Start of Lecture 1: COVER & INTRODUCTION

Lecture Notes in Software Engineering: From Domains to Requirements

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

Begun: Tuesday June 22, 2010. Compiled: November 12, 2010: 11:34

0. Abstract

- We present "standard" domain description and requirements prescription examples using the RAISE [RaiseMethod] Specification Language, RSL [RSL].
- The illustrated example is that of transportation networks.
  - These notes shalll serve as lecture notes for my lectures at Uppsala, Nov.8-19, 2010.
  - The present document is the ordinary "book-form"-like notes.
  - A separate document, compiled from the same files, present 11 sets of lecture slides.
  - The "funny" small numbers you see in the present document, in margins and at almost end of display lines refer to slide page numbers of the slides document.

Lecture Notes

A Tentative Lecture Schedule

1. Introduction
1.1. The Problem

- The problem to be solved by this technical note is to present in one specific formal specification language, `RSL` [RaiseMethod],
  - a domain description and
  - a requirements prescription developed according to the "triptych approach" [TheSEBook3].

1.2. The Triptych Approach

- The "triptych approach" calls for
  - a thorough description (cum analysis) of the domain
  - before one attempts prescribing requirements for specific software.
- As part of the triptych approach to domain engineering one starts by exploring the description ontology of specification entities:
  - *simple entities*,
  - *actions*,
  - *events* and
  - *behaviours*

- before delving into the description ontology of facets:
  - *intrinsics*,
  - *support technologies*,
  - *rules & regulations*,
  - *scripts (licenses and contracts)*,
  - *management & organisation* and
  - *human behaviour*.

- And, as part of the triptych approach to requirements engineering
  - one starts by exploring the reengineering of business processes
  - before delving into *domain requirements* concepts of
    - ∗ *projection*,
    - ∗ *instantiation*,
    - ∗ *determination*,
    - ∗ *extension* and
    - ∗ *fitting* –

    followed by a number of *interface requirements* stages.

End of Lecture 1: COVER & INTRODUCTION

Start of Lecture 2: ONTOLOGY

2. An Ontology of Specification Entities

Definition: Ontology.

- *In philosophy: A systematic account of Existence.*
- *To us:*
  - *An explicit formal specification of how to represent the phenomena and concepts*
  - *that are assumed to exist in some area of interest (some universe of discourse)*
  - *and the relationships that hold among them.*

  Further clarification:
  - *An ontology is a catalogue of* concept*s and their relationships* —
  - *including properties as relationships to other concepts.*

Definition: Specification.

- *We use the term 'specification'*
- *to cover the concepts of* domain descriptions, requirements prescriptions *and* software designs.
- *More specifically a specification is a* definition, *usually consisting of many definitions.*

Definition: Entity. *By an entity we shall understand*

- *either a* simple entity,
- *an* action,
- *an* event
- *or a* behaviour.

2.1. Simple Entities

Definition: Simple Entity. *By a simple entity we shall loosely understand*

- *an individual,* static *or* inert dynamic *and that simple entities "roughly correspond" to what we shall think of as* values.
- *We shall further allow simple entities to be*
  - *either* atomic
  - *or* composite, *i.e., in the latter case having decomposable* sub-entities.

- *Simple entities have* attributes.
- *Composite entities have*
  - attributes,
  - sub-entities *and*
  - *a* mereology, *the latter explains how the sub-entities are formed into the simple entity.*

2.1.1. Net, Hubs and Links

1. There are nets, hubs and links.
2. A net contains zero, one or more hubs.
3. A net contains zero, one or more links.

type
  1. N, H, L
value
  2. obs_Hs: N → H-**set**
  3. obs_Ls: N → L-**set**

2.1.2. Unique Hub and Link Identifiers

4. There are hub identifiers and there are link identifiers.
5. From a hub one can observe its hub identifier.
6. From a link one can observe its link identifier.
7. Hubs of a net have unique hub identifiers.
8. Links of a net have unique hub identifiers.

```
type
  4.  HI, LI
value
  5.  obs_HI: H → HI
  6.  obs_LI: L → LI
axiom
  7.  ∀ n:N, h,h′:H • {h,h′}⊆obs_Hs(n) ∧ h≠h′ ⇒ obs_HI(h)≠obs_HI(h′)
  8.  ∀ n:N, l,l′:L • {l,l′}⊆obs_Ls(n) ∧ l≠l′ ⇒ obs_LI(l)≠obs_LI(l′)
```

2.1.3. Observability of Hub and Link Identifiers

9. From every hub (of a net) we can observe the identifiers of the zero, one or more distinct links (of that net) that the hub is connected to.

```
value
  9.  obs_LIs: H → LI-set
axiom
  9.  ∀ n:N,h:H•h ∈ obs_Hs(n) ⇒ ∀ li:LI•li ∈ obs_LIs(h) ⇒ L_exists(li)(n)
value
   L_exists: LI → N → Bool
   L_exists(li)(n) ≡ ∃ l:L•l ∈ obs_Ls(n)∧obs_LI(l)=li
```

10. From every link (of a net) we can observe the identifiers of the exactly two (distinct) hubs (of that net) that the link is connected to.

```
value
  10.  obs_HIs: L → HI-set
axiom
  10.  ∀ n:N,l:L•l ∈ obs_Ls(n) ⇒
  10.     card obs_HIs(l)=2 ∧ ∀ hi:HI•hi ∈ obs_HIs(l) ⇒ H_exists(hi)(n)
value
   H_exists: HI → N → Bool
   H_exists(hi)(n) ≡ ∃ h:H•h ∈ obs_Hs(n)∧obs_HI(h)=hi
```

2.1.4. A Theorem

2.1.4.1. Links implies Hubs

11. It follows from the above that if a net has at least one link then it has at least two hubs.

```
theorem:
  11.  ∀ n:N • card obs_Ls(n)≥1 ⇒ card obs_Hs(n)≥2
```

2.1.5. Hub and Link Attributes

In preparation for later descriptions, narrative and formal, we make a slight detour to deal with hub and link attributes – but we omit, at present, from describing these attributes.

12. hub and link attributes, HAtrs and LAtrs, include the hub and link identifiers that can be observed from hubs and links, respecively.
13. These can be observed from hubs and links of nets.
14. And these can be provided as arguments when construction hubs and links.

```
type
12.  HAtrs, LAtrs
value
13.  obs_HAtrs: H → HAtrs
14.  obs_LAtrs: L → LAtrs
13.  obs_HI: HAtrs → HI
13.  obs_LIs: HAtrs → LI-set
14.  obs_LI: LAtrs → LI
14.  obs_HIs: LAtrs → HI-set
```

2.1.6. Hub and Link Generators

15. From [a (full) set of] hub attributes
    (a) including an empty set of observable link identifiers

    one can generate a hub with

    (a) the hub identifier being that of the argument hub attributes,
    (b) the link identifiers of the hub being argument the empty set of link identifiers of the hub attributes and
    (c) the argument hub attributes being those of the resulting hub,

```
15.    genH: HAtrs → H
15.    genH(hatrs) as h
15(a).   pre  obs_LIs(hatrs)={}
15(a).   post obs_HI(h)=obs_HI(hatrs)
15(b).      ∧ obs_LIs(h)={}
15(c).      ∧ obs_HAtrs(h)=hatrs
```

16. From the set of hub attributes and a net one can "similarly" generate a hub which is not a hub of the net.
17. From the set of link attributes one can "similarly" generate a link.
18. From the set of link attributes and a net one can "similarly" generate a link which is not a link of the net.

where the reader is to narrate and formalise the "similarities"!

```
16.  genH: HAtrs → N → H
16.  genH(hatrs)(n) as h
16.    pre obs_LIs(hatrs)={}
16.      ∧ ∼∃ h′:H•h′ ∈ obs_Hs(n) ∧ obs_HI(h′)=obs_HI(hatrs)
16.    post  h ∉ obs_Hs(n)
16.      ∧ obs_HI(h)=obs_HI(hatrs)
16.      ∧ obs_LIs(h)={}
16.      ∧ obs_HAtrs(h)=hatrs

17.  genL: LAtrs → L
17.  genL(latrs) as l
17.    pre  card obs_HIs(latrs)=2
17.    post obs_LI(l)=obs_LI(latrs)
17.       ∧ obs_LI(l)=obs_LI(latrs)
17.       ∧ obs_HIs(l)=obs_HIs(latrs)

18.  genL: LAtrs → N → L

18.  genL(latrs)(n) as l
18.    pre  card obs_LIs(latrs)=2
18.       ∧ obs_LIs(latrs)⊆xtr_LIs(n)
18.    post l ∉ obs_Ls(n)
18.       ∧ obs_LI(l)=obs_LI(latrs)
18.       ∧ obs_HIs(l)⊆obs_HIs(latrs)
18.       ∧ obs_LAtrs(l)=latrs
```

2.2. States

Definition: State. *By a state we shall understand*

- *a collection of one or more simple entities.*

2.3. Actions

Definition: Action. *By an action we shall understand*

- *something which potentially changes a* state*,*
- *that is, a function application to a state*
- *which potentially changes that state.*

2.3.1. Insert Hubs

19. One can insert a hub, h, into a net, n.

The hub to be inserted

20. must not be a hub of the net and
21. h cannot already be connected to any links.

    That is, we can only insert “isolated” hubs.

The result of inserting a hub, h, into a net, n, is a new net, n',

22. which is like n except that it now also has the hub h.

value

```
19. insertH: HAtrs → N ~→ N
19. insertH(hatrs)(n) as n′
19. let h = genH(hatrs)(n) in
20. pre   h ∉ obs_Hs(n)
21.         ∧ obs_LIs(h) = {}
22. post obs_Ls(n)=obsLs(n′)
22.       ∧ obs_Hs(n′)=obs_Hs(n)∪{h}
22.       ∧ obs_HAtrs(h)=hatrs
19.       end
```

Theorem:

- Inserting a proper hub in a well-formed net
- that is, a net satisfying all relevant axioms,
- results in a likewise well-formed net.

2.3.2. Remove Hubs

23. One can remove a hub, h, from a net, n.

    The hub to be removed

24. must be a hub of the net and
25. h cannot be connected to any links.

    That is, the hub, h, may earlier – in is membership of the net – have been connected to links, but these must already, at the time of hub removal, have been removed, see below.

    That is, we can only remove “isolated” hubs.

26. The result of removing a hub, h, from a net, n, is a new net, n',
27. which is like n
28. except that it now no longer has hub h.

```
value
 23.  removeH: H → N ~→ N
 26.  removeH(h)(n) as n′
 24.  pre  h ∈ obs_Hs(n)
 25.      ∧ obs_LIs(h) = {}
 27.  post obs_Ls(n)=obsLs(n′)
 28.      ∧ obs_Hs(n′)=obs_Hs(n)\{h}
```

- Please note the almost line-by-line similarity of the insert and remove hub descriptions
- and that the only difference between these descriptions are the
- membership, union, respectively set difference operations (∉, ∈, ∪ respectively \).

2.3.3. Insert Links

29. One can insert a link, ℓ, into a net, n.

The link to be inserted must

30. not be a link of the net,
31. but the observable hub identifiers must be those of hubs of the net.

The result of inserting a link, ℓ, into a net,

32. n, is a new net, n',
33. in which ℓ is now a member.
34. Let h_{j_i}, h_{k_i} be the two (distinct) hub identifiers of ℓ and
35. let h_j, h_k be the two (distinct) hubs of n which are identified by h_{j_i}, h_{k_i}.
36. All hubs of net n except h_j, h_k are the same as in n and are unchanged in n'.
37. The two hubs h_j, h_k of n become hubs h'_j, h'_k of n'
38. such that only the observable identifiers of connected links have changed to now also include the identifier of link ℓ,
39. and such that the observed attributes are those of the argument.

```
value
 29.  insertL: L × LAtrs → N ~→ N
 32.  insertL(l,latrs)(n) as n′
 30.  pre  l ∉ obs_Ls(n)
 31.      ∧ obs_HIs(l)⊆xtrHIs(n)
 33.  post obs_Ls(n′) = obs_Ls(n) ∪ {l}
 34.       ∧ let {hji,hki}=obs_HIs(l) in
 35.          let (hj,hk) = (getH(hji)(n),getH(hki)(n)) in
 31.          {hj,hk}⊆obs_Hs(n)
 36.       ∧ obs_Hs(n)\{hj,hk} = obs_Hs(n′)\{hj,hk}
 37.       ∧ let (hj′,hk′) = (getH(hji)(n′),getH(hki)(n′)) in
 38.          obs_LIs(hk′) = obs_LIs(hk′) ∪ {obs_LI(l)}
 38.       ∧ obs_LIs(hj′) = obs_LIs(hj′) ∪ {obs_LI(l)} end end end
 39.       ∧ obs_LAtrs(l) = latrs
```

```
xtrHIs: N → HI-set
xtrHIs(n) ≡ {obs_HI(h)|h:H·h ∈ obs_Hs(n)}

getH: HI → N ~→ H
getH(hi)(n) ≡ let h:H · h ∈ obs_Hs(n) ∧ obs_HI(h)=hi in h end
  pre ∃ h:H · h ∈ obs_Hs(n) ∧ obs_HI(h)=hi
```

2.3.4. Remove Links

40. One can remove a link, ℓ, from a net, n.

The link to be removed must

41. be a link of the net.

The result of removing a link, ℓ, from a net,

42. n, is a new net, n',
43. in which ℓ is no longer a member.
44. Let h_{j_i}, h_{k_i} be the two (distinct) hub identifiers of ℓ and
45. let h_j, h_k be the two (distinct) hubs of n which are identified by h_{j_i}, h_{k_i}.
46. h_j, h_k are in n'.
47. All hubs of net n except h_j, h_k are the same as in n and are unchanged in n'.
48. The two hubs h_j, h_k of n become hubs h'_j, h'_k of n'
49. such that only the observable identifiers of connected links have changed to now no longer include the identifier of link ℓ.

```
value
 40.  removeL: L → N ~→ N
 42.  removeL(l)(n) as n′
 41.  pre  l ∈ obs_Ls(n)
 43.  post obs_Ls(n′) = obs_Ls(n) \ {l}
 44.     ∧ let {hji,hki}=obs_HIs(l) in
 45.       let (hj,hk) = (getH(hji)(n),getH(hki)(n)) in
 46.       {hj,hk}⊆obs_Hs(n)
 47.     ∧ obs_Hs(n)\{hj,hk} = obs_Hs(n′)\{hj,hk}
 48.     ∧ let (hj′,hk′) = (getH(hji)(n′),getH(hki)(n′)) in
 49.       obs_LIs(hk′) = obs_LIs(hk′) \ {obs_LI(l)}
 49.     ∧ obs_LIs(hj′) = obs_LIs(hj′) \ {obs_LI(l)} end end end
```

2.3.5. Two Theorems

2.3.5.1. Idempotency

- With the preconditions satisfied by the insert and remove actions
- one can prove that first inserting a hub (link) into a net and
- then removing that hub (link) from the resulting net restores the original net:

theorem

$\forall$ n,n′:N,h:H,l:L ·
  pre insertH(h)(n) $\wedge$ removeH(h)(n′) $\wedge$ insertL(l)(n) $\wedge$ removeL(l)(n′) $\Rightarrow$
  removeH(h)(insertH(h)(n)) = n $\wedge$ removeL(l)(insertL(l)(n))

2.3.5.2. Reachability

- Any net that satisfies the axioms above
- can be constructed by sequences of insert hub and link actions.

theorem

let n_nil:N · obs_Hs(n_nil)=obs_Ls(n_nil)={} **in**
$\forall$ n:N $\vdash$ **axioms** 7. and 8 on page 14.; 9 on page 15. 10 on page 16. ·
  $\exists$ hl:H*, ll:L* · **let** n′ = insertHs(hl)(n_nil) **in** insertHs(ll)(n′)=n **end**
end

insertHs: $H^* \to N \xrightarrow{\sim} N$
insertLs: $L^* \to N \xrightarrow{\sim} N$

insertHs(hl)(n) ≡ **case** hl **of** $\langle\rangle \to$ n, $\langle h\rangle\frown$hl′ → insertHs(hl′)(insertH(h)(n)) **end**
insertLs(ll)(n) ≡ **case** ll **of** $\langle\rangle \to$ n, $\langle l\rangle\frown$ll′ → insertLs(ll′)(insertL(l)(n)) **end**

Informal proof: An informal proof goes like this:

- Take a net.
- For every hub, h, in that net,
  - let h' be a version of h which has
    - the same hub identifier,
    - an empty set of observable link identifiers (of connected links),
    - and otherwise all other attributes of h,
  - let h' be a member of the list of hubs – and only such hubs.
  - Let every and only such links in n be members of the list of links.
- Performing first the insertion of all hubs and then the insertions of all links will “turn the trick” !

end of informal proof.

2.4. Events

Definition: Event.

- *An event is something that occurs instantaneously.*
- *Events are manifested by certain state changes, and by certain interactions between behaviours or processes.*
- *The occurrence of events may “trigger” [further] actions.*
- *How the triggering, i.e., the invocation of functions are brought about is usually left implied, or unspecified.*

- A mudslide across a railway track or a road segment (i.e., a link) represents an event
  - that effectively "removes" the link, or at least a segment of a link.
- Similarly if
  - a train and/or automobile bridge collapses or
  - a tunnel gets flooded or catches fire.

How are we to model such, and other events?

50. We choose to model the event" *"disappearance" of a segment of a link* identified by l_i:LI as the composition of the following actions:
    (a) the removal of link l:L being affected, where l_i:LI identifies the link in the network;
    (b) the insertion of two hubs, h',h'':H, corresponding to "points" (on link l:L) on either side of the mudslide or bridge – or other; and
    (c) the insertion of two links, l',l'':L, between the hubs of the original link and the new hubs.
    (d) l_i:LI must identify a link l:L of net n:N.

50(b). newH: N $\to$ H-**set** $\to$ H
50(b). newH(n)(hs) $\equiv$ **let** h:H $\cdot$ h $\notin$ hs $\wedge$ obs_LIs(h)={} **in** h **end**
50(c). newL: N $\to$ L-**set** $\to$ (HI$\times$HI) $\to$ L
50(c). newL(n)(ls)(hi′,hi″) $\equiv$ **let** l:L $\cdot$ l $\notin$ ls $\wedge$ obs_HIs(l)={hi′,hi″} **in** l **enc**

value
50. event_link_disappearance: LI $\to$ N $\xrightarrow{\sim}$ N
50(a). **let** l = xtrL(li)(n) **in**
50(a). **let** {hi′,hi″} = obs_HIs(l) **in**
50(a). **let** n′ = removeL(l)(n) **in**
50(b). **let** h′= newH(n)(obs_Hs(n)) **in**
50(b). **let** h″ = newH(n)(obs_Hs(n)∪{h′}) **in**
50(b). **let** n″ = insertH(h′)(insertH(h″)(n)) **in**
50(c). **let** l′ = newL(n)(obs_Ls(n))(obs_HI(h′),hi′) **in**
50(c). **let** l″ = newL(n)(obs_Ls(n)∪{l′})(obs_HI(h″),hi″) **in**
50(c). insertL(l′)(insertL(l″)(n″)) **end end end end end end end end**
50(d). **pre** li $\in$ xtrLIs(n)

2.5. Behaviours

Definition: Behaviour.

- *By behaviour we shall understand the way in which something functions or operates.*
- *In the context of domain engineering behaviour is a concept associated with phenomena, in particular manifest entities.*
- *And then behaviour is that which can be observed about the value of the entity and its interaction with an environment.*
- *A simple, sequential behaviour is a sequence of zero, one or more actions and events.*

2.5.1. Behaviour Prescriptions

- Usually behaviours follow a prescription.
- In the case of net construction we refer to the prescription as a construction plan.

2.5.1.1. Construction Plans

51. The plan for constructing a net can be abstracted as
    (a) a map, PLAN, which to each hub identifier associates
    (b) a link-to-hub identifier map, LHIM, from the identifiers of links emanating from the hub to identifiers of connected hubs.

type

```
51(a). PLAN = HI →m LHIM
51(b). LHIM = LI →m HI
```

2.5.1.2. Wellformedness of Construction Plans

52. Wellformed net construction plans satisfy three conditions:
    (a) *All Links are Two-way Links:*
        i. Let h_k be any hub identifier of the construction plan.
        ii. For all link identifiers, l_j, of the LIHM, $lhim_k$, mapped into by h_k,
        iii. let h_ℓ be the hub identifier mapped into by l_j in $lhim_k$,
        iv. then l_j is in the link-to-hub-identifier map, $lhim_\ell$, mapped into by h_ℓ,

(b) *Using Hub Identifier Occurrences are Defined:*
    i. Let $lhim$ be any link-to-hub-identifier map of a construction plan.
    ii. For every hub identifier, h_i, mapped to by a link identifier, l_j, in $lhim$
    iii. there exists a hub identifier, h_k, that maps into l_j; and

(c) *No Junk:*
- To secure consistency between hub and link identifiers of a construction plan we impose:
  – all the defined hub identifiers of a construction plan are in the range of some link to hub identifier map of that plan;
  – and each of the hub identifiers of some link to hub identifier map are defined in the construction plan are in the range of some link to hub identifier map of that plan.

value

```
52.    wf_PLAN: PLAN → Bool
52.    wf_PLAN(plan) ≡
52(a).   all_links_are_two_way_links(plan) ∧
52(b).   hub_identifier_occurrences_are_defined(plan) ∧
52(c).   no_junk(plan)
```

```
52(a).      all_links_are_two_way_links: PLAN → **Bool**
52(a).      all_links_are_two_way_links(plan) ≡
52((a))i.       ∀ hk:HI · hk ∈ dom plan ⇒
52((a))ii.          ∀ lj:LI · lj ∈ dom plan(hk) ⇒
52((a))iii.             let hl = (plan(hk))(lj) in
52((a))iv.              lj ∈ dom plan(hl) end

52(b).      hub_identifier_occurrences_are_defined: PLAN → Bool
52(b).      hub_identifier_occurrences_are_defined(plan) ≡
52((b))i.       ∀ hlim:HLIM·hlim ∈ rng plan
52((b))ii.          ∀ lj:LI · lj ∈ dom lhim ⇒
52((b))iii.             ∃ hk:HI · hk ∈ dom plan ∧ lj ∈ dom plan(hk)

52(c).      no_junk: PLAN → Bool
52(c).      no_junk(plan) ≡ dom plan = ∪{rng(plan(hi))|hi:HI·hi ∈ dom plan
```

2.5.2. Augmented Construction Plans

- Hubs and links in nets possess attributes (cf. Item 4 on page 14.).
- Some attributes have already been dealt with:
  - the identifiers of hubs and links that can be observed from hubs, respectively links (cf. Items 4. and 5 on page 14.) and
  - the identifiers of hubs that can be observed from links and the identifiers of links that can be observed from hubs (cf. Items 9. and 10 on page 16.).
- In addition hubs and links in nets possess further attributes:
- spatial location of hubs and links,
- (locally ascribed) names of hubs and links,
- lengths of links,
- etcetera.

We therefore augment construction plans to also reveal these attributes.

type

$\text{APLAN} = \text{PLAN} \times \text{HInfo} \times \text{LInfo}$
$\text{HInfo} = \text{HI} \xrightarrow{m} \text{HAtrs}$
$\text{LInfo} = \text{LI} \xrightarrow{m} \text{LAtrs}$

53. The wellformedness of an augmented plan secures that
    (a) all hubs identifiers defined in the construction plan are “detailed” in the hub information component, and that
    (b) all links identifiers used in the construction plan are “detailed” in the in the link information component.

value

```
53.      wf_APLAN: APLAN → Bool
53.      wf_APLAN(plan,hinfo,linfo) ≡
53(a).       dom plan = dom hinfo ∧
53(b).       ∪{dom lhim|lhim:LHIM·lhim ∈ rang plan}=dom linfo
```

2.5.3. Sequential Construction Behaviours

54. From an augmented construction plan one can “extract” initial information about
    (a) all hubs and
    (b) all links.

```
value
54(a).  xtrH: HI → APLAN → HI × HAtrs, xtrH(hi)(_,hinfo,_) ≡ hinfo(hi)
54(b).  xtrL: LI → APLAN → LAtrs, xtrL(li)(_,_,linfo) ≡ linfo(li)
```

55. A net construction behaviour can be (functionally and non-deterministically)
    modelled as
    (a) a sequence of hub insertions followed by
    (b) a sequence of link insertions.

```
value
55.     net_construction: HInfo×LInfo → (HI-set×LI-set) → N → N
55.     net_construction(hinfo,linfo)(his,lis)(n) ≡
55.        case (his,lis) of
55(a).         ({hi}∪ his′,_) →
55(a).               net_construction(hinfo,linfo)(his′,lis)(insertH(hinfo(hi))(n)),
55(b).         ({},{li}∪ lis′) →
55(b).               net_construction(hinfo,linfo)({},lis′)(insertL(linfo(li))(n)),
55.           ({},{}) → n
55.        end
```

The net_construction function is initialised with the full sets of hub and link identifiers and with an empty net:

```
   net_construction(hinfo,linfo)(dom hinfo,dom linfo)(n_nil)
value
   n_nil:N • obs_Hs(n_nil) = {} = obs_Ls(n_nil)
```

- The net_construction behaviour shown above defines only a subset of all the valid behaviours that will construct a net according to the augmented plan (plan,hinfo,linfo).
- Other valid behaviours would start with constructing at least two hubs but could then go onto construct some of the (zero, one or more) links that connect some of the already constructed hubs, etcetera.
- We challenge the reader to precise narrate and formally define such net_construction behaviours.

End of Lecture 2: ONTOLOGY

Start of Lecture 3: DOMAINS: Intrinsics – Rules & Regulations

3. An Ontology of Domain Facets

3.0.1. Definitions

Definition: Domain. *An area of activity which some software is to support (or supports) or partially or fully automate (resp. automates).*

- The term 'application domain' is considered synonymous with the term 'domain'.

Definition: Domain Description. *A textual, informal or formal document which describes a domain* ***as it is****.*

Usually a domain description is a set of documents with many parts recording many facets of the domain: The

- *business process*es,
- *intrinsics*,
- *support technology*,
- *rules and regulations*,
- *management and organisation*, and the
- *human behaviour*s.

Definition: Domain Engineering.

- *The engineering of the development of a domain description, from*
  - *identification of domain stakeholders, via*
  - *domain acquisition,*
  - *domain analysis,*
  - *terminologisation,*

  and
  - *domain description*

  to
  - *domain validation and*
  - *domain verification.*

Definition: Domain Facet.

- *By a domain facet we understand*
  - *one amongst a finite set of generic ways of analysing a domain:*
  - *A view of the domain, such that the different facets cover conceptually different views,*
  - *and such that these views together cover the domain.*
- We consider here the following domain facets:
  - *business process*es,
  - *intrinsics*,
  - *support technology*,
  - *rules and regulations*,
  - *management and organisation*, and
  - *human behaviour*.

3.0.2. What Can Be Observed

- "Whether you can observe a thing or not depends on the theory which you use. It is the theory which decides what can be observed."
- Albert Einstein objecting to the placing of observables at the heart of the new quantum mechanics, during Heisenberg's 1926 lecture at Berlin; related by Heisenberg, quoted in *Unification of Fundamental Forces* (1990) by Abdus Salam ISBN 0521371406.

3.0.3. Business Processes

3.0.3.1. A Characterisation

- By a business process we shall understand
  - a *behaviour*
  - of an enterprise, a business, an institution, a factory.

3.0.3.2. An Example

- The business processes of transportation evolves around
  - freights or passengers
  - being transported along routes
  - by a vehicle (car, train, aircraft, ship)
  - "propelled" by some locomotive force.

3.1. Intrinsics

Definition: Intrinsics.

- *By the intrinsics of a domain we shall understand*
  - *those phenomena and concepts of a domain*
  - *which are basic to any of the other facets,*
  - *with such a domain intrinsics initially covering at least one stakeholder view.*

3.1.1. Net Topology Descriptors

Instead of dealing with the entire phenomenon of a net, that is, the real, physical, geographic "thing", we can describe essentials of a net, for example how its hub and links are connected.

56. One way of abstractly modelling a net descriptor is as a map, nd, from hub identifiers to simple maps, lihis, from link identifiers to hub identifiers,
57. such that
    (a) for all hi in (the definition set of) nd it is the case that
    (b) if hi maps to lihi,
    (c) and in that link identifier to hub identifier map, li maps to hi′,
    (d) then hi′ is different from hi and
    (e) hi′ maps to an lihi′ in which li is defined and maps to hi.
    (f) And there are only such pairings.

```
type
56.    ND′ = HI →m (LI →m HI)
56.    ND = {|nd′:ND′•wf_ND(nd′)|}
value
57.    wf_ND: ND′ → Bool
57.    wf_ND(nd) ≡
57(a).      ∀ hi:HI•hi ∈ dom nd ⇒
57(b).        let lihi = nd(hi) in
57(c).        ∀ li:LI · li ∈ dom lihi ⇒
57(c).          let hi′ = (nd(hi))(li) in
57(d).          hi ≠ hi′ ∧
57(e).          hi′ ∈ dom nd ∧ li ∈ dom(nd(hi′)) ∧ hi=(nd(hi′))(li)
57(f).        end end
```

From a net one can construct its net descriptor:

```
value
  conND: N → ND
  conND(n) ≡
    [ hi↦[ li↦hi′|li:LI,hi′:HI•li ∈ obs_LIs(getH(hi,n))∧{hi,hi′}=obs_HIs(getL(li,n)) ]|
      hi:HI•hi ∈ xtrHIs(n) ]
```

3.1.2. Link States and Link State Spaces

- We introduce the notions of
  - the state of a link,
  - the state of a hub,
  - the state space of a link and
  - the state space of a hub.
- States abstract directions of movement.
- Links are, by our previous definitions, bi-directional:
  - from one of the connected hubs to the other,
  - and vice versa.
- And hubs are multi-directional:
  - from potentially any link via the hub to potentially any link.

- Let
  - the observed hub identifiers of a link ℓ be $\{h_j, h_k\}$,
  - then link ℓ can potentially be in any one of the four link states:
  - $\{\{(h_j, h_k), (h_k, h_j)\}, \{(h_j, h_k)\}, \{(h_k, h_j)\}$ and $\{\}\}$.
- Any one particular link may
  - always remain in one and the same state,
  - or it may from time to time undergo transitions between any subset of the potential link state space.

58. Link states, $l\sigma$:$L\Sigma$, are set of pairs of hub identifiers.
59. Link state spaces are set of link states.
60. From a link one can generate the link state space of all potential link states.
61. From a link one can observe the current link state $l\sigma$:$L\Sigma$.
62. From a link one can observe the link state space $l\omega$:$L\Omega$.

```
type
58.  LΣ = (HI×HI)-set
59.  LΩ = LΣ-set
value
60.  generate_full_LΣ: L → LΣ
60.  generate_full_LΣ(l) ≡
60.      {}∪{(hi′,hi″)|hi′,hi″:HI•hi′≠hi″∧{hi′,hi″}=obs_HIs(l)}

60.  generate_LΩ: L → LΩ
60.      let fullLσ = generate_full_LΣ(l) in
60.      {{},∪{σ|σ:LΣ•σ⊆fullLσ}} end

61.  obs_LΣ: L → LΣ
62.  obs_LΩ: L → LΣ-set
```

3.1.3. Hub States and Hub State Spaces

63. Hub states, $h\sigma$:$H\Sigma$, are sets of pairs of link identifiers $((l_i, l_k))$, designating that if (l_i, l_k) is in the current hub state then movement can take place from the link designated by l_i (via hub h) to the link designated by l_k.
64. Hub state spaces are set of hub states.
65. From a hub one can generate the hub state space of all potential hub states.
66. From a hub one can observe the current hub state $h\sigma$:$H\Sigma$.
67. From a hub one can observe the hub state space $h\omega$:$H\Omega$.

type

63. $H\Sigma = (LI \times LI)$**-set**
64. $H\Omega = H\Sigma$**-set**

value

65. $\text{generate\_full\_H}\Sigma\text{: H} \rightarrow \text{H}\Sigma$
65. $\text{generate\_full\_H}\Sigma\text{(h)} \equiv$
65. $\{\}\cup\{(li',li'')|li',li'':LI\cdot\{li',li''\}\subseteq obs\_LIs(h)\}$

60. $\text{generate\_H}\Omega\text{: H} \rightarrow \text{H}\Omega$
60. **let** $\text{fullH}\sigma = \text{generate\_full\_H}\Sigma\text{(h)}$ **in**
60. $\{\{\}\cup\{\sigma|\sigma:H\Sigma\cdot\sigma\subseteq \text{fullH}\sigma\}\}$ **end**

66. $\text{obs\_H}\Sigma\text{: H} \rightarrow \text{H}\Sigma$
66. $\text{obs\_H}\Omega\text{: H} \rightarrow \text{H}\Sigma$**-set**

3.1.4. State and State Space Wellformedness

68. States must be in appropriate state spaces.
69. State spaces must be subsets of all potential appropriate states.

axiom

$\forall$ n:N,l:L,h:H $\cdot$ l $\in$ obs_Ls(n) $\wedge$ h $\in$ obs_Hs(n) $\Rightarrow$

58. $\text{obs\_L}\Sigma\text{(l)} \in \text{obs\_L}\Omega\text{(l)} \wedge$
59. $\text{obs\_L}\Omega\text{(l)} \subseteq \text{generate\_full\_L}\Sigma\text{(l)} \wedge$
58. $\text{obs\_H}\Sigma\text{(h)} \in \text{obs\_H}\Omega\text{(h)} \wedge$
59. $\text{obs\_H}\Omega\text{(h)} \subseteq \text{generate\_full\_H}\Sigma\text{(h)}$

theorems:

$\forall$ n:N,l:L,h:H $\cdot$ l $\in$ obs_Ls(n) $\wedge$ h $\in$ obs_Hs(n) $\Rightarrow$

$\text{obs\_L}\Sigma\text{(l)} \subseteq \{(hi',hi'')|hi',hi'':H\cdot\{hi',hi''\}\subseteq obs\_HIs(l)\} \wedge$

$\text{obs\_H}\Sigma\text{(h)} \subseteq \{(li',li'')|li',li'':L\cdot\{li',li''\}\subseteq obs\_LIs(h)\}$

3.1.5. Concrete Types for Simple Entities

- As an alternative for, or as a step of refinement from the earlier sorts of nets, hubs and links
- one can simplify matters by concrete types for these simple entities.

70. Nets are Cartesians of sets of hubs and links.
71. A link is a Cartesian of a link identifier, a set of exactly two hub identifiers, a link state, a link state space, and a number of presently further unspecified link attributes.
72. A hub is a Cartesian of a hub identifier, a set of zero, one or more link identifiers, a hub state, a hub state space, and a number of presently further unspecified hub attributes.

type

70. N = H**-set** $\times$ L**-set**
71. L :: obs_LI:LI $\times$ obs_HIs:HI**-set** $\times$ LΣ $\times$ LΩ $\times$ LAtrs
72. H :: obs_HI:HI $\times$ obs_LIs:LI**-set** $\times$ HΣ $\times$ HΩ $\times$ HAtrs

We leave it to the reader to narrate the wellformedness constraints.

```
axiom
  $\forall$ (hs,ls):N $\cdot$ ls$\neq${} $\Rightarrow$ card hs $\geq$ 2 $\wedge$
    $\forall$ l′,l″:L $\cdot$ {l′,l″}$\subseteq$ls $\wedge$ l′$\neq$l″ $\Rightarrow$ obs_LI(l′)$\neq$obs_LI(l″) $\wedge$
    $\forall$ h′,h″:H $\cdot$ {h′,h″}$\subseteq$hs $\wedge$ h′$\neq$h″ $\Rightarrow$ obs_HI(h′)$\neq$obs_HI(h″) $\wedge$
    $\forall$ l:(li,his,l$\sigma$,l$\omega$,latrs):L $\cdot$ l $\in$ ls $\Rightarrow$
      card his=2 $\wedge$ his$\subseteq${obs_HI(h‴)|h‴:H $\cdot$ h‴ $\in$ hs} $\wedge$
      l$\sigma$ $\in$ generate_full_L$\Sigma$(l) $\wedge$
      l$\sigma$ $\in$ l$\omega$ $\subseteq$ generate_full_L$\Sigma$(l) $\wedge$
    $\forall$ h:(hi,lis,h$\sigma$,h$\omega$,hatrs):H $\cdot$ h $\in$ hs $\Rightarrow$
      lis$\subseteq${obs_LI(l‴)|l‴:L $\cdot$ l‴ $\in$ ls} $\wedge$
      h$\sigma$ $\in$ generate_full_H$\Sigma$(h) $\wedge$
      h$\sigma$ $\in$ h$\omega$ $\subseteq$ generate_full_H$\Sigma$(h)
```

3.1.6. Example Hub Crossings

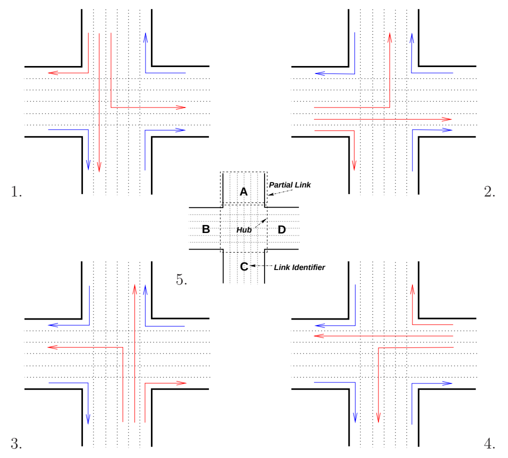


Figure 1: Four "Safe" Flows

The top left hub/link diagram (1.) thus can be claimed to depict hub state $\{(A,B), (A,C), (A,D), (B,C), (C,D), (D,A)\}$.

Photo 2 shows a semaphore which seems to be able to display all kinds of states.

Figure 2: A General Purpose Traffic Light

The point of this example is to show that a hub may take on many states, that not all hub states may be desirable (viz., lead to crossing traffic if so interpreted), and that to reach from one hub state to another one must change the state.

3.1.7. Actions Continued

73. The action change_HΣ takes a hub, h, in some state, and a desired next state, $h\sigma'$, and results in a hub, h', which
   (a) has the same hub identifier as h,
   is connected to the same links as h,
   has the same hub state space as h,
   has the same attributes (names and values) as h,
   (b) but whose state may have changed.

73(b). The new state of h' ought be $h\sigma'$, but electro-mechanical or other failures in setting the state may set the new state to any state of the potential states of h (i.e., h'), not just to any state in the hub state space of h.

```
value
73.    change_HΣ: H × HΣ → H
73.    change_HΣ((hi,lis,hσ,hω,hatrs),hσ′) ≡
73(b).      let hσ′′′ ∈ generate_full_HΣs in
73(a).      (hi,lis,hσ′′′,hω,hatrs) end
```

- Had we specified that the resulting state must be $h\sigma'$
- then we had prescribed a requirements to a change operation.
- As it is now we have described a domain phenomenon, namely that operations may fail.

3.2. Support Technologies

Definition: Support Technology. *By a support technology we understand*

- *a facet of a domain,*
- *one which reflects its (current) dependency on*
  - *human,*
  - *mechanical,*
  - *electro-mechanical,*
  - *electronic and/or*
  - *other technologies*

(i.e., tools) in order to carry out its business processes.

3.2.1. Traffic Signals

A traffic signal represents a technology in support of visualising hub states and in effecting state changes.

74. A hub state is now modelled as a triple: the link identifier l_i ("coming from"), a colour (**red**, **yellow**, and **green**), and another the link identifier l_j ("going to").
75. Signalling is now a sequence of one or more pairs of next hub states and time intervals:

$$< (h\sigma_1, ti_1), (h\sigma_2, ti_2), ..., (h\sigma_{n-1}, ti_{n-1}), (h\sigma_n, ti_n) >, n > 0$$

- The idea of a signalling is
  - to first change the designated hub to state $h\sigma_1$,
  - then wait ti_1 time units,
  - then set the designated hub to state $h\sigma_2$,
  - then wait ti_2 time units,
  - etcetera, ending with final state σ_n
  - and a (supposedly) long time interval ti_n
  - before any decisions are to be made as to another signalling.
- The set of hub states $\{h\sigma_1, h\sigma_2, ..., h\sigma_{n-1}\}$ of

  $$< (h\sigma_1, ti_1), (h\sigma_2, ti_2), ..., (h\sigma_{n-1}, ti_{n-1}), (h\sigma_n, ti_n) >, n > 0$$

  are called intermediate states.
- Their purpose is to secure an orderly vehicle-wise safe signal transitions from **red** to **green** etc.

76. A street signal (a semaphore) is now abstracted as a map from pairs of hub states to signalling sequences.

    The idea is that given a hub one can observe its semaphore, and given the state, $h\sigma$ (not in the above set), of the hub “to be signalled” and the state $h\sigma_n$ into which that hub is to be signalled “one looks up” under that pair in the semaphore and obtains the desired signalling.

```
type
74.  HΣ = LI × Colour × LI
74.  Colour == red | yellow | green
75.  Signalling = (HΣ × TI)*
75.  TI
76.  Sempahore = (HΣ×HΣ) →m Signalling
value
76.  obs_Semaphore: H → Sempahore
```

77. A hub semaphore, **sema**, contains only such hub states as are observed in the **hub state space**.
    (a) Let **hsps** be the set of “from/to” hub state pairs in **sema**.
    (b) Then **hs** is the set of all hub states mentioned in **hsps**.
    (c) To **hs** join all the hub states mentioned in any signalling, **sg**, of **sema**.

```
77.    hub_state_space: Sempahore → HΣ-set
77.    hub_state_space(sema) ≡
77(a).     let hsps={hsp|hsp:(HΣ×HΣ)•hsp ∈ dom sema} in
77(b).     let hs={hσ′,hσ″|hσ′,hσ″:HΣ•(hσ′,hσ″)∈ hsps} in
77(c).     hs ∪ ∪{{hσ|(hσ,ti):(HΣ×TI)•(hσ,ti)∈ elems sg}|sg:Signalling•sg ∈
77.        end end
axiom
77.    ∀ h:H • ∪ obs_HΩ(h) = hub_state_space(obs_Semaphore(h))
```

3.2.2. Traffic “Control”

78. Given two hub states, $h\sigma_{\rm init}$ and $h\sigma_{\rm end}$, where $h\sigma_{\rm init}$ designates a present hub state and $h\sigma_{\rm end}$ designates a desired next hub state after signalling.

79. Now **signalling** is a sequence of one or more successful hub state changes.

```
value
78.  signalling: HΣ × HΣ → H → H
79.  signalling(hσ_init,hσ_end)(h) ≡
79.      let sema = obs_Semaphore(h) in
79.      let sg = sema(hσ_init,hσ_end) in
79.      signal_sequence(sg)(h) end end
79.  pre (hσ_init,hσ_end) ∈ dom obs_Semaphore(h)
```

```
79.  signal_sequence(⟨⟩)(h) ≡ h
79.  signal_sequence(⟨(hσ,ti)⟩^sg)(h) ≡
79.      let hσ′ = change_HΣ(h)(hσ) in
79.      if hσ′ ≠ hσ then chaos
79.      else wait(ti); signal_sequence(sg)(h) end end
```

- If a desired hub state change fails (**chaos**) then we do not define the outcome of signalling.

3.3. Rules and Regulations

Definition: Rule. *A rule stipulates a regulating principle.*

- *In the context of modelling domain rules we shall understand a domain rule*
  - *as some text*
  - *whose meaning is a predicate*
  - *over a pair of suitably chosen domain states.*
- *We may assume that*
  - *a domain action or a domain event*
  - *takes place in the first of these states and*
  - *results in the second of these states.*
- *If the predicate is true*
  - *then we say that the rule has been obeyed,*

  - *otherwise that it has been violated.*

Usually a domain rule is paired with a possibly remedying regulation.

Definition: Regulation.

- *A regulation stipulates that*
  - *an action be taken*
  - *in order to remedy a previous action which violated a rule.*
- *That is,*
  - *a regulation is some text*
  - *which designates a possibly composite action,*
  - *that is, a state-to-state change*
  - *which ostensibly results in a state*
  - *in which the rule, "attached" to the regulation, now holds.*

3.3.1. Vehicles

80. Vehicles are further undefined quantities except that
    (a) vehicles have unique identifiers,
    (b) vehicles are either positioned
        i. at/in hubs
        ii. or on links, in some fractional (non-zero) distance from a hub toward the connecting hub.
81. From a net (sort) one can observe all the vehicles of the net.[1]
82. No two vehicles so observed have the same identifier.

[1]Thus a concrete net type, in addition to hubs and links (now) also contains vehicles.

```
type
80.        V
80(a).     VI
80(b).     VP   = HP | LP
80((b))i.  HP == atH(hi:HI)
80((b))ii.  LP == onL(li:LI,fhi:HI,f:F,thi:HI)
80((b))ii.  F = {|f:F•0<f<1|}
value
80(a).      obs_VI: V → VI
80(b).      obs_VP: V → VP
81.         obs_Vs: N → V-set
axiom
82.    ∀ v:V • v ∈ obs_Vs(n) ⇒
82.        ∃ onL(li,fhi,f,thi):VP • onL(li,fhi,f,thi)=obs_VP(v) ⇒
82.           ∃ l:L•l ∈ obs_Ls(n)∧li=obs_LI(l)∧{fhi,thi}=obs_HIs(l) ∨
82.        ∃ atH(hi):VP • atH(hi)=obs_VP(v) ⇒
82.           ∃ h:H•h ∈ obs_Hs(n)∧hi=obs_HI(h)
```

3.3.2. Traffic

83. By traffic we understand a continuous function from time to a pair of nets and position of vehicles.
84. By time we understand a dense set of points with dense and points being mathematical concepts [wayne.d.blizard.90,J.van.Benthem.Logic.Time91].

```
type
83.   TF = T → (sel_net:N × sel_veh_pos:(V ⇸ VP))
84.   T
```

3.3.2.1. Wellformedness of Traffic

- Expressing the wellformedness of traffic is not a simple matter.
- We shall approach this task in a number of “small steps”.

3.3.2.1.1. • *Static Wellformedness*•

85. We define a predicate over vehicle positions.
   (a) Every vehicle in the traffic has a proper position on the net, either at a hub or along a link.
   (b) No two vehicles of the traffic can occupy exactly the same link position. (That is, the link positions onL(li,hi,f,hi′) and onL(li,hi,f′,hi′) must have the two fractions (f, f') differ – be it ever so “minutely”).

We first define two auxiliary functions:[2]

```
value
  obs_HIs: N → HI-set
  obs_HIs(n) ≡ {obs_HI(h)|h:H•h ∈ obs_Hs(n)}
  obs_LIs: N → LI-set
  obs_LIs(n) ≡ {obs_LI(h)|l:L•l ∈ obs_Ls(n)}
```

[2]They really ought to have been defined much earlier!

```
85.     proper_vehicle_positions: TF → Bool
85.     proper_vehicle_positions(tf) ≡
85.        ∀ t:T • t ∈ DOMAIN tf •
85.           let (n,vps) = tf(t) in
85(a).           ∀ v:V•v ∈ dom vp•is_net_position(vps(v))(n)
85(b).           ∀ v′:V•v′ ∈ dom vp ∧ v≠v′⇒diff_net_pos(vps(v),vps(v′))
85.           end
85(a).  is_net_position: VP → N → Bool
85(a).  is_net_position(vp)(n) ≡
85(a).     case vp of
85(a).        atH(hi) → hi ∈ obs_HIs(n),
85(a).        onL(li,fhi,f,thi) → li ∈ obs_LIs(n)∧{fhi,thi}⊆obs_HIs(n)
85(a).     end
85(b).  diff_net_pos: VP × VP → Bool
85(b).  diff_net_pos(vp,vp′) ≡
85(b).     case (vp,vp′) of
85(b).        (atH(hi),atH(hi)) → true,
85(b).        (onL(li,fhi,f,thi),onL(li,fhi,f′,thi)) → f≠f′,
85(b).        _ → true
85(b).     end
```

3.3.2.1.2. • *Dynamic Wellformedness*•

86. Vehicles, when moving, move monotonically, that is,

(a) if a vehicle, at some time, t, is at a link position onL(li,hi,f,hi′) where f is not infinitesimally close to 1, then that vehicle will, at some later time t', infinitesimally close to t, be at link position onL(li,hi,f′,hi′) where f' is infinitesimally close to f;

(b) if the vehicle, at some time, t, is at a link position onL(li,hi,f,hi′) where f is indeed infinitesimally close to 1, then that vehicle will, at some infinitesimally later time t', be at hub position atH(hi′);

(c) and if the vehicle, at some time, t, is at a hub position atHP(hi) then the vehicle will at some infinitesimally later time t' either be at hub position atHP(hi) or at some link position onL(li,hi,f,hi′) where f is infinitesimally close to 0.

```
value
86.    monotonic: TF → Bool
86.    monotonic(tf) ≡
86.      ∀ t,t′:T • {t,t′}⊆DOMAIN tf •
86.        let (n,vps) = tf(t),(n′,vps′)=tf(t′) in
86.        INFINITESIMALLY CLOSE (t,t′)∧t<t′⇒
86.          ∀ v:V•v ∈ dom vps ∩ dom vps′ •
86.            case (vps(v),vps′(v)) of
86(a).             (onL(li,fhi,f,thi),onL(li,fhi,f′,thi)) →
86(a).                 f<f′ ∧ INFINITESIMALLY CLOSE (f,f′),
86(b).             (onL(li,fhi,f,thi),atH(thi)) →
86(b).                 INFINITESIMALLY CLOSE (f,1),
86(c).             (atH(hi),atH(hi)) → true,
86(c).             (atH(hi),onL(li,hi,f,thi)) →
86(c).                 INFINITESIMALLY CLOSE (0,f),
86.                _ → true
86.        end   end
```

87. If a vehicle is (has been) moving along a link l_i and is now,

- at time t, at position onL(l_i, h_j, f, h_k), that is, moving from h_j to h_k,
- then it cannot at a subsequent, infinitesimally close time, t', be at a position
- onL(l_i, h_k, f', h_j), that is, moving in the opposite direction, h_k to h_j.

```
value
87.   God_does_not_play_dice[3]: TF → Bool
87.   God_does_not_play_dice(tf) ≡
87.     ∀ t,t′:T • {t,t′}⊆DOMAIN tf ∧ t<t′ ∧ INFINITESIMALLY CLOSE (t,t′)⇒
87.       let (n,vps) = tf(t),(n′,vps′)=tf(t′) in
87.       ∀ v:V • v ∈ dom vps ∩ dom vps′ ⇒
87.         case (vps(v),vps′(v)) of
87.           (onL(li,fhi,_,thi),onL(li,thi,_,fhi))→false,
87.           _ → true
87.       end end
```

[3]Albert Einstein: "I, at any rate, am convinced that He does not throw dice." Letter to Max Born (4 December 1926); The Born-Einstein Letters (translated by Irene Born) (Walker and Company, New York, 1971) ISBN 0-8027-0326-7. Reflects Einstein's view of Quantum Mechanics at the time.

88. If a vehicle is (has been) moving along and has,
    - at time t, been at some position p, and
    - at time t', later than t, is at some position p',
    - then it must at all times t'' between t and t' have been somewhere on the net.

```
value
88.  no_ghost_vehicles: TF → Bool
88.  no_ghost_vehicles(tf) ≡
88.     ∀ t,t′:T · {t,t′}⊆DOMAIN tf ∧ t<t′ ⇒
88.        let (n,vps) = tf(t),(n′,vps′)=tf(t′) in
88.        ∀ v:V·v ∈ dom vps ∩ dom vps′ ⇒
88.           ∀ t″:T · t<t″<t′ ⇒
88.              let (n″,vps″) = tf(t″) in v ∈ dom vps″ end
88.        end
```

3.3.3. Traffic Rules (I of II)

89. A vehicle must not move from a hub, h_i, into a link ℓ (from hub (identified by) h_i to hub (identified by) h_j) which is closed in direction (h_i, h_j), that is, where (h_i, h_j) is not in the current state of link.

```
rule:
89.  ∀ tf:TF,t:T · t ∈ DOMAIN(tf) ⇒
89.     let (n,tp) = tf(t) in
89.     ∀ v:V · v ∈ dom tp ⇒
89.        case tp(v) of
89.           atH(hi) →
89.              let t′:T · t′>t ∧ t′ ∈ DOMAIN(tr′) ∧ INFINITESIMALLY_CLOSE(t,t′) in
89.              let (n′,tp′) = tf(t′) in
89.              ∃ li:LI,hi′:HI,f:F,hi″:HI ·
89.                 hi′=hi ∧ INFINITIEIMALLY_CLOSE(f,0) ∧
89.                 tp′(v) = onL(li,hi′,f′,hi″) ∧(hi,hi″) ∉ obs_LΣ(getL(li,n′))
89.           _ → ...
89.     end end end end
```

3.3.4. Another Traffic Regulator

- We present an abstraction of a more conventional traffic signal than modelled in Items 74 on page 78 to 77 on page 81.

90. A traffic signal now simply shows an entry permit: either **red**, **yellow** or **green** at the hub when "leaving" any link, i.e., at the entry to a hub from any link.

```
type
90.  EP == red | yellow | green
90.  HΣ = LI →ₘ EP
axiom
90.  ∀ h:H · obs_LIs(h)=dom obs_HΣ(h)
```

- We leave it to the reader to express a constraint over hub state spaces as to how there must be hub states such that entry from any link is possible.

3.3.5. Traffic Rules (II of II)

91. Vehicles must not enter a hub if entry permission is not **green**.

```
rule:
91.  ∀ tf:TF,t:T : t ∈ DOMAIN(tf) ⇒
91.     let (n,vps) = tf(t) in
91.     ∀ v:V · v ∈ dom vps ⇒
91.        case vps(v) of
91.           onL(li,hi,f,hi′) →
91.              INFINITESIMALLY_CLOSE(f,1) ∧
91.              let hσ = obs_HΣ(getH(hi′,n)),
91.                 t′:T · t′>t ∧ INFINITESIMALLY_CLOSE(t,t′) in
91.              let (n′,vps′) = vps(t′) in
91.              hσ(li) ≠ green ∧ vps′(v) ≠ atH(hi′)  assert: vps′(v) = onL(li,hi,f,hi′)
91.              end end
91.           _ → ...
91.     end end
```

End of Lecture 3: DOMAINS: Intrinsics – Rules & Regulations

Start of Lecture 4: DOMAINS: Scripts – Human Behaviour

3.4. Scripts

Definition: Scripts.

- *A script is plan of action.*
- *By a domain script we shall, more specifically, understand*
  - *the structured, almost, if not outright,*
  - *formally expressed, wording of a set of*
  - *rules and regulations.*
- See also
  - *license* and
  - *contract*.

  Definitions follow.

3.4.1. Routes as Scripts

3.4.1.1. Paths

92. A path is a triple:
    (a) a hub identifier, h_i, a link identifier, l_j, and another hub identifier, h_k, distinct from h_i,
    (b) such that there is a link ℓ with identifier l_j in a net n such that $\{h_i, h_k\}$ are the hub identifiers that can be observed from ℓ.

type
92. Pth = HI × LI × HI
axiom
92(a). ∀ (hi,li,hi′):Pth · ∃ n:N,l:L · l ∈ obs_Ls(n) ⇒
92(b). obs_LI(l)=li ∧ obs_HIs(l)={hi,hi′}

93. From a net one can extract all its paths:
    (a) if l is a link of the net,
    (b) l_j its identifier,
    (c) $\{h_i, h_k\}$ the identifiers of its connected hubs,
    (d) then (h_i, l_j, h_k) and (h_k, l_j, h_j) are paths of the net.

```
value
93.    paths: N → Pth-set
93(a).   paths(n) ≡
93(d).      {(hi,lj,hk),(hk,lj,hi)|l:L,lj:LI,hi,hk:HI·l ∈ obs_Ls(n) ∧
93(b).                              lj=obs_LI(l) ∧
93(c).                              {hi,hk}=obs_HIs(l)}
```

94. From a net descriptor one can (likewise) extract all its paths:
    (a) Let h_i, h_k be any two distinct hub identifiers of the net descriptor (definition set),
    (b) such that they both map into a link identifier l_j,
    (c) then (h_i, l_j, h_k) and (h_k, l_j, h_j) are paths of the net.

```
value
93.    paths: ND → Pth-set
93.    paths(nd) ≡
94(a).     {(hi,lj,hk),(hk,lj,hi)|hi,hk:HI,lj:LI · hi≠hk ∧ {hi,hk}⊆dom nd ⇒
94(b).                               lj ∈ dom nd(hi)∩ dom nd(hk)}
```

3.4.1.2. Routes

95. A route of a net is a sequence of zero, one or more paths such that
    (a) all paths of a route are paths of the net and
    (b) adjacent paths in the sequence “share” hub identifiers.

```
type
95.    R = Pth*
axiom
95.    ∀ r:R, ∃ n:N ·
95(a).    elems r ⊆ paths(n) ∧
95(b).    ∀ i:Nat · {i,i+1}⊆inds r ⇒
95(b).       let (_,_,hi)=r(i), (hi′,_,_)=r(i+1) in hi=hi′ end
```

96. From a net, n, we can generate the possibly infinite set of finite and possibly infinite routes:
    (a) $<>$ is a path (***basis clause 1***);
    (b) if p is a path of n then $< p >$ is a path of n (***basis clause 2***);
    (c) if r and r' are non-empty routes of n
        i. and the last h_i of r is the same as the first h_j of r'
        ii. then the concatenation of r and r' is a route (***induction clause***).
    (d) Only such routes which can be formed by a (finite, respectively infinite) application of basis clauses Items 96(a) and 96(b) and induction clause Item 96(c) are routes (***extremal clause***).

```
value
96.       routes: N|ND → R-infset
96.       routes(nond) ≡
96(a).      let rs = {⟨⟩} ∪
96(b).               {⟨p⟩|p:Pth·p ∈ paths(nond)} ∪
96((c))ii.             {r⌢r′|r,r′:R · r ∈ rs ∧ r′ ∈ rs ∧
96((c))i.                  ∃ hi,hi′,hi″,hi‴:H,li:LI ·
96((c))i.                  r=r″⌢⟨(hi,li,hi′)⟩∧r′=⟨(hi″,li′,hi‴)⟩⌢r‴ ∧
96((c))i.                  hi′=hi″} in
96(d).      rs end
```

3.4.2. Bus Timetables as Scripts

3.4.2.1. Buses

97. Buses are vehicles,
98. with bus identifiers being the same as vehicle identifiers.

```
type
97.   B
98.   BI ⊆ VI
```

3.4.2.2. Bus Stops

99. A link bus stop indicates the link (by its identifier), the from and to hub identifiers, and the fraction "down the link" from the from to the to hub identifiers.

```
type
99.   BS = mkL_BS(sel_fhi:HI,sel_li:LI,sel_f:F,sel_thi:HI)
```

3.4.2.3. Bus Routes

100. A bus stop list is a sequence of two or more bus stops, *bsl*.
101. A bus route, *br*, is a pair of a net route, *r*, and a bus stop list , *bsl*, such that route *r* is a route of *n* and such that *bsl* is embedded in *r*. If
    (a) there exists an index list, *il*, of ascending indices of the route *r* and of the length of *bsl*
    (b) such that the *i*th path of *r*
    (c) share from and to hub identifiers and link identifier with the *il*(*i*)th bus stop of *bsl*

    then *bsl* is embedded in *r*.
102. We must allow for two or more stops along a bus route to be adjacent on the same link — in which case the corresponding fractions must likewise be ascending.

```
value
  n:N
type
100.      BSL = BS*
101.      BR = {|(r,bsl):(R×BSL)•wf_BR(r,bsl)|}
value
101.      wf_BR: BR → Bool
101.      wf_BR(r,bsl) ≡ ∃ n:N,r:R•r ∈ routes(n) ∧ is_embedded_in(r,bsl)

101(a).   is_embedded_in: BR → Bool
101(a).   is_embedded_in(r,bsl) ≡
101(b).     ∃ il:Nat* · len il=len bsl∧inds il⊆inds r∧ascending(il) ⇒
101(c).       ∀ i:Nat · i ∈ inds il ⇒
101(c).         let (hi,lj,hk) = r(il(i)),(hi′,lj′,f,hk′) = bsl(i) in
101(c).         hi=hi′ ∧ lj=lj′ ∧ hk=hk′ end ∧
102.        ∀ i:Nat · {i,i+1}⊆inds il ⇒
102.          let (hi,lj,f,hk)=bsl(i),(hi′,lj′,f′,hk′)=bsl(i+1) in
102.          hi=hi′ ∧ lj=lj′ ∧ hk=hk′ ⇒ f<f′ end

    ascending: Nat* → Bool, ascending(il) ≡ ∀ i:Nat•{i,i+1}⊆inds il ⇒ il(i)≤il(i+1)
```

3.4.2.4. Bus Schedule

103. A timed bus stop is a pair of a time and a bus stop.
104. A timed bus stop list is a sequence of timed bus stops.
105. A bus schedule is a pair of a route and a timed bus stop list such that
    - there is a net of which the routes is indeed a route,
    - the bus stop list of the timed bus stop list is embedded in the route, and
    - 'later" listed bus stops register later times.
106. SimpleBusSchedules remove routes from BusRoutes.

```
type
103.   TBS :: sel_T:T  sel_bs:BS
104.   TBSL = TBS*
105.   BusSched = {|(r,tbsl):(R×TBSL)•wf_BusSched(r,tbsl)|}
value
105.   wf_BusSched: BusSched → Bool
105.   wf_BusSched(r,tbsl) ≡
105.     ∃ n:N•r ∈ routes(n)
105.     ∧ let bsl:SBS = ⟨sel_BS(tbsl(i))|i:[1..len tbsl]⟩ in is_embedded_in(r,bsl) end
105.     ∧ ∀ i:Nat•{i,i+1}⊆inds tbsl ⇒ sel_T(tbsl(i))<sel_T(tbsl(i+1))
type
106.  SBS = {|bsl:BS*•∃ n:N,r:R•r ∈ routes(n)∧is_embedded_in(r,bsl)|}
```

3.4.2.5. Timetable

The concept of a bus line captures all those bus schedules which ply the same bus route but at different times. A timetable is made up from distinctly named bus lines.

107. A bus line has a unique bus line name.
108. We say that two bus schedules are the same if they are based on the same route and if they differ only in their times.
109. Each of the different bus routes of a bus line has a unique bus number.
110. A route bus schedule pairs a route with simple bus schedules for each of a number of busses (identified by their bus number).
111. A bus timetable (listing, map) maps bus line names to route bus schedules.
112. A timetable is a pair, a net and a table.
113. A well-formed timetable must satisfy same bus schedules within each bus line
114. All bus numbers are distinct across bus lines.

```
type
107.   BLNm
value
108.   same_bus_schedule: BusSched × BusSched → Bool
108.   same_bus_schedule((r1,btl1),(r2,btl2)) ≡
108.      r1 = r2 ∧ len btl1 = btl2 ∧
108.      ⟨sel_BS(btl1(i))|i:[1..len btl1]⟩=⟨sel_BS(btl2(i))|i:[1..len btl2]⟩
type
109.   BNo
110.   RBS :: sel_R:R  sel_btbl:(BNo →m SBS)
111.   TBL = BLNm →m RBS
112.   TT′ = ND × TBL
113.   TT  = {|tt:TT′•wf_TT(tt)|}
```

```
value
113. wf_TT: TT′ → Bool
113. wf_TT(_,tbl) ≡
113.   ∀ bln:BLNm•bln ∈ dom tbl ⇒
113.     ∀ bno,bno′:BNo • {bno,bno′}⊆dom sel_btbl(tbl(bln)) ⇒
113.       same_bus_schedule(sel_R(tbl(bln)),sel_btbl(tbl(bln))(bno),
113.                         sel_R(tbl(bln)),sel_btbl(tbl(bln))(bno′)) ∧
114.   ∀ bln′,bln″:BLNm • {bln′,bln″}⊆dom tbl ∧ bln′≠bln″ ⇒
114.     dom sel_btbl(tbl(bln′)) ∩ dom sel_btbl(tbl(bln″)) = {}
```

3.4.3. Route and Bus Timetable Denotations

- What are routes and bus timetables scripting ?
- Routes (list of connected link traversal designations) script that one may transport people or freight along the sequence of designated links.
- Bus timetables script (at least) two things:
  - the set of bus traffics on the net which satisfy the bus timetable, and
  - information that potential and actual bus passengers may, within some measure of statistics (and probability), rely upon for their bus transport.

- Here, we shall not develop the idea of bus timetables denoting certain traffics.
  - Instead we refer to our previously sketched model of traffics (Sect. , Pages 89–97).
- Route (designations) and bus timetables
  - script potential and actual route travels, respectively
  - script the dispatch of buses and their travelling.
- Bus timetables can also be seen as a form of contracts
  - between the bus operators offering the bus services
  - and potential and actual passengers,
  - with the contract promising timely transport.
- In the next section, Sect. , we shall sketch a language of bus service contracts and bus service actions implied by such contracts.

3.4.4. Licenses and Contracts

Definition: License.

- *A license is*
  - a *script*
  - *specifically expressing a permission to act;*
  - *is freedom of action;*
  - *is a permission granted by competent authority to engage in a business or occupation or in an activity otherwise unlawful;*
  - *a document, plate, or tag evidencing a license granted;*
  - *a grant by the holder of a copyright or patent to another of any of the rights embodied in the copyright or patent short of an assignment of all rights.*

Licenses appear more to have morally than legally binding poser.

Definition: Contract.

- *A contract*
  - *is a special kind of license*
  - *specifically expressing a legally binding agreement between two or more parties —*
  - *hence a document describing the conditions of the contract;*
  - *a contract is business arrangement for the supply of goods or services at fixed prices, times and locations.*

- *In software development a contract specifies what is to be developed:*
  - *(1) a domain description,*
  - *(2) a requirements prescription, or*
  - *(3) a software design;*

  or a combination of these (1–2, 2–3, 1–3).

- *A contract further specifies*
  - *how it might, or must be developed;*
  - *criteria for acceptance of what has been developed;*
  - *delivery dates for the developed items;*
  - *who the "parties" to the contract are:*
    - ∗ *the client and*
    - ∗ *the developer, etc.*

- For a comprehensive treatment of licenses and contracts we refer to [`Chapter 10, Sect. 10.6 (Pages 309--326) [jaist-db10]][jaist-mono]`.
- We shall illustrate fragments of a language for bus service contracts.
- The background for the bus contract language is the following.
  - In many large cities around Europe the city or provincial government secures public transport in the form of bus services operated by many different private companies.
  - Earlier lectures illustrated the concept of bus (service) timetables.
  - The bus services implied by such a timetable, for a city area — with surrounding suburbs etc. — need not be implemented by just one company, but can be contracted, by the city government public transport office, to several companies, each taking care of a subset of the timetable.

– Different bus operators then take care of non-overlapping parts and all take care of the full timetable.
– It may even be that extra buses need be scheduled, on the fly, in connection with major sports or concert or other events.
– Bus operators may experience vehicle breakdowns or bus driver shortages and may be forced to subcontract other, even otherwise competing bus operators to "step in" and alleviate the problem.

3.4.4.1. Contracts

Schematically we may represent a bus contract as follows:

Contract cn **between contractee** ci **and contractor** cj:
  This contract contracts cj **in the period** $[t,t']$ **to**
    perform the following services with respect to timetable tt:
      operate bus lines $\{blj_1,blj_2,...,blj_n\}$
      subject to the following occasional exceptions:
        cancellation of bus tours:
          $\{(blj_a,\{bno_{a_1},...,bno_{a_m}\}),...\}$ **subject to conditions** cbt
        insertion of bus tours on lines
          $\{blj_\alpha,blj_\beta,...,blj_\gamma\}$ **subject to conditions** ibt
        subcontracting bus tours on lines
          $\{blj_\delta,blj_\phi,...,blj_\omega\}$ **subject to conditions** scbt.

115. A bus contract has a header with the distinct names of a contractee and a contractor and a time interval.
116. A bus contract presents a timetable.
117. A bus contract presents a set of bus lines (by their identifiers) such that these are in the timetable.
118. And a bus contract may list one or more of three kinds of "exceptions":
    (a) cancellation of one or more named bus tours on one or more bus lines subject to certain (specified) conditions;
    (b) insertion of one or more extra bus tours on one or more bus lines subject to certain (specified) conditions;
    (c) subcontracting one or more unspecified bus tours on one or more bus lines subject to certain (specified) conditions — to further unspecified contractors.

We abstract the above quoted "one or more of three kinds of exceptions" as one possibly empty clause for each of these alternatives.

119. A bus contract now contains a header, a timetable, the subject bus lines and the exceptions,
120. such that
    (a) line names mentioned in the contract are those of the bus lines of the timetable, and
    (b) bus (tour) numbers are those of the appropriate bus lines in the timetable.
121. The calendar period is for at least one full day, midnight to midnight.
122. A named contract is a pair of a contract name and a contract.

type

115. $\mathrm{CNm, CId, D, T, CON}$

115. $\mathrm{CH} = \mathrm{CId} \times \mathrm{CId} \times (\mathrm{D}\times\mathrm{D})$

116. $\mathrm{CT} = \mathrm{TT}$

117. $\mathrm{CLs} = \mathrm{BLNm}\textbf{-set}$

118. $\mathrm{CE} = (\mathrm{CA} \times \mathrm{IN} \times \mathrm{SC}) \times \mathrm{CON}$

118(a). $\mathrm{CA} = \mathrm{BLNm} \ \overrightarrow{m} \ \mathrm{BNo}\textbf{-set}$

118(b). $\mathrm{IN} = \mathrm{BLNm} \ \overrightarrow{m} \ \mathrm{BNo}\textbf{-set}$

118(c). $\mathrm{SC} = \mathrm{BLNm}\textbf{-set}$

119. $\mathrm{CO}' = \mathrm{CH} \times \mathrm{CT} \times \mathrm{CLs} \times \mathrm{CE}$

120. $\mathrm{CO} = \{|\mathrm{co{:}CO}'\cdot\mathrm{wf\_CO(co)}|\}$

122. $\mathrm{NCO} = \mathrm{CNm} \times \mathrm{CO}$

value

120. $\mathrm{wf\_CO{:}\ CO}' \rightarrow \textbf{Bool}$

120. $\mathrm{wf\_CO((ce,cr,(d,d')),(nd,tbl),cls,((blns,blns',bls),con))} \equiv$

117. $\quad \mathrm{ce} \neq \mathrm{cr} \wedge$

120(a). $\quad \mathrm{cls} \subseteq \textbf{dom}\ \mathrm{tbl} \wedge$

120(b). $\quad \forall\ \mathrm{bli,bli'{:}BLNm} \cdot \mathrm{bli} \in \textbf{dom}\ \mathrm{blns} \wedge \mathrm{bli'} \in \textbf{dom}\ \mathrm{blns'} \Rightarrow$

120(a). $\qquad \{\mathrm{bli,bli'}\} \subseteq \textbf{dom}\ \mathrm{tbl} \wedge$

120(b). $\qquad \mathrm{blns(bli)} \cup \mathrm{blns'(bli')} \subseteq \textbf{dom}\ \mathrm{sel\_btbtl(tbl(bli))} \wedge$

120(a). $\quad \mathrm{bls} \subset \textbf{dom}\ \mathrm{tbl} \wedge$

121. $\quad \mathrm{d} < \mathrm{d'}$

3.4.4.2. Contractual Actions

For contract cn **commence bus tour, line:** bli **and bus no.:** bno

For contract cn **cancel bus tour, line:** bli **and bus no.:** bno

For contract cn **insert extra bus tour, line:** bli **and bus no.:** bno

Subcontract with respect to contract cn **the following:**
  Contract cn′: **for the calendar period** [d,d′] **contractee** ci **contracts contractor** cj
    to perform the following services with respect to timetable tt:
      operate bus lines $\{\mathrm{blj}_1,\mathrm{blj}_2,...,\mathrm{blj}_n\}$
      subject to the following occasional exceptions:
        cancellation of bus tours:
          $\{(\mathrm{blj}_c,\{\mathrm{bno}_{c_1},...,\mathrm{bno}_{c_m}\}),...\}$ **subject to conditions** cbt
        insertion of bus tours on lines
          $\{(\mathrm{blj}_i,\{\mathrm{bno}_{i_1},...,\mathrm{bno}_{i_n}\}),...\}$ **subject to conditions** ibt
        subcontracting bus tours on lines
          $\{\mathrm{blj}_\delta,\mathrm{blj}_\phi,...,\mathrm{blj}_\omega\}$ **subject to conditions** scbt.

123. A bus operator action is either a commence, a cancellation, an insertion or a subcontracting action. All actions refer to the (name of) the contract with respect to which the action is contracted.
   (a) A `commence` action designator states the bus line concerned and the bus number of that line.
   (b) A cancellation action designator states the bus line concerned and the bus number of that line.
   (c) An insertion action designator states the bus line concerned and the bus number of that line — for which an extra bus is to be inserted.[4]
   (d) A subcontracting action designator, besides the name of the contract with respect to which the subcontract is a subcontract, state a named contract (whose contract name is unique).

[4]The insertion of buses in connection with either unscheduled or extraordinary (sports, concerts, etc.) events can be handled by special, initial contracts.

type
123. Act = Com | Can | Ins | Sub
123(a). Com == mkCom(sel_cn:CNm,sel_bli:BLNm,sel_bno:BNo)
123(b). Can == mkCan(sel_cn:CNm,sel_bli:BLNm,sel_bno:BNo)
123(c). Ins == mkIns(sel_cn:CNm,sel_bli:BLNm,sel_bno:BNo)
123(d). Sub == mkSub(sel_cn:CNm,sel_con:NCO)

3.4.4.3. Wellformedness of Contractual Actions

124. In order to express wellformedness conditions, that is, pre-conditions, for the action designators we introduce a context which map contract names to contracts.

125. Wellformedness of a contract is now expressed with respect to a context.

type
124. $\mathrm{CTX} = \mathrm{CNm} \xrightarrow{m} \mathrm{CO}$
value
125. wf_Act: Act → CTX → **Bool**

- Let a defined cnm entry in ctx be a contract: ((ce,cr),(nd,tbl),cls,(blns,bls,bls′),(d,d′)).

126. If cmd is a commence command mkCom(cnm,bln,bno), then
    (a) contract name cnm must be defined in context ctx;
    (b) bus line name bln must be defined in the contract, that is, in cls, and
    (c) bus number bno must be defined in the bus table part of table tbl.

126. wf_Act(mkCom(cnm,bln,bno))(ctx) ≡
126(a). cnm ∈ **dom** ctx ∧
126. **let** ((ce,cr),(nd,tbl),cls,(blns,bls,bls′),(d,d′)) = ctx(cnm) **in**
126(b). bln ∈ cls ∧
126(c). bno ∈ **dom** sel_btbl(tbl(bln)) **end**

127. cancellation and insertion commands have the same static wellformedness conditions as have commence command.

127. wf_Act(mkCan(cnm,bln,bno))(ctx) ≡ wf_Act(mkCom(cnm,bln,bno))(ctx)
127. wf_Act(mkIns(cnm,bln,bno))(ctx) ≡ wf_Act(mkCom(cnm,bln,bno))(ctx)

128. If cmd is a subcontract command then

Let the subcontract command and the cnm named contract in ctx be
mkSub(cnm,nco:(cnm′,(ce′,cr′,(d″,d‴)),(nd′,tbl′),cls′,(blns′,bls″,bls‴)))
respectively ((ce,cr,(d,d′)), (nd,tbl), cls, (blns,bls,bls′)).

(a) contract name cnm must be defined in context ctx;
(b) contract name cnm′ must not be defined in context ctx;
(c) the calendar period of the subcontract must be within that of the contract from which it derives;
(d) the net descriptors nd and nd′ must be identical;
(e) the tables tbl and tbl′ and must be identical and
(f) the set, cls′, of bus line names that are the scope of the subcontracting must be a subset of bls′.

```
128.    wf_Act(mkSub(cnm,nco:(cnm′,co:((ce′,cr′,(d″,d‴)),(nd′,tbl′),cls′,(blns′,blns″,bls‴)))))(ctx)
128(a).     cnm ∈ dom ctx ∧
128.       let co′ = ((ce,cr,(d,d′)),(nd,tbl),cls,(blns,blns′,bls′)) = ctx(cnm) in
128(b).     cnm′ ∉ dom tbl ∧
128(c).     d ≤ d″ ≤ d‴ ≤ d′ ∧
128(d).     nd′ = nd ∧
128(e).     tbl′ = tbl ∧
128(f).     cls′ ⊆ bls′ end
```

- Wellformedness of contracts, wf_CO(co) and wf_CO(co′), secures other constraints.

- We do not here bring any narrated or formalised description of the semantics of contracts and actions.
- First such a description would be rather lengthy.
- Secondly a specification would be more of a requirements prescription.

3.5. Management and Organisation

Definition: Management.

- *Management is about resources:*
  - *their acquisition,*
  - *scheduling (over time),*
  - *allocation (over locations),*
  - *deployment (in performing actions) and*
  - *disposal ("retirement").*
- *We distinguish between*
  - *board-directed,*
  - *strategic,*
  - *tactical and*
  - *operational*

  actions.

- *Board-directed actions target mainly financial resources: obtaining new funds through conversion of goodwill into financial resources, acquiring and selling "competing" or "supplementary" business units.*
- *Strategic actions convert financial resources into production, service supplies and resources and vice-versa — and in this these actions schedule availability of such resources.*
- *Tactical actions mainly allocate resources.*
- *Operational actions order, monitor and control the deployment of resources in the performance of actions.*

Definition: Organisation.

- *Organisation is about*
  - *the "grand scale",*
    - ∗ *executive and strategic*
    - ∗ *national, continental or global (world wide)*
  - *(i) allocation of major resource (e.g., business) units, whether in a hierarchical, in a matrix, or in some other organigram-specified structure,*
  - *(ii) as well as the clearly defined relations (which information, decisions and actions are transferred) between these units, and*
  - *(iii) organisational dynamics.*

Definition: Management & Organisation.

- *The composite term management and organisation*
  - *applies in connection with management as outlined just above and*
  - *with organisation also outlined above.*
- *The term then emphasises the relations between the organisation and management of an enterprise.*

• • •

The borderlines within management actions and across organisation "layouts" are fuzzy.

3.5.1. Transport System Examples

We shall only present sketchy examples of management and organsation.

- Executive actions:
  - Deciding on major re-organisation of a transport net
    - ∗ (for example introduction of toll roads or freeways,
    - ∗ road pricing,
    - ∗ major bridges across wide waters [potentially connecting two hitherto unconnected nets],
    - ∗ and their management)

    are executive actions.
  - So are decisions on merging or splitting transport from or into several transport services.

– Reorganising an enterprise
  * from one characterised by a "deep" hierarchy of management layers (a hierarchy which may very well exemplify highly centralised both administrative and functional monitoring and control)
  * into a matrix of two "shallow" hierarchies, one which addresses tactical and operational management and one which addresses executive and strategic management — with the former (the operations) being replicated across geographical areas while the latter applies "globally" —

  such reorganisations reflect executive actions (but are carried out by strategic and tactical management).

- Strategic actions: Adding or removing transport links, or major reorganisation of bus timetables are strategic actions. Splitting a(n own) contract into what is still to be operated and subcontracting other parts, for definite, to other bus operators are also strategic actions.
- Tactical actions: Insertion and cancellation of bus services are tactical actions. Subcontracting some parts of a timetable demanded service, for a short while, to other bus operators could be considered tactical actions.
- Operational actions: Commencing and thus, in general, allocating drivers to and sending these off on bus services are operational actions. So are announcing insertion of new (unscheduled) and cancellation of scheduled routes.

3.6. Human Behaviour

Definition: Human Behaviour.

- *By human behaviour we shall here understand*
  - *the way a human follows the enterprise rules and regulations*
  - *as well as interacts with a machine:*
    * *dutifully honouring specified (machine dialogue or) protocols,*
    * *or negligently so,*
    * *or sloppily not quite so,*
    * *or even criminally not so!*
- *Human behaviour is a facet of the domain.*
  - *We shall thus model human behaviour also in terms of it failing to react properly,*
  - *i.e., humans as non-deterministic agents!*

3.7. Towards Theories of Domain Facets

3.7.1. A Theory of Intrinsics

3.7.2. Theories of Support Technologies

3.7.2.1. An Example

- Traffic (tf:TF), intrinsically, is a total function over some time interval, from time (t:T) to continuously positioned (p:P) vehicles (tn:TN).
- Conventional optical sensors sample, at regular intervals, the intrinsic train traffic.
- The result is a sampled traffic (stf:sTF).
- Hence the collection of all optical sensors, for any given net, is a partial function from intrinsic (itf) to sampled train traffics (stf).
- We need to express quality criteria that any optical sensor technology should satisfy — relative to a necessary and sufficient description of a closeness predicate.

- For all intrinsic traffics, itf, and for all optical sensor technologies, og, the following must hold:
  - Let stf be the traffic sampled by the optical gates.
  - For all time points, t, in the sampled traffic,
  - those time points must also be in the intrinsic traffic,
  - and, for all trains, tn, in the intrinsic traffic at that time,
  - the train must be observed by the optical gates, and
  - the actual position of the train and the sampled position must somehow be checkable to be close, or identical to one another.

Since hubs change state with time, n:N, the net needs to be part of any model of traffic.

```
type
  T, TN
  P = HP | LP
  NetTraffic :: net:N × trf:(V $\overrightarrow{m}$ P)
  iTF = T → NetTraffic
  sTF = T $\overrightarrow{m}$ NetTraffic
  oG = iTF $\xrightarrow{\sim}$ sTF
value
  [close] c: NetTraffic × TN × NetTraffic $\xrightarrow{\sim}$ Bool
axiom
  ∀ itt:iTF, og:OG · let stt = og(itt) in
    ∀ t:T · t ∈ dom stt ·
      t ∈ $\mathbb{DOM}$ itt ∧ ∀ Tn:TN · tn ∈ dom trf(itt(t))
        ⇒ tn ∈ dom trf(stt(t)) ∧ c(itt(t),tn,stt(t)) end
```

3.7.2.2. General

- The formal requirements can be narrated:
  - Let Θ_i and Θ_a designate the spaces of intrinsic and actual-world configurations (contexts and states).
  - For each intrinsic configuration model — that we know is support technology assisted —
  - there exists a support technology solution,
  - that is, a total function from all intrinsic configurations to corresponding actual configurations.
- If we are not convinced that there is such a function then there is little hope that we can trust this technology

```
type
  $\Theta_i$, $\Theta_a$
  ST = $\Theta_i \to \Theta_a$
axiom
  ∀ sts:ST-set, st:ST · st ∈ sts ⇒ ∀ $\theta_i$:$\Theta_i$, ∃ $\theta_a$:$\Theta_a$ · st($\theta_i$) = $\theta_a$
```

3.7.3. A Theory of Rules & Regulations

- There are, abstractly speaking, usually three kinds of languages involved wrt. (i.e., when expressing) rules and regulations (respectively when invoking actions that are subject to rules and regulations).
  - Two languages, Rules and Reg, exist for describing rules, respectively regulations; and
  - one, Stimulus, exists for describing the form of the [always current] domain action stimuli.

- A syntactic stimulus, sy_sti, denotes a function, se_sti:STI: $\Theta \to \Theta$, from any configuration to a next configuration
- A syntactic rule, sy_rul:Rule, has as its semantics, its meaning, rul:RUL,
  - a predicate over current and next configurations, $(\Theta \times \Theta) \to$ **Bool**,
  - where these next configurations have been caused, by the stimuli. These stimuli express:
  - If the predicate holds then the stimulus will result in a valid next configuration.

type
  Stimulus, Rule, Θ
  STI = $\Theta \rightarrow \Theta$
  RUL = $(\Theta \times \Theta) \rightarrow$ **Bool**
value
  meaning: Stimulus $\rightarrow$ STI
  meaning: Rule $\rightarrow$ RUL

  valid: Stimulus $\times$ Rule $\rightarrow \Theta \rightarrow$ **Bool**
  valid(sy_sti,sy_rul)(θ) $\equiv$ meaning(sy_rul)(θ,(meaning(sy_sti))(θ))

  valid: Stimulus $\times$ RUL $\rightarrow \Theta \rightarrow$ **Bool**
  valid(sy_sti,se_rul)(θ) $\equiv$ se_rul(θ,(meaning(sy_sti))(θ))

- A syntactic regulation, sy_reg:Reg (related to a specific rule), stands for, i.e., has as its semantics, its meaning,
  - a semantic regulation, se_reg:REG,
  - which is a pair.
  - This pair consists of
    - ∗ a predicate, pre_reg:Pre_REG, where Pre_REG = $(\Theta \times \Theta) \rightarrow$ **Bool**,
    - ∗ and a domain configuration-changing function, act_reg:Act_REG, where Act_REG = $\Theta \rightarrow \Theta$,
    - ∗ that is, both involving current and next domain configurations.

  - The two kinds of functions express:
    - ∗ If the predicate holds,
    - ∗ then the action can be applied.
- The predicate is almost the inverse of the rules functions.
- The action function serves to undo the stimulus function.

type
  Reg
  Rul_and_Reg = Rule $\times$ Reg
  REG = Pre_REG $\times$ Act_REG
  Pre_REG = $\Theta \times \Theta \rightarrow$ **Bool**
  Act_REG = $\Theta \rightarrow \Theta$
value
  interpret: Reg $\rightarrow$ REG

- The idea is now the following:
  - Any action of the system, i.e., the application of any stimulus,
    - may be an action in accordance with the rules,
    - or it may not.
  - Rules therefore express whether stimuli are valid or not in the current configuration.
  - And regulations therefore express whether they should be applied, and, if so, with what effort.

- More specifically,
  - there is usually, in any current system configuration, given a set of pairs of rules and regulations.
  - Let (sy_rul,sy_reg) be any such pair.
  - Let sy_sti be any possible stimulus.
  - And let θ be the current configuration.
  - Let the stimulus, sy_sti, applied in that configuration result in a next configuration, θ', where $\theta' = (\text{meaning}(\text{sy\_sti}))(\theta)$.
  - Let θ' $(= (\text{meaning}(\text{sy\_sti}))(\theta))$ violate the rule, i.e., ∼valid(sy_sti,sy_rul)(θ
  - then if predicate part, pre_reg, of the meaning of the regulation, sy_reg, holds in that violating next configuration, pre_reg(θ,θ'
  - then the action part, act_reg, of the meaning of the regulation, sy_reg, must be applied, act_reg(θ'), to remedy the situation.

```
axiom
 ∀ (sy_rul,sy_reg):Rul_and_Regs ·
   let se_rul = meaning(sy_rul),
       (pre_reg,act_reg) = meaning(sy_reg) in
   ∀ sy_sti:Stimulus, $\theta$:$\Theta$ ·
     ∼valid(sy_sti,se_rul)($\theta$)
       ⇒ let $\theta'$ = (meaning(sy_sti))($\theta$) in
         pre_reg($\theta$,$\theta'$)
         ⇒ ∃ n$\theta$:$\Theta$ · act_reg($\theta'$)=n$\theta$ ∧ se_rul($\theta$,n$\theta$)
   end end
```

- It may be that the regulation predicate fails to detect applicability of regulations actions.
- That is, the interpretation of a rule differs, in that respect, from the interpretation of a regulation.
- Such is life in the domain, i.e., in actual reality

3.7.4. A Theory of Management & Organisation

3.7.5. A Theory of Human Behaviour

- Commensurate with the above, humans interpret rules and regulations differently,
- and not always "consistently" — in the sense of repeatedly applying the same interpretations.
- Our final specification pattern is therefore:

type
  Action = $\Theta \overset{\sim}{\rightarrow} \Theta$-**infset**
value
  hum_int: Rule $\rightarrow \Theta \rightarrow$ RUL-**infset**
  action: Stimulus $\rightarrow \Theta \rightarrow \Theta$
  hum_beha: Stimulus $\times$ Rules $\rightarrow$ Action $\rightarrow \Theta \overset{\sim}{\rightarrow} \Theta$-**infset**
  hum_beha(sy_sti,sy_rul)$(\alpha)(\theta)$ **as** θset
    post
      θset $= \alpha(\theta) \wedge$ action(sy_sti)$(\theta) \in \theta$set
      $\wedge \forall \theta'$:Θ•$\theta' \in \theta$set $\Rightarrow$
        $\exists$ se_rul:RUL•se_rul $\in$ hum_int(sy_rul)$(\theta)\Rightarrow$se_rul(θ,θ')

- The above is, necessarily, sketchy:
  - There is a possibly infinite variety of ways of interpreting some rules.
  - A human, in carrying out an action, interprets applicable rules and chooses one which that person believes suits some (professional, sloppy, delinquent or criminal) intent.
  - "Suits" means that it satisfies the intent,
    - i.e., yields **true** on the pre/post-configuration pair,
    - when the action is performed —
    - whether as intended by the ones who issued the rules and regulations or not.
  - We do not cover the case of whether an appropriate regulation is applied or not

End of Lecture 4: DOMAINS: Scripts – Human Behaviour

Start of Lecture 5: REQUIREMENTS – up to and incl. Determination

4. An Ontology of Requirements Constructions

Definition: Requirements.

- *A condition or capability*
- *needed by a user*
- *to solve a problem or achieve an objective* [IEEEStd.610.12].

Definition: Machine.

- *By the machine we understand*
- *the hardware*
- *plus software*
- *that implements some requirements, i.e., a computing system.*

Definition: Requirements Unit.

- *By a requirements unit*
  - *we mean a single sentence*
  - *which expresses an “isolated” requirements.*
  - *(We omit charaterising “single sentence” and “isolated”.)*

Definition: Requirements Prescription.

- *By a requirements prescription*
  - *we mean just that: the prescription of some requirements.*
- *Sometimes, by requirements prescription,*
  - *we mean a relatively complete and consistent specification of all requirements,*
  - *and sometimes just a requirements unit.*

Definition: Requirements Engineering. *The engineering of the development of a requirements prescription,*

- *from identification of requirements stake-holders,*
- *via requirements acquisition,*
- *requirements analysis, and*
- *requirements prescription to*
- *requirements validation and*
- *requirements verification.*

- We shall just focus on *requirements prescription*,
- that is, the modelling of *requirements*.

4.1. Business Process Re-engineering

Definition: Business Process.

- *By a business process we shall understand*
  - *a behaviour of an enterprise, a business, an institution, a factory.*
  - *A business process reflects the ways in which a business conducts its affairs, and is a facet of the domain.*
- *Other facets of an enterprise are those of its*
  - *intrinsics,*
  - *support technology,*
  - *rules and regulations,*
  - *management and organisation , and*
  - *human behaviour.*

Definition: Business Process Engineering.

- *By business process engineering we shall understand*
  - *the design,*
  - *the determination,*

  of business processes.
- *In doing business process engineering*
  - *one is basically designing,*
  - *i.e., prescribing entirely new business processes.*

Definition: Business Process Re-engineering.

- *By business process reengineering we shall understand*
  - *the re-design,*
  - *the change,*

  of business processes.
- *In doing business process re-engineering*
  - *one is basically carrying out change management.*

4.1.1. The Kinds of Requirements

- We distinguish between three kinds of requirements:
  - the **domain requirements** are those requirements which can be expressed solely using terms of the domain;
  - the **machine requirements** are those requirements which can be expressed solely using terms of the machine, and
  - the **interface requirements** are those requirements which must use terms from both the domain and the machine in order to be expressed.

4.1.2. Goals Versus Requirements

- Whereas
  - a domain description presents a domain **as it is**,
  - a requirements prescription presents a domain
    - **as it would be**
    - if some required **machine**
    - was implemented (from these requirements).
- The **machine** is the **hardware** plus **software** to be designed from the requirements.
- That is, the *machine* is what the requirements are about.

- We make a distinction between **goals** and **requirements.**
- Goals are what we
  - expect satisfied by the software
  - implemented from the requirements.
- But goals could also be
  - of the system
  - for which the software is required.
- First we exemplify the latter, then the former.

4.1.2.1. 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,

4.1.2.2. 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.
- Goals are usually expressed in terms of properties.
- Requirements can then be proved to satisfy the $\mathcal{G}$oals: $\mathcal{D}, \mathcal{R} \models \mathcal{G}$.

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

4.1.2.4. Arguing Goal-satisfaction of Toll Road System Software

- By recording
  - tickets issued and collected at toll booths and
  - toll road hubs and links entered and left
  - as per the requirements specification brought in forthcoming examples (Sects. –),
- we can eventually argue that
  - the requirements of the forthcoming examples (Sects. –)
  - help satisfy the goal of the example **??** on page ??.

- We shall assume that the (goal and) requirements engineer elicit both $\mathcal{G}$oals and $\mathcal{R}$equirements from requirements stake-holders.
- $\mathcal{D}, \mathcal{R} \models \mathcal{G}$
  - The $\mathcal{G}$oals can be argued to hold
  - by reasoning over the $\mathcal{R}$equirements
  - and the $\mathcal{D}$omain.
- But we shall focus only on
  - domain and
  - interface

  requirements such as "derived" from domain descriptions.

4.1.3. Re-engineered Nets

- The nets defined in Lecture 3 could be of any topology.
  - They could consist of two or more nets that were not linked to one another;
  - they could consist of connected nets or nets that were acyclic; etc.;
  - and the nets were not specifically road, rail, sea lane or air lane nets.

- We shall now consider a special kind of road nets: basically the road nets we have in mind
  - are linear sequences of pairs of links of opposite direction link "states",
  - where these links, let us call them toll road links, are connected to toll road hubs;
  - where, in addition, these toll road hubs are linked, via toll plazas (i.e., "special" hubs) to toll road hubs
  - by means of on/off links.

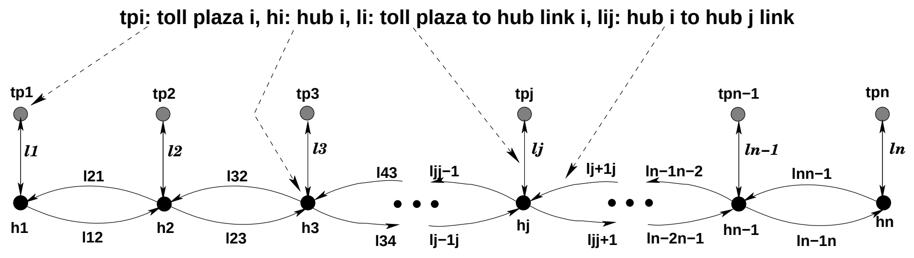


Figure 3: A Toll Road System

- We do not consider the general nets that are (possibly) connected to the toll plazas.
- The pragmatics behind these nets is the following:
  - Drivers enter and leave the toll road nets at toll road plazas;
  - collect tickets from toll road plaza ticket-issuing booths when entering the toll road net and
  - present these at toll road plaza ticket-collection booths and pay according to some function of the time and length (from entry to exit plaza) driven on the toll road net when leaving the net;
  - drivers are otherwise free to "circle" the toll road net as they see fit:
    * multiple times "up and down" the net,
    * circling toll road hubs,
    * etc.

- Our sketch centers around a toll road net with toll booth plazas.
- The BPR focuses
  - first on entities, actions, events and behaviours,
  - then on the six domain facets.

129. **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.
   - (Sect. brings more details.)

130. **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.

131. **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.

132. **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 — with
  - receipt 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).

133. **Re-engineered Intrinsics:**

- Toll plazas and abstracted booths are added to domain intrinsics.

134. **Re-engineered Support Technologies:**

- There is a definite need for domain-describing the failure-prone toll plaza entry and exit booths.

135. **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.

136. **Re-engineered Scripts:**

- No need.

137. **Re-engineered Management and Organisation:**

- There is a definite need for domain describing
- the management and possibly distributed organisation
- of toll booth plazas.

138. **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".

4.2. Domain Requirements

Definition: Domain Requirements.

- *By domain requirements we understand*
  - *such requirements*
  - *(save those of business process reengineering)*
  - *which can be expressed sôlely by using professional terms of the domain.*

Definition: Domain Requirements Facet.

- *By domain requirements facets we understand*
  - *such domain requirements*
  - *that basically arise from either of the following operations on domain descriptions (cum requirements prescriptions):*
    - ∗ *domain projection,*
    - ∗ *domain determination,*
    - ∗ *domain extension,*
    - ∗ *domain instantiation and*
    - ∗ *domain fitting.*

4.2.1. Projection

Definition: Projection.

- *By projection we shall here, in a somewhat narrow sense, mean*
  - *a technique that applies to domain descriptions and*
  - *yields requirements prescriptions.*
- *Basically projection "reduces" a domain description*
  - *by "removing" (or, but rarely, hiding)*
    - ∗ *entities,*
    - ∗ *functions,*
    - ∗ *events and*
    - ∗ *behaviours*

  from the domain description.

- *If the domain description is an informal one, say in English,*
  - *it may have expressed that certain entities, functions, events and behaviours*
  - *might be in (some instantiations of) the domain.*
  - *If not "projected away" the similar, i.e., informal requirements prescription*
  - *will express that these entities, functions, events and behaviours*
  - *shall be in the domain and*
  - *hence will be in the environment of the machine being requirements prescribed.*

4.2.1.1. Example

Keep

- N, H, L,
- obs_Hs,
- obs_Ls,
- HI, LI,
- obs_HI,
- obs_LI,
- obs_LIs,
- obs_HIs,
- PLAN, LHIM,
- wf_PLAN,
- ND, wf_ND,
- $L\Sigma$, $L\Omega$,
- obs_$L\Sigma$, obs_$L\Sigma$,
- $H\Sigma$, $H\Omega$,
- obs_$H\Sigma$, obs_$H\Sigma$,
- V, VI, VP,
- obs_VI, obs_VP,
- TF, T and
- wf_TF.

4.2.2. Instantiation

Definition: Instantiation.

- *'To represent (an abstraction) by a concrete instance'* [mw2004].
- Domain instantiation is a *domain requirements facet*.
- It is an operation performed on a *domain description* (cum *requirements prescription*).
- Where, in a domain description certain *entities* and *function*s are left undefined,
  - domain instantiation means
  - that these entities or functions are now instantiated
  - into constant *value*s.

4.2.2.1. Example

- The following instantiation prescription only covers the static aspects of the toll road net, i.e., simple entities.
- That is, the states of hubs and links will first be dealt with in Sect. .

139. A toll road net (a subnet of a larger previously described net) consists of a pair: toll road links and toll road to plaza hubs and links.
   (a) The toll road links component is a linear sequence of one or more pairs of toll road links.
   (b) The toll road to plaza hubs and links component is a linear sequence of two or more triples of a plaza, a (plaza to toll road hub) link and a toll road hub.

(c) The wellformedness of toll road nets are expressed next.
   i. The length of the toll road links sequence is one less than the length of the toll road to plaza hubs and links sequence. The idea is that the toll road links at position i connect the toll road hubs at positions i and $i+1$ of the toll road to plaza hubs and links sequence — i being the indexes of the toll road links sequence.
   ii. All links have distinct link identifiers.
   iii. All hubs and plazas have distinct hub identifiers.
   iv. From the links in the pairs of links, (l_i, l_i'), of position i in the toll road links component one observes exactly the same two element set of hub identifiers,
   v. and these are the identifiers of the hubs at positions i and $i+1$ of the toll road to plaza hubs and links sequence.
   vi. The plaza to toll road hub links are indeed connected to these plazas and hubs; and
   vii. the plaza and toll road hubs are connected only to the links as mentioned above.

(d) A toll road plaza is like a hub, with an observable hub identifier (and equipped with ticket-issuing tool booths and ticket-collection and payment toll booths).

type

```
139.     TRN′ = TRLs × PHLs
139.     TRN = {|trn:TRN′•wf_TRN(trn)|}
139(a).  TRLs = (L × L)*
139(b).  PHLs = (PZ × L × H)*
```

```
value
139(c).          wf_TRN: TRN′ → Bool
139(c).          wf_TRN(trn:(trls,phls)) ≡
139((c))i.           len trls +1 = len phls ∧
139((c))ii.          card xtr_Hs(trn) = card xtr_HIs(trn) ∧
139((c))iii.         card xtr_Ls(trn) = card xtr_LIs(trn)
139((c))iv.          ∀ i:Nat•i ∈ inds trls ⇒
139((c))iv.             let (l,l′)=trsl(i),(p,l″,hi)=phls(i),(_,l‴,hj)=phls(i+1) in
139((c))iv.             obs_HIs(l) = obs_HIs(l′) =
139((c))v.             {obs_HI(hi),obs_HI(hj)}  ∧
139((c))vii.            case i of
139((c))vii.               1 → obs_LIs(hi) = xtr_LIs({l,l′,l″}),
139((c))vii.               len trsl − 1 → obs_LIs(hj) = xtr_LIs({l,l′,l‴}),
139((c))vii.               _ → let (l⁗,l⁗′)=trsl(i) in obs_LIs(hi)=xtr_LIs({l,l′,l″,l⁗,l⁗′}) end
139((c))vii.            end end ∧
139((c))vii.         ∀ i:Nat•i ∈ inds phls ⇒
139((c))vii.            let (p,l,h)=phls(i) in obs_HIs(l)=xtr_HIs({p,h}) ∧
139((c))vii.            obs_LIs(p) = {obs_LI(l)} end
```

```
type
139(d).  PZ
value
139(d).  obs_HI: PZ → HI

    xtr_Hs: TRN → H-set
    xtr_Hs(_,phls) ≡ {pz,h|(pz,l,h):(PZ×L×H)•(pz,l,h)∈ elems phls}
    xtr_Ls: TRN → L-set
    xtr_Ls(trls,phls) ≡
       {l,l′|l,l′:L•(l,l′)∈ elems trls} ∪ {l|(pz,l,h):(PZ×L×H)•(pz,l,h)∈ elems phls}

    xtr_HIs: TRN → HI-set,   xtr_HIs(trn) ≡ {obs_HI(h)|h:(H|PZ)•h ∈ xtr_Hs(trn)}
    xtr_LIs: TRN → LI-set,   xtr_LIs(trn) ≡ {obs_LI(l)|l:L•l ∈ xtr_Ls(trn)}
    xtr_HIs: H-set → HI-set, xtr_HIs(hs) = {obs_LI(h)|h:H•h ∈ hs}
    xtr_LIs: L-set → LI-set, xtr_LIs(ls) = {obs_LI(l)|l:L•l ∈ ls}
```

4.2.2.2. Abstraction: From Concrete Toll Road Nets to Abstract Nets

140. From concrete toll road nets, trn:TRN, one can abstract the nets, n:N, of Items 1–11.
    (a) the abstract net contains the hubs of the concrete net,
    (b) and the links likewise.

```
value
140.     abs_N: TRN → N
140.     abs_N(trn) as n
140(a).      obs_Hs(n) = xtr_Hs(trn) ∧
140(b).      obs_Ls(n) = xtr_Ls(trn)
```

4.2.2.3. Theorem

141. One can prove the following theorem:
    - If trn satisfies wf_TRN(trn)
    - then abs_N(trn)
    - satisfies Axioms 7–8 (Page 14).

```
141.  ∀ trn:TRN · wf_TRN(trn) ⊨
         abs_N(trn) satisfies axioms 7.–10.
```

4.2.3. Determination

Definition: Determination.

- *Domain determination is a domain requirements facet.*
- *It is an operation performed on a domain description cum requirements prescription.*
- *Any nondeterminism expressed by either of these specifications which is not desirable for some required software design must be made deterministic (by this requirements engineer performed operation).*

4.2.3.1. Example

- We shall focus on making more specific the rather generically defined nets, hubs and links.
- There are no traffic signals within the toll road net and pairs of toll road links are "one way, opposite direction" links.

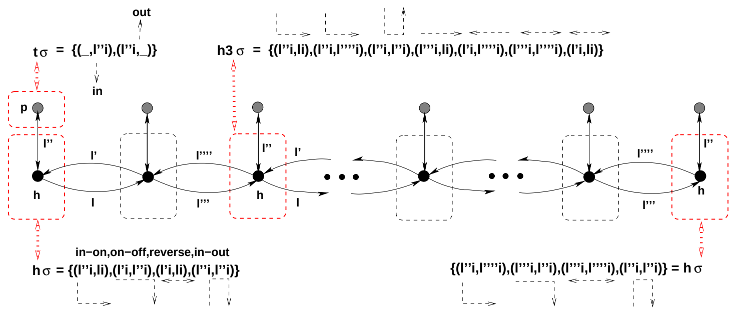

out
tσ = {(_,l''i),(l''i,_)}
in
h3 σ = {(l''i,li),(l''i,l''''i),(l''i,l''i),(l'''i,li),(l'i,l''''i),(l'''i,l''''i),(l'i,li)}
p
l''
l'
l''''
l''
l'
h
l
l'''
h
l
l''''
l''
l'''
h
in−on,on−off,reverse,in−out
hσ = {(l''i,li),(l'i,l''i),(l'i,li),(l''i,l''i)}
{(l''i,l''''i),(l'''i,l''i),(l'''i,l''''i),(l''i,l''i)} = hσ

Figure 4: Four example hub states: plaza, end hubs, "middle" hub

142. Pairs of toll road links, l, l', connecting adjacent hubs hj, hk, of identifiers hj_i, hk_i, respectively, always and only allow traffic in opposite directions, that is, are always in respective states $\{(hj_i, hk_i)\}$ and $\{(hk_i, hj_i)\}$.
143. Hub, h, states, $h\sigma$, are constant and allow traffic onto connected links not closed for traffic in directions from hub h.
144. Plazas allow traffic only onto connected plaza to hub links of the toll road net. (Whatever other links, "outside" the toll road net, the plazas may be connected to is covered in the last line of the axiom below.)

```
axiom
  ∀ (trls,phls):TRN •
    ∀ i:Nat • i ∈ inds trls
      let (l,l′) = trls(i), (p,l″,h) = phls(i) in
      case i of
        1 → obs_HΣ(h) = {(obs_LI(l″),obs_LI(l)),
                         (obs_LI(l′),obs_LI(l″)),(obs_LI(l′),obs_LI(l)),
                         (obs_LI(l″),obs_LI(l″))},
        _ → let (l‴,l⁗) = trls(i−1) in
            obs_HΣ(h) = {(obs_LI(l″),obs_LI(l)),
                         (obs_LI(l″),obs_LI(l⁗)),(obs_LI(l″),obs_LI(l″)),
                         (obs_LI(l″),obs_LI(l″)),(obs_LI(l‴),obs_LI(l)),
                         (obs_LI(l′),obs_LI(l⁗)),(obs_LI(l‴),obs_LI(l⁗)),
                         (obs_LI(l′),obs_LI(l))} end end end ∧
    let (l‴,l⁗) = trls(len trsl), (p,l″,h) = phls(1 + len trsl) in
    obs_HΣ(h) = {(obs_LI(l″),obs_LI(l⁗)),
                 (obs_LI(l‴),obs_LI(l″)),(obs_LI(l‴),obs_LI(l⁗)),
                 (obs_LI(l″),obs_LI(l″))} end ∧
    ∀ (p,l″,_):(PZ×L×H)•(p,l″,_) ∈ elems phls ⇒
      let lis = obs_LIs(p) assert: obs_LI(l″) ∈ lis in
      obs_HΣ(p) = {(li,obs_LI(l″)),(obs_LI(l″),li)|li:LI•li ∈ lis} end
```

- In the last line of the wellformedness axiom above we express that
  - the plaza maybe connected to many links not in the toll road net and
  - that the plaza is open for all traffic from these into the net (via l′′),
  - from l′′ to these
  - and that traffic may even reverse at the plazas,
  - that is, decide to not enter the toll road net after having just visited the plaza.

End of Lecture 5: REQUIREMENTS – up to and incl. Determination

Start of Lecture 6: REQUIREMENTS – from Extension “out”

4.2.4. Extension

Definition: Extension.

- *Domain extension is a domain requirements facet.*
- *It is an operation performed on a domain description or a requirements prescription.*
- *It effectively extends a domain description by entities, functions, events and/or behaviours conceptually possible, but not necessarily humanly or technologically feasible in the domain (as it was).*
- Figure 5 on the following page abstracts some of the extensions to nets: the plaza entry and exit booths.

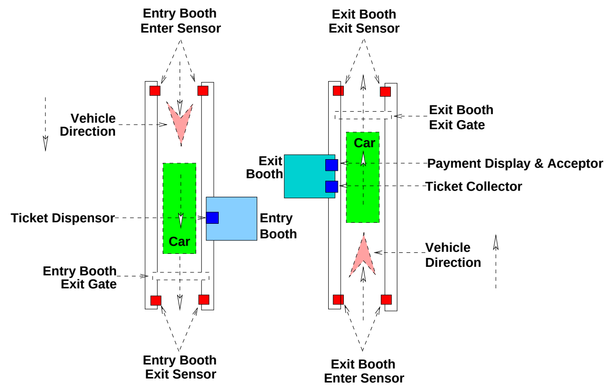


Figure 5: Entry and Exit Tool Booths

- The following is a prolonged example.
- It contains three kinds of formalisations:
  - a `RAISE/CSP` model,
  - a `Duration Calculus` model [zcc+mrh2002,olderogdirks2008] and
  - a `Timed Automata` model [AluDil:94,olderogdirks2008].
- The narrative for all three models are given when narrating the `RAISE/CSP` model.

4.2.4.1. Intuition

- A toll road system is delimited by toll plazas with entry and exit booths with their gates.
- To get access, from outside, to the roads within the toll road system, a car must pass through an entry booth and its entry gate. To leave the roads within the toll road system a car must pass through an exit booth and its exit gate.
- Cars collect tickets upon entry and return these tickets upon exit and pay a fee for having driven on the toll roads.
- The gates help ensure that cars have collected tickets and have paid their dues.

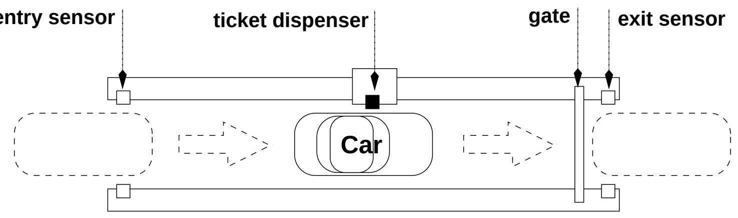


Figure 6: A toll plaza entry booth

4.2.4.2. Descriptions

4.2.4.2.1. • A RAISE/CSP Model•

We use the CSP property [TheSEBook123,CARH:Electronic] of RSL.

⊕ *Toll Booth Plazas* ⊕

- With respect to toll road systems we focus on just their plazas: that is, where cars enter and leave the systems.
- The below description is grossly simplified: instead of plazas having one or more entry and one or more exit booths (both with gates), we just assume one (pair: booth/gate) of each.

145. A toll plaza consists of a one pair of an entry booth and and entry gate and one pair of an exit booth and an exit gate.
146. Entry booths consist of an entry sensor, a ticket dispenser and an exit sensor.
147. Exit booths consist of an entry sensor, a ticket collector, a payment display and a payment component.

type
145. PZ = (EB×G) × (XB×G)
146. EB = ...
147. XB = ...

⊕ *Cars* ⊕

148. There are vehicles.
149. Vehicles have unique vehicle identifications.

type
148. V
149. VId
value
149. obs_VId: V → VId
axiom
149. $\forall$ v,v′:V · v≠v′ ⇒ obs_VId(v) ≠ obs_VId(v′)

⊕ *Entry Booths* ⊕

- The description now given is an idealisation.
- It assumes that everything works:
  - that the vehicles behave as expected and
  - that the electro-mechanics of booths and gates do likewise.

150. An entry_sensor registers whether a car is entering the entry booth or not,
   (a) that is, for the duration of the car passing the entry_sensor that sensor senses the car identification cid
   (b) otherwise it senses “nothing”.

151. A ticket_dispenser
   (a) either holds a ticket or does not hold a ticket, i.e., no_ticket;
   (b) normally it does not hold a ticket;
   (c) the ticket_dispenser holds a ticket soon after a car has passed the entry_sensor;
   (d) the passing car collects the ticket –
   (e) after which the ticket_dispenser no longer holds a ticket.
152. An exit_sensor
   (a) registers the identification of a car leaving the toll booth
   (b) otherwise it senses "nothing".

⊕ *Gates* ⊕

153. A gate
   (a) is either closed or open;
   (b) it is normally closed;
   (c) if a car is entering it is secured set to close (as a security measure);
   (d) once a car has collected a ticket it is set to open;
   (e) and once a car has passed the exit_sensor it is again set to close.

⊕ *The Entry Plaza System* ⊕

```
type
  C, CI
  G = open | close
  TK == Ticket | no_ticket
value
  obs_CI: (C|Ticket) → CI
channel
  entry_sensor:CI
  ticket_dispenser:Ticket
  exit_sensor:CI
  gate_ch:G
value
  vs:V-set
  eb:EB,xb:XB,eg,xg:G
```

```
system: G × EB × V-set × XB × G
system(eg,eb,vs,xb,xg) ≡
  ‖{car(obs_CI(c),c)|c:C·c ∈ cs} ‖ entry_booth(eb) ‖ entry_gate(eg) ‖ ...

car: CI × C → out entry_sensor,exit_sensor
          in ticket_dispenser  Unit
car(ci,c) ≡
  entry_sensor ! ci ;
  let ticket = ticket_dispenser ? assert: ticket ≠ no_ticket in
  ticket_dispenser ! no_ticket ;
  exit_sensor ! ci ;
  car(add(ticket,c)) end
```

```
entry_booth: Unit → in entry_sensor, exit_sensor
          out ticket_dispenser
          out gate_ch  Unit
entry_booth(b) ≡
  gate_ch ! close ;
  let ci = entry_sensor ? in
  ticket_dispenser ! make_ticket(cid) ;
  let res = ticket_dispenser ? in assert: res = no_ticket ;
  gate_ch ! open ;
  let ci′ = exit_sensor ? in assert: ci′ = ci ;
  gate_ch ! close ;
  entry_booth(add_Ticket(ticket,b)) end end end
```

```
entry_gate: G → in gate  Unit
entry_gate(g) ≡
  case gate_ch ? of
    close → exit_gate(close) assert: g = open,
    open → exit_gate(open) assert: g = close
  end

add_Ticket: Ticket × C $\overset{\sim}{\to}$ C
  pre add_Ticket(t,c): ∼has_Ticket(c)
  post: add_Ticket(t,c): has_Ticket(c)
```

```
has_Ticket: (C|B) → Bool

obs_Ticket: (C|B) $\overset{\sim}{\to}$ Ticket
  pre obs_Ticket(cb): has_Ticket(cb)

rem_Ticket: (C $\overset{\sim}{\to}$ C) | (B $\overset{\sim}{\to}$ B)
  pre rem_Ticket(cb): has_Ticket(cb)
  post rem_Ticket(cb): ∼has_Ticket(cb)
```

- In the next section, "A `Duration Calculus` Model", we shall start refining the descriptions given above.
- We do so in order to handle failures of vehicles to behave as expected and of the electro-mechanics of booths and gates.

4.2.4.2.2. • *A* `Duration Calculus` ***Model***•

- We use the `Duration Calculus` [zcc+mrh2002,olderogdirks2008] extension to `RSL`.
- We abstract the channels of the `RAISE/CSP` model
- to now be Boolean-valued variables.

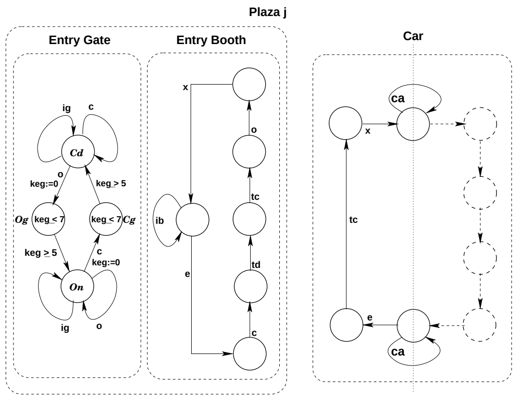
type
  ES = **Bool** [**true**=passing, **false**=not_passing]
  TD = **Bool** [**true**=ticket, **false**=no_ticket]
  G = **Bool** [**true**=open, **false**=closing⌈⌉closed⌈⌉opening]
  XS = **Bool** [**true**=car_has_just_passed, **false**=car_passing⌈⌉no-one_passing
variable
  entry_sensor:ES := **false** ;
  ticket_dispenser:TD := **false** ;
  gate:G := **false** ;
  exit_sensor:XS := **false** ;

154. No matter its position, the gate must be closed within no more than δ_{eg} time units after the entry_sensor has registered that a car is entering the toll booth.
155. A ticket must be in the ticket_dispenser within δ_{et} time units after the entry_sensor has registered that a car is entering the toll booth.
156. The ticket is in the ticket_dispenser at most δ_{tdc} time units
157. The gate must be open within δ_{go} time units after a ticket has been collected.
158. The exit sensor is registering (i.e., is on) the identification of exiting cars and is not registering anything when no car is passing (i.e., is off).

154. $\sim(\lceil \text{entry\_sensor} \rceil \,;\, (\ell = \delta_{eg} \wedge \lceil \text{gate} \rceil))$
155. $\sim(\lceil \text{entry\_sensor} \rceil \,;\, (\ell = \delta_{et} \wedge \lceil \sim\text{ticket\_dispenser} \rceil))$
156. $\Box(\lceil \sim\text{ticket\_dispenser} \rceil \Rightarrow \ell < \delta_{tdc})$
157. $\sim(\lceil \text{ticket\_dispenser} \rceil \,;\, (\lceil \sim\text{ticket\_dispenser} \wedge \sim\text{gate} \rceil \wedge \ell \geq \delta_{go}))$
158. $\Box(\lceil \text{gate=closing} \rceil \Rightarrow \lceil \sim \text{exit\_sensor} \rceil)$

4.2.4.2.3. • *A* `Timed Automata` *Model*•

- A timed automaton [AluDil:94,olderogdirks2008] for a configuration of an entry gate, its entry booth and a car is shown in Fig. 7 on the next page.
- Figure 8 on page 232 shows the a car, an exit booth and its exit gate interactions.
- They are more-or-less "derived" from the example of Sect. 7.5 of [[]Alur & Dill, 1994]AluDil:94 (Pages 42–45).
- The right half of the car timed automaton of Fig. 7 on the next page
  – is to be thought of as the same as the left half of the car timed automaton of Fig. 8 on page 232,
  – cf. the vertical dotted (⋮) line.

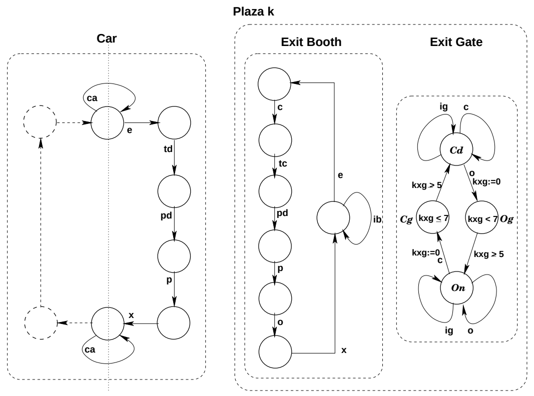


o:open, ig: idle gate, c:close, ib: idle booth, ca:cruise around,e:entry, td:ticket deposit, tc:ticket collection, x:exit
Cd: closed, Cg:closing, On:open, Og:opening

Figure 7: A `timed automata` model of gate, entry booth and car interactions

value
eg,xg:G, eb:EB, xb:XB, vs:V-**set**

System: G×EV×V-**set**×XB×G → **Unit**
System(eg,eb,vs,xb,xg) ≡
  Entry_Gate(eg) ‖ Entry_Booth(eb) ‖
  ‖{Car(obs_CId(c),c)|ci:C,v:C•c ∈ cs} ‖
  Exit_Booth(xb) ‖ Exit_Gate(xg)



ca:cruise around, ib:idle, e:entry, td:ticket deposit, pd:payment display, p: payment, x:exit, c:close, o:open, ig:idle gate

Figure 8: A `timed automata` model of car, exit booth and gate interactions

4.2.5. Fitting

Definition: Fitting.

- *By domain requirements fitting we understand an operation*
  - *which takes n domain requirements prescriptions, d_{r_i} $(i = \{1..n\})$,*
  - *claimed to share m independent sets of tightly related sets of simple entities, actions, events and/or behaviours*
- *and map these into $n{+}m$ domain requirements prescriptions, δ_{r_j} $(j = \{1..n{+}m\})$,*
  - *where m of these, $\delta_{r_{n+k}}$ $(k = \{1..m\})$*
  - *capture the m shared phenomena and concepts*
  - *and the other n prescriptions, δ_{r_ℓ} $(\ell = \{1..n\})$,*
  - *are like the n "input" domain requirements prescriptions, d_{r_i} $(i = \{1..n\})$,*
  - *except that they now,*
  - *(instead of the "more-or-less" shared prescriptions, that are now consolidated in $\delta_{r_{n+k}}$)*
  - *prescribe interfaces between δ_{r_i} and $\delta_{r_{n+k}}$ for $i : \{1..n\}$.*

4.2.5.1. Examples

TO BE WRITTEN

4.3. Interface Requirements

Definition: Interface Requirements.

- *Interface requirements are those requirements*
- *which can on be expressed using professional terms*
- *from both the domain and the machine.*

Thus, by interface requirements we understand

- *the expression of expectations*
- *as to which software-software, or software-hardware interface places (i.e., channels),*
- *inputs and outputs (including the semiotics of these input/outputs)*
- *there shall be in some contemplated computing system.*

Interface requirements can often, usefully, be classified in terms of

- *shared data initialisation requirements,*
- *shared data refreshment requirements,*
- *computational data+control requirements,*
- *man-machine dialogue requirements,*
- *man-machine physiological requirements and*
- *machine-machine dialogue requirements.*

Interface requirements constitute one requirements facet.

- *Other requirements facets are:*
  - *business process reengineering,*
  - *domain requirements and*
  - *machine requirements.*

4.3.1. But First: On Shared Phenomena and Concepts

Definition: Shared Phenomenon or Concept.

- *A shared phenomenon (or concept) is a phenomenon (respectively a concept)*
  - *which is present in some domain (say in the form of facts, knowledge or information)*
  - *and which is also represented in the machine (say in the form of some entity, simple, action, event or behaviour).*
- *A phenomenon of a domain, when shared, becomes a concept of the machine.*

- We shall give some examples – but they are just illustrative.
- Proper narration and formalisation is left to the reader !

4.3.2. Shared Simple Entities

Definition: Shared Simple Entity.

- *By a shared simple entity we mean a simple entity*
  - *which both occurs*
  - *in the* domain *(as a phenomenon or a concept)*
  - *and in the*machine.
- *Simple entities that are shared between the domain and the machine must initially be input to the machine.*
- *Dynamically arising simple entities must likewise be input*
  - *and all such machine entities*
  - *must have their attributes updated, when need arise.*
- *Requirements for shared simple entities*
  - *thus entail requirements for their representation*
  - *and for their human/machine and/or machine/machine transfer dialogue.*

4.3.2.1. Example

- Main shared entities are those of hubs and links.
- Representations of hubs and links "within" the machine
  - necessarily abstracts many of the properties of hubs and links;
  - some (such) attributes may not be represented altogether.
- 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.

4.3.3. Shared Actions

Definition: Shared Action.

- *By a shared action we mean an action*
  - *that can only be partly computed by the* machine.
  - *That is, the* machine,
    - *in order to complete an action,*
    - *may have to inquire with the* domain
    - *(in order, say, to extract some measurable, time-varying simple entity attribute value)*
    - *in order to proceed in its computation.*

4.3.3.1. Example

- In order for a car **driver** to leave an **exit toll booth** the following component actions must take place:
  - (a) the **driver** inserts the electronic pass into the exit toll booth;
  - (b) the **exit toll booth** 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;
  - (c) **exit toll booth** alerts the driver as to the cost and is requested to pay this amount;
  - (d) once the **driver** has paid
  - (e) the **exit booth toll** gate is raised.
- Actions (a,d) are **driver** actions, (b,c,e) are **machine** actions.

4.3.4. Shared Events

Definition: Shared Event.

- *By a shared event we mean*
  - *an event whose occurrence in the* domain
  - *need be communicated to the* machine

  and, vice-versa,
  - *an event whose occurrence in the* machine
  - *need be communicated to the* domain.

4.3.4.1. Examples

- 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 of a car at a toll plaza exit booth is an event
  - that must be communicated to the machine
  - 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.

4.3.5. Shared Behaviours

Definition: Shared Behaviour.

- *By a shared behaviour we mean a behaviour*
  - *many of whose actions and events occur both*
  - *in the* domain
  - *and in the* machine
  - *(in some encoded form, and in the same squence).*

4.3.5.1. Example

- 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 more
    - ∗ toll 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.

4.4. Machine Requirements

Definition: Machine Requirements.

- *Machine requirements are those requirements which, in principle,*
  - *can be expressed without using professional domain terms*
  - *(for which these requirements are established).*
- Thus, by *machine requirements*,
  - we understand *requirements* put specifically to,
  - i.e., expected specifically from, the *machine*.
- We normally analyse machine requirements into
  - *performance requirements*,
  - *dependability requirements*,
  - *maintenance requirements*,
  - *platform requirements* and
  - *documentation requirements*.

4.4.1. An Enumeration of Classes of Machine Requirements

- We shall in these lecture notes not go into any detail about machine requirements.
- But we shall classify machine requirements into a long list of specific kinds of machine requirements.

- Performance
  - Storage
  - Time
  - Software Size
- Dependability
  - Accessability
  - Availability
  - Reliability
  - Robustness
  - Safety
  - Security
- Maintenance
  - Adaptive
  - Corrective
  - Perfective
  - Preventive
- Platforms
  - Development
  - Demonstration
  - Execution
  - Maintenance
- Documentation
- Other

End of Lecture 6: REQUIREMENTS – from Extension "out"

Start of Lecture 11: CLOSING

5. Conclusion

- We discuss a number of issues.

5.1. What Have We Omitted

- Our coverage of domain and requirements engineering has focused on modelling techniques for domain and requirements facets.
- We have omitted the important software engineering tasks of
  - **stakeholder identification and liaison**,
  - **domain** and, to some extents also **requirements**, especially **goal acquisition and analysis**,
  - **terminologisation**, and
  - techniques for **domain and requirements and goal validation and [goal] verification** ($\mathcal{D},\mathcal{R} \models \mathcal{G}$).

5.2. Domain Descriptions Are Not Normative

- A description of, for example,
  - "the" domain of the *New York Stock Exchange* would describe
    - ∗ the set of rules and regulations governing the submission of sell offers and buy bids
    - ∗ as well as rules and regulations for clearing ('matching') sell offers and buy bids.
  - These rules and regulations appears to be quite different from those of the *Tokyo Stock Exchange*.
  - A normative description of stock exchanges would abstract these rules so as to be rather un-informative.
  - And, anyway, rules and regulations changes and business process re-engineering changes entities, actions, events and behaviours.
  - For any given software development one may thus have to rewrite parts of existing domain descriptions, or construct an entirely new such description.

5.3. "Requirements Always Change"

- This claim is often used as a hidden excuse for not doing a proper, professional job of requirements prescription, let alone "deriving" them, as we advocate, from domain descriptions.
- Instead we now make the following counterclaims
  - [1] "domains are far more stable than requirements" and
  - [2] "requirements changes arise more as a result of business process re-engineering than as a result of changing stakeholder ideas".

- Closer studies of a number of domain descriptions,
  - for example of a *financial service industry*,
  - reveals that the domain in terms of which an "ever expanding" variety of financial products are offered,
  - are, in effect, based on a small set of very basic domain functions which have been offered for well-nigh centuries !
- We thus claim that
  - thoroughly developed domain descriptions and
  - thoroughly "derived" requirements prescriptions
  - tend to stabilise the requirements re-design,
  - but never alleviate it.

5.4. What Can Be Described and Prescribed

- The issue of *"what can be described"* has been a constant challenge to philosophers.
  - Bertrand Russell covers, in a 1919 publication, *Theory of Descriptions*, and
  - in [Philosophy of Mathematics] a revision, as *The Philosophy of Logical Atomism*.
- The issue is not that straightforward.
- In two recent papers we try to broach the topic from the point of view of the kind of domain engineering presented in these lectures.
- Our approach is simple; perhaps too simple !
- We can describe what can be observed.

- We do so,
  - first by postulating types of observable phenomena and of derived concepts;
  - then by the introduction of *observer* functions and by axioms over these, that is, over values of postulated types and observers.
  - To this we add defined functions; usually described by pre/post-conditions.
    - ∗ The narratives refer to the "real" phenomena
    - ∗ whereas the formalisations refer to related phenomenological concepts.
- The narrative/formalisation problem is that one can 'describe' phenomena without always knowing how to formalise them.

5.5. What Have We Achieved – and What Not

- Earlier we made some claims.
- We think we have substantiated them all, albeit ever so briefly.
- Each of the domain facets
  - (intrinsics,
  - support technologies,
  - rules and regulations,
  - scripts [licenses and contracts],
  - management and organisation and
  - human behaviour)
- and each of the requirements facets
  - (projection,
  - instantiation,
  - determination,
  - extension and
  - fitting)
- provide rich grounds for both specification methodology studies and and for more theoretical studies.

5.6. Relation to Other Work

- The most obvious 'other' work is that of Michael jackson's [Problem Frames].
  - In that book Jackson, like is done here,
    - ∗ departs radically from conventional requirements engineering.
    - ∗ In his approach understandings of the domain, the requirements and possible software designs
    - ∗ are arrived at, not hierarchically, but in parallel, interacting streams of decomposition.

- Thus the 'Problem Frame' development approach iterates between concerns of
  - domains,
  - requirements and
  - software design.
- "Ideally" our approach pursues
  - domain engineering
  - prior to requirements engineering,
  - and, the latter, prior to software design.
- But see next.

- The recent book [Axel van Lamsweerde]
  - appears to represent the most definitive work on Requirements Engineering today.
  - Much of its requirements and goal acquisition and analysis techniques
  - carries over to main aspects of domain acquisition and analysis techniques
  - and the goal-related techniques of [Lamsweerde] apply to determining which
    - projections,
    - instantiation,
    - determination and
    - extension operations

    to perform on domain descriptions.

5.7. "Ideal" Versus Real Developments

- The term 'ideal' has been used in connection with 'ideal development' from domain to requirements.
- We now discuss that usage.
- Ideally software development could proceed
  - from developing domain descriptions
  - via "deriving" requirements prescriptions
  - to software design,

  each phase involving extensive
  - formal specifications,
  - verifications (formal testing, model checking and theorem proving) and validation.

- More realistically
  - less comprehensive domain description development (D)
  - may alternate with both requirements development (R) work
  - and with software design (S) –
  - in some
    - controlled,
    - contained
    - iterated and
    - "spiralling"

    manner
  - and such that it is at all times clear which development step is what: $\mathcal{D}$, $\mathcal{R}$ or $\mathcal{S}$!

5.8. Description Languages

- We have used the RSL specification language, for the formalisations of this report,
- but any of the model-oriented approaches and languages offered by
  - Alloy,
  - B, Event B,
  - RAISE,
  - VDM and
  - Z,

  should work as well.

- No single one of the above-mentioned formal specification languages, however, suffices.
- Often one has to carefully combine the above with elements of
  - Petri Nets,
  - CSP,
  - MSC,
  - Statecharts,

  and/or some temporal logic, for example
  - either DC or
  - TLA+.
- Research into how such diverse textual and diagrammatic languages can be combined is ongoing.

5.9. Entailments

- $\mathcal{D},\mathcal{R} \models \mathcal{G}$
  * From the $\mathcal{D}$omain and the $\mathcal{R}$equirements we can reason that the $\mathcal{G}$oals are met.
- $\mathcal{D},\mathcal{S} \models \mathcal{R}$
  * In a proof of correctness of $\mathcal{S}$oftware design with respect to $\mathcal{R}$equirements prescriptions one often has to refer to assumptions about the $\mathcal{D}$omain.
  * Formalising our understandings of the $\mathcal{D}$omain, the $\mathcal{R}$equirements and the $\mathcal{S}$oftware design enables proofs that the software is right and the formalisation of the "derivation" of $\mathcal{R}$equirements from $\mathcal{D}$omain specifications help ensure that it is the right software [Boehm81].

5.10. Domain Versus Ontology Engineering

- In the information science community an ontology is a
  - "formal, explicit specification of a shared conceptualisation".
- Most of the information science ontology work seems aimed primarily at axiomatisations of properties of entities.
- Apart from that there are many issues of "ontological engineering" that are similar to the triptych kind of domain engineering;
  - but then, we claim, that domain engineering goes well beyond ontological engineering and makes free use of whatever formal specification languages are needed.

6. Bibliographical Notes

6.1. Description Languages

- Besides using
  - as precise a subset of a national language, as here English, as possible, and in enumerated expressions and statements,
  - we have "paired" such narrative elements with corresponding enumerated clauses of a formal specification language.
- We have been using the RAISE Specification Language, RSL in our formal texts.
- But any of the model-oriented approaches and languages offered by
  - Alloy,
  - CafeOBJ [futatsugi2000a],
  - Event B,
  - VDM and
  - Z,

  should work as well.

- No single one of the above-mentioned formal specification languages, however, suffices.
- Often one has to carefully combine the above with elements of
  - Petri Nets,
  - CSP: Communicating Sequential Processes,
  - MSC: Message Sequence Charts,
  - Statecharts,
  - and some temporal logic, for example
    * DC: Duration Calculus
    * or TLA+.
  - And even then !

End of Lecture 11: CLOSING

Start of Lecture 7: RSL: Types

1. An RSL Primer

1.1. Types

1.1.1. Type Expressions

- Type expressions are expressions whose values are types, that is,
- possibly infinite sets of values (of "that" type).

1.1.1.1. Atomic Types

- Atomic types have (atomic) values.
- That is, values which we consider to have no proper constituent (sub-)values,
- i.e., cannot, to us, be meaningfully "taken apart".

type
[1] **Bool**
[2] **Int**
[3] **Nat**
[4] **Real**
[5] **Char**
[6] **Text**

1. The Boolean type of truth values **false** and **true**.
2. The integer type on integers ..., –2, –1, 0, 1, 2, ...
3. The natural number type of positive integer values 0, 1, 2, ...
4. The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period ("."), followed by a natural number (the fraction).
5. The character type of character values ″a″, ″b″, ...
6. The text type of character string values ″aa″, ″aaa″, ..., ″abc″, ...

Example 1 .... **Basic Net Attributes:**

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

type
  MAX = **Nat**
  LEN = **Real**
  is_Two_Way_Link = **Bool**
value
  obs_Max: (H|L) $\rightarrow$ MAX
  obs_Len: L $\rightarrow$ LEN
  is_two_way_link: L $\rightarrow$ is_Two_Way_Link
  is_two_way_link(l) $\equiv \exists\ l\sigma{:}L\Sigma \cdot l\sigma \in$ obs_HΣ(l)$\wedge$**card** $l\sigma$=2

.... **End of Example 1**

1.1.1.2. Composite Types

- Composite types have composite values.
- That is, values which we consider to have proper constituent (sub-)values,
- i.e., can, to us, be meaningfully "taken apart".

[7] A-**set**	[13] $A \rightarrow B$
[8] A-**infset**	[14] $A \overset{\sim}{\rightarrow} B$
[9] $A \times B \times \ldots \times C$	[15] (A)
[10] A^*	[16] A | B | ... | C
[11] A^ω	[17] mk_id(sel_a:A,...,sel_b:B)
[12] $A \underset{m}{\rightarrow} B$	[18] sel_a:A ... sel_b:B

Example 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
  obs_HIs: L → HI-set
  obs_LIs: H → LI-set
  obs_HΣ: H → HT-set
  set_HΣ: H × HΣ → H
```

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

```
type
  N = H-set × L-set
  HT = LI × HI × LI
  LT′ = HI × LI × HI
```

.... **End of Example 2**

1.1.2. Type Definitions

1.1.2.1. Concrete Types

- Types can be concrete
- in which case the structure of the type
- is specified by type expressions:

type
  A = Type_expr
schematic examples:
  A1 = B1-**set**, A2 = B1-**infset**
  $A3 = B2 \times C1 \times D1$
  $B1 = E^*$, $B2 = E^\omega$
  $C1 = F \underset{m}{\rightarrow} G$
  $D1 = H \rightarrow J$, $D2 = H \overset{\sim}{\rightarrow} J$
  K = L | M

Example 3 . . . . 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.

  type
    [sorts] N_α, H, L, HI, LI
  value
    obs_Hs: $N_\alpha \rightarrow$ H-**set**
    obs_Ls: $N_\alpha \rightarrow$ L-**set**
  axiom
    $\forall$ n:N_α $\cdot$ **card** obs_Hs(n)>0 $\Rightarrow$ **card** obs_Ls(n)$\geq$1 $\wedge$ ...

- **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: $N_\beta \rightarrow$ **Bool**
    wf_N_β(hs,ls) $\equiv$ **card** hs>1 $\Rightarrow$ **card** ls>0 ....
    inject_N_β: $N_\alpha \xrightarrow{\sim} N_\beta$ **pre**: wf_N_β(hs,ls)
    inject_N_β(n_α) $\equiv$ (obs_Hs(n_α),obs_Ls(n_α))

- **Cartesians + Maps + Wellformedness:** Or a net can be described

  a as a triple of b-c-d:

  b hubs (modelled as a map from hub identfiers to hubs),

  c links (modelled as a map from link identfiers to links), and

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

  – [b,c] *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.

. . . . **End of Example 3**

```
[1] Type_name = Type_expr /* without |s or subtypes */
[2] Type_name = Type_expr_1 | Type_expr_2 | ... | Type_expr_n
[3] Type_name ==
      mk_id_1(s_a1:Type_name_a1,...,s_ai:Type_name_ai) |
      ... |
      mk_id_n(s_z1:Type_name_z1,...,s_zk:Type_name_zk)
[4] Type_name :: sel_a:Type_name_a ... sel_z:Type_name_z
[5] Type_name = {| v:Type_name′ • 𝒫(v) |}
```

- where a form of [2–3] is provided by combining the types:

```
Type_name = A | B | ... | Z
A == mk_id_1(s_a1:A_1,...,s_ai:A_i)
B == mk_id_2(s_b1:B_1,...,s_bj:B_j)
...
Z == mk_id_n(s_z1:Z_1,...,s_zk:Z_k)

axiom
  ∀ a1:A_1, a2:A_2, ..., ai:Ai •
    s_a1(mk_id_1(a1,a2,...,ai))=a1 ∧ s_a2(mk_id_1(a1,a2,...,ai))=a2 ∧
    ... ∧ s_ai(mk_id_1(a1,a2,...,ai))=ai ∧
  ∀ a:A • let mk_id_1(a1′,a2′,...,ai′) = a in
    a1′ = s_a1(a) ∧ a2′ = s_a2(a) ∧ ... ∧ ai′ = s_ai(a) end
```

Example 4 . . . . Net Record Types: Insert Links:

7. 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.
8. From a net one can remove a link.[5] The removal command specifies a link identifier.

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

```
type
  7      Insert == Ins(s_ins:Ins)
  7      Ins = 2xHubs | 1x1nH | 2nHs
  7(a)     2xHubs == 2oldH(s_hi1:HI,s_l:L,s_hi2:HI)
  7(b)     1x1nH == 1oldH1newH(s_hi:HI,s_l:L,s_h:H)
  7(c)     2nHs == 2newH(s_h1:H,s_l:L,s_h2:H)
  8      Remove == Rmv(s_li:LI)
axiom
  7(d)   ∀ 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″)}
```

Example 16 on page 351 presents the semantics functions for *int_Insert* and *int_Remove*.

. . . . **End of Example 4**

1.1.2.2. Subtypes

- In RSL, each type represents a set of values. Such a set can be delimited by means of predicates.
- The set of values b which have type B and which satisfy the predicate $\mathcal{P}$, constitute the subtype A:

type
  A = {| b:B • $\mathcal{P}$(b) |}

Example 5 . . . . Net Subtypes:

- In Example 3 on page 274 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 3
  - 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 wellformedness 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** obs_Hs(n)≥0 ∧ **card** obs_Ls(n)≥0 ∧
    axioms 2.–3., 5.–6., and 10., (Page 14)

- In the **Cartesians + Wellformedness** part of Example 3
  - 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 3
  - 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 →m (LI →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*.

. . . . **End of Example 5**

1.1.2.3. Sorts — Abstract Types

- Types can be (abstract) sorts
- in which case their structure is not specified:

```
type
  A, B, ..., C
```

Example 6 . . . . **Net Sorts:**

- In formula lines of Examples 3–5
- we have indicated those **type** clauses which define *sorts*,
- by bracketed [sorts] literals.

. . . . **End of Example 6**

End of Lecture 7: RSL: Types

Start of Lecture 8: RSL: Values & Operations

1.2. Concrete RSL Types: Values and Operations

1.2.1. Arithmetic

type
  Nat, Int, Real
value
  $+,-,*$: $\mathrm{Nat}\times\mathrm{Nat}\to\mathrm{Nat} \mid \mathrm{Int}\times\mathrm{Int}\to\mathrm{Int} \mid \mathrm{Real}\times\mathrm{Real}\to\mathrm{Real}$
  $/$: $\mathrm{Nat}\times\mathrm{Nat}\overset{\sim}{\to}\mathrm{Nat} \mid \mathrm{Int}\times\mathrm{Int}\overset{\sim}{\to}\mathrm{Int} \mid \mathrm{Real}\times\mathrm{Real}\overset{\sim}{\to}\mathrm{Real}$
  $<,\leq,=,\neq,\geq,>$ $(\mathrm{Nat}|\mathrm{Int}|\mathrm{Real}) \times (\mathrm{Nat}|\mathrm{Int}|\mathrm{Real}) \to \mathrm{Bool}$

1.2.2. Set Expressions

1.2.2.1. Set Enumerations

Let the below a's denote values of type A, then the below designate simple set enumerations:

$\{\{\}, \{a\}, \{e_1,e_2,...,e_n\}, ...\} \subseteq$ A-**set**
$\{\{\}, \{a\}, \{e_1,e_2,...,e_n\}, ..., \{e_1,e_2,...\}\} \subseteq$ A-**infset**

Example 7 . . . . 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.
- For 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 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 student 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.

. . . . **End of Example 7**

1.2.2.2. Set Comprehension

- The expression, last line below, to the right of the ≡, expresses set comprehension.
- The expression "builds" the set of values satisfying the given predicate.
- It is abstract in the sense that it does not do so by following a concrete algorithm.

type
  A, B
  P = A → **Bool**
  Q = A $\xrightarrow{\sim}$ B
value
  comprehend: A-**infset** × P × Q → B-**infset**
  comprehend(s,P,Q) ≡ { Q(a) | a:A • a ∈ s ∧ P(a)}

Example 8 . . . . **Set Comprehensions:**
Item 52 on page 43, the *wf_N(hs,ls,g)* wellformedness predicate definition, includes:

type
51(a). PLAN = HI $\overrightarrow{m}$ LHIM
51(b). LHIM = LI $\overrightarrow{m}$ HI-**set**
value
52(c). no_junk: PLAN → **Bool**
52(c). no_junk(plan) ≡ **dom** plan = ∪{**rng**(plan(hi))|hi:HI•hi ∈ **dom** plan}

. . . . **End of Example 8**

1.2.3. Cartesian Expressions
1.2.3.1. Cartesian Enumerations

- Let e range over values of Cartesian types involving A, B, . . ., C,
- then the below expressions are simple Cartesian enumerations:

type
  A, B, ..., C
  A × B × ... × C
value
  (e1,e2,...,en)

Example 9 .... 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 × ...
```

.. End of Example 9

1.2.4. List Expressions

1.2.4.1. List Enumerations

- Let a range over values of type A,
- then the below expressions are simple list enumerations:

$$\{\langle\rangle, \langle e\rangle, ..., \langle e1,e2,...,en\rangle, ...\} \subseteq A^*$$
$$\{\langle\rangle, \langle e\rangle, ..., \langle e1,e2,...,en\rangle, ..., \langle e1,e2,...,en,... \rangle, ...\} \subseteq A^\omega$$

$$\langle\ a\_i\ ..\ a\_j\ \rangle$$

- The last line above assumes a_i and a_j to be integer-valued expressions.
- It then expresses the set of integers from the value of e_i to and including the value of e_j.
- If the latter is smaller than the former, then the list is empty.

1.2.4.2. List Comprehension

- The last line below expresses list comprehension.

```
type
  A, B, P = A → Bool, Q = A ~→ B
value
  comprehend: A^ω × P × Q ~→ B^ω
  comprehend(l,P,Q) ≡
    ⟨ Q(l(i)) | i in ⟨1..len l⟩ · P(l(i))⟩
```

Example 10 . . . . 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 ⇒
      obs_HIs(l(i)) ∩ obs_HIs(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 ⇒ obs_LI(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•obs_Ls(n)} ∪
          ∪ {pr⌢pr′|pr,pr′:PR•{pr,pr′}⊆prs∧obs_HIs(r(len pr))∩obs_HIs(pr′(1))≠{}}
    prs end

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

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

. . . . **End of Example 10**

1.2.5. Map Expressions

1.2.5.1. Map Enumerations

- Let (possibly indexed) u and v range over values of type $T1$ and $T2$, respectively,
- then the below expressions are simple map enumerations:

```
type
  T1, T2
  M = T1 →m T2
value
  u,u1,u2,...,un:T1, v,v1,v2,...,vn:T2
  {[], [u↦v], ..., [u1↦v1,u2↦v2,...,un↦vn],...} ⊆ M
```

1.2.5.2. Map Comprehension

- The last line below expresses map comprehension:

```
type
  U, V, X, Y
  M = U →m V
  F = U →~ X
  G = V →~ Y
  P = U → Bool
value
  comprehend: M×F×G×P → (X →m Y)
  comprehend(m,F,G,P) ≡
    [ F(u) ↦ G(m(u)) | u:U • u ∈ dom m ∧ P(u) ]
```

Example 11 . . . . Concrete Net Type Construction:

- We Define a function *con[struct]_N_γ* (of the **Cartesians + Maps + Wellformedness** part of Example 3.
  - The base of the construction is the fully abstract sort definition of N_α in the **Sorts + Observers + Axioms** part of Example 3 – 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 3.
  - First we recall the ssential types of that N_γ.

```
type
  N_γ = HUBS × LINKS × GRAPH
  HUBS = HI →_m H
  LINKS = LI →_m L
  GRAPH = HI →_m (LI →_m HI)
value
  con_N_γ: N_α → N_γ
  con_N_γ(n_α) ≡
    let hubs = [obs_HI(h) ↦ h | h:H • h ∈ obs_Hs(n_α)],
       links = [obs_LI(h) ↦ l | l:L • l ∈ obs_Ls(n_α)],
       graph = [obs_HI(h) ↦ [obs_LI(l) ↦ ι(obs_HIs(l)\{obs_HI(h)})
                                  | l:L • l ∈ obs_Ls(n_α)∧li ∈ obs_LIs(h)]
                          | H:h • h ∈ obs_Hs(n_α)] in
       (hubs.links,graph) end

  ι: A-set ~→ 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.–3., 5.–6., and 10. (Page 14) ⇒ wf_N_γ(con_N_γ(n_α

. . . . **End of Example 11**

1.2.6. Set Operations

1.2.6.1. Set Operator Signatures

```
value
  ∈: A × A-infset → Bool
  ∉: A × A-infset → Bool
  ∪: A-infset × A-infset → A-infset
  ∪: (A-infset)-infset → A-infset
  ∩: A-infset × A-infset → A-infset
  ∩: (A-infset)-infset → A-infset
  \: A-infset × A-infset → A-infset
  ⊂: A-infset × A-infset → Bool
  ⊆: A-infset × A-infset → Bool
  =: A-infset × A-infset → Bool
  ≠: A-infset × A-infset → Bool
  card: A-infset ~→ Nat
```

1.2.6.2. Set Examples

examples

$a \in \{a,b,c\}$
$a \notin \{\}$, $a \notin \{b,c\}$
$\{a,b,c\} \cup \{a,b,d,e\} = \{a,b,c,d,e\}$
$\cup\{\{a\},\{a,b\},\{a,d\}\} = \{a,b,d\}$
$\{a,b,c\} \cap \{c,d,e\} = \{c\}$
$\cap\{\{a\},\{a,b\},\{a,d\}\} = \{a\}$
$\{a,b,c\} \setminus \{c,d\} = \{a,b\}$
$\{a,b\} \subset \{a,b,c\}$
$\{a,b,c\} \subseteq \{a,b,c\}$
$\{a,b,c\} = \{a,b,c\}$
$\{a,b,c\} \neq \{a,b\}$
card $\{\} = 0$, **card** $\{a,b,c\} = 3$

1.2.7. Cartesian Operations

type
A, B, C
g0: G0 = A × B × C
g1: G1 = (A × B × C)
g2: G2 = (A × B) × C
g3: G3 = A × (B × C)

value
va:A, vb:B, vc:C, vd:D
(va,vb,vc):G0,
(va,vb,vc):G1
((va,vb),vc):G2
(va3,(vb3,vc3)):G3

decomposition expressions
let (a1,b1,c1) = g0,
  (a1′,b1′,c1′) = g1 **in** .. **end**
let ((a2,b2),c2) = g2 **in** .. **end**
let (a3,(b3,c3)) = g3 **in** .. **end**

1.2.8. List Operations
1.2.8.1. List Operator Signatures

value
hd: $A^\omega \xrightarrow{\sim} A$
tl: $A^\omega \xrightarrow{\sim} A^\omega$
len: $A^\omega \xrightarrow{\sim}$ **Nat**
inds: $A^\omega \rightarrow$ **Nat-infset**
elems: $A^\omega \rightarrow$ A-**infset**
.(.): $A^\omega \times$ **Nat** $\xrightarrow{\sim} A$
$\frown$: $A^* \times A^\omega \rightarrow A^\omega$
=: $A^\omega \times A^\omega \rightarrow$ **Bool**
≠: $A^\omega \times A^\omega \rightarrow$ **Bool**

1.2.8.2. List Operation Examples

examples
hd$\langle a1,a2,...,am\rangle=a1$
tl$\langle a1,a2,...,am\rangle=\langle a2,...,am\rangle$
len$\langle a1,a2,...,am\rangle=m$
inds$\langle a1,a2,...,am\rangle=\{1,2,...,m\}$
elems$\langle a1,a2,...,am\rangle=\{a1,a2,...,am\}$
$\langle a1,a2,...,am\rangle(i)=ai$
$\langle a,b,c\rangle\frown\langle a,b,d\rangle = \langle a,b,c,a,b,d\rangle$
$\langle a,b,c\rangle=\langle a,b,c\rangle$
$\langle a,b,c\rangle \neq \langle a,b,d\rangle$

1.2.9. Map Operations

1.2.9.1. Map Operator Signatures and Map Operation Examples

value

$\mathrm{m(a)}: \mathrm{M} \to \mathrm{A} \overset{\sim}{\to} \mathrm{B},\ \mathrm{m(a)} = \mathrm{b}$

dom: $\mathrm{M} \to \mathrm{A}$-**infset** [domain of map]
dom $[\mathrm{a1}\mapsto\mathrm{b1},\mathrm{a2}\mapsto\mathrm{b2},...,\mathrm{an}\mapsto\mathrm{bn}] = \{\mathrm{a1},\mathrm{a2},...,\mathrm{an}\}$

rng: $\mathrm{M} \to \mathrm{B}$-**infset** [range of map]
rng $[\mathrm{a1}\mapsto\mathrm{b1},\mathrm{a2}\mapsto\mathrm{b2},...,\mathrm{an}\mapsto\mathrm{bn}] = \{\mathrm{b1},\mathrm{b2},...,\mathrm{bn}\}$

$\dagger$: $\mathrm{M} \times \mathrm{M} \to \mathrm{M}$ [override extension]
$[\mathrm{a}\mapsto\mathrm{b},\mathrm{a}'\mapsto\mathrm{b}',\mathrm{a}''\mapsto\mathrm{b}''] \dagger [\mathrm{a}'\mapsto\mathrm{b}'',\mathrm{a}''\mapsto\mathrm{b}'] = [\mathrm{a}\mapsto\mathrm{b},\mathrm{a}'\mapsto\mathrm{b}'',\mathrm{a}''\mapsto\mathrm{b}']$

$\cup$: $\mathrm{M} \times \mathrm{M} \to \mathrm{M}$ [merge $\cup$]
$[\mathrm{a}\mapsto\mathrm{b},\mathrm{a}'\mapsto\mathrm{b}',\mathrm{a}''\mapsto\mathrm{b}''] \cup [\mathrm{a}'''\mapsto\mathrm{b}'''] = [\mathrm{a}\mapsto\mathrm{b},\mathrm{a}'\mapsto\mathrm{b}',\mathrm{a}''\mapsto\mathrm{b}'',\mathrm{a}'''\mapsto\mathrm{b}''']$

$\backslash$: $\mathrm{M} \times \mathrm{A}$-**infset** $\to \mathrm{M}$ [restriction by]
$[\mathrm{a}\mapsto\mathrm{b},\mathrm{a}'\mapsto\mathrm{b}',\mathrm{a}''\mapsto\mathrm{b}'']\backslash\{\mathrm{a}\} = [\mathrm{a}'\mapsto\mathrm{b}',\mathrm{a}''\mapsto\mathrm{b}'']$

$/$: $\mathrm{M} \times \mathrm{A}$-**infset** $\to \mathrm{M}$ [restriction to]
$[\mathrm{a}\mapsto\mathrm{b},\mathrm{a}'\mapsto\mathrm{b}',\mathrm{a}''\mapsto\mathrm{b}'']/\{\mathrm{a}',\mathrm{a}''\} = [\mathrm{a}'\mapsto\mathrm{b}',\mathrm{a}''\mapsto\mathrm{b}'']$

$=,\neq$: $\mathrm{M} \times \mathrm{M} \to$ **Bool**

$\circ$: $(\mathrm{A} \underset{m}{\to} \mathrm{B}) \times (\mathrm{B} \underset{m}{\to} \mathrm{C}) \to (\mathrm{A} \underset{m}{\to} \mathrm{C})$ [composition]
$[\mathrm{a}\mapsto\mathrm{b},\mathrm{a}'\mapsto\mathrm{b}'] \circ [\mathrm{b}\mapsto\mathrm{c},\mathrm{b}'\mapsto\mathrm{c}',\mathrm{b}''\mapsto\mathrm{c}''] = [\mathrm{a}\mapsto\mathrm{c},\mathrm{a}'\mapsto\mathrm{c}']$

End of Lecture 8: RSL: Values & Operations

Start of Lecture 9: RSL: Logic, Λ-Calculus, Fctl. Specs.

1.3. The RSL Predicate Calculus
1.3.1. Propositional Expressions

- Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values (**true** or **false** [or **chaos**]).
- Then:

false, **true**
a, b, ..., c $\sim$a, a$\wedge$b, a$\vee$b, a$\Rightarrow$b, a=b, a$\neq$b

- are propositional expressions having Boolean values.
- $\sim$, $\wedge$, $\vee$, $\Rightarrow$, $=$ and $\neq$ are Boolean connectives (i.e., operators).
- They can be read as: *not*, *and*, *or*, *if then* (or *implies*), *equal* and *not equal*.

1.3.2. Simple Predicate Expressions

- Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values,
- let x, y, ..., z (or term expressions) designate non-Boolean values
- and let i, j, . . ., k designate number values,
- then:

false, **true**
a, b, ..., c
$\sim$a, a$\wedge$b, a$\vee$b, a$\Rightarrow$b, a=b, a$\neq$b
x=y, x$\neq$y,
i<j, i$\leq$j, i$\geq$j, i$\neq$j, i$\geq$j, i>j

- are simple predicate expressions.

1.3.3. Quantified Expressions

- Let X, Y, . . ., C be type names or type expressions,
- and let $\mathcal{P}(x)$, $\mathcal{Q}(y)$ and $\mathcal{R}(z)$ designate predicate expressions in which x, y and z are free.
- Then:

$\forall$ x:X $\cdot$ $\mathcal{P}(x)$
$\exists$ y:Y $\cdot$ $\mathcal{Q}(y)$
$\exists$! z:Z $\cdot$ $\mathcal{R}(z)$

- are quantified expressions — also being predicate expressions.

Example 12 . . . . Predicates Over Net Quantities:

- From earlier examples we show some predicates:
- Example 1: Right hand side of function definition *is_two_way_link(l)*:
  $\exists$ *lσ:LΣ* $\cdot$ *lσ* $\in$ *obs_HΣ(l)*$\wedge$**card** *lσ=2*

- Example 3:
  - The **Sorts + Observers + Axioms** part:
    - ∗ Right hand side of the wellformedness function *wf_N(n)* definition:
      $\forall$ *n:N* · **card** *obs_Hs(n)*$\geq$*2* $\wedge$ **card** *obs_Ls(n)*$\geq$*1* $\wedge$ **axioms** 2.–3., 5.–6., and 10., (Page 14)
    - ∗ Right hand side of the wellformedness function *wf_N(hs,ls)* definition:
      card *hs*$\geq$*2* $\wedge$ **card** *ls*$\geq$*1* ...

- The **Cartesians + Maps + Wellformedness** part:
  - ∗ Right hand side of the *wf_HUBS* wellformedness function definition:
    $\forall$ *hi:HI* · *hi* $\in$ **dom** *hubs* $\Rightarrow$ *obs_HIhubs(hi)=hi*
  - ∗ Right hand side of the *wf_LINKS* wellformedness function definition:
    $\forall$ *li:LI* · *li* $\in$ **dom** *links* $\Rightarrow$ *obs_LIlinks(li)=li*
  - ∗ Right hand side of the *wf_N(hs,ls,g)* wellformedness function definition:
    [*c*] **dom** *hs* = **dom** *g* $\wedge$
    [*d*] $\cup$ {**dom** *g(hi)*|*hi:HI* · *hi* $\in$ **dom** *g*} = **dom** *links* $\wedge$
    [*e*] $\cup$ {**rng** *g(hi)*|*hi:HI* · *hi* $\in$ **dom** *g*} = **dom** *g* $\wedge$
    [*f*] $\forall$ *hi:HI* · *hi* $\in$ **dom** *g* $\Rightarrow$ $\forall$ *li:LI* · *li* $\in$ **dom** *g(hi)* $\Rightarrow$ *(g(hi))(li)*$\neq$*hi*
    [*g*] $\forall$ *hi:HI* · *hi* $\in$ **dom** *g* $\Rightarrow$ $\forall$ *li:LI* · *li* $\in$ **dom** *g(hi)* $\Rightarrow$
      $\exists$ *hi′:HI* · *hi′* $\in$ **dom** *g* $\Rightarrow$ $\exists$! *li:LI* · *li* $\in$ **dom** *g(hi)* $\Rightarrow$
        (g(hi))(li) = *hi′* $\wedge$ *(g(hi′))(li)* = *hi*

....**End of Example 12**

1.4. λ-Calculus + Functions

1.4.1. The λ-Calculus Syntax

type /∗ A BNF Syntax: ∗/
  ⟨L⟩ ::= ⟨V⟩ | ⟨F⟩ | ⟨A⟩ | (⟨A⟩)
  ⟨V⟩ ::= /∗ variables, i.e. identifiers ∗/
  ⟨F⟩ ::= λ⟨V⟩ · ⟨L⟩
  ⟨A⟩ ::= (⟨L⟩⟨L⟩)
value /∗ Examples ∗/
  ⟨L⟩: e, f, a, ...
  ⟨V⟩: x, ...
  ⟨F⟩: λ x · e, ...
  ⟨A⟩: f a, (f a), f(a), (f)(a), ...

1.4.2. Free and Bound Variables

Let x, y be variable names and e, f be λ-expressions.

- ⟨V⟩: Variable x is free in x.
- ⟨F⟩: x is free in $\lambda y \cdot e$ if $x \neq y$ and x is free in e.
- ⟨A⟩: x is free in $f(e)$ if it is free in either f or e (i.e., also in both).

1.4.3. Substitution

Substitution of an expression N for all free free x in M is expressed: **subst**([N/x]M).

- **subst**([N/x]x) ≡ N;
- **subst**([N/x]a) ≡ a,

  for all variables a≠ x;
- **subst**([N/x](P Q)) ≡ (**subst**([N/x]P) **subst**([N/x]Q));
- **subst**([N/x](λx·P)) ≡ λy·P;
- **subst**([N/x](λy·P)) ≡ $\lambda y\cdot$ **subst**([N/x]P),

  if x≠y and y is not free in N or x is not free in P;
- **subst**([N/x](λy·P)) ≡ λz·**subst**([N/z]**subst**([z/y]P)),

  if y≠x and y is free in N and x is free in P

  (where z is not free in (N P)).

1.4.4. α-Renaming and β-Reduction

- α-renaming: λx·M

  If x, y are distinct variables then replacing x by y in λx·M results in λy·**subst**([y/x]M). We can rename the formal parameter of a λ-function expression provided that no free variables of its body M thereby become bound.
- β-reduction: (λx·M)(N)

  All free occurrences of x in M are replaced by the expression N provided that no free variables of N thereby become bound in the result. (λx·M)(N) ≡ **subst**([N/x]M)

Example 13 . . . . 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.
- 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 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
  $\forall$ ts:TS $\cdot$ $\exists$ tmin,tmax:T: tmin $\in$ ts $\wedge$ tmax $\in$ ts $\wedge$ $\forall$ t:T $\cdot$ t $\in$ ts $\Rightarrow$ tmin $\leq$ t $\leq$ tmax
  [that is: ts = $\{\mathcal{MIN}$(ts)..$\mathcal{MAX}$(ts)$\}$]
type
  VP = V $\overrightarrow{m}$ Pos
  TF′ = T $\rightarrow$ 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$\rightarrow$ N $\rightarrow$ **Bool**
  wf_TF(tf)(n) $\equiv$ ...
  $\mathcal{DOMAIN}$: TF $\rightarrow$ TS
  $\mathcal{MIN}$, $\mathcal{MAX}$: TS $\rightarrow$ 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 $\times$ T $\times$ TF $\rightarrow$ TF $\rightarrow$ 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: $\mathcal{MIN}$($\mathcal{DOMAIN}$(tf)$\leq$t$\leq$$\mathcal{MAX}$($\mathcal{DOMAIN}$(tf)) $\wedge$
  $\forall$ t′:T $\cdot$ t′ $\in$ $\mathcal{DOMAIN}$(tf) $\Rightarrow$ v $\notin$ **dom** tf(t′) $\wedge$
  $\mathcal{MIN}$($\mathcal{DOMAIN}$(vtf)) = t $\wedge$
  $\forall$ t′:T·t′ $\in$ $\mathcal{DOMAIN}$(vtf) $\Rightarrow$ **dom** vtf(t′)={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: tf′ = λt″·
        let ts = 𝒟𝒪ℳ𝒜ℐ𝒩(tf) ∪ 𝒟𝒪ℳ𝒜ℐ𝒩(vtf) in
        if ℳℐ𝒩(ts) ≤ t″ ≤ ℳ𝒜𝒳(ts)
          then
            ((t″<t) → tf(t″),
            (t″≥t) → [ v↦ if v≠v then (tf(t))(v′) else (vtf(t))(v) end
                       | v′:V·v′ ∈ vehicles(tf) ])
          else chaos end
        end
  assumption: wf_TF(vtf)∧wf_TF(tf)
  theorem: wf_TF(tf′)
value
  vehicles: TF → V-set
  vehicles(tf) ≡ {v|t:T,v:V·t ∈ 𝒟𝒪ℳ𝒜ℐ𝒩(tf)∧v ∈ dom tf(t)}
```

1.4.5. Function Signatures

For sorts we may want to postulate some functions:

```
type
  A, B, ..., C
value
  obs_B: A → B
  ...
  obs_C: A → C
```

- These functions cannot be defined.
- Once a domain is presented
  - in which sort A and sorts or types B, ... and C occurs
  - these observer functions can be demonstrated.

Example 14 .... **Hub and Link Observers:**

- Let a net with several hubs and links be presented.
- Now observer functions
  - obs_Hs and
  - obs_Ls

  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
  - obs_HI and
  - obs_LI

  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.

.... **End of Example 14**

1.4.6. Function Definitions

Functions can be defined explicitly:

type
  A, B
value
  f: A $\rightarrow$ B [a total function]
  f(a_expr) $\equiv$ b_expr

  g: A $\overset{\sim}{\rightarrow}$ B [a partial function]
  g(a_expr) $\equiv$ b_expr
  pre P(a_expr)
  P: A $\rightarrow$ **Bool**

- a_expr, b_expr are
- A, respectively B valued expressions
- of any of the kinds illustrated in earlier and later sections of this primer.

Or functions can be defined implicitly:

value
  f: A$\rightarrow$B
  f(a_expr) **as** b
  post P(a_expr,b)
  P: A$\times$B$\rightarrow$**Bool**

  g: A$\overset{\sim}{\rightarrow}$B
  g(a_expr) **as** b
  pre P′(a_expr)
  post P(a_expr,b)
  P′: A$\rightarrow$**Bool**

where b is just an identifier.

- Finally functions, f, g, ..., h, can be defined in terms of axioms
- over function identifiers, f, g, ..., h, and over identifiers of function arguments and results.

type
  A, B, ..., C, D
value
  f: A $\rightarrow$ B, g: B $\rightarrow$ C, ..., h: C $\rightarrow$ D
axiom
  $\forall$ a:A, b:B, ..., c:C, d:D
    $\mathcal{P}_1$(f,g,...,h,a,b,...,c,d) $\wedge$ ... $\wedge$ $\mathcal{P}_n$(f,g,...,h,a,b,...,c,d)

Example 15 . . Axioms over Hubs, Links and Their Observers:

- The axioms displayed in Items 7–10 on Page 14 of Sect.
- demonstrates how a number of entities and observer functions are constrained
- (that is, partially defined) by function signatures.

. . . . **End of Example 15**

1.5. Other Applicative Expressions

1.5.1. Simple let Expressions

Simple (i.e., nonrecursive) **let** expressions:

let a = $\mathcal{E}_d$ **in** $\mathcal{E}_b$(a) **end**

is an “expanded” form of:

(λa.$\mathcal{E}_b$(a))($\mathcal{E}_d$)

1.5.2. Recursive let Expressions

Recursive **let** expressions are written as:

let f = λa·E(f,a) **in** B(f,a) **end**
let f = (λg·λa·E(g,a))(f) **in** B(f.a) **end**
let f = F(f) **in** E(f,a) **end where** F $\equiv$ λg·λa·E(g,a)
let f = **Y**F **in** B(f,a) **end where Y**F = F(**Y**F)

- We read f = **Y**F as “*f is a fix point of F*”.

1.5.3. Non-deterministic let Clause

- The non-deterministic **let** clause:

let a:A · $\mathcal{P}$(a) **in** $\mathcal{B}$(a) **end**

- expresses the non-deterministic selection of a value a of type A
- which satisfies a predicate $\mathcal{P}$(a) for evaluation in the body $\mathcal{B}$(a).
- If no a:A • P(a) the clause evaluates to **chaos**.

1.5.4. Pattern and “Wild Card” let Expressions

Patterns and *wild cards* can be used:

let {a} ∪ s = set **in** ... **end**
let {a,_} ∪ s = set **in** ... **end**

let (a,b,...,c) = cart **in** ... **end**
let (a,_,...,c) = cart **in** ... **end**

let $\langle$a$\rangle$$\frown\ell$ = list **in** ... **end**
let $\langle$a,_,b$\rangle$$\frown\ell$ = list **in** ... **end**

let [a↦b] ∪ m = map **in** ... **end**
let [a↦b,_] ∪ m = map **in** ... **end**

1.5.5. Conditionals

```
if b_expr then c_expr else a_expr
end

if b_expr then c_expr end ≡ /∗ same as: ∗/
  if b_expr then c_expr else skip end

if b_expr_1 then c_expr_1
elsif b_expr_2 then c_expr_2
elsif b_expr_3 then c_expr_3
...
elsif b_expr_n then c_expr_n end

case expr of
  choice_pattern_1 → expr_1,
  choice_pattern_2 → expr_2,
  ...
  choice_pattern_n_or_wild_card → expr_n end
```

Example 16 . Choice Pattern Case Expressions: Insert Links: We consider the meaning of the Insert operation designators.

9. 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.
10. 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.

11. 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.
12. 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.

13. 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
  9 int_Insert: Insert → N ∼→ N
  10′ pre_int_Insert: Ins → N → Bool
  10″ pre_int_Insert(Ins(op))(hs,ls) ≡
⋆2          s_l(op)∉ ls ∧ obs_LI(s_l(op)) ∉ iols(ls) ∧
    case op of
  11)     2oldH(hi′,l,hi″) → {hi′,hi″}∈ iohs(hs),
  12)     1oldH1newH(hi,l,h) →
          hi ∈ iohs(hs) ∧ h∉ hs ∧ obs_HI(h)∉ iohs(hs),
  13)     2newH(h′,l,h″) →
          {h′,h″}∩ hs={} ∧ {obs_HI(h′),obs_HI(h″)}∩ iohs(hs)={}
    end
```

14. 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.
15. 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
14: xtr_H: HI → N ∼→ H
14: xtr_H(hi)(hs,_) ≡ let h:H•h ∈ hs ∧ obs_HI(h)=hi in h end
          pre hi ∈ iohs(hs)
15: xtr_L: HI → N ∼→ H
15: xtr_L(li)(_,ls) ≡ let l:L•l ∈ ls ∧ obs_LI(l)=li in l end
          pre li ∈ iols(ls)
```

16. 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.
17. 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 ∼→ H
  16: aLI(h,li) as h′
    pre li ∉ obs_LIs(h)
    post obs_LIs(h′) = {li} ∪ obs_LIs(h) ∧ non_l_eq(h,h′)
  17: rLI(h′,li) as h
    pre li ∈ obs_LIs(h′) ∧ card obs_LIs(h′)≥2
    post obs_LIs(h) = obs_LIs(h′) \ {li} ∧ non_l_eq(h,h′)
```

18. 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}.
19. If the Insert command is of kind 1oldH1newH(hi,l,h) then the updated net of hubs and links, has

    19.1 : the hub identified by hi updated, hi′, to reflect the link connected to that hub.

    19.2 : The set of hubs has the hub identified by hi replaced by the updated hub hi′ and the new hub.

    19.2 : The set of links augmented by the new link.
20. If the Insert command is of kind 2oldH(hi',l,hi'') then

    20.1–.2 : the two connecting hubs are updated to reflect the new link,

    20.3 : and the resulting sets of hubs and links updated.

```
int_Insert(op)(hs,ls) ≡
⋆i   case op of
18        2newH(h′,l,h″) → (hs ∪ {h′,h″},ls ∪ {l}),
19        1oldH1newH(hi,l,h) →
19.1          let h′ = aLI(xtr_H(hi,hs),obs_LI(l)) in
19.2          (hs\{xtr_H(hi,hs)}∪{h,h′},ls ∪{l}) end,
20        2oldH(hi′,l,hi″) →
20.1          let hsδ = {aLI(xtr_H(hi′,hs),obs_LI(l)),
20.2                     aLI(xtr_H(hi″,hs),obs_LI(l))} in
20.3          (hs\{xtr_H(hi′,hs),xtr_H(hi″,hs)}∪ hsδ,ls ∪{l}) end
⋆j   end
⋆k   pre pre_int_Insert(op)(hs,ls)
```

21. The remove command is of the form Rmv(li) for some li.
22. 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
  21  int_Remove: Rmv → N ~→ N
  22  int_Remove(Rmv(li))(hs,ls) ≡
  22(a))  let l = xtr_L(li)(ls), {hi′,hi″} = obs_HIs(l) in
  22(b))  let {h′,h″} = {xtr_H(hi′,hs),xtr_H(hi″,hs)} in
  22(c))  let hs′ = cond_rmv(h′,hs) ∪ cond_rmv_H(h″,hs) in
  22(d))  (hs\{h′,h″} ∪ hs′,ls\{l}) end end end
  22(a))  pre li ∈ iols(ls)

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

. . . . **End of Example 16**

1.5.6. Operator/Operand Expressions

```
⟨Expr⟩ ::=
      ⟨Prefix_Op⟩ ⟨Expr⟩
    | ⟨Expr⟩ ⟨Infix_Op⟩ ⟨Expr⟩
    | ⟨Expr⟩ ⟨Suffix_Op⟩
    | ...
⟨Prefix_Op⟩ ::=
      − | ∼ | ∪ | ∩ | card | len | inds | elems | hd | tl | dom | rng
⟨Infix_Op⟩ ::=
      = | ≠ | ≡ | + | − | ∗ | ↑ | / | < | ≤ | ≥ | > | ∧ | ∨ | ⇒
    | ∈ | ∉ | ∪ | ∩ | \ | ⊂ | ⊆ | ⊇ | ⊃ | ^ | † | ∘
⟨Suffix_Op⟩ ::= !
```

End of Lecture 9: RSL: Logic, Λ-Calculus, Fctl. Specs.

Start of Lecture 10: RSL: Imperative & Process Specs.

1.6. Imperative Constructs

1.6.1. Statements and State Changes

```
  Unit
value
  stmt: Unit → Unit
  stmt()
```

- The **Unit** clause, in a sense, denotes "an underlying state"
  - which we, for simplicity, can consider as
  - a mapping from identifiers of declared variables into their values.
- Statements accept no arguments and, usually, operate on the state
  - through "reading" the value(s) of declared variables and
  - through "writing", i.e., assigning values to such declared variables.
- Statement execution thus changes the state (of declared variables).
- **Unit** → **Unit** designates a function from states to states.
- Statements, stmt, denote state-to-state changing functions.
- Affixing () as an "only" arguments to a function "means" that () is an argument of type **Unit**.

1.6.2. Variables and Assignment

0. **variable** v:Type := expression
1. v := expr

1.6.3. Statement Sequences and skip

2. **skip**
3. stm_1;stm_2;...;stm_n

1.6.4. Imperative Conditionals

4. **if** expr **then** stm_c **else** stm_a **end**
5. **case** e **of**: p_1→S_1(p_1),...,p_n→S_n(p_n) **end**

1.6.5. Iterative Conditionals

6. **while** expr **do** stm **end**
7. **do** stmt **until** expr **end**

1.6.6. Iterative Sequencing

8. **for** i **in** list · P(list(i)) **do** S(list(i)) **end**
9. **for** e **in** set · P(e) **do** S(e) **end**

1.7. Process Constructs

1.7.1. Process Channels

- Let A, B and C stand for three types of (channel) messages
- and i:IIdx, j:JIdx for channel array indexes, then:

```
channel
  c:A
channel
  {k[ i ]|i:IIdx}:B
  {ch[ i,j ]i:IIdx,j:JIdx}:C
```

Example 17 . . . . Modelling Connected Links and Hubs:

- Examples (17–20) 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
  obs_VI: 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** = obs_Hs(n), ls:L-**set** = obs_Ls(n)
  his:HI-**set** = {obs_HI(h)|h:H·h ∈ hs}, lis:LI-**set** = {obs_LI(h)|l:L·l ∈ ls},
  ivhs:IVH-**set** = {(obs_VI(v),obs_HI(h))|v:V,h:H·v ∈ vs∧h ∈ hs}
  ivls:IVL-**set** = {(obs_VI(v),obs_LI(l))|v:V,l:L·v ∈ vs∧l ∈ ls}
  ihls:IHL-**set** = {(hi,li)|h:H,(hi,li):IHL· h ∈ hs∧hi=obs_HI(h)∧li ∈ obs_LIs(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

. . . . **End of Example 17**

1.7.2. Process Definitions

- A process definition is a function definition.
- The below signatures are just examples.
- They emphasise that process functions must somehow express,
  - in their signature,
- via which channels they wish to engage in input and output events.
- Processes P and Q are to interact, and to do so “ad infinitum”.
- Processes R and S are to interact, and to do so “once”, and then yielding B, respectively D values.

value
  P: **Unit** → **in** c **out** {k[i]|i:IIdx} **Unit**
  Q: i:KIdx → **out** c **in** k[i] **Unit**

  P() ≡ ... c ? ... k[i] ! e ... ; P()
  Q(i) ≡ ... c ! e ... k[i] ? ... ; Q(i)

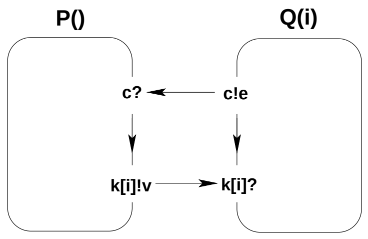


Figure 9: The P —— Q Process

Example 18 . . . . . . 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 ∈ obs_LIs(h)]}
                     in,out {vh[(vi,hi)]|vi:VI•vi ∈ vis]}  Unit
  link: li:LI × l:L → in,out {hl[(hi,li)]|hi:HI•hi ∈ obs_HIs(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]}
                        in,out {vl[(vi,li)]|li:LI•li ∈ lis]}  Unit
```

. . . . **End of Example 18**

1.7.3. Process Composition

- Let P and Q stand for names of process functions,
- i.e., of functions which express willingness to engage in input and/or output events,
- thereby communicating over declared channels.
- Let $\mathcal{P}$ and $\mathcal{Q}$ stand for process expressions,
- and let $\mathcal{P}_i$ stand for an indexed process expression, then:

$\mathcal{P} \parallel \mathcal{Q}$	Parallel composition
$\mathcal{P} \,[\!]\, \mathcal{Q}$	Nondeterministic external choice (either/or)
$\mathcal{P} \sqcap \mathcal{Q}$	Nondeterministic internal choice (either/or)
$\mathcal{P} \,\#\, \mathcal{Q}$	Interlock parallel composition
$\mathcal{O}\,\{\, \mathcal{P}_i \mid \text{i:Idx} \,\}$	Distributed composition, $\mathcal{O} = \parallel, [\!], \sqcap, \#$

Example 19 . . . . 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( obs_HI(h))(h)|h:H•h ∈ obs_Hs(n)} ‖
    ‖ {link( obs_LI(l))(l)|l:L•l ∈ obs_Ls(n)} ‖
    ‖ {vehicle(obs_VI(v))(obs_PN(v))(v)|v:V•v ∈ vs}

  obs_PN: V → (Pos×Net)
```

- We illustrate a schematic definition of simplified hub processes.
- The hub process alternates, internally non-deterministically, $\lceil\rceil$, 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 !

$$\begin{array}{l}
\mathsf{hub(hi)(h)} \equiv \\
\quad \lceil\rceil\{\mathbf{let}\ \mathsf{m} = \mathsf{hl[\,(hi,li)\,]\ ?}\ \mathbf{in}\ \mathsf{hub(hi)}(\mathcal{E}_{h_\ell}\mathsf{(li)(m)(h))}\ \mathbf{end}|\mathsf{i{:}LI{\cdot}li} \in \mathsf{obs\_LI(h)}\} \\
\lceil\rceil\ \lceil\rceil\{\mathbf{let}\ \mathsf{m} = \mathsf{vh[\,(vi,hi)\,]\ ?}\ \mathbf{in}\ \mathsf{hub(vi)}(\mathcal{E}_{h_v}\mathsf{(vi)(m)(h))}\ \mathbf{end}|\mathsf{vi{:}VI{\cdot}vi} \in \mathsf{vis}\} \\
\lceil\rceil\ \mathsf{hub(hi)}(\mathcal{E}_{h_{own}}\mathsf{(h))}
\end{array}$$

- 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.

$$\begin{array}{l}
\mathcal{E}_{h_\ell}:\ \mathsf{LI} \to \mathsf{M} \to \mathsf{H} \to \mathsf{H} \\
\mathcal{E}_{h_v}:\ \mathsf{VI} \to \mathsf{M} \to \mathsf{H} \to \mathsf{H} \\
\mathcal{E}_{h_{own}}:\ \mathsf{H} \to \mathsf{H}
\end{array}$$

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

. . . . **End of Example 19**

1.7.4. Input/Output Events

- Let c and k[i] designate channels of type A
- and e expression values of type A, then:

[1] c?, k[i]?	input A value
[2] c!e, k[i]!e	output A value

value

[3] P: ... → **out** c ..., P(...) ≡ ... c!e ...	offer an A value,
[4] Q: ... → **in** c ..., Q(...) ≡ ... c? ...	accept an A value
[5] S: ... → ..., S(...) = P(...)‖Q(...)	synchronise and communicate

- [5] expresses the willingness of a process to engage in an event that
  - [1,3] “reads” an input, respectively
  - [2,4] “writes” an output.

Example 20 . . . . **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 == mk_atH(hi:HI)
  onL == mk_onL(fhi:HI,li:LI,f:F,thi:HI)
  F = {|f:Real•0≤f≤1|}
```

- 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 == mk_L_H(li:LI,hi:HI) | mk_H_L(hi:HI,li:LI)
value
  vehicle: VI → (Pos × Net) → V → Unit
  vehicle(vi)(mk_atH(hi),net)(v) ≡
  [1]  (wait ;
  [2]   vehicle(vi)(mk_atH(hi),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,hi)]!mk_H_L(hi,li)||vl[(vi,li)]!mk_H_L(hi,li));
  [8]   vehicle(vi)(mk_onL(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 == mk_L_H(li:LI,hi:HI) | mk_H_L(hi:HI,li:LI)
value
  δ:Real = move(h,f) axiom 0<δ≪1
  vehicle(vi)( mk_onL(hi,li,f,hi′),net)(v) ≡
  [1′]  (wait ;
  [2′]   vehicle(vi)(mk_onL(hi,li,f,hi′),net)(v))
  [3′]  ⊓
  [4′]  (case f of
  [5′]     1 → ((vl[vi,hi′]!mk_L_H(li,hi′)‖vh[vi,li]!mk_L_H(li,hi′));
  [6′]          vehicle(vi)(mk_atH(hi′),net)(v)),
  [7′]     _ → vehicle(vi)(mk_onL(hi,li,f+δ,hi′),net)(v)
  [8′]   end)
  move: H × F → F
```

....End of Example 20

1.8. Simple RSL Specifications

- Besides the above constructs RSL also possesses module-oriented
  - scheme,
  - class and
  - object

  constructs.
- We shall not cover these here.
- An RSL specification is then simply
  - a sequence of one or more clusters of
    - ∗ zero, one or more sort and/or type definitions,
    - ∗ zero, one or more variable declarations,
    - ∗ zero, one or more channel declarations,
    - ∗ zero, one or more value definitions (including functions) and
    - ∗ zero, one or more and axioms.
- We can illustrate these specification components schematically:

```
type
  A, B, C, D, E, F, G
  Hf = A-set, Hi = A-infset
  J = B×C×...×D
  Kf = E*, Ki = Eω
  L = F →m G
  Mt = J → Kf, Mp = J ~→ Ki
  N == alpha | beta | ... | omega
  O == mk_Hf(as:Hf)
        | mk_Kf(el:Kf) | ...
  P = Hf | Kf | L | ...
variable
  vhf:Hf := ⟨⟩
channel
  chf:F, chg:G, {chb[i]|i:A}:B
```

```
value
  va:A, vb:B, ..., ve:E
  f1:  A → B, f2:  C ~→ D
  f1(a) ≡ ℰf1(a)
  f2:  E → in|out chf F
  f2(e) ≡ ℰf2(e)
  f3:  Unit → in chf out chg  Unit
  ...
axiom
  𝒫i(f1,va),
  𝒫j(f2,vb),
  ...
  𝒫k(f3,ve)
```

- The ordering of these clauses is immaterial.
- Intuitively the meaning of these definitions and declarations are the following.
  - The **type** clause introduces a number of user-defined type names;
    - ∗ the type names are visible anywhere in the specification;
    - ∗ and either denote sorts or concrete types.
  - The **variable** clause declares some variable names;
    - ∗ a variable name denote some value of decalred type;
    - ∗ the variable names are visible anywhere in the specification:
      - · assigned to (‘written’) or
      - · values ‘read’.
  - The **channel** clause declares some channel names;
    - ∗ either simple channels or arrays of channels of some type;
    - ∗ the channel names are visible anywhere in the specification.

– The **value** clause bind (constant) values to value names.
  * These value names are visible anywhere in the specification.
  * The specification

```
type                    value
  A                       a:A
```

  * non-deterministically binds a to a value of type A.
  * Thuis includes, for example

```
type                    value
  A, B                    f: A → B
```

  * which non-deterministically binds f to a function value of type A→B.

Example 21 . . . . 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 == mk_Enter_L(vi:VI,li:LI) | mk_Leave_L(vi:VI,li:LI)
value
  n:N
channel
  {ves[obs_HI(h),li]|h:H•h ∈ obs_Hs(n)∧li ∈ obs_LIs(h)}:VI
  {vls[li,obs_HI(h)]|h:H•h ∈ obs_Hs(n)∧li ∈ obs_LIs(h)}:VI
  rp:RPM
type
  Fee, Bal
  LVS = LI →m VI-set,  FEE = LI →m Fee,  ACC = VI →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 20, in two parts, Slide 382 and Slide 384 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
           mk_atH(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)(mk_onL(hi,li,0,hi′),net)(v)
  [9]         end end end)
           mk_onL(hi,li,f,hi′) →
  [4′]       (case f of
  [5′–6′]       1 → (vls[li,hi]!vi; vehicle(vi)(mk_atH(hi′),net)(v)),
  [7′]          _ → vehicle(vi)(mk_onL(hi,li,f+δ,hi′),net)(v)
  [8′]        end)
        end
```

- As mentioned on Slide 390 *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′} = obs_HIs(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!mk_Enter_LI(vi,li); enter_sensor(hi,li) end
  leave_sensor(li,hi) ≡
    let vi = ves[li,hi]? in rp!mk_Leave_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 student 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
    mk_Enter_LI(vi,li) →
        road_pricing(lvs†[li↦lvs(li)∪{vi}],fee,acc),
    mk_Leave_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
```

.... **End of Example 21**

End of Lecture 10: RSL: Imperative & Process Specs.

B Slide Table-of-Contents

Contents