RAISE/RSL Examples
The CoMet Project

Dines Bjørner
Fredsvej 11, DK-2840 Holte, Denmark
bjorner@gmail.com -- www.imm.dtu.dk/~db

31 December 2009: Compiled: February 2, 2010: 00:00 ECT

Contents

1. Guide to Examples

- We bring all the examples of the Lecture Notes at http://www.imm.dtu.dk/~db/comet/comet-rsl-examples.pdf
- The display line texts correspond to Lecture Note sections and appendix sections etc.
- The examples appear consecutively enumerated.

2. Specification Ontology

1 2.1 Entities

The example is that of aspects of a transportation net. You may think of such a net as being either a road net, a rail net, a shipping net or an air traffic net. Hubs are then street intersections, train stations, harbours, respectively airports. Links are then street segments between immediately adjacent intersections, rail tracks between train stations, sea lanes between harbours, respectively air lanes between airports.

1 There are hubs and links.

2 There are nets, and a net consists of a set of two or more hubs and one or more links.

3 There are hub and link identifiers.

4 Each hub (and each link) has an own, unique hub (respectively link) identifier (which can be observed (ω) from the hub [respectively link]).

type
[1] H, L,
[2] N = H-**set** × L-**set**
axiom
[2] ∀ (hs,ls):N • **card** hs≥2 ∧ **card** ks≥1
type
[3] HI, LI
value
[4] ωHI: H → HI, ωLI: L → LI
axiom
[4] ∀ h,h′:H, l,l′:L • h≠h′ ⇒ ωHI(h)≠ωHI(h′) ∧ l≠l′⇒ωLI(l)≠ωLI(l′)

In order to model the physical (i.e., domain) fact that links are delimited by two hubs and that one or more links emanate from and are, at the same time, incident upon a hub we express the following:

5 From any link of a net one can observe the two hubs to which the link is connected. We take this 'observing' to mean the following: from any link of a net one can observe the two distinct identifiers of these hubs.

6 From any hub of a net one can observe the identifiers of one or more links which are connected to the hub.

7 Extending Item [5]: the observed hub identifiers must be identifiers of hubs of the net to which the link belongs.

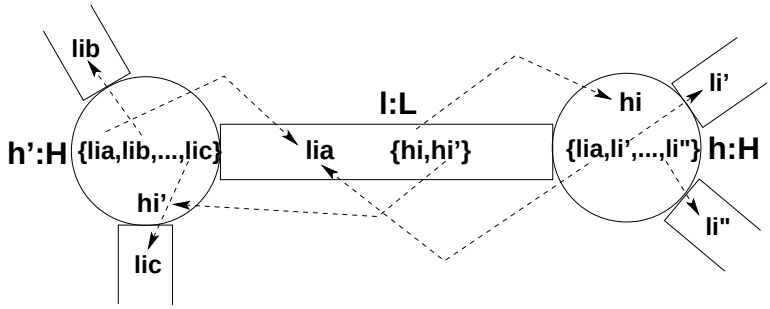

Figure 1: Connected links and hubs with observable identifiers

8 Extending Item [6]: the observed link identifiers must be identifiers of links of the net to which the hub belongs.

value
[5] ωHIs: L → HI-**set**,
[6] ωLIs: H → LI-**set**,
axiom
[5] ∀ l:L • **card** ωHIs(l)=2 ∧
[6] ∀ h:H • **card** ωLIs(h)≥1 ∧
∀ (hs,ls):N •
[5] ∀ h:H • h ∈ hs ⇒ ∀ li:LI • li ∈ ωLIs(h)
⇒ ∃ l′:L • l′ ∈ ls ∧ li=ωLI(l′) ∧ ωHI(h) ∈ ωHIs(l′) ∧
[6] ∀ l:L • l ∈ ls ⇒ ∃ h′,h″:H • {h′,h″}⊆hs ∧ ωHIs(l)={ωHI(h′),ωHI(h″)}
[7] ∀ h:H • h ∈ hs ⇒ ωLIs(h) ⊆ iols(ls)
[8] ∀ l:L • l ∈ ls ⇒ ωHIs(h) ⊆ iohs(hs)
value
iohs: H-**set** → HI-**set**, iols: L-**set** → LI-**set**
iohs(hs) ≡ {ωHI(h)|h:H•h ∈ hs}
iols(ls) ≡ {ωLI(l)|l:L•l ∈ ls}

In the above extensive example we have focused on just five entities: nets, hubs, links and their identifiers. The nets, hubs and links can be seen as separable phenomena. The hub and link identifiers are conceptual models of the fact that hubs and links are connected — so the identifiers are abstract models of 'connection', i.e., the mereology of nets, that is, of how nets are composed. These identifiers are attributes of entities.

Links and hubs have been modelled to possess link and hub identifiers. A link's "own" link identifier enables us to refer to the link, A link's two hub identifiers enables us to refer to the connected hubs. Similarly for the hub and link identifiers of hubs and links.

9 A hub, h_i, state, $h\sigma$, is a set of hub traversals.

10 A hub traversal is a triple of link, hub and link identifiers $(l_{i_{in}}, h_{i_i}, l_{i_{out}})$ such that $l_{i_{in}}$ and $l_{i_{out}}$ can be observed from hub h_i and such that h_{i_i} is the identifier of hub h_i.

11 A hub state space is a set of hub states such that all hub states concern the same hub.

type
[9] HT = (LI×HI×LI)
[10] HΣ = HT-**set**
[11] HΩ = HΣ-**set**
value
[10] ωHΣ: H → HΣ
[11] ωHΩ: H → HΩ
axiom
∀ n:N,h:H•h ∈ ωHs(n)⇒wf_HΣ(ωHΣ(h))∧wf_HΩ(h,ωHΩ(h))
value
wf_HΣ: HΣ → **Bool**, wf_HΩ: H×HΩ → **Bool**
wf_HΣ(hσ) ≡ ∀ (li,hi,li′),(_,hi′,_):HT•(li,hi,li′)∈ hσ ⇒ {li,li′}⊆ωLIs(h)∧hi=ωHI(h)∧hi′=hi
wf_HΩ(h,hω) ≡ ∀ hσ:HΣ•hσ ∈ hω⇒wf_HΣ(hσ)∧hσ≠{} ⇒
let (li,hi,li′):HT•(li,hi,li′)∈ hσ **in** hi=ωHI(h) **end**

2 2.2 Actions

12 Our example action is that of setting the state of hub.

13 The setting applies to a hub

14 and a hub state in the hub state space

13 and yields a "new" hub.

15 The before and after hub identifier remains the same.

16 The before and after hub state space remains the same.

17 The result hub is in the hub state space — subject to some probability distribution.

value
p:**Real**, **axiom** 0<p≤1, typically p≃ $1 - 10^{-7}$
$\overline{p}$:**Real**, **axiom** $\overline{p}$=1−p

[12] set_HΣ: H × HΣ → H
[13] set_HΣ(h,hσ) **as** h′
[14] **pre** hσ ∈ ωHΩ(h)

[15] **post** ωHI(h)=ωHI(h′)∧
[16] ωHΩ(h)=ωHΩ(h′)∧
[17] ωHΣ(h′)=(⌈⌉{hσ′|hσ′:HΣ•hσ′∈ ωΩ(h)\{hσ}})$_{\overline{p}}$⌈⌉$_p$hσ

The non-deterministic internal choice operator expression $s_{\overline{p}}⌈⌉_p s'$with probability p has value s' and with probability $\overline{p}$ has value s. The prefix internal choice operator expression ⌈⌉{$h\sigma_i, h\sigma_j, \ldots, h\sigma_k$} h non-deterministically as one of the values in the set $\{h\sigma_i, h\sigma_j, \ldots, h\sigma_k\}$, that is, is the same as $h\sigma_i ⌈⌉ h\sigma_j ⌈⌉ \ldots ⌈⌉ h\sigma_k$

3 2.3 Events

18 A hub is in some state, $h\sigma$.

19 An action directs it to change to state $h\sigma'$ where $h\sigma' \neq h\sigma$.

20 But after that action the hub remains either in state $h\sigma$ or is possibly in a third state, $h\sigma''$ where hσ″ ∉ {hσ,hσ′}.

21 Thus an "interesting event" has occurred !

∃ n:N,h:H,hσ,hσ′:HΣ•h ∈ ωHs(n)∧
[19,20] {hσ,hσ′}⊆ωHΩ(h)∧**card**{hσ,hσ′}=2 ∧
[18] ωHΣ(h)=hσ ;
[19] **let** h′ = set_HΣ(h,hσ′) **in**
[20] ωHΣ(h′)∈ ωHΣ(h′)\{hσ′} ⇒
[21] "interesting event" **end**

It only makes sense to change hub states if there are more than just one single such state.

4 2.4 Behaviours: Blinking Semaphores

22 Let h be a hub of a net n.

23 Let $h\sigma$ and $h\sigma'$ be two distinct states of h.

24 Let $ti : TI$ be some time interval.

25 Let h start in an initial state $h\sigma$.

26 Now let hub h undergo an ongoing sequence of n changes

26a from $h\sigma$ to $h\sigma'$ and

26b then, after a wait of ti seconds,

26c and then back to $h\sigma$.

26d After n blinks a pause, $tp : TI$, is made and blinking restarts.

```
type
  TI
value
  ti,tj:TI [axiom tj¿¿ti]
  n:Nat,
  [26] blinking: H × HΣ × HΣ → Unit
  [26] blinking(h,hσ,hσ′,m) in
  [25]   let h′ = set_HΣ(h,hσ) in
  [26c]  wait ti ;
  [26a]  let h″ = set_HΣ(h′,hσ′) in
  [26c]  wait ti ;
  [26]   if m=1
  [26]     then skip
  [26]     else blinking(h,hσ,hσ′,m−1) end end end
  [26]    wait tj ;
  [26d]   blinking(h,hσ,hσ′,m)
  [23]   pre {hσ,hσ′}⊆ωHΩ(h)∧hσ≠hσ′
  [26]       ∧ initial m=100
```

3. Domain Engineering

5 3.1 Some Transport Net Business Processes

With respect to one and the same underlying road net we suggest some business-processes and invite the reader to rough-sketch these.

27 **Private citizen automobile transports:** Private citizens use the road net for pleasure and for business, for sightseeing and to get to and from work.

28 **Public bus (&c.) transport:** Province and city councils contract bus (&c.) companies to provide regular passenger transports according to timetables and at cost or free of cost.

29 **Road maintenance and repair:** Province and city councils hire contractors to monitor road (link and hub) surface quality, to maintain set standards of surface quality, and to "emergency" re-establish sudden occurrences of low quality.

30 **Toll road traffic:** State and province governments hire contractors to run toll road nets with toll booth plazas.

31 **Net revision: road (&c.) building:** State government and province and city councils contract road building contractors to extend (or shrink) road nets.

The detailed description of the above rough-sketched business process synopses now becomes part of the domain description as partially exemplified in the previous and the next many examples.

6 3.2 Intrinsics

Most of the descriptions of Sect. **??** model intrinsics. We add a little more:

32 A link traversal is a triple of a (from) hub identifier, an along link identifier, and a (towards) hub identifier

33 such that these identifiers make sense in any given net.

34 A link state is a set of link traversals.

35 And a link state space is a set of link states.

value
  n:N
type
  [32] LT′ = HI × LI × HI
  [33] LT = {|lt:LT′•wfLT(lt)(n)|}
  [34] LΣ′ = LT-**set**
  [34] LΣ = {|lσ:LΣ′•wf_LΣ(lσ)(n)|}
  [35] LΩ′ = LΣ-**set**
  [35] LΩ = {|lω:LΩ′•wf_LΩ(lω)(n)|}
value
  [33] wfLT: LT → N → **Bool**
  [33] wfLT(hi,li,hi′)(n) ≡
  [33] ∃ h,h′:H•{h,h′}⊆ωHs(n)∧
  [33] ωHI(h)=hi∧ωHI(h′)=hi′∧
  [33] li ∈ ωLIs(h)∧li ∈ ωLIs(h′)

The wf_LΣ and wf_LΩ can be defined like the corresponding functions for hub states and hub state spaces.

7 3.3 Support Technologies

Some road intersections (i.e., hubs) are controlled by semaphores alternately shining **red**–**yellow**–**green** in carefully interleaved sequences in each of the in-directions from links incident upon the hubs. Usually these signalings are initiated as a result of road traffic sensors placed below the surface of these links. We shall model just the signaling:

36 There are three colours: **red**, **yellow** and **green**.

37 Each hub traversal is extended with a colour and so is the hub state.

38 There is a notion of time interval.

39 Signaling is now a sequence, $\langle(h\sigma', t\delta'), (h\sigma'', t\delta''), \ldots, (h\sigma'^{\cdots\prime}, t\delta'^{\cdots\prime})\rangle$ such that the first hub state $h\sigma'$ is to be set first and followed by a time delay $t\delta'$ whereupon the next state is set, etc.

40 A semaphore is now abstracted by the signalings that are prescribed for any change from a hub state $h\sigma$ to a hub state $h\sigma'$.

type
[36] Colour == red | yellow | green
[37] X = LI×HI×LI×Colour [crossings **of** a hub]
[37] HΣ = X-**set** [hub states]
[38] TI [time interval]
[39] Signalling = (HΣ × TI)*
[40] Semaphore = (HΣ × HΣ) $\xrightarrow[m]{}$ Signalling
value
[37] ωHΣ: H → HΣ
[40] ωSemaphore: H → Sema,
[41] chg_HΣ: H × HΣ → H
[41] chg_HΣ(h,hσ) **as** h′
[41] **pre** hσ ∈ ωHΩ(h) **post** ωHΣ(h′)=hσ

[39] chg_HΣ_Seq: H × HΣ → H
[39] chg_HΣ_Seq(h,hσ) ≡
[39] **let** sigseq = (ωSemaphore(h))(ωΣ(h),hσ) **in**
[39] sig_seq(h)(sigseq) **end**
[39] sig_seq: H → Signalling → H
[39] sig_seq(h)(sigseq) ≡
[39] **if** sigseq=⟨⟩ **then** h **else**
[39] **let** (hσ,tδ) = **hd** sigseq **in let** h′ = chg_HΣ(h,hσ);
[39] **wait** tδ;
[39] sig_seq(h′)(**tl** sigseq) **end end end**

8 3.4 Rules

We give two examples related to railway systems where train stations are the hubs and the rail tracks between train stations are the links:

41 Trains arriving at or leaving train stations:

a) (In China:) No two trains
b) must arrive at or leave a train station
c) in any two minute time interval.

42 Trains travelling "down" a railway track. We must introduce a notion of links being a sequence of adjacent sectors.

a) Trains must travel in the same direction;
b) and there must be at least one "free-from-trains" sector
c) between any two such trains.

We omit showing somewhat "lengthy" formalisations.

9 3.5 Timetable Scripts

43 Time is considered discrete. Bus lines and bus rides have unique names (across any set of time tables).

44 A *TimeTable* associates *Bus Line Identifiers* (*blid*) to sets of *Journies*.

45 *Journies* are designated by a pair of a *BusRoute* and a set of *BusRides*.

46 A *BusRoute* is a triple of the *Bus Stop* of origin, a list of zero, one or more intermediate *Bus Stop*s and a destination *Bus Stop*.

47 A set of *BusRides* associates, to each of a number of *Bus Identifiers* (*bid*) a *Bus Schedule*.

48 A *Bus Schedule* is a triple of the initial departure *Time*, a list of zero, one or more intermediate bus stop *Time*s and a destination arrival *Time*.

49 A *Bus Stop* (i.e., its position) is a *Fraction* of the distance along a link (identified by a *Link Identifier*) *from* an *identified hub to* an *identified hub*.

50 A *Fraction* is a **Real** properly between 0 and 1.

51 The *Journies* must be *well_formed* in the context of some net.

52 A set of journies is well-formed if

53 the bus stops are all different,
54 a bus line is embedded in some line of the net, and
55 all defined bus trips of a bus line are equivalent.

type
[43] T, BLId, BId
[44] TT = BLId $\xrightarrow[m]{}$ Journies
[45] Journies′ = BusRoute × BusRides
[46] BusRoute = BusStop × BusStop* × BusStop
[47] BusRides = BId $\xrightarrow[m]{}$ BusSched
[49] BusSched = T × T* × T
[50] BusStop == mkBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
[51] Frac = {|r:**Real**•0<r<1|}
[45] Journies = {|j:Journies′•∃ n:N • wf_Journies(j)(n)|}
value
[52] wf_Journies: Journies → N → **Bool**
[52] wf_Journies((bs1,bsl,bsn),js)(hs,ls) ≡
[53] diff_bus_stops(bs1,bsl,bsn) ∧
[54] is_net_embedded_bus_line(⟨bs1⟩^bsl^⟨bsn⟩)(hs,ls) ∧
[55] commensurable_bus_trips((bs1,bsl,bsn),js)(hs,ls)

10 3.5 Contracts

An example contract can be 'schematised':

cid: **contractor** cor **contracts sub-contractor** cee
to perform operations
`{"conduct","cancel","insert","subcontract"}`
with respect to timetable tt.

We assume a context (a global state) in which all contract actions (including contracting) takes place and in which the implicit net is defined.

Concrete examples of actions can be schematised:

(a) cid: **conduct bus ride** (blid,bid) **to start at time** t
(b) cid: **cancel bus ride** (blid,bid) **at time** t
(c) cid: **insert bus ride like** (blid,bid) **at time** t

The schematised license shown earlier is almost like an action; here is the action form:

(d) cid: **contractor** cnm′ **is granted a contract** cid′
to perform operations
{"conduct","cancel","insert",sublicense"}
with respect to timetable tt′.

All actions are being performed by a sub-contractor in a context which defines that sub-contractor *cnm*, the relevant net, say *n*, the base contract, referred here to by *cid* (from which this is a sublicense), and a timetable *tt* of which *tt′* is a subset. contract name *cnm′* is new and is to be unique. The subcontracting action can (thus) be simply transformed into a contract as shown on Page 11.

type
  Action = CNm × CId × (SubCon | SmpAct) × Time
  SmpAct = Conduct | Cancel | Insert
  Conduct == μConduct(s_blid:BLId,s_bid:BId)
  Cancel == μCancel(s_blid:BLId,s_bid:BId)
  Insert = μInsert(s_blid:BLId,s_bid:BId)
  SubCon == μSubCon(s_cid:CId,s_cnm:CNm,s_body:body)
    where body = (s_ops:Op-**set**,s_tt:TT)

We omit formalising the semantics of these syntaxes. A formalisation could be expressed (in CSP [?]) with each bus, each licensee (and licensor), time and the road net bus traffic being processes, etc.

11 3.6 Management

We relate to Example ??:

56 The **conduct, cancel** and **insert bus ride** actions are operational functions.

57 The actual **subcontract** actions are tactical functions;

58 but the decision to carry out such a tactical function may very well be a strategic function as would be the acquisition or disposal of busses.

59 Forming new timetables, in consort with the contractor, is a strategic function.

We omit formalisations.

12 3.7 Human Behaviour

Cf. Examples ??–??:

60 no failures to conduct a bus ride must be classified as diligent;

61 rare failures to conduct a bus ride must be classified as sloppy if no technical reasons were the cause;

62 occasional failures ··· as delinquent;

63 repeated patterns of failures ··· as criminal.

We omit showing somewhat "lengthy" formalisations.

4. Requirements Engineering

13 4. Goals of a Toll Road System

- A goal for a toll road system may be
  - to decrease the travel time between certain hubs and
  - to lower the number of traffic accidents between certain hubs,

14 4. Goals of Toll Road System Software

- The goal of the toll road system software is to help automate
  - the recording of vehicles entering, passing and leaving the toll road system
  - and collecting the fees for doing so.

15 4. Arguing Goal-satisfaction of a Toll Road System

- By endowing links and hubs with average traversal times for both ordinary road and for toll road links and hubs
  - one can calculate traversal times between hubs
  - and thus argue that the toll road system satisfies [significantly] "quicker" traversal times.
- By endowing links and hubs with traffic accident statics (real, respectively estimated)
  - for both ordinary road and for toll road links and hubs
  - one can calculate estimated traffic accident statics between all hubs
  - and thus argue that the combined ordinary road plus toll road system satisfies [significantly] lower traffic fatalities.

16 4. Arguing Goal-satisfaction of Toll Road System Software

- By recording
  - tickets issued and collected at toll boths,
  - toll road hubs and links entered and left
  - as per the requirements specification brought in Examples **??**-**??**,
- we can eventually argue that
  - the requirements of Examples **??**-**??**
  - help satisfy the goal of Example **??**.

17 4.1 Rough-sketching a Re-engineered Road Net

Our sketch centers around a toll road net with toll booth plazas. The BPR focuses first on entities, actions, events and behaviours (Sect. **??**), then on the six domain facets (Sects. **??**–**??**).

64 **Re-engineered Entities:** We shall focus on a linear sequence of toll road intersections (i.e., hubs) connected by pairs of one-way (opposite direction) toll roads (i.e., links). Each toll road intersection is connected by a two way road to a toll plaza. Each toll plaza contains a pair of sets of entry and exit toll booths. (Example **??** brings more details.)

65 **Re-engineered Actions:** Cars enter and leave the toll road net through one of the toll plazas. Upon entering, car drivers receive, from the entry booth, a plastic/paper/electronic ticket which they place in a special holder in the front window. Cars arriving at intermediate toll road intersections choose, on their own, to turn either "up" the toll road or "down" the toll road — with that choice being registered by the electronic ticket. Cars arriving at a toll road intersection may choose to "circle" around that intersection one or more times — with that choice being registered by the electronic ticket. Upon leaving, car drivers "return" their electronic ticket to the exit booth and pay the amount "asked" for.

66 **Re-engineered Events:** A car entering the toll road net at a toll both plaza entry booth constitutes an event. A car leaving the toll road net at a toll both plaza entry booth constitutes an event. A car entering a toll road hub constitutes an event. A car entering a toll road link constitutes an event.

67 **Re-engineered Behaviours:** The journey of a car,from entering the toll road net at a toll booth plaza, via repeated visits to toll road intersections interleaved with repeated visits to toll road links to leaving the toll road net at a toll booth plaza, constitutes a behaviour — withreceipt of tickets, return of tickets and payment of fees being part of these behaviours. Notice that a toll road visitor is allowed to cruise "up" and "down" the linear toll road net – while (probably) paying for that pleasure (through the recordings of "repeated" hub and link entries).

68 **Re-engineered Intrinsics:** Toll plazas and abstracted booths are added to domain intrinsics.

69 **Re-engineered Support Technologies:** There is a definite need for domain-describing the failure-prone toll plaza entry and exit booths.

70 **Re-engineered Rules and Regulations:** Rules for entering and leaving toll booth entry and exit booths must be described as must related regulations. Rules and regulations for driving around the toll road net must be likewise be described.

71 **Re-engineered Scripts:** No need.

72 **Re-engineered Management and Organisation:** There is a definite need for domain describing the management and possibly distributed organisation of toll booth plazas.

73 **Re-engineered Human Behaviour:** Humans, in this case car drivers, may not change their behaviour in the spectrum from diligent and accurate via sloppy and delinquent to outright traffic-law breaking – so we see no need for any "re-engineering".

18 4.2.1 Projection

Our requirements is for a simple toll road: a linear sequence of links and hubs outlined in Example **??**: see Items [1–11] of Example **??** on page ?? and Items [32–35] of Example **??** on page ??.

19 4.2.2 Instantiation

Here the toll road net topology as outlined in Example **??** on page ?? is introduced: a straight sequence of toll road hubs pairwise connected with pairs of one way links and with each hub two way link connected to a toll road plaza.

type
  H, L, P = H
  $N' = (H \times L) \times H \times ((L \times L) \times H \times (H \times L))^*$
  $N'' = \{|n{:}N' \bullet wf(n)|\}$
value
  wf_N'': $N' \rightarrow$ **Bool**
  wf_N''((h,l),h',llhpl) ≡ ... 6 lines ... !
  αN: $N'' \rightarrow N$
  αN((h,l),h',llhpl) ≡ ... 2 lines ... !

wf_N'' secures linearity; αN allows abstraction from more concrete N'' to more abstract N.

20 4.2.3 Determination

Pairs of links between toll way hubs are open in opposite directions; all hubs are open in all directions; links between toll way hubs and toll plazas are open in both directions.

type
  $L\Sigma = (HI \times HI)$-**set**, $L\Omega = L\Sigma$-**set**
  $H\Sigma = (LI \times LI)$-**set**, $H\Omega = H\Sigma$-**set**
  $N' = (H \times L) \times H \times ((L \times L) \times H \times (H \times L))^*$
value
  $\omega L\Sigma$: $L \rightarrow L\Sigma$, $\omega L\Omega$: $L \rightarrow L\Omega$

```
  ωHΣ: H → HΣ, ωHΩ: H → HΩ
axiom
  ∀ ((h,l),h′,llhhl:⟨(l′,l″),h″,(h‴,l‴)⟩^llhhl′):N″ •
    ωLΣ(l)={(ωHI(h),ωHI(h′)),(ωHI(h′),ωHI(h))}∧
    ωLΣ(l‴)={(ωHI(h″),ωHI(h‴)),(ωHI(h‴),ωHI(h″))}∧
    ∀ i,i+1:Nat • {i,i+1}⊆inds llhhl ⇒
      let ((li,li′),hi,(hi″,li″))=llhhl(i), (_,hj,(hj″,lj″))=llhhl(i+1) in
      ωLΩ(li)= {{(ωHI(hi),ωHI(hj))}}∧ωLΩ(li′)={{(ωHI(hj),ωHI(hi))}}∧
      ωHΩ(hi)= { ... } ... 3 lines end
```

21 4.2.4 Extension

We extend the domain by introducing toll road entry and exit booths as well as electronic ticket hub sensors and actuators. There should now follow a careful narrative and formalisation of these three machines: the car driver/machine "dialogues" upon entry and exit as well as the sensor/car/actuator machine "dialogues" when cars enter hubs. The description should first, we suggest, be ideal; then it should take into account failures of booth equipment, electronic tickets, car drivers, and of sensors and actuators.

22 4.2.5 Fitting

We assume three ongoing requirements development projects, all focused around road transport net software systems: (i) road maintenance, (ii) toll road car monitoring and (iii) bus services on ordinary plus toll road nets. The main shared phenomenon is the road net, i.e., the links and the hubs. The consolidated, shared road net domain requirements prescription, $\delta_{r_{n+1}}$, is to become a prescription for the domain requirements for shared hubs and links. Tuples of these relations then prescribe representation of all hub, respectively all link attributes – common to the three applications. Functions (including actions) on hubs and links become database queries and updates. Etc.

23 4.3.1 Shared Entities

Main shared entities are those of hubs and links. We suggest that eventually a relational database be used for representing hubs links in relations. As for human input, some man/machine dialogue based around a set of visual display unit screens with fields for the input of hub, respectively link attributes can then be devised. Etc.

24 4.3.2 Shared Actions

In order for a car driver to leave an exit toll both the following component actions must take place: the driver inserts the electronic pass in the exit toll booth machine; the machine scans and accepts the ticket and calculates the fee for the car journey from entry booth

via the toll road net to the exit booth; the driver is alerted to the cost and is requested to pay this amount; once paid the exit booth toll gate is raised. *Notice that a number of details of the new support technology is left out. It could either be elaborated upon here, or be part of the system design.*

25 4.3.3 Shared Events

The arrival of a car at a toll plaza entry booth is an event that must be communicated to the machine so that the entry booth may issue a proper pass (ticket). Similarly for the arrival at a toll plaza exit booth so that the machine may request the return of the pass and compute the fee. The end of that computation is an event that is communicated to the driver (in the domain) requesting that person to pay a certain fee after which the exit gate is opened.

26 4.3.4 Shared Behaviour

A typical toll road net use behaviour is as follows: Entry at some toll plaza: receipt of electronic ticket, placement of ticket in special ticket "pocket" in front window, the raising of the entry booth toll gate; drive up to [first] toll road hub (with electronic registration of time of occurrence), drive down a selected link (with electronic registration of time of occurrence of entry to and exit from link), then a repeated number of zero, one or moretoll road hub and link visits – some of which may be "repeats" – ending with a drive down from a toll road hub to a toll plaza with the return of the electronic ticket, etc. – cf. Example **??**.

A. RSL Primer

27 A.1.1 Basic Net Attributes

- For safe, uncluttered traffic, hubs and links can 'carry' a maximum of vehicles.
- Links have lengths. (We ignore hub (traveersal) lengths.)
- One can calculate whether a link is a two-way link.

type
  MAX = **Nat**
  LEN = **Real**
  is_Two_Way_Link = **Bool**
value
  ωMax: (H|L) → MAX
  ωLen: L → LEN
  is_two_way_link: L → is_Two_Way_Link
  is_two_way_link(l) ≡ ∃ lσ:LΣ • lσ ∈ ωHΣ(l)∧**card** lσ=2

28 A.1.2 Composite Net Types

There are many ways in which nets can be concretely modelled:

- **Sorts + Observers + Axioms:** First we show an example of type definitions without right-hand side, that is, of sort definitions.

  From a net one can observe many things.

  Of the things we focus on are the hubs and the links.

  A net contains two or more hubs and one or more links. Possibly other entities and net attributes may also be observable, but we shall not consider those here.

  type
  [sorts] N_α, H, L, HI, LI
  value
  ωHs: $N_\alpha \to$ H-**set**
  ωLs: $N_\alpha \to$ L-**set**
  axiom
  $\forall$ n:N_α • **card** ωHs(n)$\geq$2 $\wedge$ **card** ωLs(n)$\geq$1 $\wedge$ ...

  where the ... shall account for what has been in expressed in axioms [5–8] of Example ?? on page ??.

- **Cartesians + Wellformedness:** A net can be considered as a Cartesian of sets of two or more hubs and sets of one or more links.

  type
  [sorts] H, L
  N_β = H-**set** $\times$ L-**set**
  value
  wf_N_β: $N_\beta \to$ **Bool**
  wf_N_β(hs,ls) $\equiv$ **card** hs$\geq$2 $\wedge$ **card** ls$\geq$1 ....
  inject_N_β: $N_\alpha \xrightarrow{\sim} N_\beta$ **pre**: wf_N_β(hs,ls)
  inject_N_β(n_α) $\equiv$ (ωHs(n_α),ωLs(n_α))

  where the ... shall account for what has been in expressed in axioms [5–8] of Example ?? on page ??.

- **Cartesians + Maps + Wellformedness:** Or a net can be modelled as a triple of
  - hubs (modelled as a map from hub identfiers to hubs),
  - links (modelled as a map from link identfiers to links), and

  - a graph from hub h_i identifiers h_{i_i} to maps from identfiers l_{ij_i} of hub h_i connected links l_{ij} to the identfiers h_{j_i} of link connected hubs h_j.

type
  [sorts] H, HI, L, LI
    N_γ = HUBS × LINKS × GRAPH
  [a] HUBS = HI $\overrightarrow{m}$ H
  [b] LINKS = LI $\overrightarrow{m}$ L
  [c] GRAPH = HI $\overrightarrow{m}$ (LI −m> HI)

  - [a,b] *hs:HUBS* and *ls:LINKS* are maps from hub (link) identifiers to hubs (links) where one can still observe these identfiers from these hubs (link).

- Example **??** on page ?? defines the well-formedness predicates for the above map types.

29 A.1.2 Composite Net Type Expressions

The type clauses of function signatures:

value
  f: A → B

often have the type expressions A and/or B be composite type expressions:

value
  ωHIs: L → HI-**set** Example ?? Item [5]
  ωLIs: H → LI-**set** Example ?? Item [6]
  ωHΣ: H → HT-**set** Example ?? Item [10]
  set_HΣ: H × HΣ → H Example **??** Item [12]

Right-hand sides of type definitions often have composite type expressions:

type
  N = H-**set** × L-**set** Example ?? Item [2]
  HT = LI × HI × LI Example ?? Item [9]
  LT′ = HI × LI × HI Example ?? Item [32]

30 A.1.2 Net Record Types: Insert Links

1. To a net one can insert a new link in either of three ways:
   a) Either the link is connected to two existing hubs — and the insert operation must therefore specify the new link and the identifiers of two existing hubs;
   b) or the link is connected to one existing hub and to a new hub — and the insert operation must therefore specify the new link, the identifier of an existing hub, and a new hub;
   c) or the link is connected to two new hubs — and the insert operation must therefore specify the new link and two new hubs.
   d) From the inserted link one must be able to observe identifier of respective hubs.
2. From a net one can remove a link.[1] The removal command specifies a link identifier.

type
1 Insert == Ins(s_ins:Ins)
1 Ins = 2xHubs | 1x1nH | 2nHs
1a 2xHubs == 2oldH(s_hi1:HI,s_l:L,s_hi2:HI)
1b 1x1nH == 1oldH1newH(s_hi:HI,s_l:L,s_h:H)
1c 2nHs == 2newH(s_h1:H,s_l:L,s_h2:H)
2 Remove == Rmv(s_li:LI)
axiom
1d ∀ 2oldH(hi′,l,hi″):Ins • hi′≠hi″ ∧ obs_LIs(l)={hi′,hi″} ∧
∀ 1old1newH(hi,l,h):Ins • obs_LIs(l)={hi,obs_HI(h)} ∧
∀ 2newH(h′,l,h″):Ins • obs_LIs(l)={obs_HI(h′),obs_HI(h″)}

RSL Explanation

- 1: The type clause **type** Ins = 2xHubs | 1x1nH | 2nHs introduces the type name Ins and defines it to be the union (|) type of values of either of three types: 2xHubs, 1x1nH and 2nHs.
  - 1a): The type clause **type** 2xHubs == 2oldH(s_hi1:HI, s_l:L, s_hi2:HI) defines the type 2xHubs to be the type of values of record type 2oldH(s_hi1:HI,s_l:L,s_hi2:HI), that is, Cartesian-like, or "tree"-like values with record (root) name 2oldH and with three sub-values, like branches of a tree, of types HI, L and HI. Given a value, cmd, of type 2xHubs, applying the selectors s_hi1, s_l and s_hi2 to cmd yield the corresponding sub-values.
  - 1b): Reading of this type clause is left as exercise to the reader.
  - 1c): Reading of this type clause is left as exercise to the reader.

[1] – provided that what remains is still a proper net

  - 1d): The axiom **axiom** has three predicate clauses, one for each category of Insert commands.
    - ◊ The first clause: ∀ 2oldH(hi′,l,hi″):Ins • hi′≠hi″ ∧ obs_Hls(l) = {hi′, hi″} reads as follows:
      - For all record structures, 2oldH(hi′,l,hi″), that is, values of type Insert (which in this case is the same as of type 2xHubs),
      - that is values which can be expressed as a record with root name 2oldH and with three sub-values ("freely") named hi′, l and hi″
      - (where these are bound to be of type Hl, L and Hl by the definition of 2xHubs),
      - the two hub identifiers hi′ and hi″ must be different,
      - and the hub identifiers observed from the new link, l, must be the two argument hub identifiers hi′ and hi″.
    - ◊ Reading of the second predicate clause is left as exercise to the reader.
    - ◊ Reading of the third predicate clause is left as exercise to the reader.

  The three types 2xHubs, 1x1nH and 2nHs are disjoint: no value in one of them is the same value as in any of the other merely due to the fact that the record names, 2oldH, 1oldH1newH and 2newH, are distinct. This is no matter what the "bodies" of their record structure is, and they are here also distinct: (s_hi1:Hl,s_l:L,s_hi2:Hl), (s_hi:Hl,s_l:L,s_h:H), respectively (s_h1:H,s_l:L,s_h2:H).

- 2; The type clause **type** Remove == Rmv(s_li:Ll)
  - (as for Items 1b) and 1c))
  - defines a type of record values, say rmv,
  - with record name Rmv and with a single sub-value, say li of type Ll
  - where li can be selected from by rmv selector s_li.

End of RSL Explanation

Example **??** on page ?? presents the semantics functions for *int_Insert* and *int_Remove*.
.

31 A.1.2 Net Subtypes

In Example **??** on page ?? we gave three examples. For the first we gave an example, **Sorts + Observers + Axioms**, "purely" in terms of sets, see *Sorts — Abstract Types* below. For the second and third we gave concrete types in terms of Cartesians and Maps.

- In the **Sorts + Observers + Axioms** part of Example ??

  - a net was defined as a sort, and so were its hubs, links, hub identifiers and link identifiers;
  - axioms – making use of appropriate observer functions - make up the well-formedness condition on such nets.

We now redefine this as follows:

```
type
  [ sorts ] N′, H, L, HI, LI
            N = {|n:N′ • wf_N(n)|}
value
  wf_N: N′ → Bool
  wf_N(n) ≡
    ∀ n:N • card ωHs(n)≥2 ∧ card ωLs(n)≥1 ∧
    [ 5−−8 ] of example ??
```

- In the **Cartesians + Wellformedness** part of Example ??
  - a net was a Cartesian of a set of hubs and a set of links
  - with the wellformedness that there were at least two hubs and at least one link
  - and that these were connected appropriately (treated as ...).

We now redefine this as follows:

```
type
  N′ = H-set × L-set
  N = {|n:N′ • wf_N(n)|}
```

- In the **Cartesians + Maps + Wellformedness** part of Example ??
  - a net was a triple of hubs, links and a graph,
  - each with their wellformednes predicates.

We now redefine this as follows:

```
type
  [ sorts ] L, H, LI, HI
  N′ = HUBS × LINKS × GRAPH
  N = {|(hs,ls,g):N′ • wf_HUBS(hs)∧wf_LINKS(ls)∧wf_GRAPH(g)(hs,ls)|}
  HUBS′ = HI ~m~> H
  HUBS = {|hs:HUBS′ • wf_HUBS(hs)|}
  LINKS′ = LI → L
```

LINKS = {|ls:LINKS′ • wf_LINKS(ls)|}
GRAPH′ = HI $\overrightarrow{m}$ (LI $\overrightarrow{m}$HI)
GRAPH = {|g:GRAPH′ • wf_GRAPH(g)|}
value
wf_GRAPH: GRAPH′ → (HUBS × LINKS) → **Bool**
wf_GRAPH(g)(hs,ls) ≡ wf_N(hs,ls,g)

Example **??** on page ?? presents a definition of *wf_GRAPH*.

32 A.1.2 Net Sorts

In formula lines of Examples ??–**??** we have indicated those **type** clauses which define *sorts*, by bracketed [sorts] literals.

33 A.2.2 Set Comprehensions

Example ?? on page ?? illustrates, in the **Cartesians + Maps + Wellformedness** part the following set comprehensions in the *wf_N(hs,ls,g)* wellformedness predicate definition:

[d] ∪ {**dom** *g(hi)*|*hi:HI* • *hi* ∈ **dom** *g*}

It expresses the distributed union of sets (**dom** *g(hi)*) of link identfiers (for each of the *hi* indexed maps from (definition set, **dom**) link identiers to (range set, **rng**) hub identifiers, where *hi:HI* ranges over **dom** *g*).

[e] ∪ {**rng** *g(hi)*|*hi:HI* • *hi* ∈ **dom** *g*}

It expresses the distributed union of sets (**rng** *g(hi)*) of hub identfiers (for each of the *hi* indexed maps from (definition set, **dom**) link identiers to (range set, **rng**) hub identifiers, where *hi:HI* ranges over **deom** *g*).

34 A.2.2 Set Expressions over Nets

We now consider hubs to abstract cities, towns, villages, etcetera. Thus with hubs we can associate sets of citizens.

Let c:C stand for a citizen value c being an element in the type C of all such. Let g:G stand for any (group) of citizens, respectively the type of all such. Let s:S stand for any set of groups, respectively the type of all such. Two otherwise distinct groups are related to one another if they share at least one citizen, the liaisons. A network nw:NW is a set of groups such that for every group in the network one can always find another group with which it shares liaisons.

Solely using the set data type and the concept of subtypes, we can model the above:

type
C
G′ = C-**set**, G = {| g:G′ • g≠{} |}
S = G-**set**

```
  L′ = C-set,  L = {| ℓ:L′ • ℓ≠{} |}
  NW′ = S,  NW = {| s:S • wf_S(s) |}
value
  wf_S: S → Bool
  wf_S(s) ≡ ∀ g:G • g ∈ s ⇒ ∃ g′:G •  g′ ∈ s ∧ share(g,g′)
  share: G×G → Bool
  share(g,g′) ≡ g≠g′ ∧ g ∩ g′ ≠ {}
  liaisons: G×G → L
  liaisons(g,g′) = g ∩ g′ pre share(g,g′)
```

Annotations: L stands for proper liaisons (of at least one liaison). G′, L′ and N′ are the "raw" types which are constrained to G, L and N. {| binding:type_expr • bool_expr |} is the general form of the subtype expression. type!subtype expressionsubtypeFor G and L we state the constraints "in-line", i.e., as direct part of the subtype expression. For NW we state the constraints by referring to a separately defined predicate. wf_S(s) expresses — through the auxiliary predicate — that s contains at least two groups and that any such two groups share at least one citizen. liaisons is a "truly" auxiliary functionauxiliary!function in that we have yet to "find an active need" for this function!

The idea is that citizens can be associated with more than one city, town, village, etc. (primary home, summer and/or winter house, working place, etc.). A group is now a set of citizens related by some "interest" (Rotary club membership, political party "grassroots", religion, et.). The reader is invited to define, for example, such functions as:The set of groups (or networks) which are represented in all hubs [or in only one hub]. The set of hubs whose citizens partake in no groups [respectively networks]. The group [network] with the largest coverage in terms of number of hubs in which that group [network] is represented.
.

35 A.2.3 Cartesian Net Types

So far we have abstracted hubs and links as sorts. That is, we have not defined their types concretely. Instead we have postulated some attributes such as: observable hub identifiers of hubs and sets of observable link identifiers of links connected to hubs. We now claim the following further attributes of hubs and links.

- Concrete links have
  - link identifiers,
  - link names – where two or more connected links may have the same link name,
  - two (unordered) hub identifiers,
  - lenghts,
  - locations – where we do not presently defined what we main by locations,
  - etcetera

- Concrete hubs have
  - hub identifiers,
  - unique hub names,
  - a set of one or more observable link identifiers,
  - locations,
  - etcetera.

type
  LN, HN, LEN, LOC
  cL = LI × LN × (HI × HI) × LOC × ...
  cH = HI × HN × LI-**set** × LOC × ...

36 A.2.4 Routes in Nets

- A phenomenological (i.e., a physical) route of a net is a sequence of one or more adjacent links of that net.
- A conceptual route is a sequence of one or more link identifiers.
- An abstract route is a conceptual route
  - for which there is a phenomenological route of the net for which the link identifiers of the abstract route map one-to-one onto links of the phenomenological route.

type
  N, H, L, HI, LI
  PR′ = L*
  PR = {| pr:PR′ • ∃ n:N • wf_PR(pr)(n)|}
  CR = LI*
  AR′ = LI*
  AR = {| ar:AR′ • ∃ n:N • wf_AR(ar)(n) |}
value
  wf_PR: PR′ → N → **Bool**
  wf_PR(pr)(n) ≡
    ∀ i:**Nat** • {i,i+1}⊆**inds** pr ⇒
      ωHIs(l(i)) ∩ ωHIs(l(i+1)) ≠ {}
  wf_AR′: AR′ → N → **Bool**
  wf_AR(ar)(n) ≡
    ∃ pr:PR • pr ∈ routes(n) ∧ wf_PR(pr)(n) ∧ **len** pr=**len** ar ∧
      ∀ i:**Nat** • i ∈ **inds** ar ⇒ ωLI(pr(i))=ar(i)

- A single link is a phenomenological route.
- If r and r' are phenomenological routes
  - such that the last link r
  - and the first link of r'
  - share observable hub identifiers,

  then the concatenation $r^\frown r'$ is a route.

  This inductive definition implies a recursive set comprehension.
- A circular phenomenological route is a phenomenological route whose first and last links are distinct but share hub identifiers.
- A looped phenomenological route is a phenomenological route where two distinctly positions (i.e., indexed) links share hub identifiers.

```
value
  routes: N → PR-infset
  routes(n) ≡
    let prs = {⟨l⟩|l:L•ωLIs(n)} ∪
                  ∪ {pr⌢pr′|pr,pr′:PR•{pr,pr′}⊆prs∧ωHIs(r(len pr))∩ωHIs(pr′(1))≠{}}
    prs end

  is_circular: PR → Bool
  is_circular(pr) ≡ ωHIs(pr(1))∩ωHIs(pr(len pr))≠{}

  is_looped: PR → Bool
  is_looped(pr) ≡ ∃ i,j:Nat • i≠j∧{i,j}⊆index pr ⇒ ωHIs(pr(i))∩ωHIs(pr(j))≠{}
```

- Straight routes are Phenomenological routes without loops.
- Phenomenological routes with no loops can be constructed from phenomenological routes by removing suffix routes whose first link give rise to looping.

```
value
  straight_routes: N → PR-set
  straight_routes(n) ≡
    let prs = routes(n) in {straight_route(pr)|pr:PR•ps ∈ prs} end

  straight_route: PR → PR
  straight_route(pr) ≡
    ⟨pr(i)|i:Nat•i:[ 1..len pr ] ∧ pr(i)∉ elems⟨pr(j)|j:Nat•j:[ 1..i ]⟩⟩
```

37 A.2.5 Concrete Net Type Construction

- We Define a function *con[struct]*_N_γ (of the **Cartesians + Maps + Wellformedness** part of Example ??.
  - The base of the construction is the fully abstract sort definition of N_α in the **Sorts + Observers + Axioms** part of Example ?? – where the sorts of hub and link identifiers are taken from earlier examples.
  - The target of the construction is the N_γ of the **Cartesians + Maps + Wellformedness** part of Example ??.
  - First we recall the ssential types of that N_γ.

type
  N_γ = HUBS × LINKS × GRAPH
  HUBS = HI $\overrightarrow{m}$ H
  LINKS = LI $\overrightarrow{m}$ L
  GRAPH = HI $\overrightarrow{m}$ (LI $\overrightarrow{m}$ HI)
value
  con_N_γ: N_α → N_γ
  con_N_γ(n_α) ≡
    let hubs = [ωHI(h) ↦ h | h:H • h ∈ ωHs(n_α)],
      links = [ωLI(h) ↦ l | l:L • l ∈ ωLs(n_α)],
      graph = [ωHI(h) ↦ [ωLI(l) ↦ ι(ωHIs(l)\{li})]
                | l:L • l ∈ ωLs(n_α)∧li ∈ ωLIs(h)
              | H:h • h ∈ ωHs(n_α)] **in**
      (hubs.links,graph) **end**

  ι: A-**set** $\xrightarrow{\sim}$ A [A could be LI-**set**]
  ι(as) ≡ **if card** as=1 **then let** {a}=as **in** a **else chaos end end**

theorem:
  n_α **satisfies axioms** [2,5–8] for N of Example ?? ⇒ wf_$N_\gamma$con_N_γ(n_α)

38 A.2.9 Miscellaneous Net Expressions: Maps

Example ?? on page ?? left out defining the well-formedness of the map types:

value
    wf_HUBS: HUBS → **Bool**
  [a] wf_HUBS(hubs) ≡ ∀ hi:HI • hi ∈ **dom** hubs ⇒ ωHIhubs(hi)=hi
    wf_LINKS: LINKS → **Bool**

[b] wf_LINKS(links) ≡ ∀ li:LI • li ∈ **dom** links ⇒ ωLIlinks(li)=li
wf_N_γ: N_γ → **Bool**
wf_N_γ(hs,ls,g) ≡
  [c] **dom** hs = **dom** g ∧
  [d] ∪ {**dom** g(hi)|hi:HI • hi ∈ **dom** g} = **dom** links ∧
  [e] ∪ {**rng** g(hi)|hi:HI • hi ∈ **dom** g} = **dom** g ∧
  [f] ∀ hi:HI • hi ∈ **dom** g ⇒ ∀ li:LI • li ∈ **dom** g(hi) ⇒ (g(hi))(li)≠hi
  [g] ∀ hi:HI • hi ∈ **dom** g ⇒ ∀ li:LI • li ∈ **dom** g(hi) ⇒
    ∃ hi′:HI • hi′ ∈ **dom** g ⇒ ∃ ! li:LI • li ∈ **dom** g(hi) ⇒
      (g(hi))(li) = hi′ ∧ (g(hi′))(li) = hi

- [c] *HUBS* record the same hubs as do the net corresponding *GRAPHS* (**dom** *hs* = **dom** *g* ∧).
- [d] *GRAPHS* record the same links as do the net corresponding *LINKS* (∪ {**dom** *g(hi)|hi:HI • hi* ∈ **dom** *g*} = **dom** *links*).
- [e] The target (or range) hub identifiers of graphs are the same as the domain of the graph (∪ {**rng** *g(hi)|hi:HI • hi* ∈ **dom** *g*} = **dom** *g*), that is none missing, no new ones !
- [f] No links emanate from and are incident upon the same hub (∀ *hi:HI • hi* ∈ **dom** *g* ⇒ ∀ *li:LI • li* ∈ **dom** *g(hi)* ⇒ *(g(hi))(li)≠hi*).
- [g] If there is a link from one hub to another in the *GRAPH*, then the same link also connects the other hub to the former (∀ *hi:HI • hi* ∈ **dom** *g* ⇒ ∀ *li:LI • li* ∈ **dom** *g(hi)* ⇒ ∃ *hi′:HI • hi′* ∈ **dom** *g* ⇒ ∃ ! *li:LI • li* ∈ **dom** *g(hi)* ⇒ *(g(hi))(li)* = *hi′* ∧ *(g(hi′))(li)* = *hi*).

39 A.3.3 Predicates Over Net Quantities

From earlier examples we show some predicates:

- Example **??**: Right hand side of function definition *is_two_way_link(l):*
  ∃ *lσ:LΣ • lσ* ∈ *ωHΣ(l)*∧**card** *lσ=2*
- Example ??:
  - The **Sorts + Observers + Axioms** part:
    * Right hand side of the wellformedness function *wf_N(n)* definition:
      ∀ *n:N* • **card** *ωHs(n)*≥*2* ∧ **card** *ωLs(n)*≥*1* ∧ [*5−−8*] *of example ??*
    * Right hand side of the wellformedness function *wf_N(hs,ls)* definition:
      card *hs*≥*2* ∧ **card** *ls*≥*1* ...

– The **Cartesians + Maps + Wellformedness** part:

∗ Right hand side of the *wf_HUBS* wellformedness function definition:
∀ *hi:HI* • *hi* ∈ **dom** *hubs* ⇒ *ωHIhubs(hi)=hi*

∗ Right hand side of the *wf_LINKS* wellformedness function definition:
∀ *li:LI* • *li* ∈ **dom** *links* ⇒ *ωLIlinks(li)=li*

∗ Right hand side of the *wf_N(7 hs,ls,g)* wellformedness function definition:
[*c*] **dom** *hs* = **dom** *g* ∧
[*d*] ∪ {**dom** *g(hi)*|*hi:HI* • *hi* ∈ **dom** *g*} = **dom** *links* ∧
[*e*] ∪ {**rng** *g(hi)*|*hi:HI* • *hi* ∈ **dom** *g*} = **dom** *g* ∧
[*f*] ∀ *hi:HI* • *hi* ∈ **dom** *g* ⇒ ∀ *li:LI* • *li* ∈ **dom** *g(hi)* ⇒ *(g(hi))(li)≠hi*
[*g*] ∀ *hi:HI* • *hi* ∈ **dom** *g* ⇒ ∀ *li:LI* • *li* ∈ **dom** *g(hi)* ⇒
∃ *hi′:HI* • *hi′* ∈ **dom** *g* ⇒ ∃ ! *li:LI* • *li* ∈ **dom** *g(hi)* ⇒
(g(hi))(li) = *hi′* ∧ *(g(hi′))(li)* = *hi*

40 A.4.5 Network Traffic

We model traffic by introducing a number of model concepts. We simplify – without loosing the essence of this example, namely to show the use of λ–functions – by omitting consideration of dynamically changing nets. These are introduced next:

- Let us assume a net, *n:N*.
- There is a dense set, *T*, of times – for which we omit giving an appropriate definition.
- There is a sort, *V*, of vehicles.
- *TS* is a dense subset of *T*.
- For each *ts:TS* we can define a minimum and a maximum time.
- The $\mathcal{MIN}$ and $\mathcal{MAX}$ functions are meta-linguistic, that is, are not defined in our formal specification language RSL, but can be given a satisfactory meaning.
- At any moment some vehicles, *v:V*, have a *pos:Pos*ition on the net and *VP* records those.
- A *Pos*ition is either on a link or at a hub.
- An *onL*ink position can be designated by the link identifier, the identifiers of the from and to hubs, and the fraction, *f:F*, of the distance down the link from the from hub to the to hub.
- An *atH*ub position just designates the hub (by its identifier).
- Traffic, *tf:TF*, is now a continuous function from *T*ime to *NP* ("recordings").

- Modelling traffic in this way, in fact, in whichever way, entails a ("serious") number of well-formedness conditions. These are defined in *wf_TF* (omitted: ...).

value
  n:N
type
  T, V
  TS = T-**infset**
axiom
  ∀ ts:TS • ∃ tmin,tmax:T: tmin ∈ ts ∧ tmax ∈ ts ∧ ∀ t:T • t ∈ ts ⇒ tmin ≤ t ≤ tmax
  [that is: ts = {$\mathcal{MIN}$(ts)..$\mathcal{MAX}$(ts)}]
type
  VP = V $\overrightarrow{m}$ Pos
  TF′ = T → VP, TF = {|tf:TF′•wf_TF(tf)(n)|}
  Pos = onL | atH
  onL == mkLPos(hi:HI,li:LI,f:F,hi:HI), atH == mkHPos(hi:HI)
value
  wf_TF: TF→ N → **Bool**
  wf_TF(tf)(n) ≡ ...
  $\mathcal{DOMAIN}$: TF → TS
  $\mathcal{MIN}$, $\mathcal{MAX}$: TS → T

We have defined the continuous, composite entity of traffic. Now let us define an operation of inserting a vehicle in a traffic.

- To insert a vehicle, v, in a traffic, tf, is prescribable as follows:
  - the vehicle, v, must be designated;
  - a time point, t, "inside" the traffic tf must be stated;
  - a traffic, vtf, from time t of vehicle v must be stated;
  - as well as traffic, tf, into which vtf is to be "merged".
- The resulting traffic is referred to as tf'.

value
  insert_V: V × T × TF → TF → TF
  insert_V(v,t,vtf)(tf) **as** tf′

- The function *insert_V* is here defined in terms of a pair of pre/post conditions.
- The pre-condition can be prescribed as follows:

  - The insertion time t must be within to open interval of time points in the traffic tf to which insertion applies.
  - The vehicle v must not be among the vehicle positions of tf.
  - The vehicle must be the only vehicle "contained" in the "inserted" traffic vtf.

pre: $\mathcal{MIN}(\mathcal{DOMAIN}(\text{tf}) \leq \text{t} \leq \mathcal{MAX}(\mathcal{DOMAIN}(\text{tf})) \wedge$
$\forall\ \text{t}':\text{T} \bullet \text{t}' \in \mathcal{DOMAIN}(\text{tf}) \Rightarrow \text{v} \notin \textbf{dom}\ \text{tf}(\text{t}') \wedge$
$\mathcal{MIN}(\mathcal{DOMAIN}(\text{vtf})) = \text{t} \wedge$
$\forall\ \text{t}':\text{T} \bullet \text{t}' \in \mathcal{DOMAIN}(\text{vtf}) \Rightarrow \textbf{dom}\ \text{vtf}(\text{t}') = \{\text{v}\}$

- The post condition "defines" tf', the traffic resulting from merging vtf with tf:
  - Let ts be the time points of tf and vtf, a time interval.
  - The result traffic, tf', is defines as a λ-function.
  - For any t'' in the time interval
  - if t'' is less than t, the insertion time, then tf' is as tf;
  - if t'' is t or larger then tf' applied to t'', i.e., $tf'(t'')$
    - for any $v' : V$ different from v yields the same as $(tf(t))(v')$,
    - but for v it yields $(vtf(t))(v)$.

post: $\text{tf}' = \lambda\text{t}'' \bullet$
let $\text{ts} = \mathcal{DOMAIN}(\text{tf}) \cup \mathcal{DOMAIN}(\text{vtf})$ **in**
if $\mathcal{MIN}(\text{ts}) \leq \text{t}'' \leq \mathcal{MAX}(\text{ts})$
then
$((\text{t}'' < \text{t}) \rightarrow \text{tf}(\text{t}''),$
$(\text{t}'' \geq \text{t}) \rightarrow [\,\text{v}' \mapsto$ **if** $\text{v}' \neq \text{v}$ **then** $(\text{tf}(\text{t}))(\text{v}')$ **else** $(\text{vtf}(\text{t}))(\text{v})$ **end**$\,]$)
else chaos end
end
assumption: wf_TF(vtf)$\wedge$wf_TF(tf)
theorem: wf_TF(tf$'$)

We leave it as an exercise for the reader to define functions for: removing a vehicle from a traffic, changing to course of a vehicle from an originally (or changed) vehicle traffic to another. etcetera.

41 A.4.6 Hub and Link Observers

Let a net with several hubs and links be presented. Now observer functions ωHs and ωLs can be demonstrated: one simply "walks" along the net, pointing out this hub and that link, one-by-one until all the net has been visited.

The observer functions ωHI and ωLI can be likewise demonstrated, for example: when a hub is "visited" its unique identification can be postulated (and "calculated") to be the unique geographic position of the hub one which is not overlapped by any other hub (or link), and likewise for links.

42 A.4.7 Axioms over Hubs, Links and Their Observers

Example ?? on page ?? Items [4]–[8] clearly demonstrates how a number of entities and observer functions are constrained (that is, partially defined) by function signatures and axioms.

.

43 A.5.5 Choice Pattern Case Expressions: Insert Links

We consider the meaning of the Insert operation designators.

3. The insert operation takes an Insert command and a net and yields either a new net or **chaos** for the case where the insertion command "is at odds" with, that is, is not semantically well-formed with respect to the net.
4. We characterise the "is not at odds", i.e., is semantically well-formed, that is:
   - pre_int_Insert(op)(hs,ls),

   as follows: it is a propositional function which applies to Insert actions, op, and nets, (hs.ls), and yields a truth value if the below relation between the command arguments and the net is satisfied. Let (hs,ls) be a value of type N.
5. If the command is of the form 2oldH(hi′,l,hi′) then
   - ⋆1 hi′ must be the identifier of a hub in hs,
   - ⋆s2 l must not be in ls and its identifier must (also) not be observable in ls, and
   - ⋆3 hi″ must be the identifier of a(nother) hub in hs.
6. If the command is of the form 1oldH1newH(hi,l,h) then
   - ⋆1 hi must be the identifier of a hub in hs,
   - ⋆2 l must not be in ls and its identifier must (also) not be observable in ls, and
   - ⋆3 h must not be in hs and its identifier must (also) not be observable in hs.
7. If the command is of the form 2newH(h′,l,h″) then
   - ⋆1 h′ — left to the reader as an exercise (see formalisation !),
   - ⋆2 l — left to the reader as an exercise (see formalisation !), and

⋆3 h″ — left to the reader as an exercise (see formalisation !).

Conditions concerning the new link (second ⋆s, ⋆2, in the above three cases) can be expressed independent of the insert command category.

```
value
  3  int_Insert: Insert → N ~→ N
  4′ pre_int_Insert: Ins → N → Bool
  4″ pre_int_Insert(Ins(op))(hs,ls) ≡
⋆2        s_l(op)∉ ls ∧ obs_LI(s_l(op)) ∉ iols(ls) ∧
    case op of
  5)    2oldH(hi′,l,hi″) → {hi′,hi″}∈ iohs(hs),
  6)    1oldH1newH(hi,l,h) →
            hi ∈ iohs(hs) ∧ h∉ hs ∧ obs_HI(h)∉ iohs(hs),
  7)    2newH(h′,l,h″) →
            {h′,h″}∩ hs={} ∧ {obs_HI(h′),obs_HI(h″)}∩ iohs(hs)={}
    end
```

RSL Explanation

- 3: The value clause **value** int_Insert: Insert → N ~→ N names a value, int_Insert, and defines its type to be Insert → N ~→ N, that is, a partial function (~→) from Insert commands and nets (N) to nets.
  (int_Insert is thus a function. What that function calculates will be defined later.)
- 4′: The predicate pre_int_Insert: Insert → N → **Bool** function (which is used in connection with int_Insert to assert semantic well-formedness) applies to Insert commands and nets and yield truth value **true** if the command can be meaningfully performed on the net state.
- 4″: The action pre_int_Insert(op)(hs,ls) (that is, the effect of performing the function pre_int_Insert on an Insert command and a net state is defined by a case distinction over the category of the Insert command. But first we test the common property:
- ⋆2: s_l(op)∉ls∧obs_LI(s_l(op))∉iols(ls), namely that the new link is not an existing net link and that its identifier is not already known.
  - 5): If the Insert command is of kind 2oldH(hi',l,hi'') then {hi′,hi″}∈ iohs(hs), that is, then the two distinct argument hub identifiers must not be in the set of known hub identifiers, i.e., of the existing hubs hs.
  - 6): If the Insert command is of kind 1oldH1newH(hi,l,h) then ... exercise left as an exercises to the reader.
  - 7): If the Insert command is of kind 2newH(h',l,h'') ... exercise left as an exercises to the reader. The set intersection operation is defined in Sect. **??** on page ?? Item **??** on page ??.

End of RSL Explanation

8. Given a net, (hs,ls), and given a hub identifier, (hi), which can be observed from some hub in the net, xtr_H(hi)(hs,ls) extracts the hub with that identifier.
9. Given a net, (hs,ls), and given a link identifier, (li), which can be observed from some link in the net, xtr_L(li)(hs,ls) extracts the hub with that identifier.

value
8: xtr_H: HI → N $\xrightarrow{\sim}$ H
8: xtr_H(hi)(hs,_) ≡ **let** h:H•h ∈ hs ∧ obs_HI(h)=hi **in** h **end**
pre hi ∈ iohs(hs)
9: xtr_L: HI → N $\xrightarrow{\sim}$ H
9: xtr_L(li)(_,ls) ≡ **let** l:L•l ∈ ls ∧ obs_LI(l)=li **in** l **end**
pre li ∈ iols(ls)

RSL Explanation

- 8: Function application xtr_H(hi)(hs,_) yields the hub h, i.e. the value h of type H, such that (•) h is in hs and h has hub identifier hi.
- 8: The wild-card, _, expresses that the extraction (xtr_H) function does not need the L-set argument.
- 9: Left as an exercise for the reader.

End of RSL Explanation

10. When a new link is joined to an existing hub then the observable link identifiers of that hub must be updated to reflect the link identifier of the new link.
11. When an existing link is removed from a remaining hub then the observable link identifiers of that hub must be updated to reflect the removed link (identifier).

value
aLI: H × LI → H, rLI: H × LI $\xrightarrow{\sim}$ H
10: aLI(h,li) **as** h′
pre li ∉ obs_LIs(h)
post obs_LIs(h′) = {li} ∪ obs_LIs(h) ∧ non_I_eq(h,h′)
11: rLI(h′,li) **as** h
pre li ∈ obs_LIs(h′) ∧ **card** obs_LIs(h′)≥2
post obs_LIs(h) = obs_LIs(h′) \ {li} ∧ non_I_eq(h,h′)

RSL Explanation

- 10: The add link identifier function aLI:

  - The function definition clause aLI(h,li) **as** h′ defines the application of aLI to a pair (h,li) to yield an update, h′ of h.
  - The pre-condition **pre** li ∉ obs_LIs(h) expresses that the link identifier li must not be observable h.
  - The post-condition **post** obs_LIs(h) = obs_LIs(h′) \ {li} ∧ non_l_eq(h,h′) expresses that the link identifiers of the resulting hub are those of the argument hub except (\) that the argument link identifier is not in the resulting hub.

- 11: The remove link identifier function rLI:
  - The function definition clause rLI(h′,li) **as** h defines the application of rLI to a pair (h′,li) to yield an update, h of h′.
  - The pre-condition clause **pre** li ∈ obs_LIs(h′) ∧ **card** obs_LIs(h′)≥2 expresses that the link identifier li must not be observable h.
  - post-condition clause **post** obs_LIs(h) = obs_LIs(h′) \ {li} ∧ non_l_eq(h,h′) expresses that the link identifiers of the resulting hub are those of the argument hub except that the argument link identifier is not in the resulting hub.

End of RSL Explanation

12. If the Insert command is of kind 2newH(h',l,h'') then the updated net of hubs and links, has
    - the hubs hs joined, ∪, by the set {h′,h″} and
    - the links ls joined by the singleton set of {l}.
13. If the Insert command is of kind 1oldH1newH(hi,l,h) then the updated net of hubs and links, has
    - 13.1 : the hub identified by hi updated, hi′, to reflect the link connected to that hub.
    - 13.2 : The set of hubs has the hub identified by hi replaced by the updated hub hi′ and the new hub.
    - 13.2 : The set of links augmented by the new link.
14. If the Insert command is of kind 2oldH(hi',l,hi'') then
    - 14.1–.2 : the two connecting hubs are updated to reflect the new link,
    - 14.3 : and the resulting sets of hubs and links updated.

```
int_Insert(op)(hs,ls) ≡
⋆i  case op of
12      2newH(h′,l,h″) → (hs ∪ {h′,h″},ls ∪ {l}),
```

```
13     1oldH1newH(hi,l,h) →
13.1     let h′ = aLI(xtr_H(hi,hs),obs_LI(l)) in
13.2     (hs\{xtr_H(hi,hs)}∪{h,h′},ls ∪{l}) end,
14     2oldH(hi′,l,hi″) →
14.1     let hsδ = {aLI(xtr_H(hi′,hs),obs_LI(l)),
14.2                aLI(xtr_H(hi″,hs),obs_LI(l))} in
14.3     (hs\{xtr_H(hi′,hs),xtr_H(hi″,hs)}∪ hsδ,ls ∪{l}) end
⋆j  end
⋆k  pre pre_int_Insert(op)(hs,ls)
```

RSL Explanation

- $\star_i$–$\star_j$: The clause **case** op **of** $p_1 \rightarrow c_1$, $p_2 \rightarrow c_2$, ... $p_n \rightarrow c_n$ **end** is a conditional clause.
- $\star_k$: The pre-condition expresses that the insert command is semantically well-formed — which means that those reference identifiers that are used are known and that the new link and hubs are not known in the net.
- $\star_i$ + 12: If op is of the form 2newH(h′,l,h″ then — the narrative explains the rest;

  else
- $\star_i$ + 13: If op is of the form 1oldH1newH(hi,l,h) then
  - 13.1: h′ is the known hub (identified by hi) updated to reflect the new link being connected to that hub,
  - 13.2: and the pair [(updated hs,updated ls)] reflects the new net: the hubs have the hub originally known by hi replaced by h′, and the links have been simple extended (∪) by the singleton set of the new link;

  else
- $\star_i$ + 14: 14: If op is of the form 2oldH(hi′,l,hi″) then
  - 14.1: the first element of the set of two hubs (hsδ) reflect one of the updated hubs,
  - 14.2: the second element of the set of two hubs (hsδ) reflect the other of the updated hubs,
  - 14.3: the set of two original hubs known by the argument hub identifiers are removed and replaced by the set hsδ;

  else — well, there is no need for a further 'else' part as the operator can only be of either of the three mutually exclusive forms !

End of RSL Explanation

15. The remove command is of the form Rmv(li) for some li.
16. We now sketch the meaning of removing a link:
    a) The link identifier, li, is, by the pre_int_Remove pre-condition, that of a link, l, in the net.
    b) That link connects to two hubs, let us refer to them as h′ and h′.
    c) For each of these two hubs, say h, the following holds wrt. removal of their connecting link:
        i. If l is the only link connected to h then hub h is removed. This may mean that
            - either one
            - or two hubs

            are also removed when the link is removed.
        ii. If l is not the only link connected to h then the hub h is modified to reflect that it is no longer connected to l.
    d) The resulting net is that of the pair of adjusted set of hubs and links.

value
15 int_Remove: Rmv → N $\xrightarrow{\sim}$ N
16 int_Remove(Rmv(li))(hs,ls) ≡
16a) **let** l = xtr_L(li)(ls), {hi′,hi″} = obs_HIs(l) **in**
16b) **let** {h′,h″} = {xtr_H(hi′,hs),xtr_H(hi″,hs)} **in**
16c) **let** hs′ = cond_rmv(h′,hs) ∪ cond_rmv_H(h″,hs) **in**
16d) (hs\{h′,h″} ∪ hs′,ls\{l}) **end end end**
16a) **pre** li ∈ iols(ls)

cond_rmv: LI × H × H-**set** → H-**set**
cond_rmv(li,h,hs) ≡
16(c)i) **if** obs_HIs(h)={li} **then** {}
16(c)ii) **else** {sLI(li,h)} **end**
pre li ∈ obs_HIs(h)

RSL Explanation

- 15: The int_Remove operation applies to a remove command Rmv(li) and a net (hs,ls) and yields a net — provided the remove command is semantically well-formed.
- 16: To Remove a link identifier by li from the net (hs,ls) can be formalised as follows:
  - 16a): obtain the link l from its identifier li and the set of links ls, and

- 16a): obtain the identifiers, {hi′,hi″}, of the two distinct hubs to which link l is connected;
- 16b): then obtain the hubs {h′,h″} with these identifiers;
- 16c): now examine cond_rmv each of these hubs (see Lines 16(c)i–16(c)ii)).
  - The examination function cond_rmv either yields an empty set or the singleton set of one modified hub (a link identifier has been removed).
  - 16c) The set, hs′, of zero, one or two modified hubs is yielded.
  - That set is joined to the result of removing the hubs {h′,h″}
  - and the set of links that result from removing l from ls.

  The conditional hub remove function cond_rmv
- 16(c)i): either yields the empty set (of no hubs) if li is the only link identifier inh,
- 16(c)ii): or yields a modification of h in which the link identifier li is no longer observable.

End of RSL Explanation

44 A.7.1 Modelling Connected Links and Hubs

Examples (??–??) of this section, i.e., Sect. ?? are building up a model of one form of meaning of a transport net. We model the movement of vehicles around hubs and links. We think of each hub, each link and each vehicle to be a process. These processes communicate via channels.

- We assume a net, $n : N$, and a set, vs, of vehicles.
- Each vehicle can potentially interact
  - with each hub and
  - with each link.
- Array channel indices *(vi,hi):IVH* and *(vi,li):IVL* serve to effect these interactions.
- Each hub can interact with each of its connected links and indices *(hi,li):IHL* serves these interactions.

type
  N, V, VI
value
  n:N, vs:V**-set**
  ωVI: V → VI
type
  H, L, HI, LI, M
  IVH = VI×HI, IVL = VI×LI, IHL = HI×LI

- We need some auxiliary quantities in order to be able to express subsequent channel declarations.
- Given that we assume a net, $n : N$ and a set of vehicles, $vs : VS$, we can now define the following (global) values:
  - the sets of hubs, hs, and links, ls of the net;
  - the set, $ivhs$, of indices between vehicles and hubs,
  - the set, $ivls$, of indices between vehicles and links, and
  - the set, $ihls$, of indices between hubs and links.

value
hs:H-**set** = ωHs(n), ls:L-**set** = ωLs(n)
his:HI-**set** = {ωHI(h)|h:H•h ∈ hs}, lis:LI-**set** = {ωLI(h)|l:L•l ∈ ls},
ivhs:IVH-**set** = {(ωVI(v),ωHI(h))|v:V,h:H•v ∈ vs∧h ∈ hs}
ivls:IVL-**set** = {(ωVI(v),ωLI(l))|v:V,l:L•v ∈ vs∧l ∈ ls}
ihls:IHL-**set** = {(hi,li)|h:H,(hi,li):IHL• h ∈ hs∧hi=ωHI(h)∧li ∈ ωLIs(h)}

- We are now ready to declare the channels:
  - a set of channels, {vh[i]|i:IVH•i∈ivhs} between vehicles and all potentially traversable hubs;
  - a set of channels, {vh[i]|i:IVH•i∈ivhs} between vehicles and all potentially traversable links; and
  - a set of channels, {hl[i]|i:IHL•i∈ihls}, between hubs and connected links.

channel
{vh[i] | i:IVH • i ∈ ivhs} : M
{vl[i] | i:IVL • i ∈ ivls} : M
{hl[i] | i:IHL • i ∈ ihls} : M

45 A.7.2 Communicating Hubs, Links and Vehicles

- Hubs interact with links and vehicles:
  - with all immediately adjacent links,
  - and with potentially all vehicles.
- Links interact with hubs and vehicles:
  - with both adjacent hubs,

  - and with potentially all vehicles.

- Vehicles interact with hubs and links:
  - with potentially all hubs.
  - and with potentially all links.

```
value
  hub: hi:HI × h:H → in,out {hl[ (hi,li)|li:LI•li ∈ ωLIs(h) ]}
                           in,out {vh[ (vi,hi)|vi:VI•vi ∈ vis ]}  Unit
  link: li:LI × l:L → in,out {hl[ (hi,li)|hi:HI•hi ∈ ωHIs(l) ]}
                          in,out {vh[ (vi,li)|vi:VI•vi ∈ vis ]}  Unit
  vehicle: vi:VI → (Pos × Net) → v:V → in,out {vh[ (vi,hi)|hi:HI•hi ∈ his ]}  Unit
                            in,out {vh[ (vi,li)|li:LI•li ∈ lis ]}  Unit
```

46 A.7.3 Modelling Transport Nets

- The net, with vehicles, potential or actual, is now considered a process.
- It is the parallel composition of
  - all hub processes,
  - all link processes and
  - all vehicle processes.

```
value
  net: N → V-set → Unit
  net(n)(vs) ≡
    ‖ {hub( ωHI(h))(h)|h:H•h ∈ ωHs(n)} ‖
    ‖ {link( ωLI(l))(l)|l:L•l ∈ ωLs(n)} ‖
    ‖ {vehicle(ωVI(v))(v)|v:V•v ∈ vs}
```

- We illustrate a schematic definition of simplified hub processes.
- The hub process alternates, internally non-deterministically, $\sqcap$, between three sub-processes
  - a sub-process which serves the link-hub connections,
  - a sub-process which serves thos vehicles which communicate that they somehow wish to enter or leave (or do something else with respect to) the hub, and
  - a sub-process which serves the hub itself — whatever that is !

hub(hi)(h) ≡
  []{**let** m = hl[li] ? **in** hub(hi)($\mathcal{E}_{h_\ell}$(li)(m)(h)) **end**|i:LI•li ∈ ωLI(h)}
  [] []{**let** m = vh[vi] ? **in** hub(vi)($\mathcal{E}_{h_v}$(vi)(,)(h)) **end**|vi:VI•vi ∈ vis}
  [] hub(hi)($\mathcal{E}_{h_{own}}$(h))

- The three auxiliary processes:
  - $\mathcal{E}_{h_\ell}$ update the hub with respect to (wrt.) connected link, *li*, information *m*,
  - $\mathcal{E}_{h_v}$ update the hub with wrt. vehicle, *vi*, information *m*,
  - $\mathcal{E}_{h_{own}}$ update the hub with wrt. whatever the hub so decides. An example could be signalling dependent on previous link-to-hub communicated information, say about traffic density.

$\mathcal{E}_{h_\ell}$: LI → M → H → H
$\mathcal{E}_{h_v}$: VI → M → H → H
$\mathcal{E}_{h_{own}}$: H → H

The reader is encouraged to sketch/define similarly schematic link and vehicle processes.
.

47 A.7.4 Modelling Vehicle Movements

- Whereas hubs and links are modelled as basically static, passive, that is, inert, processes we shall consider vehicles to be "highly" dynamic, active processes.
- We assume that a vehicle possesses knowledge about the road net.
  - The road net is here abstracted as an awareness of
  - which links, by their link identifiers,
  - are connected to any given hub, designated by its hub identifier,
  - the length of the link,
  - and the hub to which the link is connected "at the other end", also by its hub identifier
- A vehicle is further modelled by its current position on the net in terms of either hub or link positions
  - designated by appropriate identifiers
  - and, when "on a link" "how far down the link", by a measure of a fraction of the total length of the link, the vehicle has progressed.

```
type
  Net = HI ~m~> (LI ~m~> HI)
  Pos = atH | onL
  atH == μatH(hi:HI)
  onL == μonL(fhi:HI,li:LI,f:F,thi:HI)
  F = {|f:Real•0≤f≤1|}
value
  ωNet: V → Net
  ωPos: V → POS
```

- We first assume that the vehicle is at a hub.
- There are now two possibilities (1–2] versus [4–8]).
  - Either the vehicle remains at that hub
    - [1] which is expressed by some non-deterministic *wait*
    - [2] followed by a resumption of being that vehicle at that location.
  - [3] Or the vehicle (driver) decides to "move on":
    - [5] Onto a link, *li*,
    - [4] among the links, *lis*, emanating from the hub,
    - [6] and towards a next hub, *hi′*.
  - [4,6] The *lis* and *hi′* quantities are obtained from the vehicles own knowledge of the net.
  - [7] The hub and the chosen link are notified by the vehicle of its leaving the hub and entering the link,
  - [8] whereupon the vehicle resumes its being a vehicle at the initial location on the chosen link.
- The vehicle chooses between these two possibilities by an internal non-deterministic choice ([3]).

```
type
  M == μL_H(li:LI,hi:HI) | μH_L(hi:HI,li:LI)
value
  vehicle: VI → (Pos × Net) → V → Unit
  vehicle(vi)(μatH(hi),net)(v) ≡
  [1] (wait ;
  [2]  vehicle(vi)(pos,net)(v))
  [3] ⌈⌉
  [4] (let lis=dom net(hi) in
```

```
[ 5 ]  let li:LI•li ∈ lis in
[ 6 ]  let hi′=(net(hi))(li) in
[ 7 ]  (vh[ vi ]!μH_L(hi,li)‖vl[ vi ]!μH_L(hi,li));
[ 8 ]  vehicle(vi)(μonL(hi,li,0,hi′),net)(v)
[ 9 ]  end end end)
```

- We then assume that the vehicle is on a link and at a certain distance "down", *f*, that link.
- There are now two possibilities ([1–2] versus [4–7]).
  - Either the vehicle remains at that hub
    - [1′] which is expressed by some non-deterministic *wait*
    - [2′] followed by a resumption of being that vehicle at that location.
  - [3′] Or the vehicle (driver) decides to "move on".
  - [4′] Either
    - [5′] The vehicle is at the very end of the link and signals the link and the hub of its leaving the link and entering the hub,
    - [6′] whereupon the vehicle resumes its being a vehicle at hub h'.
  - [7′] or the vehicle moves further down, some non-zero fraction down the link.
- The vehicle chooses between these two possibilities by an internal non-deterministic choice ([3]).

```
type
  M == μL_H(li:LI,hi:HI) | μH_L(hi:HI,li:LI)
value
  δ:Real = move(h,f) axiom 0<δ≪1
  vehicle(vi)( μonL(hi,li,f,hi′),net)(v) ≡
  [ 1′ ]  (wait ;
  [ 2′ ]   vehicle(vi)(pos,net)(v))
  [ 3′ ]  ⌈⌉
  [ 4′ ]  (case f of
  [ 5′ ]     1 → ((vl[ vi ]!μL_H(li,hi′)‖vh[ vi ]!μL_H(li,hi′));
  [ 6′ ]           vehicle(vi)(μatH(hi′),net)(v)),
  [ 7′ ]     _ → vehicle(vi)(μonL(hi,li,f+δ,hi′),net)(v)
  [ 8′ ]   end)
  move: H × F → F
```

48 A.8 A Neat Little "System"

We present a self-contained specification of a simple system: The system models vehicles moving along a net, *vehicle*, the recording of vehicles entering links, *enter_sensor*, the recording of vehicles leaving links, *leave_sensor*, and the *road_pricing payment* of a vehicle having traversed (*entered* and *left*) a link. Note that vehicles only pay when completing a link traversal; that 'road pricing' only commences once a vehicle enters the first link after possibly having left an earlier link (and hub); and that no *road_pricing payment* is imposed on vehicles entering, staying-in (or at) and leaving hubs.

We assume the following: that each *link* is somehow associated with two pairs of *sensors:* a pair of *enter* and *leave sensors* at one end, and a pair of *enter* and *leave sensors* at the other end; and a *road pricing* process which records pairs of link enterings and leavings, first one, then, after any time interval, the other, with leavings leading to debiting of traversal fees; Our first specification define types, assume a net value, declares channels and state signatures of all processes.

- *ves* stand for vehicle entering (link) sensor channels,
- *vls* stand for vehicle leaving (link) sensor channels,
- *rp* stand for 'road pricing' channel
- *enter_sensor(hi,li)* stand for vehicle entering [sensor] process from hub *hi* to link *(*li).
- *leave_sensor(li,hi)* stand for vehicle leaving [sensor] process from link *li* to hub *(*hi).
- *road_pricing()* stand for the unique 'road pricing' process.
- *vehicle(vi)(...)* stand for the vehicle *vi* process.

type
  N, H, HI, LI, VI
  RPM == μEnter_L(vi:VI,li:LI) | μLeave_L(vi:VI,li:LI)
value
  n:N
channel
  {ves[ωHI(h),li]|h:H•h $\in$ ωHs(n)$\wedge$li $\in$ ωLIs(h)}:VI
  {vls[li,ωHI(h)]|h:H•h $\in$ ωHs(n)$\wedge$li $\in$ ωLIs(h)}:VI
  rp:RPM
type
  Fee, Bal
  LVS = LI $\xrightarrow[m]{}$ VI-**set**, FEE = LI $\xrightarrow[m]{}$ Fee, ACC = VI $\xrightarrow[m]{}$ Bal
value
  link: (li:LI × L) → **Unit**
  enter_sensor: (hi:HI × li:LI) → **in** ves[hi,li],**out** rp **Unit**
  leave_sensor: (li:LI × hi:HI) → **in** vls[li,hi],**out** rp **Unit**
  road_pricing: (LVS×FEE×ACC) → **in** rp **Unit**

To understand the sensor behaviours let us review the vehicle behaviour. In the *vehicle* behaviour defined in Example **??**, in two parts, Page 42 and Page 43 we focus on the events [7] where the vehicle enters a link, respectively [5′] where the vehicle leaves a link. These are summarised in the schematic reproduction of the vehicle behaviour description. We redirect the interactions between vehicles and links to become interactions between vehicles and enter and leave sensors.

```
value
  δ:Real = move(h,f) axiom 0<δ≪1
  move: H × F → F
  vehicle: VI → (Pos × Net) → V → Unit
  vehicle(vi)(pos,net)(v) ≡
  [1] (wait ;
  [2]  vehicle(vi)(pos,net)(v))
  [3]  ⌈⌉
       case pos of
          μatH(hi) →
  [4−6]   (let lis=dom net(hi) in let li:LI•li ∈ lis in let hi′=(net(hi))(li) in
  [7]       ves[hi,li]!vi;
  [8]       vehicle(vi)(μonL(hi,li,0,hi′),net)(v)
  [9]       end end end)
          μonL(hi,li,f,hi′) →
  [4′]      (case f of
  [5′−6′]     1 → (vls[li,hi]!vi; vehicle(vi)(μatH(hi′),net)(v)),
  [7′]        _ → vehicle(vi)(μonL(hi,li,f+δ,hi′),net)(v)
  [8′]      end)
       end
```

- As mentioned on Page 44 *link* behaviours are associated with two pairs of sensors:
  - a pair of *enter* and *leave sensors* at one end, and
  - a pair of *enter* and *leave sensors* at the other end;

```
value
  link(li)(l) ≡
     let {hi,hi′} = ωHIs(l) in
     enter_sensor(hi,li) ‖ leave_sensor(li,hi) ‖
     enter_sensor(hi′,li) ‖ leave_sensor(li,hi′) end
  enter_sensor(hi,li) ≡
     let vi = ves[hi,li]? in rp!μEnter_LI(vi,li); enter_sensor(hi,li) end
  leave_sensor(li,hi) ≡
     let vi = ves[li,hi]? in rp!μLeave_LI(vi,li); enter_sensor(li,hi) end
```

- The *LVS* component of the *road_pricing* behaviour serves,
  - among other purposes that are not mentioned here,
  - to record whether the movement of a vehicles "originates" along a link or not.
- Otherwise we leave it to the reader to carefully read the formulas.

```
value
  payment: VI × LI → (ACC × FEE) → ACC
  payment(vi,li)(fee,acc) ≡
    let bal′ = if vi ∈ dom acc then add(acc(vi),fee(li)) else fee(li) end
    in acc † [ vi ↦ bal′ ] end
  add: Fee × Bal → Bal [ add fee to balance ]
  road_pricing(lvs,fee,acc) ≡ in rp
    let m = rp? in
    case m of
      μEnter_LI(vi,li) →
          road_pricing(lvs†[ li↦lvs(li)∪{vi} ],fee,acc),
      μLeave_LI(vi,li) →
          let lvs′ = if vi ∈ lvs(li) then lvs†[ li↦lvs(li)\{vi} ] else lvs end,
              acc′ = payment(vi,li)(fee,acc) in
          road_pricing(lvs′,fee,acc′)
    end end end
```

B. Mereology

49 B.1.2 Simple and Composite Net Entities

We repeat some of the material from Example ?? on page ??.

- [1] A road, train, airlane (air traffic) or sea lane (shipping) net
- [2] consists, amongst other things, of hubs and links.

```
type
  [ 1 ] N
  [ 2 ] H, L
value
  [ 2 ] ωHs: N → H-set, ωLs: N → L-set,
```

We can consider nets as composite and, for the time being, hubs and links as simple.
.

50 B.1.2 Simple and Composite Net Functions

- [3] With every link we associate a length.
- [4] A journey is a pair of a link and a continuation.
- [5] A continuation is either `"nil"` or is a journey.
- [6] Journies have lengths:
  - [6.1] the length of the link of the journey pair,
  - [6.2] and the length of the continuation – where a `"nil"` continuation has length 0.

type
[3] LEN
[4] Journey = L × C
[5] C = ″`nil`″ | Journey
value
[3] zero_LEN:LEN
[3] ωLEN: L → LEN
[6] length: Journey → LEN
[6] length(l,c) ≡
[6.1] **let** ll = ωLEN(l),
[6.2] cl = **if** c=″`nil`″ **then** zero_LEN **else** length(c) **end in**
[6] sum(ll,cl) **end**
sum: LEN × LEN → LEN

Both the journey and continuation entities, *j* and *c* , and the *length* function are composite Both the link entities, *ll*, the *ωLEN* function are atomic.

51 B.1.2 Simple and Composite Net Events

- [7] The isolated crash of two vehicles, at time t, in a traffic, at a hub or along a link can be construed as a single atomic event.
- [8] The crash, within a few seconds $(t, t', t \sim t')$, in a traffic, of three or more vehicles,
  - [8.1] in a hub,
  - [8.2] or along a short segment of a link,

  can be considered a composite event.

We shall model this event by the predicates which holds of vehicles in a traffic at given times.

type
TF = T → (V $\overrightarrow{m}$ Pos)
Pos == μatH(hi:HI) | μonL(πhi:HI,πli:LI,πf:F,πhi′:HI)
type
value
[7] atomic_crash: V × V → TF → T → **Bool**
[7] atomic_crash(v,v′)(tf)(t) ≡ (tf(t))(v)=(tf(t))(v′)
[7] **pre** t ∈ $\mathcal{DOMAIN}$tf ∧ {v,v′}⊆**dom**(tf(t))∧v≠v′

[8] composite_crash: V-**set** → TF → (T×T) → **Bool**
[8] composite_crash(vs)(tf)(t,t′) ≡
[8.1] ∃ hi:HI • **card**{v|v:V• ∈ vs∧(tf(t″))(v)=μatH(hi)∧t≤t″≤t′}≥3∨
[8.2] ∃ hi′,hi″:HI,li:LI,fs:F-**set** •
[8.2] fs={r..r′} **where** 0≤r≃r′≤1 ∧
[8.2] **card**{(tf(t″))(v)=μonL(hi′,li,f,hi″)|v:V,f:F•v ∈ vs∧f ∈ fs∧t≤t″≤t′}≥3
[8] **pre** {t,t′}⊆$\mathcal{DOMAIN}$tf ∧ t∼t′ ∧ ∧ vs⊆**dom**(tf(t)) ∧ **card** vs≥3

52 B.1.2 Simple and Composite Net Behaviours

Pipeline Systems and Their Units

17. We focus on nets, $n : N$, of pipes, $\pi : \Pi$, valves, $v : V$, pumps, $p : P$, forks, $f : F$, joins, $j : J$, wells, $w : W$ and sinks, $s : S$.

18. Units, $u : U$, are either pipes, valves, pumps, forks, joins, wells or sinks.

19. Units are explained in terms of disjoint types of PIpes, VAlves, PUmps, FOrks, JOins, WElls and SKs.[2]

type
17 N, PI, VA, PU, FO, JO, WE, SK
18 U = Π | V | P | F | J | S| W
18 Π == mkΠ(pi:PI)
18 V == mkV(va:VA)
18 P == mkP(pu:PU)
18 F == mkF(fo:FO)
18 J == mkJ(jo:JO)
18 W == mkW(we:WE)
18 S == mkS(sk:SK)

[2]This is a mere specification language technicality.

Unit Identifiers and Unit Type Predicates

20. We associate with each unit a unique identifier, $ui : UI$.
21. From a unit we can observe its unique identifier.
22. From a unit we can observe whether it is a pipe, a valve, a pump, a fork, a join, a well or a sink unit.

type
20 UI
value
21 obs_UI: U → UI
22 is_Π: U → **Bool**, is_V: U → **Bool**, ..., is_J: U → **Bool**
is_Π(u) ≡ **case** u **of** mkPI(_) → **true**, _ → **false end**
is_V(u) ≡ **case** u **of** mkV(_) → **true**, _ → **false end**
...
is_S(u) ≡ **case** u **of** mkS(_) → **true**, _ → **false end**

Unit Connections

A connection is a means of juxtaposing units. A connection may connect two units in which case one can observe the identity of connected units from "the other side".

23. With a pipe, a valve and a pump we associate exactly one input and one output connection.
24. With a fork we associate a maximum number of output connections, m, larger than one.
25. With a join we associate a maximum number of input connections, m, larger than one.
26. With a well we associate zero input connections and exactly one output connection.
27. With a sink we associate exactly one input connection and zero output connections.

value
23 obs_InCs,obs_OutCs: Π|V|P → {|1:**Nat**|}
24 obs_inCs: F → {|1:**Nat**|}, obs_outCs: F → **Nat**
25 obs_inCs: J → **Nat**, obs_outCs: J → {|1:**Nat**|}
26 obs_inCs: W → {|0:**Nat**|}, obs_outCs: W → {|1:**Nat**|}
27 obs_inCs: S → {|1:**Nat**|}, obs_outCs: S → {|0:**Nat**|}
axiom
24 ∀ f:F • obs_outCs(f) ≥ 2
25 ∀ j:J • obs_inCs(j) ≥ 2

If a pipe, valve or pump unit is input-connected [output-connected] to zero (other) units, then it means that the unit input [output] connector has been sealed. If a fork is input-connected to zero (other) units, then it means that the fork input connector has been sealed. If a fork is output-connected to n units less than the maximum fork-connectability, then it means that the unconnected fork outputs have been sealed. Similarly for joins: "the other way around".

Net Observers and Unit Connections

28. From a net one can observe all its units.

29. From a unit one can observe the the pairs of disjoint input and output units to which it is connected:

    a) Wells can be connected to zero or one output unit — a pump.

    b) Sinks can be connected to zero or one input unit — a pump or a valve.

    c) Pipes, valves and pumps can be connected to zero or one input units and to zero or one output units.

    d) Forks, f, can be connected to zero or one input unit and to zero or n, $2 \leq n \leq$obs_Cs(f) output units.

    e) Joins, j, can be connected to zero or n, $2 \leq n \leq$obs_Cs(j) input units and zero or one output units.

```
value
  28  obs_Us: N → U-set
  29  obs_cUIs: U → UI-set × UI-set
      wf_Conns: U → Bool
      wf_Conns(u) ≡
        let (iuis,ouis) = obs_cUIs(u) in iuis ∩ ouis = {} ∧
        case u of
  29a  mkW(_) → card iuis ∈ {0} ∧ card ouis ∈ {0,1},
  29b  mkS(_) → card iuis ∈ {0,1} ∧ card ouis ∈ {0},
  29c  mkΠ(_) → card iuis ∈ {0,1} ∧ card ouis ∈ {0,1},
  29c  mkV(_) → card iuis ∈ {0,1} ∧ card ouis ∈ {0,1},
  29c  mkP(_) → card iuis ∈ {0,1} ∧ card ouis ∈ {0,1},
  29d  mkF(_) → card iuis ∈ {0,1} ∧ card ouis ∈ {0}∪{2..obs_inCs(j)},
  29e  mkJ(_) → card iuis ∈ {0}∪{2..obs_inCs(j)} ∧ card ouis ∈ {0,1}
        end end
```

Well-formed Nets, Actual Connections

30. The unit identifiers observed by the obs_cUIs observer must be identifiers of units of the net.

```
axiom
  30 ∀ n:N,u:U • u ∈ obs_Us(n) ⇒
  30   let (iuis,ouis) = obs_cUIs(u) in
  30   ∀ ui:UI • ui ∈ iuis ∪ ouis ⇒
  30     ∃ u′:U • u′ ∈ obs_Us(n) ∧ u′≠u ∧ obs_UI(u′)=ui end
```

Well-formed Nets, No Circular Nets

31. By a route we shall understand a sequence of units.

32. Units form routes of the net.

```
type
  31 R = UI^ω
value
  32 routes: N → R-infset
  32 routes(n) ≡
  32   let us = obs_Us(n) in
  32   let rs = {⟨u⟩|u:U•u ∈ us} ∪ {r⌢r′|r,r′:R• {r,r′}⊆rs∧adj(r,r′)} in
  32   rs end end
```

33. A route of length two or more can be decomposed into two routes

34. such that the least unit of the first route "connects" to the first unit of the second route.

```
value
  33   adj: R × R → Bool
  33   adj(fr,lr) ≡
  33     let (lu,fu)=(fr(len fr),hd lr) in
  34     let (lui,fui)=(obs_UI(lu),obs_UI(fu)) in
  34     let ((_,luis),(fuis,_))=(obs_cUIs(lu),obs_cUIs(fu)) in
  34     lui ∈ fuis ∧ fui ∈ luis end end end
```

35. No route must be circular, that is, the net must be acyclic.

```
value
  35 acyclic: N → Bool
  35   let rs = routes(n) in
  35   ∼∃ r:R•r ∈ rs⇒∃ i,j:Nat•{i,j}⊆inds r∧i≠j∧r(i)=r(j) end
```

Pipeline Processes

We now add connectors to our model:

36. From an oil pipeline system one can observe units and connectors.
37. Units are either well, or pipe, or pump, or valve, or join, or fork or sink units.
38. Units and connectors have unique identifiers.
39. From a connector one can observe the ordered pair of the identity of the two from-, respectively to-units that the connector connects.

type
36 OPLS, U, K
38 UI, KI
value
36 obs_Us: OPLS → U-**set**, obs_Ks: OPLS → K-**set**
37 is_WeU, is_PiU, is_PuU, is_VaU, is_JoU, is_FoU, is_SiU: U → **Bool** [mutually exclusive]
38 obs_UI: U → UI, obs_KI: K → KI
39 obs_UIp: K → (UI|{nil}) × (UI|{nil})

Above, we think of the types OPLS, U, K, UI and KI as denoting semantic entities. Below, in the next section, we shall consider exactly the same types as denoting syntactic entities !

40. There is given an oil pipeline system, opls.
41. To every unit we associate a CSP behaviour.
42. Units are indexed by their unique unit identifiers.
43. To every connector we associate a CSP channel.

    Channels are indexed by their unique "k"onnector identifiers.
44. Unit behaviours are cyclic and over the state of their (static and dynamic) attributes, represented by u.
45. Channels, in this model, have no state.
46. Unit behaviours communicate with neighbouring units — those with which they are connected.
47. Unit functions, $\mathcal{U}_i$, change the unit state.
48. The pipeline system is now the parallel composition of all the unit behaviours.

Editorial Remark: Our use of the term unit and the RSL literal **Unit** may seem confusing, and we apologise. The former, unit, is the generic name of a well, pipe, or pump, or valve, or join, or fork, or sink. The literal **Unit**, in a function signature, before the → "announces" that the function takes no argument.[3] The literal **Unit**, in a function signature, after the → "announces", as used here, that the function never terminates.

```
value
40  opls:OPLS
channel
43  {ch[ ki ]|k:KI,k:K•k ∈ obs_Ks(opls)∧ki=obs_KI(k)} M
value
48  pipeline_system: Unit → Unit
48  pipeline_system() ≡
41    ‖ {unit(ui)(u)|u:U•u ∈ obs_Us(opls)∧ui=obs_UI(u)}

42  unit: ui:UI → U →
46        in,out {ch[ ki ]|k:K,ki:KI•k ∈ obs_Ks(opls)∧ki=obs_KI(k)∧
46                    let (ui′,ui″)=obs_UIp(k) in ui ∈{ui′,ui″}\{nil} end}  Unit
44  unit(ui)(u) ≡ let u′ = U_i(ui)(u) in unit(ui)(u′) end

47  U_i: ui:UI → U →
47        in,out {ch[ ki ]|k:K,ki:KI•k ∈ obs_Ks(opls)∧ki=obs_KI(k)∧
47                    let (ui′,ui″)=obs_UIp(k) in ui ∈{ui′,ui″}\{nil} end}  U
```

53 B.5 Pipeline Transport Functions and Events

We need introduce a number of auxiliary concepts in order to show examples of atomic and composite functions and events.

Well-formed Nets, Special Pairs, wfN_SP

49. We define a "special-pairs" well-formedness function.
    a) Fork outputs are output-connected to valves.
    b) Join inputs are input-connected to valves.
    c) Wells are output-connected to pumps.
    d) Sinks are input-connected to either pumps or valves.

```
value
  49  wfN_SP: N → Bool
```

[3] **Unit** is a type name; () is the only value of type **Unit**.

```
49  wfN_SP(n) ≡
49   ∀ r:R • r ∈ routes(n) in
49    ∀ i:Nat • {i,i+1}⊆inds r ⇒
49      case r(i) of ∧
49a       mkF(_) → ∀ u:U•adj(⟨r(i)⟩,⟨u⟩) ⇒ is_V(u),_→true end ∧
49      case r(i+1) of
49b       mkJ(_) → ∀ u:U•adj(⟨u⟩,⟨r(i)⟩) ⇒ is_V(u),_→true end ∧
49      case r(1) of
49c       mkW(_) → is_P(r(2)),_→true end ∧
49      case r(len r) of
49d       mkS(_) → is_P(r(len r−1))∨is_V(r(len r−1)),_→true end
```

The **true** clauses may be negated by other **case** distinctions' is_V or is_V clauses.

Special Routes, I

50. A pump-pump route is a route of length two or more whose first and last units are pumps and whose intermediate units are pipes or forks or joins.
51. A simple pump-pump route is a pump-pump route with no forks and joins.
52. A pump-valve route is a route of length two or more whose first unit is a pump, whose last unit is a valve and whose intermediate units are pipes or forks or joins.
53. A simple pump-valve route is a pump-valve route with no forks and joins.
54. A valve-pump route is a route of length two or more whose first unit is a valve, whose last unit is a pump and whose intermediate units are pipes or forks or joins.
55. A simple valve-pump route is a valve-pump route with no forks and joins.
56. A valve-valve route is a route of length two or more whose first and last units are valves and whose intermediate units are pipes or forks or joins.
57. A simple valve-valve route is a valve-valve route with no forks and joins.

```
value
  50-57  ppr,sppr,pvr,spvr,vpr,svpr,vvr,svvr: R → Bool
           pre {ppr,sppr,pvr,spvr,vpr,svpr,vvr,svvr}(n): len n≥2

  50  ppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_P(fu) ∧ is_P(lu) ∧ is_πfjr(ℓ)
  51  sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
  52  pvr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_P(fu) ∧ is_V(r(len r)) ∧ is_πfjr(ℓ)
  53  sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
  54  vpr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_V(fu) ∧ is_P(lu) ∧ is_πfjr(ℓ)
```

55 sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
56 vvr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_V(fu) ∧ is_V(lu) ∧ is_πfjr(ℓ)
57 sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)

is_πfjr,is_πr: R → **Bool**
is_πfjr(r) ≡ ∀ u:U•u ∈ **elems** r⇒is_Π(u)∨is_F(u)∨is_J(u)
is_πr(r) ≡ ∀ u:U•u ∈ **elems** r⇒is_Π(u)

Special Routes, II

Given a unit of a route,

58. if they exist (∃),
59. find the nearest pump or valve unit,
60. "upstream" and
61. "downstream" from the given unit.

value
58 ∃UpPoV: U × R → **Bool**
58 ∃DoPoV: U × R → **Bool**
60 find_UpPoV: U × R ~→ (P|V), **pre** find_UpPoV(u,r): ∃UpPoV(u,r)
61 find_DoPoV: U × R ~→ (P|V), **pre** find_DoPoV(u,r): ∃DoPoV(u,r)
58 ∃UpPoV(u,r) ≡
58 ∃ i,j **Nat**•{i,j}⊆**inds** r∧i≤j∧{is_V|is_P}(r(i))∧u=r(j)
58 ∃DoPoV(u,r) ≡
58 ∃ i,j **Nat**•{i,j}⊆**inds** r∧i≤j∧u=r(i)∧{is_V|is_P}(r(j))
60 find_UpPoV(u,r) ≡
60 **let** i,j:**Nat**•{i,j}⊆indsr∧i≤j∧{is_V|is_P}(r(i))∧u=r(j) **in** r(i) **end**
61 find_DoPoV(u,r) ≡
61 **let** i,j:**Nat**•{i,j}⊆indsr∧i≤j∧u=r(i)∧{is_V|is_P}(r(j)) **in** r(j) **end**

State Attributes of Pipeline Units

By a state attribute of a unit we mean either of the following three kinds: (i) the open/close states of valves and the pumping/not_pumping states of pumps; (ii) the maximum (laminar) oil flow characteristics of all units; and (iii) the current oil flow and current oil leak states of all units.

62. Oil flow, $\phi : \Phi$, is measured in volume per time unit.

63. Pumps are either pumping or not pumping, and if not pumping they are closed.

64. Valves are either open or closed.

65. Any unit permits a maximum input flow of oil while maintaining laminar flow. We shall assume that we need not be concerned with turbulent flows.

66. At any time any unit is sustaining a current input flow of oil (at its input(s)).

67. While sustaining (even a zero) current input flow of oil a unit leaks a current amount of oil (within the unit).

```
type
  62  Φ
  63  PΣ == pumping | not_pumping
  63  VΣ == open | closed
value
      −,+: Φ × Φ → Φ, <,=,>: Φ × Φ → Bool
  63   obs_PΣ: P → PΣ
  64   obs_VΣ: V → VΣ
  65–67  obs_LamiΦ.obs_CurrΦ,obs_LeakΦ: U → Φ
  is_Open: U → Bool
    case u of
      mkΠ(_)→true,mkF(_)→true,mkJ(_)→true,mkW(_)→true,mkS(_)→true,
      mkP(_)→obs_PΣ(u)=pumping,
      mkV(_)→obs_VΣ(u)=open
    end
  acceptable_LeakΦ, excessive_LeakΦ: U → Φ
axiom
  ∀ u:U • excess_LeakΦ(u) > accept_LeakΦ(u)
```

Flow Laws

The sum of the current flows into a unit equals the the sum of the current flows out of a unit minus the (current) leak of that unit. This is the same as the current flows out of a unit equals the current flows into a unit minus the (current) leak of that unit. The above represents an interpretation which justifies the below laws.

68. When, in Item 66, for a unit u, we say that at any time any unit is sustaining a current input flow of oil, and when we model that by obs_CurrΦ(u) then we mean that obs_CurrΦ(u) - obs_LeakΦ(u) represents the flow of oil from its outputs.

value
68 obs_inΦ: U → Φ
68 obs_inΦ(u) ≡ obs_CurrΦ(u)
68 obs_outΦ: U → Φ
law:
68 ∀ u:U • obs_outΦ(u) = obs_CurrΦ(u)−obs_LeakΦ(u)

69. Two connected units enjoy the following flow relation:
    a) If
        i. two pipes, or
        ii. a pipe and a valve, or
        iii. a valve and a pipe, or
        iv. a valve and a valve, or
        v. a pipe and a pump, or
        vi. a pump and a pipe, or
        vii. a pump and a pump, or
        viii. a pump and a valve, or
        ix. a valve and a pump

        are immediately connected
    b) then
        i. the current flow out of the first unit's connection to the second unit
        ii. equals the current flow into the second unit's connection to the first unit

law:
69a ∀ u,u′:U • {is_Π,is_V,is_P,is_W}(u′|u″) ∧ adj(⟨u⟩,⟨u′⟩)
69a is_Π(u)∨is_V(u)∨is_P(u)∨is_W(u) ∧
69a is_Π(u′)∨is_V(u′)∨is_P(u′)∨is_S(u′)
69b ⇒ obs_outΦ(u)=obs_inΦ(u′)

A similar law can be established for forks and joins. For a fork output-connected to, for example, pipes, valves and pumps, it is the case that for each fork output the out-flow equals the in-flow for that output-connected unit. For a join input-connected to, for example, pipes, valves and pumps, it is the case that for each join input the in-flow equals the out-flow for that input-connected unit. We leave the formalisation as an exercise.

Possibly Desirable Properties

70. Let r be a route of length two or more, whose first unit is a pump, p, whose last unit is a valve, v and whose intermediate units are all pipes: if the pump, p is pumping, then we expect the valve, v, to be open.

71. Let r be a route of length two or more, whose first unit is a pump, p, whose last unit is another pump, p' and whose intermediate units are all pipes: if the pump, p is pumping, then we expect pump p'', to also be pumping.

72. Let r be a route of length two or more, whose first unit is a valve, v, whose last unit is a pump, p and whose intermediate units are all pipes: if the valve, v is closed, then we expect pump p, to not be pumping.

73. Let r be a route of length two or more, whose first unit is a valve, v', whose last unit is a valve, v'' and whose intermediate units are all pipes: if the valve, v' is in some state, then we expect valve v'', to also be in the same state.

desirable properties:

```
70  ∀ r:R • spvr(r) ∧
70    spvr_prop(r): obs_PΣ(hd r)=pumping ⇒ obs_PΣ(r(len r))=open

71  ∀ r:R • sppr(r) ∧
71    sppr_prop(r): obs_PΣ(hd r)=pumping⇒obs_PΣ(r(len r))=pumping

72  ∀ r:R • svpr(r) ∧
72    svpr_prop(r): obs_PΣ(hd r)=open⇒obs_PΣ(r(len r))=pumping

73  ∀ r:R • svvr(r) ∧
73    svvr_prop(r): obs_PΣ(hd r)=obs_PΣ(r(len r))
```

Pipeline Actions

- ***Simple Pump and Valve Actions***

74. Pumps may be set to pumping or reset to not pumping irrespective of the pump state.

75. Valves may be set to be open or to be closed irrespective of the valve state.

76. In setting or resetting a pump or a valve a desirable property may be lost.

```
value
74  pump_to_pump, pump_to_not_pump: P → N → N
75  valve_to_open, valve_to_close: V → N → N


value
74  pump_to_pump(p)(n) as n′
74    pre p ∈ obs_Us(n)
74    post let p′:P•obs_UI(p)=obs_UI(p′) in
74          obs_PΣ(p′)=pumping∧else_equal(n,n′)(p,p′) end
74  pump_to_not_pump(p)(n) as n′
74    pre p ∈ obs_Us(n)
```

```
74    post let p′:P•obs_UI(p)=obs_UI(p′) in
74          obs_PΣ(p′)=not_pumping∧else_equal(n,n′)(p,p′) end
75  valve_to_open(v)(n) as n′
74    pre v ∈ obs_Us(n)
75    post let v′:V•obs_UI(v)=obs_UI(v′) in
74          obs_VΣ(v′)=open∧else_equal(n,n′)(v,v′) end
75  valve_to_close(v)(n) as n′
74    pre v ∈ obs_Us(n)
75    post let v′:V•obs_UI(v)=obs_UI(v′) in
74          obs_VΣ(v′)=close∧else_equal(n,n′)(v,v′) end
```

```
value
  else_equal: (N×N) → (U×U) → Bool
  else_equal(n,n′)(u,u′) ≡
     obs_UI(u)=obs_UI(u′)
   ∧ u ∈ obs_Us(n)∧u′ ∈ obs_Us(n′)
   ∧ omit_Σ(u)=omit_Σ(u′)
   ∧ obs_Us(n)\{u}=obs_Us(n)\{u′}
   ∧ ∀ u″:U•u″ ∈ obs_Us(n)\{u} ≡ u″ ∈ obs_Us(n′)\{u′}

  omit_Σ: U → U_no_state −−− ″magic″ function

  =: U_no_state × U_no_state → Bool
axiom
  ∀ u,u′:U•omit_Σ(u)=omit_Σ(u′) ≡ obs_UI(u)=obs_UI(u′)
```

Events

- ***Unit Handling Events***

77. Let n be any acyclic net.
77. If there exists p, p', v, v', pairs of distinct pumps and distinct valves of the net,
77. and if there exists a route, r, of length two or more of the net such that
78. all units, u, of the route, except its first and last unit, are pipes, then
79. if the route "spans" between p and p' and the *simple desirable property*, sppr(r), does not hold for the route, then we have a possibly undesirable event — that occurred as soon as sppr(r) did not hold;
80. if the route "spans" between p and v and the *simple desirable property*, spvr(r), does not hold for the route, then we have a possibly undesirable event;

81. if the route "spans" between v and p and the *simple desirable property*, svpr(r), does not hold for the route, then we have a possibly undesirable event; and

82. if the route "spans" between v and v' and the *simple desirable property*, svvr(r), does not hold for the route, then we have a possibly undesirable event.

events:

```
77  ∀ n:N • acyclic(n) ∧
77    ∃ p,p′:P,v,v′:V • {p,p′,v,v′}⊆obs_Us(n)⇒
77      ∧ ∃ r:R • routes(n) ∧
78          ∀ u:U • u ∈ elems(r)\{hd r,r(len r)} ⇒ is_Π(i) ⇒
79            p=hd r∧p′=r(len r) ⇒ ∼sppr_prop(r) ∧
80            p=hd r∧v=r(len r) ⇒ ∼spvr_prop(r) ∧
81            v=hd r∧p=r(len r) ⇒ ∼svpr_prop(r) ∧
82            v=hd r∧v′=r(len r) ⇒ ∼svvr_prop(r)
```

- ***Foreseeable Accident Events***

A number of foreseeable accidents may occur.

83. A unit ceases to function, that is,
    a) a unit is clogged,
    b) a valve does not open or close,
    c) a pump does not pump or stop pumping.
84. A unit gives rise to excessive leakage.
85. A well becomes empty or a sunk becomes full.
86. A unit, or a connected net of units gets on fire.
87. Or a number of other such "accident".

Well-formed Operational Nets

88. A well-formed operational net
89. is a well-formed net
    a) with at least one well, w, and at least one sink, s,
    b) and such that there is a route in the net between w and s.

```
value
  88  wf_OpN: N → Bool
  88  wf_OpN(n) ≡
  89    satisfies axiom 30 on page 51 ∧ acyclic(n): Item 35 on page 51  ∧
  89    wfN_SP(n): Item 49 on page 53 ∧
  89    satisfies flow laws, 68 on page 56 and 69 on page 57 ∧
 89a     ∃ w:W,s:S • {w,s}⊆obs_Us(n) ⇒
 89b       ∃ r:R• ⟨w⟩⌢r⌢⟨s⟩ ∈ routes(n)
```

Orderly Action Sequences

- ***Initial Operational Net***

90. Let us assume a notion of an initial operational net.

91. Its pump and valve units are in the following states

    a) all pumps are not_pumping, and

    b) all valves are closed.

```
value
  90  initial_OpN: N → Bool
  91  initial_OpN(n) ≡ wf_OpN(n) ∧
 91a    ∀ p:P • p ∈ obs_Us(n) ⇒ obs_PΣ(p)=not_pumping ∧
 91b    ∀ v:V • v ∈ obs_Us(n) ⇒ obs_VΣ(p)=closed
```

Oil Pipeline Preparation and Engagement

92. We now wish to prepare a pipeline from some well, $w : W$, to some sink, $s : S$, for flow.

    a) We assume that the underlying net is operational wrt. w and s, that is, that there is a route, r, from w to s.

    b) Now, an orderly action sequence for engaging route r is to "work backwards", from s to w

    c) setting encountered pumps to pumping and valves to open.

In this way the system is well-formed wrt. the desirable sppr, spvr, svpr and svvr properties. Finally, setting the pump adjacent to the (preceding) well starts the system.

```
value
  92   prepare_and_engage: W × S → N ~→ N
  92   prepare_and_engage(w,s)(n) ≡
  92a    let r:R • ⟨w⟩^r^⟨s⟩ ∈ routes(n) in
  92b    action_sequence(⟨w⟩^r^⟨s⟩)(len⟨w⟩^r^⟨s⟩)(n) end
  92     pre ∃ r:R • ⟨w⟩^r^⟨s⟩ ∈ routes(n)

  92c  action_sequence: R → Nat → N → N
  92c  action_sequence(r)(i)(n) ≡
  92c    if i=1 then n else
  92c    case r(i) of
  92c      mkV(_) → action_sequence(r)(i−1)(valve_to_open(r(i))(n)),
  92c      mkP(_) → action_sequence(r)(i−1)(pump_to_pump(r(i))(n)),
  92c      _ → action_sequence(r)(i−1)(n)
  92c    end end
```

Emergency Actions

93. If a unit starts leaking excessive oil
    a) then nearest up-stream valve(s) must be closed,
    b) and any pumps in-between this (these) valves and the leaking unit must be set to not_pumping — following an orderly sequence.
94. If, as a result, for example, of the above remedial actions, any of the desirable properties cease to hold
    a) then — a ha !
    b) Left as an exercise.

Compiled: February 2, 2010: 00:00 ECT
Fredsvej 11, DK-2840 Holte, Denmark; bjorner@gmail.com