process *detect* and the channel *train_detected* would be unnecessary, and *sensor* could directly access the channel *detect_train*.

The process *sensor* controls the sensor state *sensor_state*: In each cycle, right after receiving a message (on *train_detected*) from the *detect* process that a train has arrived, *sensor_state* stays "high" for exactly δ time units and then becomes low. (Hence, it satisfies the DC requirement.)

Note that correct behaviour of *detect*, in the sense of only reporting actual errors (trains too close together), assumes that the value t is the same as the time since the last train, i.e. since the last communication on *detect_train*. This will only be true if there is no wait anywhere in the loop except for the communication on *detect_train*. This in particular means that the *sensor* process must always be ready to input on *train_detected* when *detect* is ready to do output on *train_detected*, i.e. *sensor* must have a cycle time of at most δ. This is clearly satisfied by *sensor*.

Satisfaction Relation: The following satisfaction relation expresses that the *sensor* process satisfies the previously stated DC requirement:

sensor() **satisfies**
$\Box((\lceil \text{sensor\_state=low} \rceil \bullet \lceil \text{sensor\_state=high} \rceil \bullet \lceil \text{sensor\_state=low} \rceil) \Rightarrow \ell \geq \delta)$

It can be proved using proof rules in [30] and DC proof rules.

7 Petri Nets and RAISE

We assume basic knowledge of Petri Nets: [56, 62, 37, 61, 38, 41].

7.1 The RAISE Part (DB)

First we augment our model of railway nets with dynamics of these railway nets. We introduce defined concepts such as paths through rail units, state of rail units, rail unit state spaces, routes through a railway network, open and closed routes, trains on the railway net, and train movement on the railway net.

Informal description:

18. A path, $p : P$, is a pair of connectors, (c, c'),
19. which are distinct,
20. and of some unit.[4]
21. A state, $\sigma : \Sigma$, of a unit is the set of all open paths of that unit (at the time observed).[5]
22. A unit may, over its operational life, attain any of a (possibly small) number of different states ω, Ω.
23. A route is a sequence of pairs of units and paths —
24. such that the path of a unit/path pair is a possible path of some state of the unit, and such that "neighbouring" connectors are identical.
25. An open route is a route such that all its paths are open.
26. A train is modelled as a route.
27. Train movement is modelled as a discrete function (ie., a map) from time to routes
28. such that for any two adjacent times the two corresponding routes differ by at most one of the following:
    (a) a unit path pair has been deleted (removed) from one end of the route;
    (b) a unit path pair has been deleted (removed) from the other end of the route;

[3] An input or output can optionally return the time that it waited for synchronisation: this supports time dependence, i.e. following behaviour can depend on the value of this time.

[4] A path of a unit designate that a train may move across the unit in the direction from c to c'. We say that the unit is open in the direction of the path.

[5] The state may be empty: the unit is closed.

(c) a unit path pair has been added (joined) from one end of the route;
(d) a unit path pair has been added (joined) from the other end of the route;
(e) a unit path pair has been added (joined) from one end of the route, and another unit path par has been deleted (removed) from the other end of the route;
(f) a unit path pair has been added (joined) from the other of the route, and another unit path par has been deleted (removed) from the one end of the route;
(g) or there has been no changes with respect to the route (yet the train may have moved);

29. and such that the new route is a well–formed route.

Formalisation:

type
18. $P' = C \times C$
19. $P = \{|\ (c,c'):P' \bullet c \neq c'\ |\}$
21. Σ = **P-set**
22. $\Omega = \Sigma$**-set**
23. $R' = (U \times P)^*$
24. $R = \{|\ r:R' \bullet wf\_R(r)\ |\}$
26. $Trn = R$
27. $Mov' = T\ \overrightarrow{m}\ Trn$
28. $Mov = \{|\ m:Mov' \bullet wf\_Mov(m)\ |\}$

value
21. obs_Σ: $U \to \Sigma$
22. obs_Ω: $U \to \Omega$

axiom
∀ u:U •
  let ω = obs_Ω(u), σ = obs_Σ(u) **in**
  $\sigma \in \omega \wedge$ 20.
  let cs = obs_Cs(u) **in**
  ∀ (c,c′):P • (c,c′) ∈ ∪ ω ⇒ {c,c′} ⊆ obs_Cs(u)
  end end

24. wf_R: $R' \to$ **Bool**
  wf_R(r) ≡
    len r > 0 ∧
    ∀ i:**Nat** • i ∈ **inds** r **let** (u,(c,c′)) = r(i) **in**
      (c,c′) ∈ ⋃ obs_Ω(u) ∧ i+1 ∈ **inds** r ⇒
      let (_,(c″,_)) = r(i+1) **in** c′ = c″ **end end**

25. open_R: R → **Bool**
  open_R(r) ≡
    ∀ (u,p):U×P • (u,p) ∈ **elems** r ∧ p ∈ obs_Σ(u)

27. wf_Mov: Mov → **Bool**
  wf_Mov(m) ≡ **card dom** m ≥ 2 ∧
    ∀ t,t′:T • t,t′ ∈ **dom** m ∧ t < t′
    ∧ adjacent(t,t′) ⇒
      let (r,r′) = (m(t),m(t′))
        (u,p):U×P • p ∈ ⋃ obs_Ω(u) **in**
  28a. (ld(r,r′,(u,p)) ∨ 28b. rd(r,r′,(u,p)) ∨
  28c. la(r,r′,(u,p)) ∨ 28d. ra(r,r′,(u,p)) ∨
  28e. ldra(r,r′,(u,p))∨ 28f. rdla(r,r′,(u,p))∨
  28g. r=r′) ∧ wf_R(r′)
      end

  adjacent: T × T → **Bool**
  adjacent(t,t′) ≡ ∼∃ t″:T • t″ ∈ **dom** m ∧ t < t″ < t′

  ld,rd,la,ra,ldra,rdla: R × R × P → **Bool**

  ld(r,r′,(u,p)) ≡ r′ = **tl** r **pre len** r>1
  rd(r,r′,(u,p)) ≡ r′ = fst(r) **pre len** r>1
  la(r,r′,(u,p)) ≡ r′ = ⟨(u,p)⟩⌢r
  ra(r,r′,(u,p)) ≡ r′ = r⌢⟨(u,p)⟩
  ldra(r,r′,(u,p)) ≡ r′ = **tl** r⌢⟨(u,p)⟩
  rdla(r,r′,(u,p)) ≡ r′ = ⟨(u,p)⟩⌢fst(r)

  fst: R $\xrightarrow{\sim}$ R′
  fst(r) ≡ ⟨ r(i) | i **in** ⟨1..**len** r−1⟩ ⟩

So the above models that rail units change state. What makes rail units change state ? Well, firstly, external stimuli may change the state of a switch or a crossover switch; secondly signals, in stations and along lines imply the closing of sequences of units. Thirdly these signals and switches are according to certain rail line and station switch interlocking protocols. How the latter protocols are specified will be the subject of the next subsection, Sect. 7.2 and of Sect. 8.

7.2 The Petri Net Part (MP+CKM)

We shall, in this section, model one set of proper interlocking control requirements. We shal do so by means of Petri Nets. There are other ways of doing that: [52–55] uses `ccs` ([51]), [39, 3] uses `Z` ([70, 70, 76]), [69] uses `CSP` ([33, 66, 68]), and [47, 28] uses `RAISE`, and so forth. Others have used Petri Nets: [4, 5, 72]. What we shal show is another approach.

We shall be using *Place Transition Nets* for our example.

Route Descriptions: Since interlocking has to do with setting up proper routes from station approach ("line departure") signals to platform (ec.) tracks, and from these to the lines connecting to other stations, we shall focus on constructing, for all such "interesting" routes of a station a Petri Net that models a proper interlocking control scheme.

Routes are described in terms of Units, Switches and Signals. In the previous section (Sect. 7.1) formulas 23 and 24 defined routes as sequences of pairs of units and paths, such that the path of a unit/path pair is a possible path of some state of the unit, and such that "neighbouring" connectors are identical. There can be many such routes in a station. We are interested only in routes which start at an approach signal and ends either at the track or on the line. In the example station of Fig. 4 there are 16 such routes.

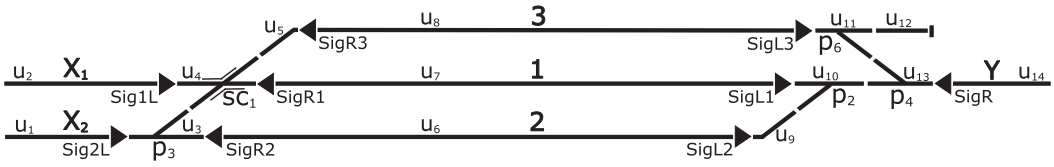


Fig. 4. Example Station

Interlocking Tables: Now, depending on the local, or national traditions and rules & regulations, there are such rules & regulations which stipulate how signals and switches are to be set (and reset) in order to facilitate the safe movement of trains within a station.

One can formalise such rules (see, for example, [39]). From a mechanisation of such a formalisation and from the specific topology of a station layout, for example that abstracted in Fig. 4, one can then construct an interlocking table, such as for example the one given Table 1. Each row in this table corresponds to a proper route. The table expresses for each interesting route the requirements for switches (points and switchable crossovers) and the requirements for signal states. The table also lists all units which compose the route. If there are no requirements on the setting of switch or signal, it is marked with dash (–). In this paper, we do not show, how to formally construct such table, but we refer to [21, 39, 22, 69].

Requirements:	Switches					Signals									Units
Routes	sc_1	p_2	p_3	p_4	p_6	Sig_{1L}	Sig_{2L}	Sig_{L1}	Sig_{L2}	Sig_{L3}	Sig_R	Sig_{R1}	Sig_{R2}	Sig_{R3}	
1. $Sig_{1L}-1$	S	–	S	–	–	G	–	–	–	–	–	R	–	R	u_2, u_4, u_7
2. $Sig_{1L}-3$	T	–	S	–	–	G	–	–	–	–	–	R	–	R	u_2, u_4, u_5, u_8
3. $Sig_{2L}-1$	T	–	T	–	–	R	G	–	–	–	–	R	R	R	u_1, u_3, u_4, u_7
4. $Sig_{2L}-2$	–	–	S	–	–	–	G	–	–	–	–	–	R	–	u_1, u_3, u_6
5. $Sig_{2L}-3$	S	–	T	–	–	R	G	–	–	–	–	R	R	R	u_1, u_3, u_4, u_5, u_8
6. $Sig_{L1}-Y$	–	S	–	S	S	–	–	G	R	R	R	–	–	–	u_{10}, u_{13}, u_{14}
7. $Sig_{L2}-Y$	–	T	–	S	S	–	–	R	G	R	R	–	–	–	$u_9, u_{10}, u_{13}, u_{14}$
8. $Sig_{L3}-Y$	–	–	–	T	T	–	–	R	R	G	R	–	–	–	u_{11}, u_{13}, u_{14}
9. Sig_R-1	–	S	–	S	S	–	–	R	R	R	G	–	–	–	u_{13}, u_{10}, u_7
10. Sig_R-2	–	T	–	S	S	–	–	R	R	R	G	–	–	–	u_{13}, u_{10}, u_9, u_6
11. Sig_R-3	–	–	–	T	T	–	–	R	R	R	G	–	–	–	$u_{13}, u_{10}, u_{11}, u_8$
12. $Sig_{R1}-X_1$	S	–	S	–	–	R	–	–	–	–	–	G	–	R	u_4, u_2
13. $Sig_{R1}-X_2$	T	–	T	–	–	R	R	–	–	–	–	G	R	R	u_4, u_3, u_1
14. $Sig_{R2}-X_2$	–	–	S	–	–	–	R	–	–	–	–	–	G	–	u_3, u_1
15. $Sig_{R3}-X_1$	T	–	S	–	–	R	–	–	–	–	–	R	–	G	u_5, u_4, u_2
16. $Sig_{R3}-X_2$	S	–	T	–	–	R	R	–	–	–	–	R	R	G	u_5, u_4, u_3, u_1

Table 1. Interlocking Table for Routes through the Example Station

We can now start to build up Petri Nets for a partial railway net from four subparts: Petri Net for a Unit, for a Switch (ie., Point or Switchable Crossover), for a Signal, and Petri Net for a Route. Pls. observe that all units have a basic Petri Net. Additionally Switches have additional basic Petri Nets — as we shall soon see. And, finally, although Routes are basically sequences of Units, also Routes have their separate basic Petri Nets. The Petri Net of a Route is then a composition of all its Unit, all its Switch, and all its Signal Petri Nets — where the composition is specified by the Interlocking Table.

Petri Net for Units: A Unit can be in two basic states. It is either free (a new route can be opened through the unit) or not (ie., blocked, there is an already opened route through the unit).

The Petri Net for Units is shown in Fig, 5(a). Two places represent the two states Free and Blocked. The initial marking consists of a token at the Free place.

One can notice, that Petri Net for a Unit in Fig. 5(a) will interminably circulate ("oscillate"). But this is not the final Petri Net for a route. It is just one component. Later on, extra arcs will be added. They will prevent "oscillations".

Petri Net for Switches: A Switch can be either a point or switchable-crossover. A typical switch has two states: Straight and Turn. A switch may be required to be set in certain state in two ways: as a direct part of a route, or because it must be set for side protection (to avoid trains touching each other). In the both cases, if there is a open route through switches, these switches must never change their states.

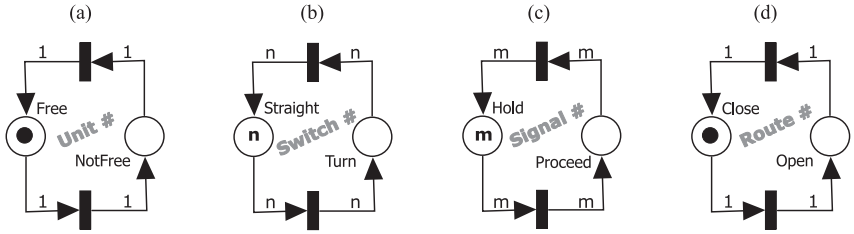


Fig. 5. Petri Nets for (a) Units, (b) Switches, (c) Signals, and (d) Routes

Thus the Petri Net for a switch has two places representing the two mentioned states Straight and Turn. The initial marking consists of n tokens at the Straight place, where n is the total number of routes which require settings of that switch. This number can be found from the Interlocking Table (here Table 1) as a count of required setting in the switch column. For the example station in Fig. 4, one finds that for switchable–crossover $sc1$, n is 8; for point $p2$, n is 4; etc.

The switch can change state if and only if all n tokens are available. Later on, when the whole Petri Net will be constructed, open routes though the switch cause decreases of switch token numbers. This will ensure that the switch can only change its state when no route — that requires the actual state — is active. But still the switch can be part of several routes, as long as these routes require the switch to be in the same state. These requirements are captured by the Petri Net in Fig. 5(b).

Petri Net for Signals: A signal has two states: Hold and Proceed[6]. The Petri Net for a signal has two places representing the two settings Hold and Proceed. The initial marking consists of m tokens at the Hold place, where m is the number of routes which require setting of that signal. With Table 1, for the example station in Fig. 4, one finds that for for signal Sig_{1L}, m is 8, for signal Sig_{2L}, n is 6, etc.

The signal can only change setting if all m tokens are available. This will ensure that the signal can only change its state when no route that requires the actual state is active; but still the signal can be part of several routes, as long as these routes require the signal to be in the same state. These requirements are captured by the Petri Net in Fig. 5(c).

[6] This is a simplistic view – a real signal is able to indicate the speed with which it may be passed.

Petri Net for Routes: In formula 25 of Sect. 7.1 you can find that routes can be open or close. A route can be open only when all its requirements on switch settings, signal settings and units occupancies are fulfilled.

The Petri Net for a route also has two places representing the two states: Open and Closed. The initial marking consists of one token at the Closed place. The basic Petri Net for a route is shown in Figure 5(d). This corresponds to the route that has no requirements on switches, signals or units.

Construction of Petri Net for Interlocking Tables: In this paragraph we will show, how to construct the Petri Net, for the interlocking table of a station, from the four components already described (unit, switch, signal and route). This Petri Net will be made by adding extra pairs of arcs for each requirement between these components.

The example station of Fig. 4 will be composed by these components: 16 Petri Nets for routes, 14 Petri Nets for units, 5 Petri Nets for switches and 9 Petri Nets for signals — the station shown has these numbers.

A route can be open, when all units, that the route is composed from, are free (not occupied by train or blocked by another route in the station). To satisfy this requirement, between each route Petri Net and all unit Petri Nets that make up the route, a pair of arcs needs to added. Fig. 6.A shows how.

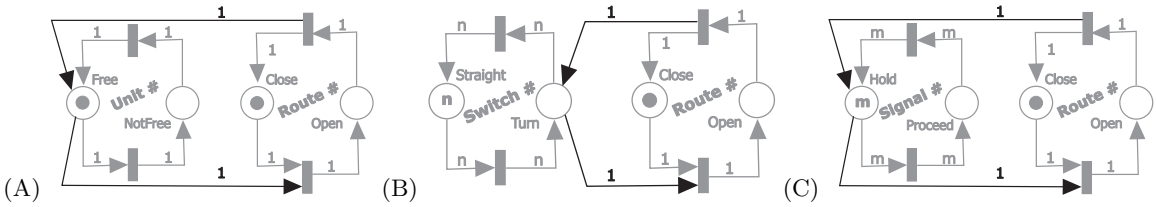


Fig. 6. Arc additions for Route (A) Units, (B) Switches and (C) Signals

For each switch requirement it must be ensured that the switch cannot change state while the route thought that switch is open. To satisfy this requirement, between each route Petri Net and all switch Petri Nets of that route, a pair of arcs have to be added. The particular insertion of arcs depend on the required state of the switch (as given in the Interlocking table). This insertion is captured in the Petri Net of Fig. 6.B. Note, that in the figure it is assumed the route requires the switch to be set to the Turn state. The case for Straight follows.

The signal can be in Proceed state only and only if the route that starts at the signal is open. How to add a pair of arcs for a signal is illustrated in Figure 6.C. This is clearly the pre–condition for opening the route, the same as the pre–condition for adding switches.

Summary: The full Petri net for the example railway station and interlocking table thus contains 16 Petri Nets for routes, 14 Petri Nets for units, 5 Petri Nets for switches, and 9 Petri Nets for signals. The interlocking table then dictates "zillions" of arcs to be inserted — so many that "readable" diagrams become impossible. Clearly, though, a case for tools. These tools can then create the complete control program, based on Petri Nets, for a station, and can check for liveness, deadlock, etc.

7.3 Integrating RAISE and Petri Nets

In [48][7] `RSL` models are given of the static and the dynamic semantics of Condition Event, Place Transition and Coloured Petri Nets. In ongoing work we are, amongst many other things, exploring the usefulness of translating Petri Nets to `RSL` for control purposes [2].

[7] See `http://www.krog-madsen.dk/page.php?id=10`

8 RAISE with Live Sequence Charts and Statecharts (CKM+MP)

Live Sequence Charts (LSCs) derive from Message Sequence Charts: [12, 42, 43, 50, 44, 64, 63, 34, 35] and are first proposed in [13] and further studied in [40, 31].

In [48, 49][8] RSL, respectively process algebraic models are given of Message and of Live Sequence Charts and of their relation to RSL.

Statecharts were introduced by and in: [23, 24, 26, 27, 25].

In [48, 49][9] models are given of Statecharts in, and of their relation to RSL.

Live Sequence Charts (on one hand) are used to specify the sequences of communication, i.e. the protocol, between two or more entities. These may be physical phenomena, processes, objects, etc.

Statecharts (on the other hand) are used to describe the sequences of states an entity may pass through in response to external stimuli.

When combined, these two methods specify both the external behaviour (LSC) and the internal behaviour (Statechart) of an entitiy.

8.1 Problem Description

The most important safety property of a railway line is that two trains are not allowed to move in opposite directions on that line. In order to ensure that the two stations at either end of the line agree on the direction trains are allowed to move at a given time. What is called a Line Direction Agreement System (LDAS) is thus introduced.

If a station wants to send a train along the line, it must first check with the LDAS if the line is open in the required direction. If so, the train may proceed along the line. If not, the opposite station must agree to changing the direction. This is only possible if there are currently no trains en–route.

8.2 External Communication: LSCs

The externally visible behaviour of the LDAS is illustrated using Live Sequence Charts. The three entities are Station A (SA), the Line Direction Agreement System (LDAS) and Station B (SB). In addition, the station managers are represented using the notation traditionally used for UML actors.

The charts in Figure 7(a) illustrate the situation when the LDAS has been turned off. One of the stations asks the LDAS to open the line, and the LDAS passes the request on to the other station, awaiting a response. The process of reversing the direction of the line is similar, see Figure 7(b).

If the station manager approves the request to reverse the direction, the LDAS will instruct the stations to open, respectively, close their end of the line, thus effecting the direction reversal, see Figure 7(c).

If the station manager rejects the request to reverse the direction, the LDAS will notify the requesting station to keep its end of the line closed, see Figure 7(d).

8.3 Internal Behaviour: Statecharts

The internal behaviour of the LDAS is illustrated by a Statechart, see Figure 8. The LDAS has some initial state called DEAD, in which no direction along the line is open. This state can only occur when the LDAS is powered up after having been shutdown (due to a failure or emergency stop). The LDAS stays in the initial state until a request to open the line in either the A to B or B to A direction arrives in the form of the InitAB or InitBA signals. Next, the opposite station will send an Agree or Disagree signal to either approve or veto the opening of the line. If the opening is vetoed, the LDAS returns to the DEAD state. If the opening is approved, the LDAS moves into a state where the line is open in one direction, represented by LockedAB or LockedBA. The station whose end of the line is closed may request the direction to be reversed by sending the AskChange signal to the LDAS. The LDAS passes the request on to the other station, awaiting the response. If it is approved, the LDAS moves into the state where the opposite direction is locked.

[8] See http://www.krog-madsen.dk/page.php?id=10

[9] See http://www.krog-madsen.dk/page.php?id=10

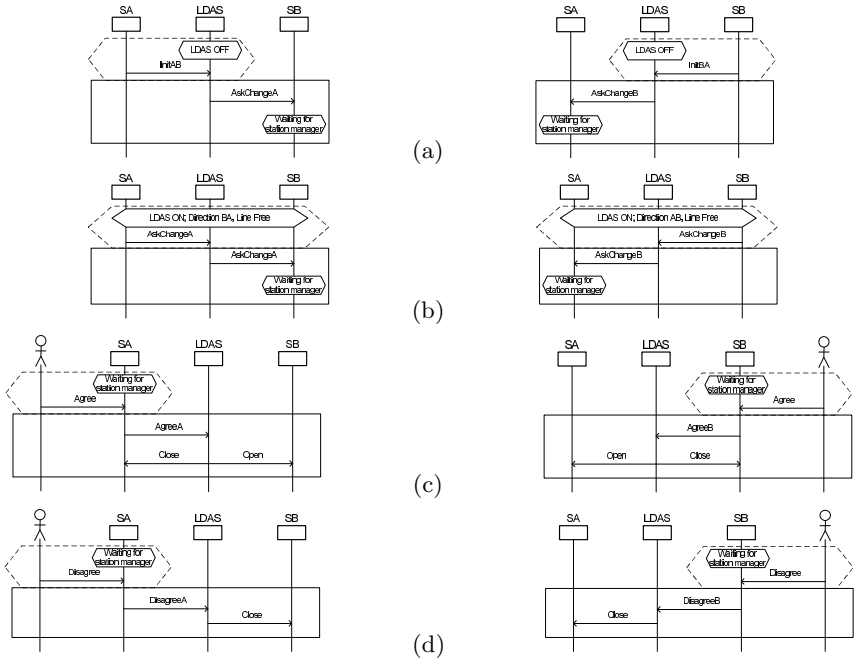


Fig. 7. (a) Initial LDAS, (b) Request Direction Reversal, (c) Request Approval (d) Request Rejection

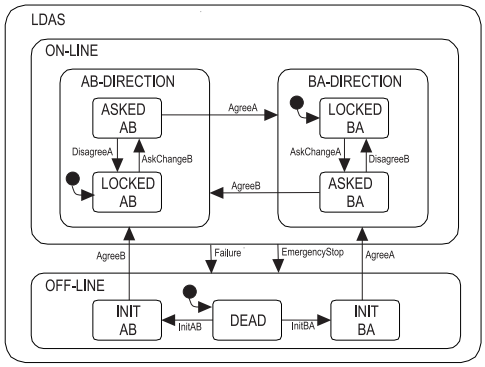


Fig. 8. Statechart for LDAS

8.4 Relation to RSL Model, Satisfaction Relation

The LSC and the Statechart models prescribe requirements to an orderly protocol aiming at a secure change of line direction. That is: That protocol is not specified in terms of RSL, but as shown, by the diagrams. Still the RSL model "survives": The actions and the state changes implied by the diagrams and to be effected, can now be individually prescribed in RSL. These RSL prescriptions are rather directly concerned with the setting of states (ie., signals) as expressed in the dynamic model of the railway net states.

In [2], and based on the work of [48, 49] we shall explore tools for translating Lice Sequence Charts (LSCs) and Statecharts (SCs) into RSL.

In [48] it is shown how to establish and verify a criterion of correctness, ie., a satisfaction relation, between LSC and SC specificatioms, on one hand, and RSL specifications on the other hand. To express and prove this satisfaction relation, as is noted in [48], clearly needs tool support.

9 Review and Future "Challenges"

9.1 Some Review Comments

The present paper has but shown a number of examples. We claim that several of these link two or more "formalisms". Yet not much, really, was said about it (except in Sect. 6.2). To properly "link–up" is a nice "challenge" — one which is next for several of us. [48] (based on ideas of [30, 29]) provides several such "link–ups".

9.2 A Research Programme: Challenge # 1

In Sect. 1.2 we mentioned: [75, 17, 14, 32, 19, 11, 60, 77, 49, 48, 65] as indicative of the research in the "integration" area. We have mostly followed ideas of George and Haxthausen [29]. These, many other publications, and annual conferences, *IFMs: Integrating Formal Methods,* together amply cover the problem area touched upon in this paper. We see it as a Grand Challenge, as a *"Man on Mars"* project: To device, ie., to research and develop a complementary set of formal specification languages (SLs), with comprehensive, cross–SL proof systems, that "covers the ground". A 20+ year challenge !

9.3 A Software Engineering Programme: Challenge # 2

But all this, ie., the R&D hinted at in Sect. 9.2, is in vain if industry, the developers of software, do not take software (and, in general hardware + software) development seriously. So, commensurate with advances in our ability to actually develop provably correct, pleasing and effective computing systems, goes, hand–in–hand, the task, the pedagogic, didactive, educational, training and socio–economically based challenge of making sure that the software (etc.) engineering graduates that have been taught this ability, also actually deploy their skills, responsibly, when in industry. Another Grand Challenge, another *"Man on Mars"* project. Another, or the same 20 years, to turn our industry into a responsible one ?

9.4 Acknowledgements

The first author acknowledges, with thanks his co–authors with whom it is a joy to work. Thanks are also due the organisers of *INT 2004,* the Third International Workshop on *Integration of Specification Techniques for Applications in Engineering,* Barcelona, Spain, March 28, 2004, namely the partners in the German Research Council's Priority Programme of the *Integration* project alluded to above. In particular the first author's thanks goes to Prof. Wolfgang Reif of Augsburg for inviting him to write and present this paper. It's been a very worthwhile and "revealing" effort !

References

1. Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, England, 1996.
2. Steffen Andersen and Steffen Holmslykke. "UML–ised" Formal Tools for the RAISE Tool Set. M.Sc. Thesis Project, Department of Computer Science and Engineering, Institute of Informatics and Mathematical Modelling, Technical University of Denmark, Building 322, Richard Petersens Plads, DK–2800 Kgs.Lyngby, Denmark, 2004–2005 2003. Pre–MSc Thesis project: Spring 2004 Lyngby; main M.Sc. Thesis Project Fall/Winter 2004/2005 UNU–IIST Macau / NUS Singapore.
3. Ales J. Anot. Using Z Specification for Railway Interlocking Safety. Periodica Polytechnica, Transport Engineering Series vol.28, no. 1–2, pp 39–53, Department of Information and Safety Systems Faculty of Electrical Engineering University of Zilina, Vel'ký diel, Zilina 010 26, Slovak Republic, 2000. .
4. Gérard Berthelot and Laure Petrucci. Specification and validation of a concurrent system: an educational project. *International Journal on Software Tools for Technology Transfer*, 3(4):372–381, September 2001. Special section on the practical use of high-level Petri Nets.
5. J. Billington and C. Janczura. Removing Deadlock from a Railway Network Specification. In *Australian Engineering Mathematics Conference (AEMC'96)*, pages 193–200, Sydney, Australia, July 1996. (Australian Engineering Mathematical Society ?).
6. Dines Bjørner. Formal Software Techniques in Railway Systems. In Eckehard Schnieder, editor, *9th IFAC Symposium on Control in Transportation Systems*, pages 1–12, Technical University, Braunschweig, Germany, 13–15 June 2000. VDI/VDE-Gesellschaft Mess– und Automatisieringstechnik, VDI-Gesellschaft für Fahrzeug– und Verkehrstechnik. Invited talk.
7. Dines Bjørner. Dynamics of Railway Nets: On an Interface between Automatic Control and Software Engineering. In *CTS2003: 10th IFAC Symposium on Control in Transportation Systems*, Oxford, UK, August 4-6 2003. Elsevier Science Ltd. Symposium held at Tokyo, Japan. Editors: S. Tsugawa and M. Aoki.
8. Dines Bjørner, Chris W. George, and Søren Prehn. Computing Systems for Railways — A Rôle for Domain Engineering. Relations to Requirements Engineering and Software for Control Applications. In *Integrated Design and Process Technology. Editors: Bernd Kraemer and John C. Petterson*, P.O.Box 1299, Grand View, Texas 76050-1299, USA, 24–28 June 2002. Society for Design and Process Science.
9. Dines Bjørner, C.W. George, and S. Prehn. *Scheduling and Rescheduling of Trains*, chapter 8, pages 157–184. *Industrial Strength Formal Methods in Practice,* Eds.: Michael G. Hinchey and Jonathan P. Bowen. FACIT, Springer–Verlag, London, England, 1999.
10. Grady Booch, Jim Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
11. Robert Büssow, Robert Geisler, and Marcus Klar. Specifying safety-critical embedded systems with Statecharts and Z: A case study. In E. Astesiano, editor, *Fundamental Approaches to Software Engineering: First International Conference, FASE'98, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March/April 1998*, volume 1382 of *Lecture Notes in Computer Science*, pages 71–87. Springer-Verlag, 1998.
12. CCITT. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992.
13. Werner Damm and David Harel. LSCs: Breathing life into Message Sequence Charts. *Formal Methods in System Design*, 19:45–80, 2001. Early version appeared as Weizmann Institute Tech. Report CS98-09, April 1998. An abridged version appeared in *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-based Distributed Systems* (FMOODS'99), Kluwer, 1999, pp. 293–312.
14. Clemens Fischer. CSP-OZ: A combination of Object-Z and CSP. Technical Report TRCF-97-2, Universitt Oldenburg, 1997.
15. John Fitzgerald and Peter Gorm Larsen. *Software System Design: Formal Methods into Practice*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1997. To appear.
16. Ana Funes and Chris W. George. Formal Foundations in RSL for UML Class Diagrams. Research Report 253, UNU/IIST, P.O. Box 3058, Macau, May 2002. Published as chapter VIII Formalizing UML Class Diagrams of UML and the Unified Process, Liliana Favre (ed.).
17. A. Galloway. *Integrated Formal Methods*. PhD thesis, University of Teeside, 1996.
18. Chris W. George and Yong Xia. An Operational Semantics for Timed RAISE. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods*, pages 1008–1027. FME, Springer–Verlag, 1999.
19. W. Grieskamp, M. Heiseland, and H. Dörr. Specifying embedded systems with Statecharts and Z: An agenda for cyclic software components. In E. Astesiano, editor, *Fundamental Approaches to Software Engineering: First International Conference, FASE'98, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March/April 1998*, volume 1382 of *Lecture Notes in Computer Science*, pages 88–106. Springer-Verlag, 1998.
20. Object Management Group. *OMG Unified Modelling Language Specification*. OMG/UML, http://www.omg.org/uml/, version 1.5 edition, March 2003. www.omg.org/cgi-bin/doc?formal/03-03-01.

21. Kirsten Mark Hansen. Validation of a railway interlocking model. In M. Bertran M. Naftalin, T. Denvir, editor, *FME'94: Industrial Benefit of Formal Methods*, pages 582–601. Springer-Verlag, October 1994.
22. K.M. Hansen. *Linking Safety Analysis to Safety Requirements*. PhD thesis, Department of Computer Science, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, August 1996.
23. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
24. David Harel. On visual formalisms. *Communications of the ACM*, 33(5), 514–530 1988.
25. David Harel and Eran Gery. Executable object modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.
26. David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark B. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *Software Engineering*, 16(4):403–414, 1990.
27. David Harel and Amnon Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.
28. A. Haxthausen and T. Gjaldbæk. Modelling and Verification of Interlocking Systems for Railway Lines. In *10th IFAC Symposium on Control in Transportation Systems, Tokyo, Japan*, August 4–6 2003.
29. Anne Haxthausen. Some approaches for integration of specification techniques (invited extended abstract), 2000.
30. Anne Haxthausen and Yong Xia. Linking DC together with TRSL. In *Proceedings of 2nd International Conference on Integrated Formal Methods (IFM'2000), Schloss Dagstuhl, Germany, November 2000*, number 1945 in Lecture Notes in Computer Science, pages 25–44. Springer-Verlag, 2000.
31. Patrick Heymans and Yves Bontemps. Turning high-level Live Sequence Charts into automata. In Tarja Systa and Albert Zundorf, editors, *Proceedings of the First International Workshop on Scenarios and State Machines (SCESM), (ICSE'02 workshop)*, 2002.
32. C. A. R. Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
33. C.A.R. Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985.
34. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1996.
35. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1999.
36. Ivar Jacobson, Grady Booch, and Jim Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
37. Kurt Jensen. *Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use, Volume 1 Basic Concepts*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1992.
38. Ekkart Kindler, Wolfgang Reisig, Hagen Volzer, and Rolf Walter. Petri net based verification of distributed algorithms: An example. *Formal Aspects of Computing*, 9(4):409–424, 1997.
39. T. King. Formalising British Rail's Signalling Rules. In M. Bertran M. Naftalin, T. Denvir, editor, *FME'94: Industrial Benefit of Formal Methods*, pages 45–54. Springer-Verlag, October 1994.
40. Jochen Klose and Hartmut Wittke. An automata based interpretation of Live Sequence Charts. In T. Margaria and W. Yi, editors, *TACAS 2001*, LNCS 2031, pages 512–527. Springer-Verlag, 2001.
41. Lars M. Kristensen, Soren Christensen, and Kurt Jensen. The practitioner's guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.
42. Peter B. Ladkin and Stefan Leue. Analysis of Message Sequence Charts. Technical Report IAM 92-013, Institute for Informatics and Applied Mathematics, University of Berne, Bern, Switzerland, 1992.
43. Peter B. Ladkin and Stefan Leue. What do Message Sequence Charts mean? In *FORTE*, pages 301–316, 1993.
44. Peter B. Ladkin and Stefan Leue. Interpreting Message Flow Graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.
45. L. Li and Jifeng He. A denotational semantics of Timed RSL using Duration Calculus. Research Report 168, UNU/IIST, P.O.Box 3058, Macau, 1999. Published in Proceedings of The Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99), pp. 492-503, IEEE Computer Society Press.
46. L. Li and Jifeng He. Towards a denotational semantics of Timed RSL using Duration Calculus. Research Report 161, UNU/IIST, P.O.Box 3058, Macau, April 1999. Accepted for publication by Chinese Journal of Advanced Software Research.
47. Morten Peter Lindegaard, Peter Viuf, and Anne Haxthausen. Modelling Railway Interlocking Systems. In *Proceedings of the 9th IFAC Symposium on Control in Transportation Systems 2000, June 13-15, 2000, Braunschweig, Germany*, pages 211–217, 2000.
48. Christian Krog Madsen. Integration of Specification Techniques. M.Sc. Thesis Project, Department of Computer Science and Engineering, Institute of Informatics and Mathematical Modelling, Technical University of Denmark, Building 322, Richard Petersens Plads, DK–2800 Kgs.Lyngby, Denmark, 30 November 2003.
49. Christian Krog Madsen. Study of Graphical and Temporal Specification Techniques. Pre–Thesis Project, Department of Computer Science and Engineering, Institute of Informatics and Mathematical Modelling, Technical University of Denmark, Building 322, Richard Petersens Plads, DK–2800 Kgs.Lyngby, Denmark, June 2003.

50. S. Mauw and M. A. Reniers. An algebraic semantics of basic Message Sequence Charts. *The Computer Journal*, 37(4):269–277, 1994.
51. R. Milner. *Communication and Concurrency*. C.A.R. Hoare Series in Computer Science. Prentice Hall, 1989.
52. M.J. Morley. Modelling British Rail's Interlocking Logic: Geographic Data Correctness. Technical Report ECS-LFCS-91-186, University of Edinburgh, 1991.
53. M.J. Morley. Safety in railway signalling data: A behavioural analysis. In J. Joyce and C. Seger, editors, *Proc. 6th annual workshop on higher order logic and its applications, Vancouver, 4-6 August*, pages 465–474. Springer–Verlag Lecture Notes in Computer Science, Vol.780, 1993–4.
54. M.J. Morley. *Safety Assurance in Interlocking Design*. PhD thesis, University of Edinburgh, 1996.
55. M.J. Morley. Safety-level communication in railway interlockings. *Science of Computer Programming*, 29(1-2):147–170, July 1997.
56. Carl Adam Petri. *Kommunikation mit Automaten*. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.
57. Martin Pěnička, Albena Kirilova Strupchanska, and Dines Bjørner. Train Maintenance Routing. In *FORMS'2003: Symposium on Formal Methods for Railway Operation and Control Systems*. L'Harmattan Hongrie, 15–16 May 2003. Conf. held at Techn.Univ. of Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany.
58. RAISE Language Group. *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall Int., 1992.
59. RAISE Method Group. *The RAISE Development Method*. BCS Practitioner Series. Prentice Hall Int., 1995.
60. G. Reggio and L. Repetto. Casl-Chart: A combination of Statecharts and of the algebraic specification language Casl. Technical Report DISI-TR-00-2, DISI, Università di Genova, 2000.
61. W. Reisig. *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets*. Springer Verlag, 1998.
62. Wolfgang Reisig. *A Primer in Petri Net Design*. Springer-Verlag, 1992.
63. M. Reniers. Static semantics of Message Sequence Charts, 1995.
64. M.A. Reniers. Syntax requirements of Message Sequence Charts. In R. Braek and A. Sarma, editors, *Proceedings of the 7th SDL Forum*, 1995.
65. Martin-Große Rhode. *Semantic Integration of Heterogeneous Software Specifications*. Monographs in Theoretical Computer Science. An EATCS Series. Springer–Verlag, Heidelberg and Berlin, Germany, 2004.
66. A.W. Roscoe. *Theory and Practice of Concurrency*. C.A.R. Hoare Series in Computer Science. Prentice–Hall, 1997.
67. Jim Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
68. Steve Schneider. *Concurrent and Real–time Systems — The CSP Approach*. Worldwide Series in Computer Science. John Wiley & Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, January 2000.
69. A.C. Simpson, J.C.P. Woodcock, and J.W. Davies. The mechanical verification of Solid State Interlocking geographic data. In L. Groves and S. Reeves, editors, *Proceedings of Formal Methods Pacific*, pages 223–242, Wellington, New Zealand, 9–11 July 1997. Springer–Verlag.
70. J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, January 1988.
71. Albena Kirilova Strupchanska, Martin Pěnička, and Dines Bjørner. Railway Staff Rostering. In *FORMS2003: Symposium on Formal Methods for Railway Operation and Control Systems*. L'Harmattan Hongrie, 15–16 May 2003. Conf. held at Techn.Univ. of Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany.
72. W.M.P. van der Aalst and M.A. Odijk. Analysis of railway stations by means of interval timed colored Petri Nets. *Real-Time Systems*, 9(3):241–263, 1995. .
73. Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley Publ. Co., October 13 1998. 144 pages, ASIN: 0201379406, Paperback.
74. Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Publ. Co., 2nd edition, August 29 2003. 240 pages, ISBN: 0321179366, Paperback.
75. M. Weber. Combining Statecharts and Z for the design of safety-critical control systems. In M. Gaudel and J. Woodcock, editors, *FME 96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 307–326. Springer-Verlag, 1996.
76. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.
77. J. C. P. Woodcock and Arthur Hughes. Unifying theories of parallel programming. In Chris George and H. Miao, editors, *Formal Methods and Software Engineering: 4th International Conference on Formal Engineering Methods, ICFEM 2002 Shanghai, China*, volume 2495 of *Lecture Notes in Computer Science*, pages 24–37. Springer-Verlag, October 21–25 2002.
78. Wang Yi. *A Calculus of Real Time Systems*. PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1991.
79. Chaochen Zhou and Michael R. Hansen. *Duration Calculus: A Formal Approach to Real–time Systems*.

Monographs in Theoretical Computer Science. An EATCS Series. Springer–Verlag, 2004.

80. Chaochen Zhou, C.A.R. Hoare, and A.P. Ravn. A Calculus of Durations. *Information Proc. Letters*, 40(5), 1992.