(-1.)

Start of Lecture 1: SUMMARY & INTRODUCTION

(-1.)

FROM DOMAINS TO REQUIREMENTS

April 16–30, 2010 Lectures, TUWien

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

0. Abstract

- We shall present core aspects of the Triptych approach to software engineering.
- The benefits from deploying this approach are that we both achieve *the right software* and *software that is right*[Boehm 1981]).
- *The right software* is software that meets all of the customers' expectations and only those.
- *Software that is right* is software that is correct with respect to specific requirements prescriptions.

(0. Abstract)

- Experience has shown that using also the formal techniques part of the Triptych approach has lead to projects that are on time and at initially estimated costs.
- To achieve **the right software** we "prefix" the phase of requirements engineering with a phase of domain engineering – and these lecture slides will present core aspects of domain engineering.
- To achieve **software that is right** we do two things:
  - (i) "derive" requirements prescriptions from domain descriptions and software design from requirements prescriptions – and this these lecture slides will present core aspects of a somewhat different approach to requirements engineering, and
  - (ii) formulate descriptions and prescriptions both informally, in precise, say English narratives, and formally. The latter is not shown in these lecture slides.

(0. **Abstract**)

- The "somewhat" different approach to requirements engineering, however, and as we shall see, fits reasonably "smoothly" with current requirements engineering approaches[van Lamsweerde].
- Precursors of the 'triptych' approach was used in DDC's 44 man-year Ada Compiler development project [Bjørner and Oest].
  - That project was on time and at cost,
  - and time and cost were significantly below those of other commercial Ada compiler developments .

(0. **Abstract**)

- The 'triptych' approach has been in partial use since the early 1990s,
  - including at the United Nations University's International Institute for Software Technology (`www.iist.unu.edu`).
  - Young software engineers, while being tutored by UNU-IIST's science & engineering staff,
    - ∗ domain engineered,
    - ∗ requirements engineered
    - ∗ and software designed (incl. implemented)[2002, LNCS 2757]
      - · trustworthy software systems
      - · that have met customer expectations –
      - · with what seems be substantially fewer man-power resources than usually experienced and within planned time limits.

(0. **Abstract**)

- Domain engineering, in the sense of these lecture slides,
  - is offered as a means to help secure that software engineers deliver *the right software* –
  - where formalisation of relevant stages and steps of software development helps secure that *the software is right*.
- In these lecture slides we shall present the essence of a software development *triptych*:
  - from domains
  - via requirements
  - to software design.

(0. **Abstract**)

- We emphasize the two first phases: *domain engineering* and *requirements engineering*.
  - We show the pragmatic stages of the construction of *domain descriptions*: the facets of
  - *intrinsics*,
  - *support technologies*,
  - *rules & regulations*,
  - *script (licenses and contracts)*,
  - *management & organisation*, and
  - *human behaviour*.

(0. **Abstract**)

- And we show how to construct main facets of *requirements prescriptions*:
  - *domain requirements* and
  - *interface requirements*.
- In this respect we focus in particular on the *domain requirements* development stages of
  - *projection*,
  - *instantiation*,
  - *determination* and
  - *extension*.

Lecture Notes for T U Wien, April 2010

- The present version of this document is intended as the "written" support for my April 2010 lectures at the Technical University of Vienna. Austria.
  - The **www.imm.dtu.dk/~db/wien** web page gives details.
  - From there you can see that Sects. 1–5 covers 5 lectures
  - and that Appendix A covers 8 lectures.

(0. **Abstract**)

- To examples of sections 2–4 we have "added" formalisations.
- These formalisations are in the `RAISE` specification languages `RSL`.
- And I have additionally added an extensive appendix,
  - An RSL Primer[1],
- so that students can also learn `RSL`, the specification language for a **r**igorous **a**pproach to **i**ndustrial **s**oftware **e**ngineering, `RAISE`.
- The primer contains many examples which expands on the examples of sections 2–4.

[1]a small introductory book on a subject

(0. **Abstract**)

"Formalisation–Parametrised" Examples and Primer

- The formalisations of the examples of sections 2–4 could as well be expressed in one of the other prominent formal specifications languages current at this time (April 22, 2010), for example:
  - `Alloy`,
  - `Event B`,
  - `VDM-SL` or
  - `Z`.
- It could be interesting
  - if this little book could entice
  - my Alloy, Event B, VDM-SL and Z colleagues
  - to "rewrite/reformulate" the formal parts of all examples
  - into their main tool of formal expression (besides mathematics).
- I would be very willing to engage in such a project
  - having the aim of making my and their notes
  - Internet-based and thus publically available.

(0. Abstract)

On Studying the Examples

- In order to learn to **write** poems one must **read** poetry.
- In order to learn th **write** formal specifications one must **read** formal specifications
- We have ourselves found
  - that even if students attend pedagogically and didactically exciting and sound lectures
  - they must still, in the quiet of their study room, without listening to Ipod (or the like),
  - carefully study the examples we are presenting.
- And we are presenting many examples, 49 in all !
  - To begin with little explanation is given of the formulas.
  - Instead we rely on the student's ability to relate the numbered formulas to the numbered annotation textst.
  - As from Appendix we present a schematic syntax and informal semantics of the spexification language, RSL, used in these lectures.
- Students are well adviced in studying all examples.

(0. Abstract)

On Course Lectures Based on these Slides

1. Summary of Lectures and Introduction	0–36
2. The Triptych Specification Ontology	37–59
3. Domain Engineering	60–97
4. Requirements Engineering	98–139
5. Entities	312–347

6. **RSL** Types	162–188	11. **RSL** Specifications	298–311
7. **RSL** Values & Operations	189–234	12. Mereology	361–453
8. **RSL** Logic and λ-Calculus	235–258	13. Discussion and Conclusion	140–161
9. **RSL** Applicativeness	259–274	14. Discussion	
10. **RSL** Imperativeness and Concurrency (CSP)	275–297	15. Exam	

1. Introduction

1.1. Some Observations

- Current software development,
  - when it is pursued in a state-of-the-art,
  - but still a conventional manner,
  - starts with requirements engineering and
  - proceeds to software design.
- Current software development practices
  - appears to be focused on processes
  - (viz.: "best practices': tools and techniques').

(1. Introduction 1.1. Some Observations)

- An aeronautics engineer to be hired by `Airbus` to their design team for a next generation aircraft must be pretty well versed in applied mathematics and in aerodynamics.
- A radio communications engineer to be hired by `Ericsson` to their design team for a next generation mobile telephony antennas must be likewise well versed in applied mathematics and in the physics of electromagnetic wave propagation in matter.
- And so forth.

- Software engineers hired for the development of software
  - for hospitals,
  - or for railways,
- know little, if anything, about
  - health care,
  - respectively rail transportation (scheduling, rostering, signalling, etc.).
- The `Ericsson` radio communications engineer can be expected to understand Maxwell's Equations, and to base the design of antenna characteristics on the transformation and instantiation of these equations.

- It is therefore quite reasonable to expect the domain-specific software engineer to understand proper, including formal descriptions of their domains:
  - for railways cf. `www.railwaydomain.org`,
  - and for pipelines `pipelines.pdf`,
  - logistics `logistics.pdf`
  - and for container lines `container-paper.pdf` –
  - all at `www.imm.dtu.dk/~db/`.
- For the Vienna course the above — and other such — examples are temporarily blocked !

- The process knowledge and "best" practices of the triptych software engineering
  - is well-founded and takes place in
  - the context of established domain model
  - and an established, carefully phrased (and formalised) requirements model.
- The 24 hour 7 days a week trustworthy operation of many software systems
  - is so crucial that utmost care must be taken
  - to ensure that they
    - ∗ fulfill all (and only) the customers expectations
    - ∗ and are correct.

- Barry Boehm has coined the statement: *it is the right software and the software is right*.
- Extra care must be taken to ensure those two "rights".
- And here it is not enough to only follow current "best process, technique and tool practices".

1.2. A Triptych of Software Engineering

Dogma:

- *Before we can design software*
- *we must have a robust understanding of its requirements.*
- *And before we can prescribe requirements*
- *we must have a robust understanding of the environment,*
  - *or, as we shall call it, the domain in which the software is to serve*
  - *and as it is at the time such software is first being contemplated.*

- In consequence we suggest that software, "ideally"[2], be developed in three phases.

[2]Section [Item 5] will discuss renditions of "idealism"!

- First a phase of **domain engineering.**
  - In this phase a reasonably comprehensive description is constructed from an analysis of the domain.
  - That description, as it evolves, is analysed with respect to inconsistencies, conflicts and relative completeness.
  - $\mathcal{P}$roperties, as stated by domain stakeholders, are proved with respect to the domain description ($\mathcal{D}\models\mathcal{P}$).
  - This phase is the most important, we think, when it comes to secure the first of the two "rights": that we are on our way to develop the right software.

- Then a phase of **requirements engineering.**
  - This phase is strongly based on an available, necessary and sufficient domain description.
  - Guided by the domain and requirements engineers the *requirements stakeholders* points out which domain description parts are

    ∗ to be kept (*projected*) out of the *domain requirements*,

    and for those kept in,

    ∗ what *instantiations*,

    ∗ *determinations*

    ∗ and *extensions* are required.

  - Similarly the requirements stakeholders, guided by the domain and requirements engineers, informs as to

    ∗ which domain *entities: simple, actions, events* and *behaviours*

    ∗ are *shared* between the domain and the *machine*,
  - that is, the *hardware* and the *software* being required.
- In these lectures we shall only very briefly cover aspects of *machine requirements*.

- And finally a phase of **software design.**
  - We shall not cover this phase in these lectures –
  - other than saying this:
    - ∗ the design is "derived" from the requirements model.

- To ensure that the software being developed is right, that is, correct,
  - we can then rigorously
  - argue, informally,
  - or formally – test, model check and/or prove,
  - that the $\mathcal{S}$oftware is correct
    - ∗ with respect to the $\mathcal{R}$equirements
    - ∗ in the context of the $\mathcal{D}$omain:
    - ∗ $\mathcal{D}, \mathcal{S} \models \mathcal{R}$.

1.3. What are Domains ?

- By a domain we shall here understand a universe of discourse,
  - an area of nature subject to laws of physics and study by physicists, or
  - an area of human activity subject to its interfaces with other domains and to nature.
- There are other domains – which we shall ignore.
- We shall focus on the human-made domains.

- "Large scale" examples are
  - *the financial service industry: banking, insurance, securities trading, portfolio management, etc.*;
  - *health care: hospitals, clinics, patients, medical staff, etc.*;
  - *transportation: road, rail/train, sea/shipping*, and *air/aircraft transport (vehicles, transport nets, etc.)*;
  - *oil and gas systems: pumps, pipes, valves, refineries, distribution, etc.*

- "Intermediate scale" examples are
  - *automobiles: manufacturing* or *monitoring and control, etc.*;
  - *heating systems*;
  - *heart pumps*;
  - etc.
- The above explication was "randomised":
  - for some domains, to wit, *the financial service industry*, we mentioned major functionalities,
  - for others, to wit, *health care*, we mentioned major entities.

1.4. What is a Domain Description ?

- By a *domain description* we understand a description of
  - the *simple entities*,
  - the *actions*,
  - the *events* and
  - the *behaviours*

  of the domain, including its *interfaces* to other domains.
- A domain description describes the domain **as it is.**
- A domain description does not contain requirements let alone references to any software.

- A description is *syntax*.
- The meaning (*semantics*) of a domain description is usually a set of *domain models*.
- We shall take domain models to be *mathematical structures (theories)*.
- The form of domain descriptions that we shall advocate "come in pairs": precise, say, English text alternates with clearly related formula text.

1.5. Description Languages

- Besides using
  - as precise a subset of a national language, as here English, as possible, and in enumerated expressions and statements,
  - we "pair" such narrative elements with corresponding enumerated clauses of a formal specification language.
- We shall be using the `RAISE` Specification Language, `RSL` in our formal texts.
- But any of the model-oriented approaches and languages offered by
  - `Alloy`,
  - `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
    - either `DC: Duration Calculus`
    - or `TLA+`.

1.6. Contributions of these Lectures

- We claim that the major contributions of the Triptych approach to software engineering as presented in this paper are the following:
  - (1) the clear *identification* of domain engineering, or, for some, its clear *separation* from requirements engineering;
  - (2) the *identification* and *'elaboration'* of the pragmatically determined domain *facets* of *intrinsics, support technologies, management and organisation, rules and regulations, scripts (licenses and contracts)* and *human behaviour* whereby 'elaboration' we mean that we provide principles and techniques for the construction of these facet description parts;

- (3) the *re-identification* and *'elaboration'* of the concept of *business process re-engineering* on the basis of the notion of *business processes*;
- (4) the *identification* and *'elaboration'* of the technically determined *domain requirements facets* of *projection, instantiation, determination, extension* and *fitting* requirements principles and techniques – and, in particular the *"discovery"* that these requirements engineering stages are strongly dependent on necessary and sufficient domain descriptions ;

and

- (5) the *identification* and *'elaboration'* of the technically determined *interface requirements facets* of *shared entity, shared action, shared event* and *shared behaviour* requirements principles and techniques. We claim that the facets of (2, 3, 4) and (5) are all *novel*.

1.7. Structure of Lectures

- Before going into some details on domain enginering and requirements engineering
- cover the basic concepts of specifications, whether domain descriptions or requirements prescriptions.
- These are:
  - entities,
  - actions,
  - events and
  - behaviours.

End of Lecture 1: SUMMARY & INTRODUCTION

Start of Lecture 2: A SPECIFICATION ONTOLOGY

2. A Specification Ontology

- In order to describe domains we postulate the following related specification components:
  - *entities*,
  - *actions*,
  - *events* and
  - *behaviours*.

(2. A Specification Ontology)

2.1. Entities

- By an entity we shall understand
  - a phenomenon we can point to in the domain
  - or a concept formed from such phenomena.

(2. A Specification Ontology 2.1. Entities)

Example 1 – Entities

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

(2. A Specification Ontology 2.1. Entities)

1 There are hubs and links.

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

3 There are hub and link identifiers.

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

(2. A Specification Ontology 2.1. Entities)

type
  [1] H, L,
  [2] N = H-**set** × L-**set**
axiom [nets−hubs−links−1]
  [2] $\forall$ (hs,ls):N · **card** hs$\geq$2 $\wedge$ **card** ls$\geq$1
type
  [3] HI, LI
value
  [4] ωHI: H $\rightarrow$ HI, ωLI: L $\rightarrow$ LI
axiom [nets−hubs−links−2]
  [4] $\forall$ h,h′:H, l,l′:L · h$\neq$h′ $\Rightarrow$ ωHI(h)$\neq$$\omega$HI(h′) $\wedge$ l$\neq$l′$\Rightarrow$$\omega$LI(l)$\neq$$\omega$LI(l′)

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

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

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

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

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

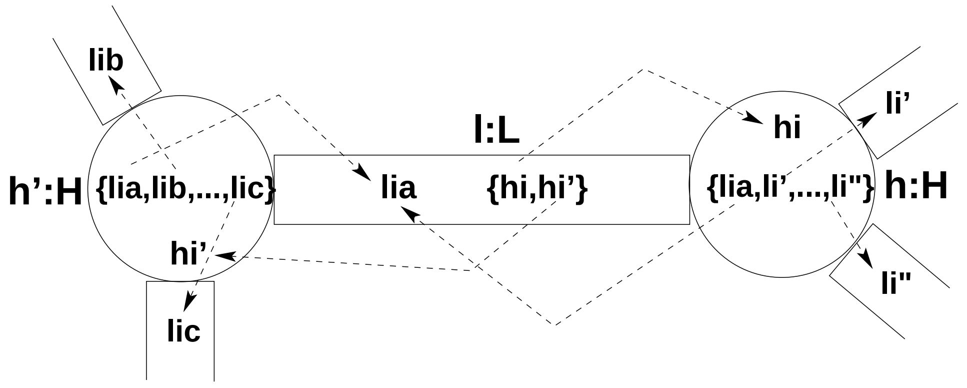


Figure 1: Connected links and hubs with observable identifiers

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

- In the above extensive example we have focused on just five entities: nets, hubs, links and their identifiers.
- The nets, hubs and links can be seen as separable phenomena.
- The hub and link identifiers are conceptual models of the fact that hubs and links are connected
  - so the identifiers are abstract models of 'connection',
  - i.e., the mereology of nets, that is, of how nets are composed.
- These identifiers are attributes of entities.
- Links and hubs have been modelled to possess link and hub identifiers.
  - A link's "own" link identifier enables us to refer to the link,
  - A link's two hub identifiers enables us to refer to the connected hubs.
  - Similarly for the hub and link identifiers of hubs and links.

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

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

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

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

■ End of Example 1

2.2. Actions

- A set of entities form a domain state.
- It is the domain engineer which decides on such states.
- A function is an action if,
  - when applied
    - to zero, one or more arguments
    - and a state,
  - it then results in a state change.
- (Arguments could be other entities or just values of entity attributes.)

Example 2 – Deterministic Hub State Setting

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

13 The setting applies to a hub

14 and a hub state in the hub state space

13 and yields a "new" hub.

15A The before and after hub identifier remains the same.

15B The before and after link identifiers remain the same.

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

17 The result hub state is that being set (i.e., the argument hub state).

```
value
 [12]  set_H$\Sigma$: H $\times$ H$\Sigma$ $\rightarrow$ H
 [13]  set_H$\Sigma$(h,h$\sigma$) as h$'$
 [14]    pre h$\sigma$ $\in$ $\omega$H$\Omega$(h)
 [15A]   post $\omega$HI(h)=$\omega$HI(h$'$)$\wedge$
 [15B]        $\omega$LIs(h)=$\omega$LIs(h$'$)$\wedge$
 [16]         $\omega$H$\Omega$(h)=$\omega$H$\Omega$(h$'$)$\wedge$
 [17]         $\omega$H$\Sigma$(h$'$)=h$\sigma$
```

■ End of Example 2

- Example 2 illustrated a deterministic action:
  - one that always succeeded
  - in carrying out the prescribed operation.
- But, as we shall see later,
  - the domain technology may be faulty and
  - an action, as carried out by such a technology,
  - may fail to have the desired effect.

Example 3 – Non-Deterministic Hub State Setting

17 The result hub state is one of the hub states of the hub state space.

```
value
 [12]  set_H$\Sigma$: H $\times$ H$\Sigma$ $\rightarrow$ H
 [13]  set_H$\Sigma$(h,h$\sigma$) as h$'$
 [14]    pre h$\sigma$ $\in$ $\omega$H$\Omega$(h)
 [15A]   post $\omega$HI(h)=$\omega$HI(h$'$)$\wedge$
 [15B]        $\omega$LIs(h)=$\omega$LIs(h$'$)$\wedge$
 [16]         $\omega$H$\Omega$(h)=$\omega$H$\Omega$(h$'$)$\wedge$
 [17]         $\omega$H$\Sigma$(h$'$) $\in$ $\omega$H$\Omega$(h)
```

2.3. Events

- Any domain state change is an event.
- A situation
  - in which a (specific) state change was expected
  - but none (or another) occurred is an event.
- Some events are more "interesting" than other events.
- Not all state changes are caused by actions of the domain.

Example 4 – Events: Failure State Transitions

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

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

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

21 Thus an "interesting event" has occurred !

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

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

■ End of Example 4

2.4. Behaviours

- A behaviour is a set of
  - zero, one or more sequences of sets of
    - ∗ actions
    - ∗ or behaviours,
    - ∗ including events.

Example 5 – Behaviours: Blinking Semaphores

22 Let h be a hub of a net n.

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

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

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

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

- 26a from $h\sigma$ to $h\sigma'$ and
- 26b then, after a wait of ti seconds,
- 26c and then back to $h\sigma$.
- 26d After n blinks a pause, $tp : TI$, is made and blinking restarts.

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

■ End of Example 5

End of Lecture 2: A SPECIFICATION ONTOLOGY

Start of Lecture 3: DOMAIN ENGINEERING

3. Domain Engineering

- We focus on the *facet* components of a domain description
- and shall not here cover such aspects of domain engineering as
  - stakeholder identification and liaison,
  - domain acquisition and analysis,
  - terminologisation,
  - verification, testing, model-checking, validation and
  - domain theory formation.

- By understanding, first, the *facet* components
  - the domain engineer is in a better position to effectively
  - establish the regime of stakeholders,
  - pursue acquisition and analysis,
  - and construct a necessary and sufficient terminology.
- The domain description components each cover their domain facet.

- We outline six such facets:
  - intrinsics,
  - support technology,
  - rules and regulations,
  - scripts (licenses and contracts),
  - management and organisation, and
  - human behaviour.
- But first we cover a notion of business processes.

3.1. Business Processes

- By a business process we understand
  - a set of one or more, possibly interacting behaviours
  - which fulfill a business objective.
- We advocate that domain engineers,
  - typically together with domain stakeholder groups,
  - rough-sketch their individual business processes.

Example 6 – Some Transport Net Business Processes

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

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

A private citizen automobile transport "business process rough-sketch" might be:

> A car owner drives to work: *Drives out, onto the street, turns left, goes down the street, straight through the next three intersections, then turns left, two blocks straight, etcetera, finally arrives at destination, and finally turns into a garage.*

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

A public bus transport "business process rough-sketch" might be:

> A bus drive from station of origin to station of final destination: *Bus driver starts from station of origin at the designated time for this drive; drives to first passenger stop; open doors to let passenger in; leaves stop at time table designated time; drives to next stop adjusting speed to traffic conditions and to "keep time" as per the time table; repeats this process: "from stop to stop", letting passengers off and on the bus; after having (thus, i.e., in this manner) completed last stop "turns" bus around to commence a return drive.*

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

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

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

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

■ End of Example 6

- Rough-sketching such business processes helps bootstrap the process of domain acquisition.

3.2. Intrinsics

- By intrinsics we shall understand
  - the very basics,
  - that without which none of the other facets can be described,
  - i.e., that which is common to two or more, usually all of these other facets.

Example 7 – Intrinsics

- Most of the descriptions of earlier examples model intrinsics.
- We add a little more:

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

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

34 A link state is a set of link traversals.

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

value
  n:N
type
  [32] $\mathsf{LT}' = \mathsf{HI} \times \mathsf{LI} \times \mathsf{HI}$
  [33] $\mathsf{LT} = \{|\mathsf{lt}{:}\mathsf{LT}' \cdot \mathsf{wfLT}(\mathsf{lt})(\mathsf{n})|\}$
  [34] $\mathsf{L}\Sigma' = \mathsf{LT}\textbf{-set}$
  [34] $\mathsf{L}\Sigma = \{|\mathsf{l}\sigma{:}\mathsf{L}\Sigma' \cdot \mathsf{wf\_L}\Sigma(\mathsf{l}\sigma)(\mathsf{n})|\}$
  [35] $\mathsf{L}\Omega' = \mathsf{L}\Sigma\textbf{-set}$
  [35] $\mathsf{L}\Omega = \{|\mathsf{l}\omega{:}\mathsf{L}\Omega' \cdot \mathsf{wf\_L}\Omega(\mathsf{l}\omega)(\mathsf{n})|\}$
value
  [33] $\mathsf{wfLT}{:}\ \mathsf{LT} \to \mathsf{N} \to \mathbf{Bool}$
  [33] $\mathsf{wfLT}(\mathsf{hi},\mathsf{li},\mathsf{hi}')(\mathsf{n}) \equiv$
  [33] $\exists\ \mathsf{h},\mathsf{h}'{:}\mathsf{H} \cdot \{\mathsf{h},\mathsf{h}'\} \subseteq \omega\mathsf{Hs}(\mathsf{n}) \wedge$
  [33] $\omega\mathsf{HI}(\mathsf{h}) = \mathsf{hi} \wedge \omega\mathsf{HI}(\mathsf{h}') = \mathsf{hi}' \wedge$
  [33] $\mathsf{li} \in \omega\mathsf{LIs}(\mathsf{h}) \wedge \mathsf{li} \in \omega\mathsf{LIs}(\mathsf{h}')$

■ End of Example 7

3.3. Support Technologies

- By support technologies we shall understand
  - the ways and means by which
    - ∗ humans and/or
    - ∗ technologies
    - ∗ support
      - · the representation of entities and
      - · the carrying out of actions.

Example 8 – Support Technologies

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

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

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

38 There is a notion of time interval.

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

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

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

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

■ End of Example 8

3.4. Rules and Regulations

- By a **rule** we shall understand
  - a text which describe how the domain is — i.e., how people and technology are —
  - expected to behave.
- The meaning of a rule is
  - a predicate over "before/after" states of actions (simple, one step behaviours):
  - if the predicate holds then the rule has been obeyed.

- By a **regulation** we shall understand
  - a text which describes actions to be performed
  - should its corresponding rule fail to hold.
- The meaning of a regulation is therefore
  - a state-to-state transition,
  - one that brings the domain into a rule-holding “after” state.

Example 9 – **Rules** We give two examples related to railway systems where train stations are the hubs and the rail tracks between train stations are the links:

41 Trains arriving at or leaving train stations:

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

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

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

We omit showing somewhat “lengthy” formalisations.
. ■ End of Example 9

We omit exemplification of regulations.

3.5. Scripts, Licenses and Contracts

3.5.1. Scripts

- By a script we understand
  - a usually structured set of pairs of rules and regulations —
  - structured, for example, as a simple “algorithm description”.

Example 10 – Timetable Scripts

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

44 A *T*ime*T*able associates *B*us *L*ine *Id*entifiers (*blid*) to sets of *Journies*.

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

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

47 A set of *BusRides* associates, to each of a number of *B*us *Id*entifiers (*bid*) a *Bus Sched*ule.

48 A *Bus Sched*ule is a triple of the initial departure *T*ime, a list of zero, one or more intermediate bus stop *T*imes and a destination arrival *T*ime.

49 A *Bus Stop* (i.e., its position) is a *Frac*tion of the distance along a link (identified by a *L*ink *I*dentifier) *f*rom an *i*dentified *h*ub *t*o an *i*dentified *h*ub.

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

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

52 A set of journies is well-formed if

53 the bus stops are all different,

54 a bus line is embedded in some line of the net, and

55 all defined bus trips of a bus line are equivalent.

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

■ End of Example 10

- Timetables are used in Example 11 on the following page.

3.5.2. Licenses and Contracts

- By a **license** (a **contract**) language we understand a pair of languages
  - of licenses and
  - of the set of actions allowed by the license
  - such that non-allowable license (contract) actions
    - incur moral obligations
    - (respectively legal responsibilities).

Example 11 – Public Bus Transport Contracts

- An example contract can be 'schematised':

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

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

- Concrete examples of actions can be schematised:

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

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

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

- All actions are being performed by a sub-contractor in a context which defines
  - that sub-contractor *cnm*,
  - the relevant net, say *n*,
  - the base contract, referred here to by *cid* (from which this is a sublicense), and
  - a timetable *tt* of which *tt′* is a subset.
- contract name *cnm′* is new and is to be unique.
- The subcontracting action can (thus) be simply transformed into a contract as shown on Slide 84.

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

■ End of Example 11

3.6. Management and Organisation

- By management we shall understand
  - the set of behaviours which perform
    - strategic,
    - tactical and
    - operational

    actions.

- By organisation we shall understand
  - the decomposition of these behaviours into, for example, clearly separate
    - strategic,
    - tactical and
    - operational

    "areas",
    - possibly further decomposed
    - by geographical and/or
    - "subject matter" concerns.

- To explain differences between strategic, tactical and operational issues we introduce notions of
  - *strategic*, *tactical* and *operational funds*, $\mathbb{F}_{\mathcal{S},\mathcal{T},\mathcal{O}}$,
  - and other *resources*, $\mathbb{R}$,
  - a notion of *contexts*, $\mathbb{C}$,
  - and a notion of *states*, $\mathbb{S}$.
- Contexts bind resources to bindings from locations to disjoint time intervals (allocation and scheduling),
- states bind resource identifiers to resource values.

- Simplified types of the strategic, tactical and operational actions are now of the following types:
  - executive functions apply to contexts, states and funds and obtain and redistribute funds;
  - strategic functions apply to contexts and strategic funds and create new contexts and states and consume some funds;
  - tactical functions apply to resources, contexts, states tactical funds and create new contexts while consuming some tactical funds;
  - etcetera.

type
$\mathbb{R}, \mathbb{RID}, \mathbb{RVAL}, \mathbb{F}_{\mathcal{S}}, \mathbb{F}_{\mathcal{T}}, \mathbb{F}_{\mathcal{O}}$
$\mathbb{C} = \mathbb{R} \overrightarrow{m} ((\mathbb{T} \times \mathbb{T}) \overrightarrow{m} \mathbb{L})$
$\mathbb{S} = \mathbb{RID} \overrightarrow{m} \mathbb{RVAL}$

value
$\omega\mathbb{RID}\colon \mathbb{R} \to \mathbb{RID}$
$\omega\mathbb{RVAL}\colon \mathbb{R} \to \mathbb{RVAL}$
Executive_functions: $\mathbb{C} \times \mathbb{S} \times \mathbb{F}_{\mathcal{S},\mathcal{T},\mathcal{O}} \to \mathbb{F}_{\mathcal{S},\mathcal{T},\mathcal{O}}$
Strategic_functions: $\mathbb{C} \times \mathbb{F}_{\mathcal{S}} \to \mathbb{F}_{\mathcal{S}} \times \mathbb{R} \times \mathbb{C} \times \mathbb{S}$
Tactic_functions: $\mathbb{R} \times \mathbb{C} \times \mathbb{S} \times \mathbb{F}_{\mathcal{T}} \to \mathbb{C} \times \mathbb{F}_{\mathcal{T}}$
Operational_functions: $\mathbb{C} \times \mathbb{S} \times \mathbb{F}_{\mathcal{O}} \to \mathbb{S} \times \mathbb{F}_{\mathcal{O}}$

Example 12 – Public Bus Transport Management We relate to Example 11:

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

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

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

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

We omit formalisations. ■ End of Example 12

3.7. Human Behaviour

- By human behaviour we shall understand
  - those aspects of the behaviour of domain stakeholders
  - which have a direct bearing on the "functioning" of the domain
- Behaviours "fall" in a spectrum
  - from diligent
  - via sloppy
  - to delinquent and
  - outright criminal neglect

  in the observance of maintaining
  - entities,
  - carrying our actions and
  - responding to events.

Example 13 – Human Behaviour Cf. Examples 11–12:

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

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

62 occasional failures ··· as delinquent;

63 repeated patterns of failures ··· as criminal.

We omit showing somewhat "lengthy" formalisations.
. ■ End of Example 13

3.8. Discussion

- We have briefly outlined six concepts of domain facets and we have exemplified each of these.
- Real-scale domain descriptions are, of course, much larger than what we can show. Typically, say for the domain of logistics, a basic description is approximately 30 pages; for "small" parts of railway systems we easily get up to 100–200 pages – both including formalisations.
- You should now have gotten a reasonably clear idea as to what constitutes a domain description.

- As mentioned, in the introduction to this lecture, we shall not cover post-modelling activities such a validation and domain theory formation. The latter is usually part of the verification (theorem proving, model checking and formal testing) of the formal domain description.
- Final validation of a domain description is with respect to the narrative part of the narrative/formalisation pairs of descriptions.
- The reader should also be able to form a technical opinion about what can be formalised, and that not all can be formalised within the framework of a single formal specification language, cf. Sect. .

End of Lecture 3: DOMAIN ENGINEERING

Start of Lecture 4: REQUIREMENTS ENGINEERING

4. Requirements Engineering

- 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 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.

- 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 exemply the latter, then the former.

Example 14 – 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,

■ End of Example 14

Example 15 – 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.

■ End of Example 15

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

Example 16 – 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.

■ End of Example 16

Example 17 – Arguing Goal-satisfaction of Toll Road System Software

- By recording
  - tickets issued and collected at toll boths and
  - toll road hubs and links entered and left
  - as per the requirements specification brought in (forthcoming) Examples 19-23,
- we can eventually argue that
  - the requirements of (the forthcoming) Examples 19-23
  - help satisfy the goal of Example 15 on page 102.

■ End of Example 17

- We shall assume that the (goal and) requirements engineer elicit both $\mathcal{G}$oals and $\mathcal{R}$equirements from requirements stakeholders.
- But we shall focus only on
  - domain and
  - interface

  requirements such as "derived" from domain descriptions.

4.1. Business Process Re-engineering

- There are the business processes of the domain before installation of the required computing systems.
- The potential of installing computing systems invariably requires revision of established business processes.
- Business process re-engineering (BPR) is a development of new business processes
  - – whether or not complemented by computing and communication.
- BPR, such as we advocate it,
  - proceeds on the basis of an existing domain description and
  - outlines needed changes (additions, deletions, modifications) to entities, actions, events and behaviours
  - following the six domain facets.
- The goals help us formulate the BPR prescriptions.

Example 18 – Rough-sketching a Re-engineered Road Net

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

64 Re-engineered Entities:

- We shall focus on a linear sequence of toll road intersections (i.e., hubs) connected by pairs of one-way (opposite direction) toll roads (i.e., links).
- Each toll road intersection is connected by a two way road to a toll plaza.
- Each toll plaza contains a pair of sets of entry and exit toll booths.
- (Example 20 brings more details.)

65 Re-engineered Actions:

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

66 Re-engineered Events:

- A car entering the toll road net at a toll both plaza entry booth constitutes an event.
- A car leaving the toll road net at a toll both plaza entry booth constitutes an event.
- A car entering a toll road hub constitutes an event.
- A car entering a toll road link constitutes an event.

67 Re-engineered Behaviours:

- The journey of a car,
  - from entering the toll road net at a toll booth plaza,
  - via repeated visits to toll road intersections
  - interleaved with repeated visits to toll road links
  - to leaving the toll road net at a toll booth plaza,

  constitutes a behaviour — 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).

68 Re-engineered Intrinsics:

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

69 Re-engineered Support Technologies:

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

70 Re-engineered Rules and Regulations:

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

71 Re-engineered Scripts:

- No need.

72 Re-engineered Management and Organisation:

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

73 Re-engineered Human Behaviour:

- Humans, in this case car drivers, may not change their behaviour in the spectrum from diligent and accurate via sloppy and delinquent to outright traffic-law breaking – so we see no need for any “re-engineering”.

■ End of Example 18

4.2. Domain Requirements

- For the phase of domain requirements the requirements stakeholders "sit together" with the domain cum requirements engineers and read the domain description, line-by-line, in order to "derive" the domain requirements.
- They do so in five rounds (in which the BPR rough sketch is both regularly referred to and possibly, i.e., most likely regularly updated).
- Domain requirements are "derived" from the domain description.
- The goals then determine the derivations: which projections, instantiations, determinations, etcetera, to perform.

4.2.1. Projection

By *domain projection* we understand an operation

- that applies to a domain description
- and yields a domain requirements prescription.
- The latter represents a projection of the former
- in which only those parts of the domain are present
- that shall be of interest in the ongoing requirements development

Example 19 – Projection

- Our requirements is for a simple toll road:
  - a linear sequence of links and hubs outlined in Example 18:
    - ∗ see Items [1–11] of Example 1 on page 39
    - ∗ and Items [32–35] of Example 7 on page 68.

■ End of Example 19

4.2.2. Instantiation

- By *domain instantiation* we understand an operation
  - that applies to a (projected) domain description, i.e., a requirements prescription,
  - and yields a domain requirements prescription,
  - where the latter has been made more specific, usually by constraining a domain description.

Example 20 – Instantiation

- Here the toll road net topology as outlined in Example 18 on page 107 is introduced:
  - a straight sequence of toll road hubs
  - pairwise connected with pairs of one way links
  - and with each hub two way link connected to a toll road plaza.

type
$\mathsf{H, L, P = H}$
$\mathsf{N' = (H \times L) \times H \times ((L \times L) \times H \times (H \times L))^*}$
$\mathsf{N'' = \{|n:N' \cdot wf(n)|\}}$
value
$\mathsf{wf\_N'': N' \to}$ **Bool**
$\mathsf{wf\_N''((h,l),h',llhpl) \equiv}$ ... 6 lines ... !
$\alpha\mathsf{N: N'' \to N}$
$\alpha\mathsf{N((h,l),h',llhpl) \equiv}$ ... 2 lines ... !

- $\mathsf{wf\_N''}$ secures linearity;
- αN allows abstraction from more concrete $\mathsf{N''}$ to more abstract N.

■ End of Example 20

4.2.3. Determination

- By *domain determination* we understand an operation
  - that applies to a (projected and possibly instantiated) domain description, i.e., a requirements prescription,
  - and yields a domain requirements prescription,
  - where (attributes of) entities, actions, events and behaviours have been made less indeterminate.

Example 21 – Determination

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

```
type
  LΣ = (HI×HI)-set, LΩ = LΣ-set
  HΣ = (LI×LI)-set, HΩ = HΣ-set
  N′ = (H × L) × H × ((L × L) × H × (H × L))*
value
  ωLΣ: L → LΣ, ωLΩ: L → LΩ
  ωHΣ: H → HΣ, ωHΩ: H → HΩ
axiom
  ∀ ((h,l),h′,llhhl:⟨(l′,l″),h″,(h‴,l‴)⟩⌢llhhl′):N″ •
    ωLΣ(l)={(ωHI(h),ωHI(h′)),(ωHI(h′),ωHI(h))}∧
    ωLΣ(l‴)={(ωHI(h″),ωHI(h‴)),(ωHI(h‴),ωHI(h″))}∧
    ∀ i,i+1:Nat • {i,i+1}⊆inds llhhl ⇒
      let ((li,li′),hi,(hi″,li″))=llhhl(i), (_,hj,(hj″,lj″))=llhhl(i+1) in
      ωLΩ(li)= {{(ωHI(hi),ωHI(hj))}}∧ωLΩ(li′)={{(ωHI(hj),ωHI(hi))}}∧
      ωHΩ(hi)= { ... } ... 3 lines end
```

■ End of Example 21

4.2.4. Extension

- By *domain extension* we understand an operation
  - that applies to a (projected and possibly determined and instantiated) domain description, i.e., a (domain) requirements prescription,
  - and yields a (domain) requirements prescription.
  - The latter prescribes that a software system is to support, partially or fully, entities, operations, events and/or behaviours that were not feasible (or not computable in reasonable time or space) in a domain without computing support, but which are now are not only feasible but also computable in reasonable time and space.

Example 22 – Extension

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

■ End of Example 22

4.2.5. Fitting

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

Example 23 – Fitting

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

■ End of Example 23

4.2.6. Discussion:

- This section has very briefly surveyed and illustrated domain requirements.
- The reader should take cognizance of the fact that these are indeed “derived” from the domain description.
- They are not domain descriptions, but, once the business process re-engineering has been adopted
- and the required software has been installed,
- then the domain requirements become part of a revised domain description !

4.3. Interface Requirements

- By interface requirements we understand such requirements which are concerned with the phenomena and concepts *shared* between the domain and the machine.
- Thus such requirements can only be expressed using terms from both the domain and the machine.
- We tackle the problem of “deriving”, i.e., constructing interface requirements by tackling four “smaller” problems:
  - those of “deriving” interface requirements for
    - entities,
    - actions,
    - events and
    - behaviours

    respectively.
  - Again goals help state which phenomena and concepts are to be shared.

4.3.1. Entity Interfaces

- Entities that are shared between the domain and the machine must initially be input to the machine.
- Dynamically arising or attribute value changing entities must likewise be input and all such machine entities must have their attributes updated, when need arise.
- Requirements for shared entities thus entail
  - requirements for their representation
  - and for their human/machine and/or machine/machine transfer-dialogues.

Example 24 – Shared Entities

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

  can then be devised.
- Etc.

■ End of Example 24

4.3.2. Action Interfaces

- 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
  - (some measurable, time-varying entity attribute value, or some domain stakeholder)
  - in order to proceed in its computation.

Example 25 – Shared Actions

- In order for a car driver to leave an exit toll both the following component actions must take place:
  - the driver inserts the electronic pass in the exit toll booth machine;
  - the machine scans and accepts the ticket and calculates the fee for the car journey from entry booth via the toll road net to the exit booth;
  - the driver is alerted to the cost and is requested to pay this amount;
  - once paid the exit booth toll gate is raised.
- *Notice that a number of details of the new support technology is left out.*
- *It could either be elaborated upon here, or be part of the system design.*

■ End of Example 25

4.3.3. Event Interfaces

- 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.

Example 26 – Shared Events

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

■ End of Example 26

4.3.4. Behaviour Interfaces

- By a shared behaviour we understand
  - a sequence of zero, one or more
    - ∗ shared actions and
    - ∗ shared events.

Example 27 – Shared Behaviour

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

■ End of Example 27

4.3.5. Discussion

- Once the machine has been installed
- it, the machine, is part of the new domain !

4.4. Machine Requirements

- We shall not cover this stage of requirements development other than saying that it consists of the following concerns:
  - performance requirements (storage, speed, other resources),
  - dependability requirements (availability, accessibility, integrity, reliability, safety, security),
  - maintainability requirements (adaptive, extensional, corrective, perfective, preventive),
  - portability requirements (development platform, execution platform, maintenance platform, demo platform) and
  - documentation requirements.

- Only dependability seems to be subjectable to rigorous, formal treatment.
- The **discussions** of earlier carry over to this paragraph.
- That is, once the machine has been installed it, the machine, is part of the new domain !

End of Lecture 4: REQUIREMENTS ENGINEERING

Start of Lecture 13: CONCLUDING DISCUSSION & CONCLUSION

5. Discussion

- We discuss a number of issues that were left open above.

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

- The 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 those of 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 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.
  - Bertran 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 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,
  - 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. $\mathcal{D},\mathcal{S} \models \mathcal{R}$

- 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 .

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 needes.

6. Conclusion

- These lecture slides are based on the paper:

```
From Domains to Requirements
Submitted for publication
December 7, 2009
```

- Versions of that paper are found on the Internet”

```
www.imm.dtu.dk/~db/short-from-domains-to-requirements.p
www.imm.dtu.dk/~db/long-from-domains-to-requirements.pd
```

  - The examples of the short version are without formulas.
  - The examples of the long version are with formulas.

(6. **Conclusion**)

- The idea of extending that (8-11 page two column) paper
  - into a brief set of lectures notes and slides
  - arose in connection with the author's
  - April 2010 lectures at the Technical University of Vienna.
- In addition to a normal format paper
  - a full-fledged "RSL primer",
  - a number of clarifying methodology sections and
  - further examples

  have been added as appendices.

(6. **Conclusion**)

- The formalisations of these lecture notes (and slides)
  - which are expressed in RSL,
  - the RAISE Specification Languange,
  - can be expressed in either of
    - ∗ Alloy,
    - ∗ Event B,
    - ∗ VDM-SL or
    - ∗ Z.
  - The present author
    - ∗ would like to work with "enthusiasts" (i.e., "followers")
    - ∗ of the above-listed specification languages
    - ∗ to achieve versions of these lecture notes (and slides)
    - ∗ for any and all of these other formal specification languages.

(6. **Conclusion**)

End of Lecture 13: CONCLUDING DISCUSSION & CONCLUSION

(. **Conclusion**)

Start of Lecture 6: RSL: TYPES

A. An RSL Primer

A.1. Types

A.1.1. Type Expressions

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

A.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 28 – Basic Net Attributes:

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

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

■ End of Example 28

A.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 29 – Composite Net Type Expressions:

- The type clauses of function signatures:

value
  f: $A \rightarrow B$

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

value

ωHIs: L $\rightarrow$ HI-**set**	Example 1 Item [5]
ωLIs: H $\rightarrow$ LI-**set**	Example 1 Item [6]
ωHΣ: H $\rightarrow$ HT-**set**	Example 1 Item [10]
set_HΣ: H $\times$ HΣ $\rightarrow$ H	Example 2 Item [12]

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

type

N = H-**set** $\times$ L-**set**	Example 1 Item [2]
HT = LI $\times$ HI $\times$ LI	Example 1 Item [9]
LT$'$ = HI $\times$ LI $\times$ HI	Example 7 Item [32]

■ End of Example 29

A.1.2. Type Definitions

A.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

Example 30 – 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
  ωHs: $N_\alpha \to$ H-**set**
  ωLs: $N_\alpha \to$ L-**set**
axiom
  $\forall$ n:N_α · **card** ωHs(n)$\geq$2 $\wedge$ **card** ωLs(n)$\geq$1 $\wedge$ ...

- **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** × L-**set**
value
  wf_N_β: $N_\beta \to$ **Bool**
  wf_N_β(hs,ls) ≡ **card** hs≥2 ∧ **card** ls≥1 ....
  inject_N_β: $N_\alpha \xrightarrow{\sim} N_\beta$ **pre**: wf_N_β(hs,ls)
  inject_N_β(n_α) ≡ (ωHs(n_α),ωLs(n_α))

- **Cartesians + Maps + Wellformedness:** Or a net can be modelled as a triple of
  - hubs (modelled as a map from hub identfiers to hubs),
  - links (modelled as a map from link identfiers to links), and
  - a graph from hub h_i identifiers h_{i_i} to maps from identfiers l_{ij_i} of hub h_i connected links l_{ij} to the identfiers h_{j_i} of link connected hubs h_j.

```
type
  [sorts] H, HI, L, LI
     N_γ = HUBS × LINKS × GRAPH
  [a] HUBS = HI →m H
  [b] LINKS = LI →m L
  [c] GRAPH = HI →m (LI −m> HI)
```

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

- Example 39 on page 231 defines the well-formedness predicates for the above map types.

■ End of Example 30

- Schematic type definitions:

```
[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 31 – 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.[3] The removal command specifies a link identifier.

[3]– 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 ?? on page ?? presents the semantics functions for *int_Insert* and *int_Remove*. ■ End of Example 31

A.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
  $A = \{|\ b{:}B \cdot \mathcal{P}(b)\ |\}$

Example 32 – Net Subtypes:

- In Example 30 on page 171 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 30
  - 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 ωHs(n)≥2 ∧ card ωLs(n)≥1 ∧
    [5−−8] of example 1
```

- In the **Cartesians + Wellformedness** part of Example 30
  - 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 30
  - 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 39 on page 231 presents a definition of *wf_GRAPH*.

■ End of Example 32

A.1.2.3. Sorts — Abstract Types

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

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

Example 33 – Net Sorts:

- In formula lines of Examples 30–32
- we have indicated those **type** clauses which define *sorts*,
- by bracketed [sorts] literals.

■ End of Example 33

End of Lecture 6: RSL: TYPES

Start of Lecture 7: RSL: VALUES & OPERATIONS

A.2. Concrete RSL Types: Values and Operations

A.2.1. Arithmetic

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

A.2.2. Set Expressions

A.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 34 – 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
  $G' = C\text{-}\mathbf{set},\ G = \{|\ g{:}G' \cdot g \neq \{\}\ |\}$
  $S = G\text{-}\mathbf{set}$
  $L' = C\text{-}\mathbf{set},\ L = \{|\ \ell{:}L' \cdot \ell \neq \{\}\ |\}$
  $NW' = S,\ NW = \{|\ s{:}S \cdot wf\_S(s)\ |\}$
value
  $wf\_S{:}\ S \to \mathbf{Bool}$
  $wf\_S(s) \equiv \forall\ g{:}G \cdot g \in s \Rightarrow \exists\ g'{:}G \cdot\ g' \in s \wedge share(g,g')$
  $share{:}\ G \times G \to \mathbf{Bool}$
  $share(g,g') \equiv g \neq g' \wedge g \cap g' \neq \{\}$
  $liaisons{:}\ G \times G \to L$
  $liaisons(g,g') = g \cap g'\ \mathbf{pre}\ share(g,g')$

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 34

A.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
$P = A \rightarrow$ **Bool**
$Q = A \overset{\sim}{\rightarrow} B$
value
comprehend: A-**infset** $\times$ P $\times$ Q $\rightarrow$ B-**infset**
comprehend(s,P,Q) $\equiv$ { Q(a) | a:A $\cdot$ a $\in$ s $\wedge$ P(a)}

Example 35 – Set Comprehensions:

- Example 30 on page 171 illustrates, in the **Cartesians + Maps + Wellformedness** part the following set comprehensions in the *wf_N(hs,ls,g)* wellformedness predicate definition:
  - [d] ∪ {**dom** *g(hi)*|*hi:HI* · *hi* ∈ **dom** *g*}
    - ∗ It expresses the distributed union
    - ∗ of sets (**dom** *g(hi)*) of link identfiers
    - ∗ (for each of the *hi* indexed maps from (definition set, **dom**) link identiers
    - ∗ to (range set, **rng**) hub identifiers, where *hi:HI* ranges over **dom** *g*).

- [e] ∪ {**rng** *g(hi)*|*hi:HI* · *hi* ∈ **dom** *g*}
  - ∗ It expresses the distributed union
  - ∗ of sets (**rng** *g(hi)*) of hub identfiers
  - ∗ (for each of the *hi* indexed maps from (definition set, **dom**) link identiers
  - ∗ to (range set, **rng**) hub identifiers, where *hi:HI* ranges over **deom** *g*).

■ End of Example 35

A.2.3. Cartesian Expressions

A.2.3.1. Cartesian Enumerations

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

type
A, B, ..., C
A $\times$ B $\times$ ... $\times$ C
value
(e1,e2,...,en)

Example 36 – 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 36

A.2.4. List Expressions

A.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.

A.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 37 – Routes in Nets:

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

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

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

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

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

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

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

  is_looped: PR → Bool
  is_looped(pr) ≡ ∃ i,j:Nat • i≠j∧{i,j}⊆index pr ⇒ ωHls(pr(i))∩ωHls(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 37

A.2.5. Map Expressions

A.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
  $M = T1 \xrightarrow[m]{} T2$
value
  u,u1,u2,...,un:T1, v,v1,v2,...,vn:T2
  $\{[\,], [u \mapsto v], ..., [u1 \mapsto v1, u2 \mapsto v2, ..., un \mapsto vn], ...\} \subseteq M$

A.2.5.2. Map Comprehension

- The last line below expresses map comprehension:

type
  U, V, X, Y
  $M = U \xrightarrow[m]{} V$
  $F = U \xrightarrow{\sim} X$
  $G = V \xrightarrow{\sim} Y$
  $P = U \to \textbf{Bool}$
value
  $\text{comprehend}: M \times F \times G \times P \to (X \xrightarrow[m]{} Y)$
  comprehend(m,F,G,P) ≡
    $[\, F(u) \mapsto G(m(u)) \mid u{:}U \cdot u \in \textbf{dom}\ m \wedge P(u) \,]$

Example 38 – Concrete Net Type Construction:

- We Define a function *con[struct]_*N_γ (of the **Cartesians + Maps + Wellformedness** part of Example 30.
  - The base of the construction is the fully abstract sort definition of N_α in the **Sorts + Observers + Axioms** part of Example 30 – 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 30.
  - First we recall the ssential types of that N_γ.

```
type
  $N_\gamma$ = HUBS × LINKS × GRAPH
  HUBS = HI $\overrightarrow{m}$ H
  LINKS = LI $\overrightarrow{m}$ L
  GRAPH = HI $\overrightarrow{m}$ (LI $\overrightarrow{m}$ HI)
value
  con_$N_\gamma$: $N_\alpha \to N_\gamma$
  con_$N_\gamma(n_\alpha) \equiv$
    let hubs = [ωHI(h) ↦ h | h:H • h ∈ ωHs($n_\alpha$)],
        links = [ωLI(h) ↦ l | l:L • l ∈ ωLs($n_\alpha$)],
        graph = [ωHI(h) ↦ [ωLI(l) ↦ ι(ωHIs(l)\{ωHI(h)})
                              | l:L • l ∈ ωLs($n_\alpha$)∧li ∈ ωLIs(h)]
                   | H:h • h ∈ ωHs($n_\alpha$)] in
        (hubs.links,graph) end

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

theorem:

n_α **satisfies axioms** [2,5–8] for N of Example 1 $\Rightarrow$ wf_$N_\gamma$con_$N_\gamma(n_\alpha)$

■ End of Example 38

A.2.6. Set Operations

A.2.6.1. Set Operator Signatures

```
value
  9  ∈: A × A-infset → Bool
  10 ∉: A × A-infset → Bool
  11 ∪: A-infset × A-infset → A-infset
  12 ∪: (A-infset)-infset → A-infset
  13 ∩: A-infset × A-infset → A-infset
  14 ∩: (A-infset)-infset → A-infset
  15 \: A-infset × A-infset → A-infset
  16 ⊂: A-infset × A-infset → Bool
  17 ⊆: A-infset × A-infset → Bool
  18 =: A-infset × A-infset → Bool
  19 ≠: A-infset × A-infset → Bool
  20 card: A-infset $\xrightarrow{\sim}$ Nat
```

A.2.6.2. Set Examples

examples

$a \in \{a,b,c\}$

$a \notin \{\}$, $a \notin \{b,c\}$

$\{a,b,c\} \cup \{a,b,d,e\} = \{a,b,c,d,e\}$

$\cup\{\{a\},\{a,b\},\{a,d\}\} = \{a,b,d\}$

$\{a,b,c\} \cap \{c,d,e\} = \{c\}$

$\cap\{\{a\},\{a,b\},\{a,d\}\} = \{a\}$

$\{a,b,c\} \setminus \{c,d\} = \{a,b\}$

$\{a,b\} \subset \{a,b,c\}$

$\{a,b,c\} \subseteq \{a,b,c\}$

$\{a,b,c\} = \{a,b,c\}$

$\{a,b,c\} \neq \{a,b\}$

$\mathbf{card}\ \{\} = 0$, $\mathbf{card}\ \{a,b,c\} = 3$

A.2.6.3. Informal Explication

9. $\in$: The membership operator expresses that an element is a member of a set.
10. $\notin$: The nonmembership operator expresses that an element is not a member of a set.
11. $\cup$: The infix union operator. When applied to two sets, the operator gives the set whose members are in either or both of the two operand sets.
12. $\cup$: The distributed prefix union operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.
13. $\cap$: The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.
14. $\cap$: The prefix distributed intersection operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.

15. $\setminus$: The set complement (or set subtraction) operator. When applied to two sets, the operator gives the set whose members are those of the left operand set which are not in the right operand set.
16. $\subseteq$: The proper subset operator expresses that all members of the left operand set are also in the right operand set.
17. $\subset$: The proper subset operator expresses that all members of the left operand set are also in the right operand set, and that the two sets are not identical.
18. $=$: The equal operator expresses that the two operand sets are identical.
19. $\neq$: The nonequal operator expresses that the two operand sets are *not* identical.
20. **card**: The cardinality operator gives the number of elements in a finite set.

A.2.6.4. Set Operator Definitions

value

$s' \cup s'' \equiv \{\ a \mid a{:}A \cdot a \in s' \vee a \in s''\ \}$

$s' \cap s'' \equiv \{\ a \mid a{:}A \cdot a \in s' \wedge a \in s''\ \}$

$s' \setminus s'' \equiv \{\ a \mid a{:}A \cdot a \in s' \wedge a \notin s''\ \}$

$s' \subseteq s'' \equiv \forall\ a{:}A \cdot a \in s' \Rightarrow a \in s''$

$s' \subset s'' \equiv s' \subseteq s'' \wedge \exists\ a{:}A \cdot a \in s'' \wedge a \notin s'$

$s' = s'' \equiv \forall\ a{:}A \cdot a \in s' \equiv a \in s'' \equiv s{\subseteq}s' \wedge s'{\subseteq}s$

$s' \neq s'' \equiv s' \cap s'' \neq \{\}$

$\mathbf{card}\ s \equiv$

$\quad \mathbf{if}\ s = \{\}\ \mathbf{then}\ 0\ \mathbf{else}$

$\quad \mathbf{let}\ a{:}A \cdot a \in s\ \mathbf{in}\ 1 + \mathbf{card}\ (s \setminus \{a\})\ \mathbf{end}\ \mathbf{end}$

$\quad \mathbf{pre}\ s$ /∗ is a finite set ∗/

$\mathbf{card}\ s \equiv \mathbf{chaos}$ /∗ tests for infinity of s ∗/

A.2.7. Cartesian Operations

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

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**

A.2.8. List Operations

A.2.8.1. List Operator Signatures

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

A.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$\rangle$

A.2.8.3. Informal Explication

- **hd**: Head gives the first element in a nonempty list.
- **tl**: Tail gives the remaining list of a nonempty list when Head is removed.
- **len**: Length gives the number of elements in a finite list.
- **inds**: Indices give the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems**: Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$: Indexing with a natural number, i larger than 0, into a list ℓ having a number of elements larger than or equal to i, gives the ith element of the list.

- $\frown$: Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.
- $=$: The equal operator expresses that the two operand lists are identical.
- $\neq$: The nonequal operator expresses that the two operand lists are *not* identical.

The operations can also be defined as follows:

A.2.8.4. List Operator Definitions

```
value
  is_finite_list: A^ω → Bool

  len q ≡
    case is_finite_list(q) of
      true → if q = ⟨⟩ then 0 else 1 + len tl q end,
      false → chaos end

  inds q ≡
    case is_finite_list(q) of
      true → { i | i:Nat • 1 ≤ i ≤ len q },
      false → { i | i:Nat • i≠0 } end

  elems q ≡ { q(i) | i:Nat • i ∈ inds q }
```

```
  q(i) ≡
    case (q,i) of
      (⟨⟩,1) → chaos,
      (_,1) → let a:A,q′:Q • q=⟨a⟩⌢q′ in a end
      _ → q(i−1)
    end

  fq ⌢ iq ≡
      ⟨ if 1 ≤ i ≤ len fq then fq(i) else iq(i − len fq) end
       | i:Nat • if len iq≠chaos then i ≤ len fq+len end ⟩
    pre is_finite_list(fq)

  iq′ = iq″ ≡
    inds iq′ = inds iq″ ∧ ∀ i:Nat • i ∈ inds iq′ ⇒ iq′(i) = iq″(i)

  iq′ ≠ iq″ ≡ ∼(iq′ = iq″)
```

A.2.9. Map Operations

A.2.9.1. Map Operator Signatures and Map Operation Examples

```
value
  m(a): M → A $\overset{\sim}{\rightarrow}$ B, m(a) = b

  dom: M → A-infset [domain of map]
    dom [a1↦b1,a2↦b2,...,an↦bn] = {a1,a2,...,an}

  rng: M → B-infset [range of map]
    rng [a1↦b1,a2↦b2,...,an↦bn] = {b1,b2,...,bn}

  †: M × M → M [override extension]
    [a↦b,a′↦b′,a″↦b″] † [a′↦b″,a″↦b′] = [a↦b,a′↦b″,a″↦b′]
```

$\cup$: M × M → M [merge ∪]
  $[a \mapsto b, a' \mapsto b', a'' \mapsto b''] \cup [a''' \mapsto b'''] = [a \mapsto b, a' \mapsto b', a'' \mapsto b'', a''' \mapsto b''']$

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

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

$=, \neq$: M × M → **Bool**

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

A.2.9.2. Map Operation Explication

- $m(a)$: Application gives the element that a maps to in the map m.
- **dom**: Domain/Definition Set gives the set of values which *maps to* in a map.
- **rng**: Range/Image Set gives the set of values which *are mapped to* in a map.
- †: Override/Extend. When applied to two operand maps, it gives the map which is like an override of the left operand map by all or some "pairings" of the right operand map.
- ∪: Merge. When applied to two operand maps, it gives a merge of these maps.

- \: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements that are not in the right operand set.
- /: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements of the right operand set.
- =: The equal operator expresses that the two operand maps are identical.
- ≠: The nonequal operator expresses that the two operand maps are *not* identical.
- ∘: Composition. When applied to two operand maps, it gives the map from definition set elements of the left operand map, m_1, to the range elements of the right operand map, m_2, such that if a is in the definition set of m_1 and maps into b, and if b is in the definition set of m_2 and maps into c, then a, in the composition, maps into c.

Example 39 – Miscellaneous Net Expressions: Maps: Example 30 on page 171 left out defining the well-formedness of the map types:

```
value
    wf_HUBS: HUBS → Bool
  [a] wf_HUBS(hubs) ≡ ∀ hi:HI • hi ∈ dom hubs ⇒ ωHIhubs(hi)=hi
    wf_LINKS: LINKS → Bool
  [b] wf_LINKS(links) ≡ ∀ li:LI • li ∈ dom links ⇒ ωLIlinks(li)=li
  wf_Nγ: Nγ → Bool
  wf_Nγ(hs,ls,g) ≡
    [c] dom hs = dom g ∧
    [d] ∪ {dom g(hi)|hi:HI • hi ∈ dom g} = dom links ∧
    [e] ∪ {rng g(hi)|hi:HI • hi ∈ dom g} = dom g ∧
    [f] ∀ hi:HI • hi ∈ dom g ⇒ ∀ li:LI • li ∈ dom g(hi) ⇒ (g(hi))(li)≠hi
    [g] ∀ hi:HI • hi ∈ dom g ⇒ ∀ li:LI • li ∈ dom g(hi) ⇒
          ∃ hi′:HI • hi′ ∈ dom g ⇒ ∃ ! li:LI • li ∈ dom g(hi) ⇒
            (g(hi))(li) = hi′ ∧ (g(hi′))(li) = hi
```

- [c] *HUBS* record the same hubs as do the net corresponding *GRAPHS* ($\mathbf{dom}\ hs = \mathbf{dom}\ g \wedge$).
- [d] *GRAPHS* record the same links as do the net corresponding *LINKS* ($\cup \{\mathbf{dom}\ g(hi) | hi{:}HI \cdot hi \in \mathbf{dom}\ g\} = \mathbf{dom}\ links$).
- [e] The target (or range) hub identifiers of graphs are the same as the domain of the graph ($\cup \{\mathbf{rng}\ g(hi) | hi{:}HI \cdot hi \in \mathbf{dom}\ g\} = \mathbf{dom}\ g$), that is none missing, no new ones !
- [f] No links emanate from and are incident upon the same hub ($\forall\ hi{:}HI \cdot hi \in \mathbf{dom}\ g \Rightarrow \forall\ li{:}LI \cdot li \in \mathbf{dom}\ g(hi) \Rightarrow (g(hi))(li) \neq hi$).
- [g] If there is a link from one hub to another in the *GRAPH*, then the same link also connects the other hub to the former ($\forall\ hi{:}HI \cdot hi \in \mathbf{dom}\ g \Rightarrow \forall\ li{:}LI \cdot li \in \mathbf{dom}\ g(hi) \Rightarrow \exists\ hi'{:}HI \cdot hi' \in \mathbf{dom}\ g \Rightarrow \exists\ !\ li{:}LI \cdot li \in \mathbf{dom}\ g(hi) \Rightarrow (g(hi))(li) = hi' \wedge (g(hi'))(li) = hi$).

■ End of Example 39

A.2.9.3. Map Operation Redefinitions

value

$$\mathbf{rng}\ \mathrm{m} \equiv \{\ \mathrm{m(a)}\ |\ \mathrm{a{:}A} \cdot \mathrm{a} \in \mathbf{dom}\ \mathrm{m}\ \}$$

$$\begin{array}{l} \mathrm{m1} \dagger \mathrm{m2} \equiv \\ \quad [\ \mathrm{a} \mapsto \mathrm{b}\ |\ \mathrm{a{:}A,b{:}B} \cdot \\ \quad\quad \mathrm{a} \in \mathbf{dom}\ \mathrm{m1} \setminus \mathbf{dom}\ \mathrm{m2} \wedge \mathrm{b{=}m1(a)} \vee \mathrm{a} \in \mathbf{dom}\ \mathrm{m2} \wedge \mathrm{b{=}m2(a)}\ ] \end{array}$$

$$\begin{array}{l} \mathrm{m1} \cup \mathrm{m2} \equiv [\ \mathrm{a} \mapsto \mathrm{b}\ |\ \mathrm{a{:}A,b{:}B} \cdot \\ \quad\quad \mathrm{a} \in \mathbf{dom}\ \mathrm{m1} \wedge \mathrm{b{=}m1(a)} \vee \mathrm{a} \in \mathbf{dom}\ \mathrm{m2} \wedge \mathrm{b{=}m2(a)}\ ] \end{array}$$

$$\begin{array}{l} \mathrm{m} \setminus \mathrm{s} \equiv [\ \mathrm{a} \mapsto \mathrm{m(a)}\ |\ \mathrm{a{:}A} \cdot \mathrm{a} \in \mathbf{dom}\ \mathrm{m} \setminus \mathrm{s}\ ] \\ \mathrm{m}\ /\ \mathrm{s} \equiv [\ \mathrm{a} \mapsto \mathrm{m(a)}\ |\ \mathrm{a{:}A} \cdot \mathrm{a} \in \mathbf{dom}\ \mathrm{m} \cap \mathrm{s}\ ] \end{array}$$

$$\begin{array}{l} \mathrm{m1} = \mathrm{m2} \equiv \\ \quad \mathbf{dom}\ \mathrm{m1} = \mathbf{dom}\ \mathrm{m2} \wedge \forall\ \mathrm{a{:}A} \cdot \mathrm{a} \in \mathbf{dom}\ \mathrm{m1} \Rightarrow \mathrm{m1(a)} = \mathrm{m2(a)} \\ \mathrm{m1} \neq \mathrm{m2} \equiv \sim(\mathrm{m1} = \mathrm{m2}) \end{array}$$

$$\begin{array}{l} \mathrm{m}^\circ\mathrm{n} \equiv \\ \quad [\ \mathrm{a} \mapsto \mathrm{c}\ |\ \mathrm{a{:}A,c{:}C} \cdot \mathrm{a} \in \mathbf{dom}\ \mathrm{m} \wedge \mathrm{c} = \mathrm{n(m(a))}\ ] \\ \quad \mathbf{pre}\ \mathbf{rng}\ \mathrm{m} \subseteq \mathbf{dom}\ \mathrm{n} \end{array}$$

End of Lecture 7: RSL: VALUES & OPERATIONS

Start of Lecture 8: RSL: PREDICATE CALCULUS and λ–CALCULUS

A.3. The RSL Predicate Calculus

A.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 ∼a, a∧b, a∨b, a⇒b, a=b, a≠b

- are propositional expressions having Boolean values.
- ∼, ∧, ∨, ⇒, = and ≠ are Boolean connectives (i.e., operators).
- They can be read as: *not*, *and*, *or*, *if then* (or *implies*), *equal* and *not equal*.

A.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
∼a, a∧b, a∨b, a⇒b, a=b, a≠b
x=y, x≠y,
i<j, i≤j, i≥j, i≠j, i≥j, i>j

- are simple predicate expressions.

A.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:

∀ x:X · $\mathcal{P}(x)$
∃ y:Y · $\mathcal{Q}(y)$
∃ ! z:Z · $\mathcal{R}(z)$

- are quantified expressions — also being predicate expressions.

Example 40 – Predicates Over Net Quantities:

- From earlier examples we show some predicates:
- Example 28: Right hand side of function definition *is_two_way_link(l)*:
  $\exists\ l\sigma{:}L\Sigma \cdot l\sigma \in \omega H\Sigma(l) \wedge \mathbf{card}\ l\sigma{=}2$

- Example 30:
  - The **Sorts + Observers + Axioms** part:
    - ∗ Right hand side of the wellformedness function *wf_N(n)* definition:
      $\forall\ n{:}N \cdot \mathbf{card}\,\omega Hs(n){\geq}2 \wedge \mathbf{card}\,\omega Ls(n){\geq}1 \wedge [\,5{-}{-}8\,]$ *of example 1*
    - ∗ Right hand side of the wellformedness function *wf_N(hs,ls)* definition:
      $\mathbf{card}\ hs{\geq}2 \wedge \mathbf{card}\ ls{\geq}1$ ...

- The **Cartesians + Maps + Wellformedness** part:
  - ∗ Right hand side of the *wf_HUBS* wellformedness function definition:
    $\forall\ hi{:}HI \cdot hi \in \mathbf{dom}\ hubs \Rightarrow \omega HIhubs(hi){=}hi$
  - ∗ Right hand side of the *wf_LINKS* wellformedness function definition:
    $\forall\ li{:}LI \cdot li \in \mathbf{dom}\ links \Rightarrow \omega LIlinks(li){=}li$
  - ∗ Right hand side of the *wf_N(7 hs,ls,g)* wellformedness function definition:

$[\,c\,]\ \mathbf{dom}\ hs = \mathbf{dom}\ g \wedge$
$[\,d\,] \cup \{\mathbf{dom}\ g(hi) | hi{:}HI \cdot hi \in \mathbf{dom}\ g\} = \mathbf{dom}\ links \wedge$
$[\,e\,] \cup \{\mathbf{rng}\ g(hi) | hi{:}HI \cdot hi \in \mathbf{dom}\ g\} = \mathbf{dom}\ g \wedge$
$[\,f\,]\ \forall\ hi{:}HI \cdot hi \in \mathbf{dom}\ g \Rightarrow \forall\ \ li{:}LI \cdot li \in \mathbf{dom}\ g(hi) \Rightarrow (g(hi))(li){\neq}hi$
$[\,g\,]\ \forall\ hi{:}HI \cdot hi \in \mathbf{dom}\ g \Rightarrow \forall\ li{:}LI \cdot li \in \mathbf{dom}\ g(hi) \Rightarrow$
$\quad \exists\ hi'{:}HI \cdot hi' \in \mathbf{dom}\ g \Rightarrow \exists\ !\ li{:}LI \cdot li \in \mathbf{dom}\ g(hi) \Rightarrow$
$\quad\quad (g(hi))(li) = hi' \wedge (g(hi'))(li) = hi$

■ End of Example 40

A.4. λ-Calculus + Functions

A.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), ...

A.4.2. Free and Bound Variables

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

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

A.4.3. Substitution

- **subst**([N/x]x) $\equiv$ N;
- **subst**([N/x]a) $\equiv$ a,

  for all variables a$\neq$ x;
- **subst**([N/x](P Q)) $\equiv$ (**subst**([N/x]P) **subst**([N/x]Q));
- **subst**([N/x]($\lambda x \cdot P$)) $\equiv$ λ y$\cdot$P;
- **subst**([N/x](λ y$\cdot$P)) $\equiv$ $\lambda y \cdot$ **subst**([N/x]P),

  if x$\neq$y and y is not free in N or x is not free in P;
- **subst**([N/x](λy$\cdot$P)) $\equiv$ λz$\cdot$**subst**([N/z]**subst**([z/y]P)),

  if y$\neq$x and y is free in N and x is free in P

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

A.4.4. α-Renaming and β-Reduction

- α-renaming: λx$\cdot$M

  If x, y are distinct variables then replacing x by y in λx$\cdot$M results in λy$\cdot$**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$\cdot$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$\cdot$M)(N) $\equiv$ **subst**([N/x]M)

A.4.5. An Example

Example 41 – 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 *Position* is either on a link or at a hub.
- An *onLink* 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 *atHub* position just designates the hub (by its identifier).
- Traffic, *tf:TF*, is now a continuous function from *Time* 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
  ∀ ts:TS • ∃ tmin,tmax:T: tmin ∈ ts ∧ tmax ∈ ts ∧ ∀ t:T • t ∈ ts ⇒ tmin ≤ t ≤ tmax
  [that is: ts = {MIN(ts)..MAX(ts)}]
type
  VP = V →m Pos
  TF′ = T → VP,                              TF = {|tf:TF′•wf_TF(tf)(n)|}
  Pos = onL | atH
  onL == mkLPos(hi:HI,li:LI,f:F,hi:HI),    atH == mkHPos(hi:HI)
value
  wf_TF: TF→ N → Bool
  wf_TF(tf)(n) ≡ ...
  DOMAIN: TF → TS
  MIN,MAX: TS → T
```

- We have defined the continuous, composite entity of traffic.
- Now let us define an operation of inserting a vehicle in a traffic.
- To insert a vehicle, v, in a traffic, tf, is prescribable as follows:
  - the vehicle, v, must be designated;
  - a time point, t, “inside” the traffic tf must be stated;
  - a traffic, vtf, from time t of vehicle v must be stated;
  - as well as traffic, tf, into which vtf is to be “merged”.
- The resulting traffic is referred to as tf'.

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

- The function *insert_V* is here defined in terms of a pair of pre/post conditions.
- The pre-condition can be prescribed as follows:
  - The insertion time t must be within to open interval of time points in the traffic tf to which insertion applies.
  - The vehicle v must not be among the vehicle positions of tf.
  - The vehicle must be the only vehicle “contained” in the “inserted” traffic vtf.

```
pre: MIN(DOMAIN(tf)≤t≤MAX(DOMAIN(tf)) ∧
     ∀ t′:T • t′ ∈ DOMAIN(tf) ⇒ v ∉ dom tf(t′) ∧
     MIN(DOMAIN(vtf)) = t ∧
     ∀ t′:T•t′ ∈ DOMAIN(vtf) ⇒ 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)}
```

A.4.6. Function Signatures

For sorts we may want to postulate some functions:

```
type
 A, B, ..., C
value
 ωB: A → B
 ...
 ωC: 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 42 – Hub and Link Observers:

- Let a net with several hubs and links be presented.
- Now observer functions
  - ωHs and
  - ωLs

  can be demonstrated:
  - one simply "walks" along the net, pointing out
  - this hub and
  - that link,
  - one-by-one
  - until all the net has been visited.

- The observer functions
  - ωHI and
  - ωLI

  can be likewise demonstrated, for example:
  - when a hub is "visited"
  - its unique identification
  - can be postulated (and "calculated")
  - to be the unique geographic position of the hub
  - one which is not overlapped by any other hub (or link),
- and likewise for links. ■ End of Example 42

A.4.7. 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 $\xrightarrow{\sim}$ 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$\xrightarrow{\sim}$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, ..., can be defined in terms of axioms
- over function identifiers, f, g, ..., and over identiers of function arguments and results.

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

Example 43 – Axioms over Hubs, Links and Their Observers:

- Example 1 on page 39 Items [4]–[8]
- clearly demonstrates how a number of entities and observer functions are constrained
- (that is, partially defined)
- by function signatures and axioms. ■ End of Example 43

End of Lecture 8: RSL: PREDICATE CALCULUS and λ–CALCULUS

Start of Lecture 9: RSL: APPLICATIVE CONSTRUCTS

A.5. Other Applicative Expressions

A.5.1. Simple let Expressions

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

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

is an “expanded” form of:

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

A.5.2. Recursive let Expressions

Recursive **let** expressions are written as:

let f = λa•E(f,a) **in** B(f,a) **end**
let f = (λgλa•E(g,a))(f) **in** B(f.a) **end**
let f = F(f) **in** E(f,a) **end where** F ≡ λgλa•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*".

A.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**.

A.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 ⟨a⟩⌢ℓ = list **in** ... **end**
let ⟨a,_,b⟩⌢ℓ = list **in** ... **end**

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

A.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 44 – Choice Pattern Case Expressions: Insert Links: We consider the meaning of the Insert operation designators.

21. 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.
22. 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.

23. 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.
24. 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.

25. 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
  21   int_Insert: Insert → N ⥲ N
  22′  pre_int_Insert: Ins → N → Bool
  22″  pre_int_Insert(Ins(op))(hs,ls) ≡
⋆2             s_l(op)∉ ls ∧ obs_LI(s_l(op)) ∉ iols(ls) ∧
    case op of
  23)      2oldH(hi′,l,hi″) → {hi′,hi″}∈ iohs(hs),
  24)      1oldH1newH(hi,l,h) →
          hi ∈ iohs(hs) ∧ h∉ hs ∧ obs_HI(h)∉ iohs(hs),
  25)      2newH(h′,l,h″) →
          {h′,h″}∩ hs={} ∧ {obs_HI(h′),obs_HI(h″)}∩ iohs(hs)={}
    end
```

26. 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.
27. 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

26: xtr_H: HI $\rightarrow$ N $\stackrel{\sim}{\rightarrow}$ H

26: xtr_H(hi)(hs,_) $\equiv$ **let** h:H•h $\in$ hs $\wedge$ obs_HI(h)=hi **in** h **end**

  pre hi $\in$ iohs(hs)

27: xtr_L: HI $\rightarrow$ N $\stackrel{\sim}{\rightarrow}$ H

27: xtr_L(li)(_,ls) $\equiv$ **let** l:L•l $\in$ ls $\wedge$ obs_LI(l)=li **in** l **end**

  pre li $\in$ iols(ls)

28. 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.
29. 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 $\times$ LI $\rightarrow$ H, rLI: H $\times$ LI $\stackrel{\sim}{\rightarrow}$ H

28: aLI(h,li) **as** h′

  pre li $\notin$ obs_LIs(h)

  post obs_LIs(h′) $=$ {li} $\cup$ obs_LIs(h) $\wedge$ non_l_eq(h,h′)

29: rLI(h′,li) **as** h

  pre li $\in$ obs_LIs(h′) $\wedge$ **card** obs_LIs(h′)$\geq$2

  post obs_LIs(h) $=$ obs_LIs(h′) $\setminus$ {li} $\wedge$ non_l_eq(h,h′)

30. If the Insert command is of kind 2newH(h',l,h'') then the updated net of hubs and links, has
    - the hubs hs joined, $\cup$, by the set {h′,h″} and
    - the links ls joined by the singleton set of {l}.
31. If the Insert command is of kind 1oldH1newH(hi,l,h) then the updated net of hubs and links, has

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

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

31.2 : The set of links augmented by the new link.

32. If the Insert command is of kind 2oldH(hi',l,hi'') then

32.1–.2 : the two connecting hubs are updated to reflect the new link,

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

```
int_Insert(op)(hs,ls) ≡
⋆i   case op of
30     2newH(h′,l,h″) → (hs ∪ {h′,h″},ls ∪ {l}),
31     1oldH1newH(hi,l,h) →
31.1     let h′ = aLI(xtr_H(hi,hs),obs_LI(l)) in
31.2     (hs\{xtr_H(hi,hs)}∪{h,h′},ls ∪{l}) end,
32     2oldH(hi′,l,hi″) →
32.1     let hsδ = {aLI(xtr_H(hi′,hs),obs_LI(l)),
32.2                aLI(xtr_H(hi″,hs),obs_LI(l))} in
32.3     (hs\{xtr_H(hi′,hs),xtr_H(hi″,hs)}∪ hsδ,ls ∪{l}) end
⋆j   end
⋆k   pre pre_int_Insert(op)(hs,ls)
```

33. The remove command is of the form Rmv(li) for some li.
34. 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
  33 int_Remove: Rmv → N $\xrightarrow{\sim}$ N
  34 int_Remove(Rmv(li))(hs,ls) ≡
  34(a))  let l = xtr_L(li)(ls), {hi′,hi″} = obs_HIs(l) in
  34(b))  let {h′,h″} = {xtr_H(hi′,hs),xtr_H(hi″,hs)} in
  34(c))  let hs′ = cond_rmv(h′,hs) ∪ cond_rmv_H(h″,hs) in
  34(d))  (hs\{h′,h″} ∪ hs′,ls\{l}) end end end
  34(a))  pre li ∈ iols(ls)

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

■ End of Example 44

A.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: APPLICATIVE CONSTRUCTS

Start of Lecture 10: RSL: IMPERATIVE & PARALLEL CONSTRUCTS

A.6. Imperative Constructs

A.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**.

A.6.2. Variables and Assignment

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

A.6.3. Statement Sequences and skip

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

A.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**

A.6.5. Iterative Conditionals

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

A.6.6. Iterative Sequencing

8. **for** e **in** list_expr · P(b) **do** S(b) **end**

A.7. Process Constructs

A.7.1. Process Channels

Let A, B and D stand for two types of (channel) messages and i:KIdx for channel array indexes, then:

channel
  c,c′:A
channel
  {k[i]|i:KIdx}:B
  {ch[i]i:KIdx}:B

Example 45 – Modelling Connected Links and Hubs:

- Examples (45–48) are building up a model of one form of meaning of a transport net.
  - We model the movement of vehicles around hubs and links.
  - We think of each hub, each link and each vehicle to be a process.
  - These processes communicate via channels.

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

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

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

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

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

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

■ End of Example 45

A.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] Unit
  Q: i:KIdx →  out c in k[i] Unit
  P() ≡ ... c ? ... k[i] ! e ... ; P()
  Q(i) ≡ ... k[i] ? ... c ! e ... ; Q(i)

  R:  Unit →  out c in k[i]  B
  S:  i:KIdx →  out c in k[i]  D
  R() ≡ ... c′ ? ... ch[i] ! e ... ; B_Val_Expr
  S(i) ≡ ... ch[i] ? ... c ! e ...; D_Val_Expr
```

Example 46 – Communicating Hubs, Links and Vehicles:

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

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

■ End of Example 46

A.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
$\mathcal{O}$ { $\mathcal{P}_i$ | i:Idx }	Distributed composition, $\mathcal{O} = \parallel, [], \sqcap, \#$

Example 47 – Modelling Transport Nets:

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

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

  ωPN: V → (Pos×Net)
```

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

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

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

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

- The student is encouraged to sketch/define similarly schematic link and vehicle processes. ■ End of Example 47

A.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 48 – Modelling Vehicle Movements:

- Whereas hubs and links are modelled as basically static, passive, that is, inert, processes we shall consider vehicles to be "highly" dynamic, active processes.
- We assume that a vehicle possesses knowledge about the road net.
  - The road net is here abstracted as an awareness of
  - which links, by their link identifiers,
  - are connected to any given hub, designated by its hub identifier,
  - the length of the link,
  - and the hub to which the link is connected "at the other end", also by its hub identifier

- A vehicle is further modelled by its current position on the net in terms of either hub or link positions
  - designated by appropriate identifiers
  - and, when "on a link" "how far down the link", by a measure of a fraction of the total length of the link, the vehicle has progressed.

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

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

- The vehicle chooses between these two possibilities by an internal non-deterministic choice ([3]).

```
type
  M == μL_H(li:LI,hi:HI) | μH_L(hi:HI,li:LI)
value
  vehicle: VI → (Pos × Net) → V → Unit
  vehicle(vi)(μatH(hi),net)(v) ≡
  [1]  (wait ;
  [2]   vehicle(vi)(μatH(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)]!μH_L(hi,li)‖vl[(vi,li)]!μH_L(hi,li));
  [8]   vehicle(vi)(μonL(hi,li,0,hi′),net)(v)
  [9]   end end end)
```

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

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

■ End of Example 48

End of Lecture 10: RSL IMPERATIVE & PARALLEL CONSTRUCTS

Start of Lecture 11: RSL SPECIFICATIONS

A.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 = E^ω
  L = F →m G
  Mt = J → Kf, Mp = J ~→ Ki
  N == alpha | beta | ... | omega
  O == μHf(as:Hf)
     | μKf(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) ≡ ℰ_f1(a)
  f2: E → in|out chf F
  f2(e) ≡ ℰ_f2(e)
  f3: Unit → in chf out chg Unit
  ...
axiom
  𝒫_i(f1,va),
  𝒫_j(f2,vb),
  ...
  𝒫_k(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.

- The **axiom** clause is usually expressed as several "comma (,) separated" predicates:

$$\mathcal{P}_i(\overline{A_i},\overline{f_i},\overline{v_i}),\mathcal{P}_j(\overline{A_j},\overline{f_j},\overline{v_j}),\ldots,\mathcal{P}_k(\overline{A_k},\overline{f_k},\overline{v_k})$$

- where $(\overline{A_k},\overline{f_\ell},\overline{v\ell})$ is an abbreviation for A_{ℓ_1}, A_{ℓ_2}, $\ldots$, A_t, f_{ℓ_1}, f_{ℓ_2}, $\ldots$, f_{ℓ_f}, v_{ℓ_1}, v_{ℓ_2}, $\ldots$, v_{ℓ_v}.
- The indexed sort or type names, A and the indexed function names, d, are defined elsewhere in the specification.
- The index value names, v are usually names of bound 'variables' of universally or existentially quantified predicates of the indexed ("comma"-separated) $\mathcal{P}$.

Example 49 – A Neat Little "System":

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

- We assume the following:
  - ∗ that each *link* is somehow associated with two pairs of *sensors:*
    - · a pair of *enter* and *leave sensors* at one end, and
    - · a pair of *enter* and *leave sensors* at the other end;

    and
  - ∗ a *road pricing* process
    - · which records pairs of link enterings and leavings,
    - · first one, then, after any time interval, the other,
    - · with leavings leading to debiting of traversal fees;

- Our first specification
  - define types,
  - assume a net value,
  - declares channels and
  - state signatures of all processes.

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

```
type
  N, H, HI, LI, VI
  RPM == μEnter_L(vi:VI,li:LI) | μLeave_L(vi:VI,li:LI)
value
  n:N
channel
  {ves[ωHI(h),li]|h:H•h ∈ ωHs(n)∧li ∈ ωLIs(h)}:VI
  {vls[li,ωHI(h)]|h:H•h ∈ ωHs(n)∧li ∈ ωLIs(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 48, in two parts, Slide 295 and Slide 297 we focus on the events
  - [7] where the vehicle enters a link, respectively
  - [5′] where the vehicle leaves a link.
- These are summarised in the schematic reproduction of the vehicle behaviour description.
  - We redirect the interactions between vehicles and links to become
  - interactions between vehicles and enter and leave sensors.

```
value
  δ:Real = move(h,f) axiom 0<δ≪1
  move: H × F → F
```

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

- As mentioned on Slide 304 *link* behaviours are associated with two pairs of sensors:
  - a pair of *enter* and *leave sensors* at one end, and
  - a pair of *enter* and *leave sensors* at the other end;

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

- The *LVS* component of the *road_pricing* behaviour serves,
  - among other purposes that are not mentioned here,
  - to record whether the movement of a vehicles “originates” along a link or not.
- Otherwise we leave it to the 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
     μEnter_LI(vi,li) →
        road_pricing(lvs†[ li↦lvs(li)∪{vi} ],fee,acc),
     μLeave_LI(vi,li) →
        let lvs′ = if vi ∈ lvs(li) then lvs†[ li↦lvs(li)\{vi} ] else lvs end,
            acc′ = payment(vi,li)(fee,acc) in
       road_pricing(lvs′,fee,acc′)
  end end end
```

■ End of Example 49

End of Lecture 11: RSL SPECIFICATIONS

Start of Lecture 5: DOMAIN ENTITIES

B. Domain Entities

B.1. Entities

- The reason for our interest in ‘simple entities’
  - is that assemblies and units of systems
  - possess static and dynamic properties
  - which become contexts and states of
  - the processes into which we shall “transform” simple entities.

B.1.1. Observable Phenomena

- We shall just consider ‘simple entities’.
  - By a simple entity we shall here understand
    - ∗ a phenomenon that we can designate, viz.
    - ∗ see, touch, hear, smell or taste, or
    - ∗ measure by some instrument (of physics, incl. chemistry).
  - A simple entity thus has properties.
  - A simple entity is
    - ∗ either continuous
    - ∗ or is discrete, and then it is
      - · either atomic
      - · or composite.

(B. Domain Entities B.1. Entities B.1.1. Observable Phenomena)

B.1.1.1. Attributes: Types and Values

- By an attribute we mean a simple property of an entity.
  - *A simple entity has properties* $p_i, p_j, \ldots, p_k$.
- Typically we express attributes by a pair of
  - a type designator: *the attribute is of type* V, and
  - a value: *the attribute has value* v (of type V, i.e., $v : V$).
- A simple entity may have many simple properties.
  - A continuous entity, like 'oil', may have the following attributes:
    - type: *petroleum*,
    - kind: *Brent-crude*,
    - amount: *6 barrels*,
    - price: *45 US $/barrel*.

(B. Domain Entities B.1. Entities B.1.1. Observable Phenomena B.1.1.1. Attributes: Types and Values)

- An *atomic* entity, like a 'person', may have the following attributes:
  - gender: *male*,
  - name: *Dines Bjørner*,
  - birth date: *4. Oct. 1937*,
  - marital status: *married*.
- A *composite* entity, like a railway system, may have the following attributes:
  - country: *Denmark*,
  - name: *DSB*,
  - electrified: *partly*,
  - owner: *independent public enterprise owned by Danish Ministry of Transport*.

(B. Domain Entities B.1. Entities B.1.1. Observable Phenomena B.1.1.1. Attributes: Types and Values)

B.1.1.2. Continuous Simple Entities

- A simple entity is said to be continuous
  - if, within limits, reasonably sizable amounts of the simple entity, can be arbitrarily decomposed into smaller parts
  - each of which still remain simple continuous entities
  - of the same simple entity kind.
- Examples of continuous entities are:
  - oil, i.e., any fluid,
  - air, i.e., any gas,
  - time period and
  - a measure of fabric.

(B. Domain Entities B.1. Entities B.1.1. Observable Phenomena B.1.1.2. Continuous Simple Entities)

B.1.1.3. Discrete Simple Entities

- A simple entity is said to be discrete if its immediate structure is not continuous.
  - A simple discrete entity may, however, contain continuous sub-entities.
- Examples of discrete entities are:
  - persons,
  - rail units,
  - oil pipes,
  - a group of persons,
  - a railway line and
  - an oil pipeline.

B.1.1.4. Atomic Simple Entities

- A simple entity is said to be atomic
  - if it cannot be meaningfully decomposed into parts
  - where these parts has a useful "value" in the context in which the simple entity is viewed and
  - while still remaining an instantiation of that entity.
- Thus a 'physically able person', which we consider atomic,
  - can, from the point of physical ability,
  - not be decomposed into meaningful parts: a leg, an arm, a head, etc.
- Other atomic entities could be a rail unit, an oil pipe, or a hospital bed.
- The only thing characterising an atomic entity are its attributes.

B.1.1.5. Composite Simple Entities

- A simple entity, c, is said to be composite
  - if it can be meaningfully decomposed
  - into sub-entities that have separate
  - meaning in the context in which c is viewed.
- We exemplify some composite entities.
  - (1) A *railway net* can be decomposed into
    - ∗ a set of one or more *train lines* and
    - ∗ a set of two or more *train stations*.
  - Lines and stations are themselves composite entities.

- (2) An *Oil industry* whose decomposition include:
  - ∗ one or more *oil fields*,
  - ∗ one or more *pipeline systems*,
  - ∗ one or more *oil refineries* and
  - ∗ one or more *one or more oil product distribution systems*.
- Each of these sub-entities are also composite.

- Composite simple entities are thus characterisable by
  - their attributes,
  - their sub-entities, and
  - the mereology of how these sub-entities are put together.

B.1.2. Discussion

- In Sect. 3.2 we interpreted the model of mereology in six examples.
- The units of Sect. 2
  - which in that section were left uninterpreted
  - now got individuality —
    - ∗ in the form of
      - · aircraft,
      - · building rooms,
      - · rail units and
      - · oil pipes.
  - Similarly for the assemblies of Sect. 2. They became
    - ∗ pipeline systems,
    - ∗ oil refineries,
    - ∗ train stations,
    - ∗ banks, etc.

- In conventional modelling
  - the mereology of an infrastructure component,
    - ∗ of the kinds exemplified in Sect. 3.2,
  - was modelled by modelling
    - ∗ that infrastructure component's special mereology
    - ∗ together, "in line", with the modelling
    - ∗ of unit and assembly attributes.
- With the model of Sect. 2 now available
  - we do not have to model the mereological aspects,
  - but can, instead, instantiate the model of Sect. 2 appropriately.
  - We leave that to be reported upon elsewhere.
- In many conventional infrastructure component models
  - it was often difficult to separate
    - ∗ what was mereology from
    - ∗ what were attributes.

B.2. Examples of Composite Structures

- Before a semantic treatment of the concept of mereology
  - let us review what we have done and
  - let us interpret our abstraction
    - ∗ (i.e., relate it to actual societal infrastructure components).

B.2.1. What We have Done So Far ?

- We have
  - presented a model that is claimed to abstract essential mereological properties of
    - ∗ machine assemblies,
    - ∗ railway nets,
    - ∗ the oil industry,
    - ∗ oil pipelines,
    - ∗ buildings with installations,
    - ∗ hospitals,
    - ∗ etcetera.

B.2.2. Six Interpretations

- Let us substantiate the claims made in the previous paragraph.
  - We will do so, albeit informally, in the next many paragraphs.
  - Our substantiation is a form of diagrammatic reasoning.
  - Subsets of diagrams will be claimed to represent parts, while
  - Other subsets will be claimed to represent connectors.
- The reasoning is incomplete.

B.2.2.1. Air Traffic

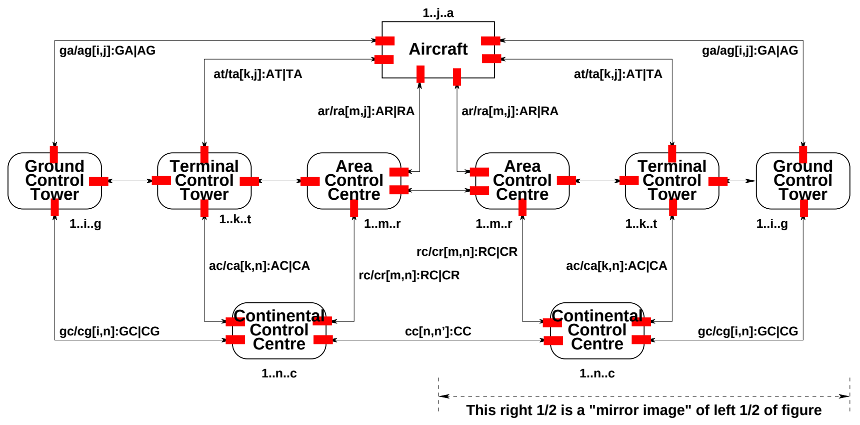


Figure 2: An air traffic system. Black (rounded or edged) boxes and lines are units; **red filled boxes** are connections

- Figure 2 on the preceding page shows nine (9) boxes and eighteen (18) lines.
  - Together they form an assembly.
  - Individually boxes and lines represent units.
    - ∗ The rounded corner boxes denote buildings.
    - ∗ The sharp corner box denote an aircraft.
    - ∗ Lines denote radio telecommunication.
  - Only where lines touch boxes do we have connections.
    - ∗ These are shown as red horisontal or vertical boxes at both ends of the double-headed arrows,
    - ∗ overlapping both the arrows and the boxes.
- The index ranges shown attached to, i.e., labelling each unit,
  - shall indicate that there are a multiple of the "single" (thus representative) unit shown.

- Notice that
  - the 'box' units are fixed installations and that
  - the double-headed arrows designate the ether where radio waves may propagate.
  - We could, for example, assume that each such line is characterised by
    - ∗ a combination of location and
    - ∗ (possibly encrypted) radio communication frequency.
  - That would allow us to consider all line for not overlapping.
  - And if they were overlapping, then that must have been a decision of the air traffic system.

B.2.2.2. Buildings

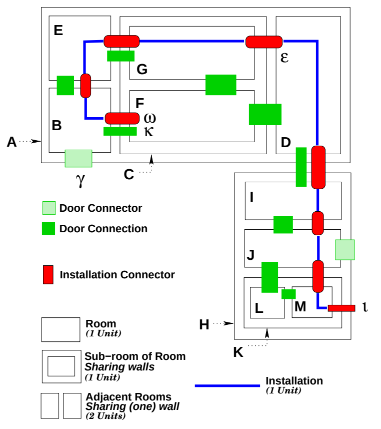


Figure 3: A building plan with installation

- Figure 3 on the previous page shows a building plan — as an assembly
  - of two neighbouring, common wall-sharing buildings, A and H,
  - probably built at different times;
  - with room sections B, C, D and E contained within A,
  - and room sections I, J and K within H;
  - with room sections L and M within K,
  - and F and G within C.

- Connector γ provides means of a connection between A and B.
- Connection κ provides "access" between B and F.
- Connectors ι and ω enable input, respectively output adaptors (receptor, resp. outlet) for electricity (or water, or oil),
- connection ϵ allow electricity (or water, or oil) to be conducted through a wall.
- Etcetera.

B.2.2.3. Financial Service Industry

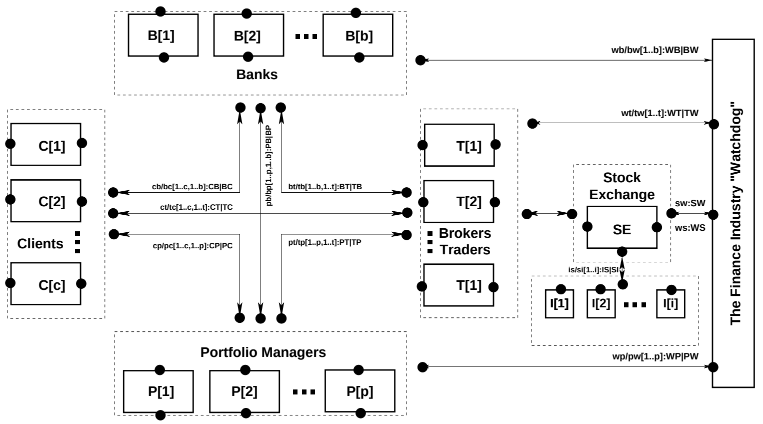


Figure 4: A financial service industry

- Figure 4 on the preceding page shows seven (7) larger boxes [6 of which are shown by dashed lines] and twelve (12) double-arrowed lines.
  - Where double-arrowed lines touch upon (dashed) boxes we have connections (also to inner boxes).
  - Six (6) of the boxes, the dashed line boxes, are assemblies, five (5) of them consisting of a variable number of units;
  - five (5) are here shown as having three units each with bullets "between" them to designate "variability".
- People,
  - not shown, access the outermost (and hence the "innermost" boxes, but the latter is not shown)
  - through connectors, shown by bullets, •.

(B. Domain Entities B.2. Examples of Composite Structures B.2.2. Six Interpretations B.2.2.3. Financial Service Industry)

B.2.2.4. Machine Assemblies

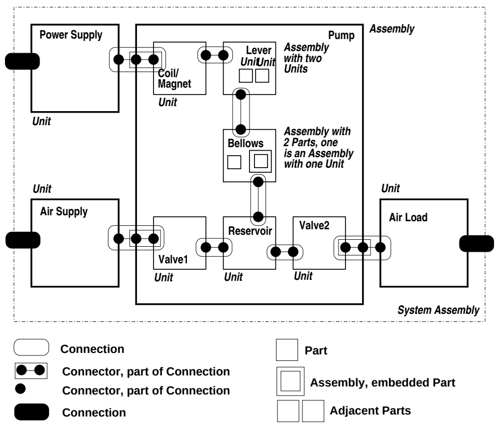


Figure 5: An air pump, i.e., a physical mechanical system

(B. Domain Entities B.2. Examples of Composite Structures B.2.2. Six Interpretations B.2.2.4. Machine Assemblies)

- Figure 5 on the previous page shows a machine assembly.
  - Square boxes show assemblies or units.
  - Bullets, •, show connectors.
  - Strands of two or three bullets on a thin line, encircled by a rounded box, show connections.
  - The full, i.e., the level 0, assembly consists of
    - four parts
    - and three internal and three external connections.
  - The Pump unit
    - is an assembly
      - of six (6) parts,
      - five (5) internal connections
      - and three (3) external connectors.

(B. Domain Entities B.2. Examples of Composite Structures B.2.2. Six Interpretations B.2.2.4. Machine Assemblies)

- Etcetera.
- One connector and some connections afford "transmission" of electrical power.
- Other connections convey torque.
- Two connectors convey input air, respectively output air.

(B. Domain Entities B.2. Examples of Composite Structures B.2.2. Six Interpretations B.2.2.4. Machine Assemblies)

B.2.2.5. Oil Industry

B.2.2.5.1. • *"The" Overall Assembly* •

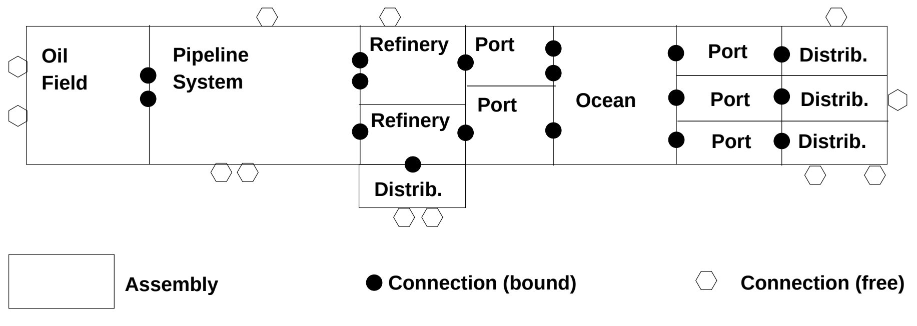


Figure 6: A Schematic of an Oil Industry

- Figure 6 on the preceding page shows
  - an assembly consisting of fourteen (14) assemblies, left-to-right:
    - ∗ one oil field,
    - ∗ a crude oil pipeline system,
    - ∗ two refineries and one, say, gasoline distribution network,
    - ∗ two seaports,
    - ∗ an ocean (with oil and ethanol tankers and their sea lanes),
    - ∗ three (more) seaports,
    - ∗ and three, say gasoline and ethanol distribution networks.
  - Between all of the assembly units there are connections,
  - and from some of the assembly units there are connectors (to an external environment).
- The crude oil pipeline system assembly unit will be concretised next.

B.2.2.5.2. • *A Concretised Assembly Unit*•

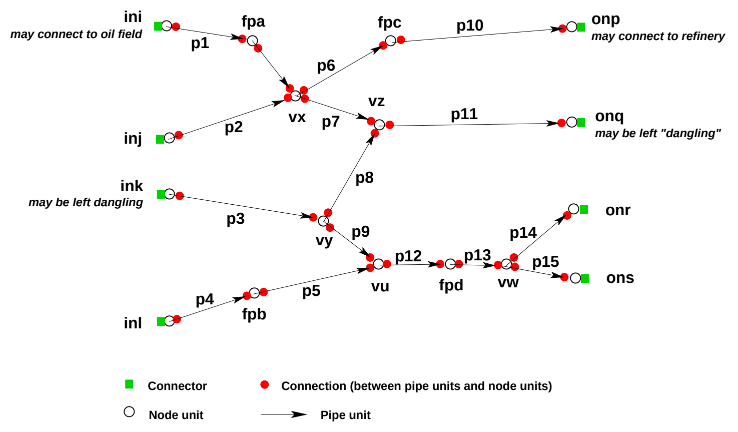


Figure 7: A Pipeline System

- Figure 7 on the previous page shows a pipeline system.
- It consists of 32 units:
  - fifteen (15) pipe units (shown as directed arrows and labelled p1–p15),
  - four (4) input node units (shown as small circles, ○, and labelled in*i*–in*ℓ*),
  - four (4) flow pump units (shown as small circles, ○, and labelled fp*a*–fp*d*),
  - five (5) valve units (shown as small circles, ○, and labelled v*x*–v*w*), and
  - four (4) output node units (shown as small circles, ○, and labelled on*p*–on*s*).

- In this example the routes through the pipeline system
  - start with node units and end with node units,
  - alternates between node units and pipe units,
  - and are connected as shown by fully filled-out **red**[4] disc connections.
  - Input and output nodes have input, respectively output connectors, one each, and shown with **green**[5]

[4]This paper is most likely not published with colours, so **red** will be shown as **darker colour**.
[5]Shown as **lighter colour**ed connections.

B.2.2.6. Railway Nets

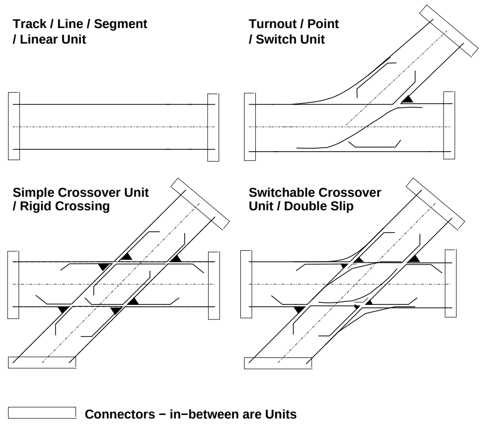


Figure 8: Four example rail units

- Figure 8 on the preceding page diagrams
  - four rail units,
  - each with their two, three or four connectors.
- Multiple instances of these rail units
  - can be assembled
  - as shown on Fig. 9 on the next page
  - into proper rail nets.

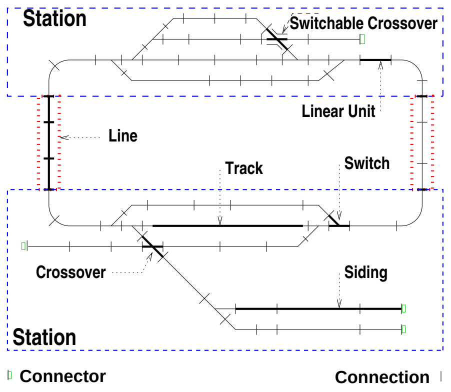


Figure 9: A "model" railway net. An Assembly of four Assemblies:
Two stations and two lines; Lines here consist of linear rail units;
stations of all the kinds of units shown in Fig. 8 on page 342.
There are 66 connections and four "dangling" connectors

- Figure 9 on the preceding page diagrams an example of a proper rail net.
  - It is assembled from the kind of units shown in Fig. 8.
  - In Fig. 9 consider just the four dashed boxes:
    * The dashed boxes are assembly units.
    * Two designate stations, two designate lines (tracks) between stations.
    * We refer to to the caption four line text of Fig. 8 on page 342 for more "statistics".
    * We could have chosen to show, instead, for each of the four "dangling' connectors, a composition of a connection, a special "end block" rail unit and a connector.

B.2.3. Discussion

- It requires a somewhat more laborious effort,
  - than just "flashing" and commenting on these diagrams,
  - to show that the modelling of essential aspects of their structures
  - can indeed be done by simple instantiation
  - of the model given in the previous part of the talk.

- We can refer to a number of documents which give rather detailed domain models of
  - air traffic,
  - container line industry,
  - financial service industry,
  - health-care,
  - IT security,
  - "the market",
  - "the" oil industry[6],
  - transportation nets[7],
  - railways, etcetera, etcetera.
- Seen in the perspective of the present paper
  - we claim that much of the modelling work done in those references
  - can now be considerably shortened and
  - trust in these models correspondingly increased.

[6]http://www2.imm.dtu.dk/˜db/pipeline.pdf
[7]http://www2.imm.dtu.dk/˜db/transport.pdf

B.3. Attributes and Sub-entities of Sort Values

B.3.1. General

- Entities are defined in terms of
  - either sorts, that is, abstract types for whose values we do not define mathematical models,
  - or concrete types whose values are sets, Cartesians, lists, maps, functions or other.
- Entities are
  - either atomic,[8] in which case they are characterised solely in terms of all their attributes (types and values),
  - or are composite, in which case they are characterised in terms of all their attributes (types and values) and all their sub-entities.

[8]As dealt with elsewhere (Appendix Sect. , Pages 313–322) in these lecture notes: attributes of atomic or composite entities are (type and value) properties of entities (save those of being a composite entity and of such composite entities sub-entities). Atomic entities are atomic in that they have no sub-entities. Sub-entities of composite entities are proper entities.

- For both atomic and composite sorts
  - we introduce, as need be, observer functions,
  - whether of attributes or (possibly, if composite) of sub-entities.[9]
- In this section we shall introduce and define an equality operator that compares entities modulo some attribute:
  - the name of the equality operator is $\simeq_{\omega_{\mathcal{A}_{attr}}}$,
  - and application of the equality operator to a pair of entities to be compared and the attribute for which comparison is left is expressed: $\simeq_{\mathcal{A}_{attr_A}}(a', a'')(\omega_\alpha)$.
- To explain this "modulo attribute" equality operator we first ιℓℓustrate[10] the concepts of functions that observe attributes and sub-entities.

[9]Till now, in these lecture notes, we have used"the same kind" of observer functions (ωB_i, ωC_j) for observing attributes (B_i) of atomic or composite entities and for observing sub-entities (C_j) of composite entities. In this section we shall distinguish between ωbserving αttributes ($\omega_\alpha B$) and ωbserving sub-εntities ($\omega_\epsilon C$). Maybe we shall have an opportunity to do so in a next version of these lecture notes.

[10]In this section we distinguish between ιℓℓustrations (formally marked with ιℓℓs) and δεφinitions (read: definitions, marked with δεφs). ιℓℓustrations are like schematic examples, but they are just that: rough-sketched generic examples. δεφinitions are valid throughout these lecture notes.

B.3.2. Constant and Variable Valued Attributes

- There are two kinds of attributes to be considered.
  - constant valued attributes, and
  - variable valued attributes.
- Attributes with variable values are also called entity state components.

- Let A be (the type name of) a set of entities,
- let $B_1, \ldots, B_m$ be all the (distinct names of) types of constant valued attributes of A and
- let $\Sigma_1, \ldots, \Sigma_n$ be all the (distinct names of) types of variable valued attributes of A.
- We $\iota\ell\ell$ustrate these:

type
  $[\iota\ell\ell]$ A, B_1, ..., B_m, Σ_1, ..., Σ_n, C_1, ..., C_k

B.3.3. Sub-Entities

- Let $C_1, \ldots, C_k$ be all the (distinct names of) types of sub-entities of A.
- We $\iota\ell\ell$ustrate these:

type
  $[\iota\ell\ell]$ C_1, ..., C_k

B.3.4. Attribute and Sub-Entity Observers

- Let $\{\omega_\alpha \mathrm{B}_1, \ldots, \omega_\alpha \mathrm{B}_m\}$ be the corresponding set of all the constant valued observers of A,
- Let $\{\omega_\alpha \Sigma_1, \ldots, \omega_\alpha \Sigma_n\}$ be the corresponding set of all the variable valued observers of A and
- let $\{\omega_\epsilon \mathrm{C}_1, \ldots, \omega_\epsilon \mathrm{C}_k\}$ be the corresponding set of all the sub-entity observers of A.
- We $\iota\ell\ell$ustrate these:

value
  $[\iota\ell\ell]$ $\omega_\alpha\mathrm{B}_1$: A $\rightarrow$ B_1, ..., $\omega_\alpha\mathrm{B}_n$: A $\rightarrow$ B_m
  $[\iota\ell\ell]$ $\omega_\alpha\Sigma_1$: A $\rightarrow$ Σ_1, ..., $\omega_\alpha\Sigma_n$: A $\rightarrow$ Σ_n,
  $[\iota\ell\ell]$ $\omega_\epsilon\mathrm{C}_1$: A $\rightarrow$ C_1, ..., $\omega_\epsilon\mathrm{C}_k$: A $\rightarrow$ C_2

B.3.5. Attribute and Sub-entity Meta-Observers

- Let $\mathcal{A}ttr_A$ name the general type of a attribute observer function for sort A.
- Let $\mathcal{E}subs_A$ name the general type of a sub-entity observer functions for sort A.
- We $\iota\ell\ell$ustrate, with respect to the above $\iota\ell\ell$ustrations, these general types:

type

$[\,\iota\ell\ell\,]\ \ \mathcal{A}ttr_A = \omega_\alpha\mathrm{B}_1 \mid \ldots \mid \omega_\alpha\mathrm{B}_m \mid \omega_\alpha\Sigma_1 \mid \ldots \mid \omega_\alpha\Sigma_n$

$[\,\iota\ell\ell\,]\ \ \mathcal{E}subs_A = \omega_\epsilon\mathrm{C}_1 \mid \ldots \mid \omega_\epsilon\mathrm{C}_k$

- Let $\omega\mathcal{A}attr_A$ denote the function which from a type (A) observes all it attribute observer functions.
- Let $\omega\mathcal{E}subs$ denote the function which from a type observes all it possible sub-entity observer functions.
- We $\delta\epsilon\varphi$ne these:

value

$[\,\delta\epsilon\phi\,]\ \ \omega\mathcal{A}ttr_A s\text{: A} \to \mathcal{A}ttr_A\textbf{-set}$

$[\,\delta\epsilon\phi\,]\ \ \omega\mathcal{E}subs_A s\text{: A} \to \mathcal{E}subs_A\textbf{-set}$

B.3.6. Meta-Observer Properties

- Let $\mathbb{A}ttr_A$ $\iota\ell\ell$ustrate the set of all attribute observers for type A, and
- let $\mathbb{E}subs_A$ $\iota\ell\ell$ustrate the set of all sub-entity observers for type A,
- then the two axioms $\iota\ell\ell_{attr}$ and $\iota\ell\ell_{subs}$ holds for the $\iota\ell\ell$ustrated type A and its observer functions:

value

$[\,\iota\ell\ell_{attr}\,]\ \ \mathbb{A}ttr_A{:}\mathcal{A}ttr_A\textbf{-set} = \{\omega_\alpha\mathrm{B}_1,\ldots,\omega_\alpha\mathrm{B}_m,\omega_\alpha\Sigma_1,\ldots,\omega_\alpha\Sigma_n\},$

$[\,\iota\ell\ell_{subs}\,]\ \ \mathbb{E}subs_A{:}\mathcal{E}subs_A\textbf{-set} = \{\omega_\epsilon\mathrm{C}_1,\ldots,\omega_\epsilon\mathrm{C}_k\}$

axiom

$[\,\iota\ell\ell_{attr}\,]\ \ \forall\ \text{a:A} \cdot \omega\mathcal{A}ttr_A s(\text{a}) = \mathbb{A}ttr_A \wedge$

$[\,\iota\ell\ell_{subs}\,]\ \ \forall\ \text{a:A} \cdot \omega\mathcal{E}subs_A s(\text{a}) = \mathbb{E}subs_A$

B.3.7. Sort Value Equality

- Now to register a possible change in but one attribute of A we meta-linguistically $\delta\epsilon\phi$ine the following equality operator:

value

$[\,\delta\epsilon\phi\,]\ \ \simeq_{\mathcal{A}attr_A}\text{: A}\times\text{A} \to \mathcal{A}ttr_A \to \textbf{Bool}$

$[\,\delta\epsilon\phi\,]\ \ \simeq_{\mathcal{A}attr_A}(\text{a}',\text{a}'')(\omega_\alpha) \equiv$

$[\,\delta\epsilon\phi\,]\qquad \forall\ \omega F{:}\omega\mathcal{A}ttr_A s(\text{a}')\backslash\{\omega_\alpha\}\Rightarrow\omega F(\text{a}')=\omega F(\text{a}'') \wedge \forall\ \omega'_\epsilon{:}\mathcal{E}subs_A\Rightarrow\omega'_\epsilon(\text{a}')=\omega'_\epsilon(\text{a}'')$

$[\,\delta\epsilon\phi\,]\qquad \textbf{pre}\ \omega\mathcal{A}ttr_A s(\text{a}') = \omega\mathcal{A}ttr_A s(\text{a}'')$

- The $\simeq_{\omega\mathcal{A}attr}$ ‘equality’ operator
  - applies to two values a′,a′′:A and an attribute observer function, $\omega\mathrm{B}_i$ (given as ω_α),
  - and yields **true** if a′ and a′′
    - ∗ have all but the same attribute values except for attribute B_i, and
    - ∗ have all exactly the same and equal sub-entities.

Example 50 – Equality of Hubs Modulo Hub States:

- Please review Examples 2 on page 50 and 3 on page 53.
  - In Example 3 on page 53 on Page 359, formula line item [17], a comparison is made between two values of a sort:
    $\omega H\Sigma(h')=(\prod\{h\sigma'|h\sigma':H\Sigma \cdot h\sigma' \in \omega\Omega(h)\backslash\{h\sigma\}\})_{\overline{p}}\prod_p h\sigma$.
  - We now redefine this comparion – which really does not capture all the value aspects of the compared hubs!

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

[12] set_HΣ: H × HΣ → H
[13] set_HΣ(h,hσ) as h′
[14]   pre hσ ∈ ωHΩ(h)
[15]   post $\simeq_{\omega\mathcal{A}_{attr_H}}$(h,h′)(ωHΣ) ∧
[17]        $\omega H\Sigma(h')=(\prod\{h\sigma'|h\sigma':H\Sigma \cdot h\sigma' \in \omega\Omega(h)\backslash\{h\sigma\}\})_{\overline{p}}\prod_p h\sigma$
```

■ End of Example 50

B.4. Unique Entity Identifiers

- In many domain and requirements modelling situations we make use of the concept of *unique entitiy identifiers.*
  - For any type A for which we introduce unique identifiers of all a:A values
  - we consider such unique identifiers as of sort AI[11].
  - The AI attribute shall be considered a constant-valued attribute.
  -
  -

[11] We may, in some immediate future, decide to instead of using the sort name AI using, for example, the sort name $\Im$A or $\Im_{\mathsf{A}}$.

End of Lecture 5: DOMAIN ENTITIES

Start of Lecture 12: MEREOLOGY

C. Mereology
C.1. Opening
C.1.1. Definition

- By mereology we understand
  - the study and knowledge about
  - parts and wholes
  - and the relationships between parts and between parts and holes.

C.1.2. Examples

Example 51 – Simple and Composite Net Entities:

- We repeat some of the material from Example 1 on page 39.
- [1] A road, train, airlane (air traffic) or sea lane (shipping) net
- [2] consists, amongst other things, of hubs and links.

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

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

■ End of Example 51

- Example 51 illustrated that entities can be either atomic of composite.
- But also functions, events and behaviours can be either atomic or composite.

Example 52 – Simple and Composite Net Functions:

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

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

- Both
  - the journey and continuation entities, *j* and *c* , and
  - the *length* function

  are composite
- Both
  - the link entities, *ll*,
  - the ω*LEN* function

  are atomic.

■ End of Example 52

Example 53 – Simple and Composite Net Events:

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

  can be considered a composite event.
- We shall model this event by the predicates which holds of vehicles in a traffic at given times.

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

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

■ End of Example 53

- In the next, long example we consider a pipeline system (or either oil or gas pipes).

Example 54 – Simple and Composite Net Behaviours:

Pipeline Systems and Their Units

35. We focus on nets, $n : N$, of pipes, $\pi : \Pi$, valves, $v : V$, pumps, $p : P$, forks, $f : F$, joins, $j : J$, wells, $w : W$ and sinks, $s : S$.
36. Units, $u : U$, are either pipes, valves, pumps, forks, joins, wells or sinks.
37. Units are explained in terms of disjoint types of PIpes, VAlves, PUmps, FOrks, JOins, WEIls and SKs.[12]

[12] This is a mere specification language technicality.

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

Unit Identifiers and Unit Type Predicates

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

```
type
  38 UI
value
  39 obs_UI: U → UI
  40 is_Π: U → Bool, is_V: U → Bool, ..., is_J: U → Bool
     is_Π(u) ≡ case u of mkPI(_) → true, _ → false end
     is_V(u) ≡ case u of mkV(_) → true, _ → false end
      ...
     is_S(u) ≡ case u of mkS(_) → true, _ → false end
```

Unit Connections

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

41. With a pipe, a valve and a pump we associate exactly one input and one output connection.
42. With a fork we associate a maximum number of output connections, m, larger than one.
43. With a join we associate a maximum number of input connections, m, larger than one.
44. With a well we associate zero input connections and exactly one output connection.
45. With a sink we associate exactly one input connection and zero output connections.

value
41 obs_InCs,obs_OutCs: Π|V|P $\rightarrow$ {|1:**Nat**|}
42 obs_inCs: F $\rightarrow$ {|1:**Nat**|}, obs_outCs: F $\rightarrow$ **Nat**
43 obs_inCs: J $\rightarrow$ **Nat**, obs_outCs: J $\rightarrow$ {|1:**Nat**|}
44 obs_inCs: W $\rightarrow$ {|0:**Nat**|}, obs_outCs: W $\rightarrow$ {|1:**Nat**|}
45 obs_inCs: S $\rightarrow$ {|1:**Nat**|}, obs_outCs: S $\rightarrow$ {|0:**Nat**|}
axiom
42 $\forall$ f:F · obs_outCs(f) $\geq$ 2
43 $\forall$ j:J · obs_inCs(j) $\geq$ 2

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

Net Observers and Unit Connections

46. From a net one can observe all its units.
47. From a unit one can observe the the pairs of disjoint input and output units to which it is connected:
    (a) Wells can be connected to zero or one output unit — a pump.
    (b) Sinks can be connected to zero or one input unit — a pump or a valve.
    (c) Pipes, valves and pumps can be connected to zero or one input units and to zero or one output units.
    (d) Forks, f, can be connected to zero or one input unit and to zero or n, $2 \leq n \leq$obs_Cs(f) output units.
    (e) Joins, j, can be connected to zero or n, $2 \leq n \leq$obs_Cs(j) input units and zero or one output units.

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

Well-formed Nets, Actual Connections

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

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

Well-formed Nets, No Circular Nets

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

50. Units form routes of the net.

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

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

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

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

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

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

Pipeline Processes

We now add connectors to our model:

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

```
type
54  OPLS, U, K
56  UI, KI
value
54  obs_Us: OPLS → U-set, obs_Ks: OPLS → K-set
55  is_WeU, is_PiU, is_PuU, is_VaU, is_JoU, is_FoU, is_SiU: U → Bool [mutua
56  obs_UI: U → UI, obs_KI: K → KI
57  obs_UIp: K → (UI|{nil}) × (UI|{nil})
```

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

58. There is given an oil pipeline system, opls.
59. To every unit we associate a CSP behaviour.
60. Units are indexed by their unique unit identifiers.
61. To every connector we associate a CSP channel.
    Channels are indexed by their unique "k"onnector identifiers.
62. Unit behaviours are cyclic and over the state of their (static and dynamic) attributes, represented by u.
63. Channels, in this model, have no state.
64. Unit behaviours communicate with neighbouring units — those with which they are connected.
65. Unit functions, $\mathcal{U}_i$, change the unit state.
66. The pipeline system is now the parallel composition of all the unit behaviours.

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

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

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

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

■ End of Example 54

C.1.3. Discussion

- In this lecture
  - we shall mainly cover
  - atomic and
  - composite

  entities.

C.2. A Conceptual Model of Composite Entities

C.2.1. Systems, Assemblies, Units

- We speak of systems as assemblies.
- From an assembly we can immediately observe a set of parts.
- Parts are either assemblies or units.
- We do not further define what assemblies and units are.

```
type
  S = A, A, U, P = A | U
value
  obs_Ps: (S|A) → P-set
```

- Parts observed from an assembly are said to be immediately embedded in, that is, within, that assembly.
- Two or more different parts of an assembly are said to be immediately adjacent to one another.

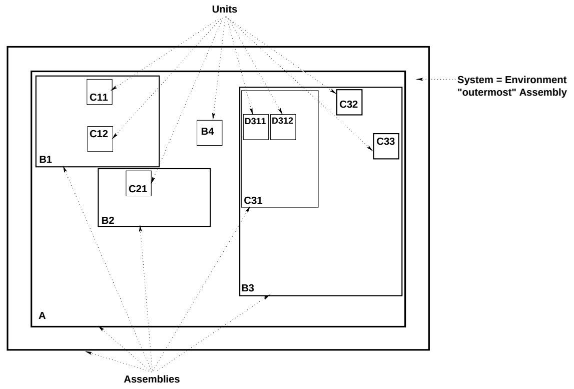


Figure 10: Assemblies and Units "embedded" in an Environment

- A system includes its environment.
- And we do not worry, so far, about the semiotics of all this !

Embeddedness and adjacency generalise to transitive relations.

- Given obs_Ps we can define a function, xtr_Ps,
  - which applies to an assembly a and
  - which extracts all parts embedded in a and including a.
- The functions obs_Ps and xtr_Ps define the meaning of embeddedness.

value
$\text{xtr\_Ps: (S|A)} \rightarrow \text{P-}\mathbf{set}$
$\text{xtr\_Ps(a)} \equiv$
$\quad \mathbf{let}\ \text{ps} = \{\text{a}\} \cup \text{obs\_Ps(a)}\ \mathbf{in}\ \text{ps} \cup \mathbf{union}\{\text{xtr\_Ps(a}')|\text{a}'\text{:A}\cdot\text{a}' \in \text{ps}\}\ \mathbf{end}$

- **union** is the distributed union operator.
- Parts have unique identifiers.
- All parts observable from a system are distinct.

type
AUI
value
$\text{obs\_AUI: P} \rightarrow \text{AUI}$
axiom
$\forall\ \text{a:A} \cdot$
$\quad \mathbf{let}\ \text{ps} = \text{obs\_Ps(a)}\ \mathbf{in}$
$\quad \forall\ \text{p}',\text{p}''\text{:P} \cdot \{\text{p}',\text{p}''\} \subseteq \text{ps} \wedge \text{p}' \neq \text{p}'' \Rightarrow \text{obs\_AUI(p}') \neq \text{obs\_AUI(p}'') \wedge$
$\quad \forall\ \text{a}',\text{a}''\text{:A} \cdot \{\text{a}',\text{a}''\} \subseteq \text{ps} \wedge \text{a}' \neq \text{a}'' \Rightarrow \text{xtr\_Ps(a}') \cap \text{xtr\_Ps(a}'') = \{\}\ \mathbf{end}$

C.2.2. 'Adjacency' and 'Within' Relations

- Two parts, p,p′, are said to be *immediately next to*, i.e., i_next_to(p,p′)(a), one another in an assembly a
  - if there exists an assembly, a' equal to or embedded in a
  - such that p and p′ are observable in that assembly a′.

value
$\text{i\_next\_to: P} \times \text{P} \rightarrow \text{A} \stackrel{\sim}{\rightarrow} \mathbf{Bool},\ \mathbf{pre}\ \text{i\_next\_to(p,p}')\text{(a): p} \neq \text{p}'$
$\text{i\_next\_to(p,p}')\text{(a)} \equiv \exists\ \text{a}'\text{:A} \cdot \text{a}'\text{=a} \vee \text{a}' \in \text{xtr\_Ps(a)} \cdot \{\text{p,p}'\} \subseteq \text{obs\_Ps(a}')$

- One part, p, is said to be *immediately within* another part, p'in an assembly a
  - if there exists an assembly, a' equal to or embedded in a
  - such that p is observable in a'.

value

$\text{i\_within}: P \times P \to A \xrightarrow{\sim} \mathbf{Bool}$
$\text{i\_within}(p,p')(a) \equiv$
$\quad \exists\, a':A \cdot\ (a=a' \vee a' \in \text{xtr\_Ps}(a)) \cdot p'=a' \wedge p \in \text{obs\_Ps}(a')$

- We can generalise the immediate 'within' property.
- A part, p, is (transitively) within a part p', within(p,p')(a), of an assembly, a,
  - either if p, is immediately within p' of that assembly, a,
  - or if there exists a (proper) part p'' of p'
  - such that within(p'',p)(a).

value

$\text{within}: P \times P \to A \xrightarrow{\sim} \mathbf{Bool}$
$\text{within}(p,p')(a) \equiv$
$\quad \text{i\_within}(p,p')(a) \vee \exists\, p'':P \cdot p'' \in \text{obs\_Ps}(p) \wedge \text{within}(p'',p')(a)$

- The function within can be defined, alternatively,
- using xtr_Ps and i_within
- instead of obs_Ps and within :

value

$\text{within}': P \times P \to A \xrightarrow{\sim} \mathbf{Bool}$
$\text{within}'(p,p')(a) \equiv$
$\quad \text{i\_within}(p,p')(a) \vee \exists\, p'':P \cdot p'' \in \text{xtr\_Ps}(p) \wedge \text{i\_within}(p'',p')(a)$

lemma: $\text{within} \equiv \text{within}'$

- We can generalise the immediate 'next to' property.
- Two parts, p, p' of an assembly, a, are adjacent if they are
  - either 'next to' one another
  - or if there are two parts p_o, p'_o
    - ∗ such that p, p' are embedded in respectively p_o and p'_o
    - ∗ and such that p_o, p'_o are immediately next to one another.

value

$\text{adjacent}: P \times P \to A \xrightarrow{\sim} \mathbf{Bool}$
$\text{adjacent}(p,p')(a) \equiv$
$\quad \text{i\_next\_to}(p,p')(a) \vee$
$\quad \exists\, p'',p''':P \cdot \{p'',p'''\} \subseteq \text{xtr\_Ps}(a) \wedge \text{i\_next\_to}(p'',p''')(a) \wedge$
$\quad\quad ((p=p'') \vee \text{within}(p,p'')(a)) \wedge ((p'=p''') \vee \text{within}(p',p''')(a))$

C.2.3. Mereology, Part I

- So far we have built a *ground mereology* model, $\mathcal{M}_{\mathcal{G}\mathrm{round}}$.
- Let $\sqsubseteq$ denote *parthood, x is part of y,* $x \sqsubseteq y$.

$$\forall x(x \sqsubseteq x)^{13} \quad (1)$$
$$\forall x, y(x \sqsubseteq y) \wedge (y \sqsubseteq x) \Rightarrow (x = y) \quad (2)$$
$$\forall x, y, z(x \sqsubseteq y) \wedge (y \sqsubseteq z) \Rightarrow (x \sqsubseteq z) \quad (3)$$

- Let $\sqsubset$ denote *proper parthood, x is part of y,* $x \sqsubset y$.
- Formula 4 defines $x \sqsubset y$. Equivalence 5 can be proven to hold.

$$\forall x \sqsubset y =_{\mathrm{def}} x(x \sqsubseteq y) \wedge \neg(x = y) \quad (4)$$
$$\forall\forall x, y(x \sqsubseteq y) \quad \Leftrightarrow \quad (x \sqsubset y) \vee (x = y) \quad (5)$$

[13]Our notation now is not RSL but some conventional first-order predicate logic notation.

- The *proper part* $(x \sqsubset y)$ relation is a strict partial ordering:

$$\forall x \neg(x \sqsubset x) \quad (6)$$
$$\forall x, y(x \sqsubset y) \Rightarrow \neg(y \sqsubset x) \quad (7)$$
$$\forall x, y, z(x \sqsubset y) \wedge (y \sqsubset z) \Rightarrow (x \sqsubset z) \quad (8)$$

- *Overlap*, $\bullet$, is also a relation of parts:
  - Two individuals overlap if they have parts in common:

$$x \bullet y =_{\mathrm{def}} \exists z(z \sqsubset x) \wedge (z \sqsubset y) \quad (9)$$
$$\forall x(x \bullet x) \quad (10)$$
$$\forall x, y(x \bullet y) \Rightarrow (y \bullet x) \quad (11)$$

- Proper overlap, $\circ$, can be defined:

$$x \circ y =_{\mathrm{def}} (x \bullet x) \wedge \neg(x \sqsubseteq y) \wedge \neg(y \sqsubseteq x) \quad (12)$$

- Whereas Formulas (1-11) holds of the model of mereology we have shown so far, Formula (12) does not.
- In the next section we shall repair that situation.
- The *proper part* relation, $\sqsubset$, reflects the *within* relation.
- The *disjoint* relation, $\oint$, reflects the *adjacency* relation.

$$x \oint y =_{\mathrm{def}} \neg(x \bullet y) \quad (13)$$

- Disjointness is symmetric:

$$\forall x, y(x \oint y) \Rightarrow (y \oint x) \quad (14)$$

- The *weak supplementation* relation, Formula 15, expresses
  - that if y is a proper part of x
  - then there exists a part z
  - such that z is a proper part of x
  - and z and y are disjoint
- That is, whenever an individual has one proper part then it has more than one.

$$\forall x, y(y \sqsubset x) \quad \Rightarrow \quad \exists z(z \sqsubset x) \wedge (z \oint y) \quad (15)$$

- Formulas 1–3 and 15 together determine the *minimal mereology*, $\mathcal{M}_{\mathcal{M}\text{inimal}}$.
- Formula 15 does not hold of the model of mereology we have shown so far..
- Formula 15 on the preceding page expresses that
  - whenever an individual has one proper part
  - then it has more than one.
- We mentioned there, Slide 400, that we would comment on the fact that our model appears to allow that assemblies may have just one proper part.

- We now do so.
  - We shall still allow assemblies to have just one proper part —
  - in the sense of a sub-assembly or a unit —
  - but we shall interpret the fact that an assembly always have at least one attribute.
  - Therefore we shall "generously" interpret the set of attributes of an assembly to constitute a part.

- In Sect. A.6
  - we shall see how attributes of both units and assemblies of the interpreted mereology
  - contribute to the state components of the unit and assembly processes.

C.2.4. Connectors

- So far we have only covered notions of
  - parts being next to other parts or
  - within one another.
- We shall now add to this a rather general notion of parts being otherwise related.
- That notion is one of connectors.

- Connectors provide for connections between parts.
- A connector is an ability be be connected.
- A connection is the actual fulfillment of that ability.
- Connections are relations between pairs of parts.
- Connections "cut across" the "classical"
  - *parts being part of the (or a) whole* and
  - *parts being related by embeddedness or adjacency*.

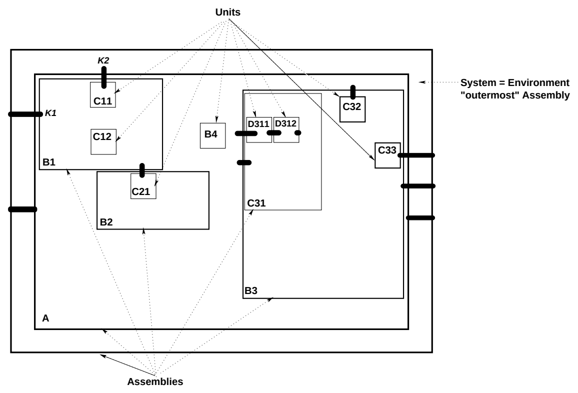


Figure 11: Assembly and Unit Connectors: Internal and External

- For now, we do not "ask" for the meaning of connectors !

- Figure 11 on the previous page "adds" connectors to Fig. 10 on page 388.
- The idea is that connectors
  - allow an assembly to be connected to any embedded part, and
  - allow two adjacent parts to be connected.
- In Fig. 11 on the previous page
  - the environment is connected, by *K2*, to part C11;
  - the "external world" is connected, by K1, to B1;
  - etcetera.

- From a system we can observe all its connectors.
- From a connector we can observe
  - its unique connector identifier and
  - the set of part identifiers of the parts that the connector connects.
- All part identifiers of system connectors identify parts of the system.
- All observable connector identifiers of parts identify connectors of the system.

```
type
  K
value
  obs_Ks: S → K-set
  obs_KI: K → KI
  obs_Is: K → AUI-set
  obs_KIs: P → KI-set
axiom
  ∀ k:K • card obs_Is(k)=2,
  ∀ s:S,k:K • k ∈ obs_Ks(s) ⇒
    ∃ p:P • p ∈ xtr_Ps(s) ⇒ obs_AUI(p) ∈ obs_Is(k),
  ∀ s:S,p:P • ∀ ki:KI • ki ∈ obs_KIs(p) ⇒
    ∃! k:K • k ∈ obs_Ks(s) ∧ ki=obs_KI(k)
```

- This model allows for a rather "free-wheeling" notion of connectors
  - one that allows internal connectors to "cut across" embedded and adjacent parts;
  - and one that allows external connectors to "penetrate" from an outside to any embedded part.
- We need define an auxiliary function.
  - xtr∀KIs(p) applies to a system
  - and yields all its connector identifiers.

```
value
  xtr∀KIs: S → KI-set
  xtr∀Ks(s) ≡ {obs_KI(k)|k:K•k ∈ obs_Ks(s)}
```

C.2.5. Mereology, Part II

(See Sect. (Slide 396) for Mereology, Part I.)
We shall interpret connections as follows:

- A connection between parts p_i and p_j
  - that enjoy a p_i adjacent to p_j relationship, means $p_i \circ p_j$,
  - that is, although parts p_i and p_j are adjacent
  - they do *share* "something", i.e., have something *in common*.
  - What that "something" is we shall comment on later, when we have "mapped" systems onto parallel compositions of CSP processes.
- A connection between parts p_i and p_j
  - that enjoy a p_i within p_j relationship,
  - does not add other meaning than
  - commented upon later, again when we have "mapped" systems onto parallel compositions of CSP processes.

- With the above interpretation we may arrive at the following, perhaps somewhat "awkward-looking" case:
  - a connection connects two adjacent parts p_i and p_j
    - where part p_i is within part p_{i_o}
    - and part p_j is within part p_{j_o}
    - where parts p_{i_o} and p_{j_o} are adjacent
    - but not otherwise connected.
  - How are we to explain that !
    - Since we have not otherwise interpreted the meaning of parts,
    - we can just postulate that "so it is" !
    - We shall, later, again when we have "mapped" systems onto parallel compositions of CSP processes, give a more satisfactory explanation.

- On Slides 396–399 we introduced the following operators:
  - $\sqsubseteq$, $\sqsubset$, $\bullet$, $\circ$, and $\oint$
- In some of the mereology literature these operators are symbolised with caligraphic letters:
  - $\sqsubseteq$: $\mathcal{P}$: part,
  - $\sqsubset$: $\mathcal{PP}$: proper part,
  - $\bullet$: $\mathcal{O}$: overlap and
  - $\oint$: $\mathcal{U}$: underlap.

C.2.6. Discussion

Summary:

- This ends our first model of a concept of mereology.
- The parts are those of assemblies and units.
- The relations between parts and the whole are,
  - on one hand, those of
    - embeddedness i.e. within, and
    - adjacency, i.e., adjacent,

    and
  - on the other hand, those expressed by connectors: relations
    - between arbitrary parts and
    - between arbitrary parts and the exterior.

Extensions:

- A number of extensions are possible:
  - one can add “mobile” parts and “free” connectors, and
  - one can further add operations that allow such mobile parts to move from one assembly to another along routes of connectors.
- Free connectors and mobility assumes static versus dynamic parts and connectors:
  - a free connector is one which allows a mobile part to be connected to another part, fixed or mobile; and
  - the potentiality of a move of a mobile part introduces a further dimension of dynamics of a mereology.

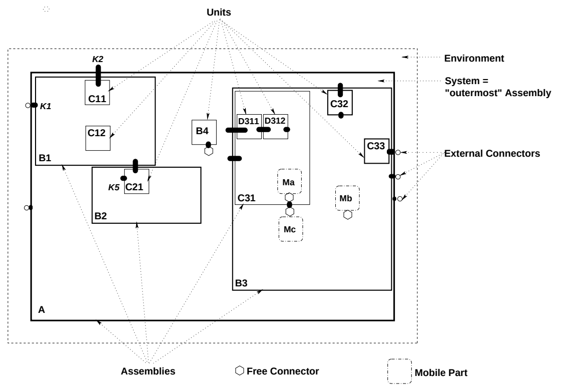


Figure 12: Mobile Parts and Free Connectors

Comments:

- We shall leave the modelling of free connectors and mobile parts to another time.
- Suffice it now to indicate that the mereology model given so far is relevant:
  - that it applies to a somewhat wide range of application domain structures, and
  - that it thus affords a uniform treatment of proper formal models of these application domain structures.

C.3. Functions and Events

-
-
-
-

Example 55 – Pipeline Transport Functions and Events:

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

Well-formed Nets, Special Pairs, wfN_SP

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

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

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

Special Routes, I

68. A pump-pump route is a route of length two or more whose first and last units are pumps and whose intermediate units are pipes or forks or joins.
69. A simple pump-pump route is a pump-pump route with no forks and joins.
70. A pump-valve route is a route of length two or more whose first unit is a pump, whose last unit is a valve and whose intermediate units are pipes or forks or joins.
71. A simple pump-valve route is a pump-valve route with no forks and joins.
72. A valve-pump route is a route of length two or more whose first unit is a valve, whose last unit is a pump and whose intermediate units are pipes or forks or joins.
73. A simple valve-pump route is a valve-pump route with no forks and joins.
74. A valve-valve route is a route of length two or more whose first and last units are valves and whose intermediate units are pipes or forks or joins.
75. A simple valve-valve route is a valve-valve route with no forks and joins.

```
value
  68-75  ppr,sppr,pvr,spvr,vpr,svpr,vvr,svvr: R → Bool
         pre {ppr,sppr,pvr,spvr,vpr,svpr,vvr,svvr}(n): len n≥2

  68  ppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_P(fu) ∧ is_P(lu) ∧ is_πfjr(ℓ)
  69  sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
  70  pvr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_P(fu) ∧ is_V(r(len r)) ∧ is_πfjr(ℓ)
  71  sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
  72  vpr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_V(fu) ∧ is_P(lu) ∧ is_πfjr(ℓ)
  73  sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
  74  vvr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_V(fu) ∧ is_V(lu) ∧ is_πfjr(ℓ)
  75  sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)

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

Special Routes, II

Given a unit of a route,

76. if they exist (∃),
77. find the nearest pump or valve unit,
78. "upstream" and
79. "downstream" from the given unit.

```
value
  76 ∃UpPoV: U × R → Bool
  76 ∃DoPoV: U × R → Bool
  78 find_UpPoV: U × R ~→ (P|V), pre find_UpPoV(u,r): ∃UpPoV(u,r)
  79 find_DoPoV: U × R ~→ (P|V), pre find_DoPoV(u,r): ∃DoPoV(u,r)
  76 ∃UpPoV(u,r) ≡
  76    ∃ i,j Nat·{i,j}⊆inds r∧i≤j∧{is_V|is_P}(r(i))∧u=r(j)
  76 ∃DoPoV(u,r) ≡
  76    ∃ i,j Nat·{i,j}⊆inds r∧i≤j∧u=r(i)∧{is_V|is_P}(r(j))
  78 find_UpPoV(u,r) ≡
  78    let i,j:Nat·{i,j}⊆indsr∧i≤j∧{is_V|is_P}(r(i))∧u=r(j) in r(i) end
  79 find_DoPoV(u,r) ≡
  79    let i,j:Nat·{i,j}⊆indsr∧i≤j∧u=r(i)∧{is_V|is_P}(r(j)) in r(j) end
```

State Attributes of Pipeline Units

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

80. Oil flow, $\phi : \Phi$, is measured in volume per time unit.
81. Pumps are either pumping or not pumping, and if not pumping they are closed.
82. Valves are either open or closed.
83. Any unit permits a maximum input flow of oil while maintaining laminar flow. We shall assume that we need not be concerned with turbulent flows.
84. At any time any unit is sustaining a current input flow of oil (at its input(s)).
85. While sustaining (even a zero) current input flow of oil a unit leaks a current amount of oil (within the unit).

```
type
  80  Φ
  81  PΣ == pumping | not_pumping
  81  VΣ == open | closed
value
       −,+: Φ × Φ → Φ, <,=,>: Φ × Φ → Bool
  81      obs_PΣ: P → PΣ
  82      obs_VΣ: V → VΣ
  83–85  obs_LamiΦ.obs_CurrΦ,obs_LeakΦ: U → Φ
  is_Open: U → Bool
    case u of
      mkΠ(_)→true,mkF(_)→true,mkJ(_)→true,mkW(_)→true,mkS(_)→true,
      mkP(_)→obs_PΣ(u)=pumping,
      mkV(_)→obs_VΣ(u)=open
    end
  acceptable_LeakΦ, excessive_LeakΦ: U → Φ
axiom
  ∀ u:U · excess_LeakΦ(u) > accept_LeakΦ(u)
```

Flow Laws

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

86. When, in Item 84, for a unit u, we say that at any time any unit is sustaining a current input flow of oil, and when we model that by obs_CurrΦ(u) then we mean that obs_CurrΦ(u) - obs_LeakΦ(u) represents the flow of oil from its outputs.

```
value
  86  obs_inΦ: U → Φ
  86  obs_inΦ(u) ≡ obs_CurrΦ(u)
  86  obs_outΦ: U → Φ
law:
  86  ∀ u:U · obs_outΦ(u) = obs_CurrΦ(u)−obs_LeakΦ(u)
```

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

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

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

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

Possibly Desirable Properties

88. Let r be a route of length two or more, whose first unit is a pump, p, whose last unit is a valve, v and whose intermediate units are all pipes: if the pump, p is pumping, then we expect the valve, v, to be open.
89. Let r be a route of length two or more, whose first unit is a pump, p, whose last unit is another pump, p' and whose intermediate units are all pipes: if the pump, p is pumping, then we expect pump p'', to also be pumping.
90. Let r be a route of length two or more, whose first unit is a valve, v, whose last unit is a pump, p and whose intermediate units are all pipes: if the valve, v is closed, then we expect pump p, to not be pumping.
91. Let r be a route of length two or more, whose first unit is a valve, v', whose last unit is a valve, v'' and whose intermediate units are all pipes: if the valve, v' is in some state, then we expect valve v'', to also be in the same state.

desirable properties:

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

89  ∀ r:R · sppr(r) ∧
89    sppr_prop(r): obs_PΣ(hd r)=pumping⇒obs_PΣ(r(len r))=pumping

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

91  ∀ r:R · svvr(r) ∧
91    svvr_prop(r): obs_PΣ(hd r)=obs_PΣ(r(len r))
```

Pipeline Actions

•*Simple Pump and Valve Actions*

92. Pumps may be set to pumping or reset to not pumping irrespective of the pump state.
93. Valves may be set to be open or to be closed irrespective of the valve state.
94. In setting or resetting a pump or a valve a desirable property may be lost.

value

```
92  pump_to_pump, pump_to_not_pump: P → N → N
93  valve_to_open, valve_to_close: V → N → N
```

```
value
  92  pump_to_pump(p)(n) as n′
  92    pre p ∈ obs_Us(n)
  92    post let p′:P•obs_UI(p)=obs_UI(p′) in
  92           obs_PΣ(p′)=pumping∧else_equal(n,n′)(p,p′) end
  92  pump_to_not_pump(p)(n) as n′
  92    pre p ∈ obs_Us(n)
  92    post let p′:P•obs_UI(p)=obs_UI(p′) in
  92           obs_PΣ(p′)=not_pumping∧else_equal(n,n′)(p,p′) end
  93  valve_to_open(v)(n) as n′
  92    pre v ∈ obs_Us(n)
  93    post let v′:V•obs_UI(v)=obs_UI(v′) in
  92           obs_VΣ(v′)=open∧else_equal(n,n′)(v,v′) end
  93  valve_to_close(v)(n) as n′
  92    pre v ∈ obs_Us(n)
  93    post let v′:V•obs_UI(v)=obs_UI(v′) in
  92           obs_VΣ(v′)=close∧else_equal(n,n′)(v,v′) end
```

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

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

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

Events

•*Unit Handling Events*

95. Let n be any acyclic net.

95. If there exists p, p', v, v', pairs of distinct pumps and distinct valves of the net,

95. and if there exists a route, r, of length two or more of the net such that

96. all units, u, of the route, except its first and last unit, are pipes, then

97. if the route "spans" between p and p' and the *simple desirable property*, sppr(r), does not hold for the route, then we have a possibly undesirable event — that occurred as soon as sppr(r) did not hold;

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

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

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

```
events:
  95   ∀ n:N · acyclic(n) ∧
  95     ∃ p,p′:P,v,v′:V · {p,p′,v,v′}⊆obs_Us(n)⇒
  95       ∧ ∃ r:R · routes(n) ∧
  96           ∀ u:U · u ∈ elems(r)\{hd r,r(len r)} ⇒ is_Π(i) ⇒
  97             p=hd r∧p′=r(len r) ⇒ ∼sppr_prop(r) ∧
  98             p=hd r∧v=r(len r) ⇒ ∼spvr_prop(r) ∧
  99             v=hd r∧p=r(len r) ⇒ ∼svpr_prop(r) ∧
  100             v=hd r∧v′=r(len r) ⇒ ∼svvr_prop(r)
```

• *Foreseeable Accident Events*

- A number of foreseeable accidents may occur.

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

Well-formed Operational Nets

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

value

106 wf_OpN: N → **Bool**
106 wf_OpN(n) ≡
107 satisfies axiom 48 on page 377 ∧ acyclic(n): Item 53 on page 379 ∧
107 wfN_SP(n): Item 67 on page 419 ∧
107 satisfies flow laws, 86 on page 428 and 87 on page 430 ∧
107(a) $\exists$ w:W,s:S · {w,s}⊆obs_Us(n) ⇒
107(b) $\exists$ r:R· ⟨w⟩⌢r⌢⟨s⟩ ∈ routes(n)

Orderly Action Sequences

•*Initial Operational Net*

108. Let us assume a notion of an initial operational net.
109. Its pump and valve units are in the following states
    (a) all pumps are not_pumping, and
    (b) all valves are closed.

value

108 initial_OpN: N → **Bool**
109 initial_OpN(n) ≡ wf_OpN(n) ∧
109(a) ∀ p:P · p ∈ obs_Us(n) ⇒ obs_PΣ(p)=not_pumping ∧
109(b) ∀ v:V · v ∈ obs_Us(n) ⇒ obs_VΣ(p)=closed

Oil Pipeline Preparation and Engagement

110. We now wish to prepare a pipeline from some well, $w : W$, to some sink, $s : S$, for flow.
    (a) We assume that the underlying net is operational wrt. w and s, that is, that there is a route, r, from w to s.
    (b) Now, an orderly action sequence for engaging route r is to "work backwards", from s to w
    (c) setting encountered pumps to pumping and valves to open.

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

```
value
  110    prepare_and_engage: $W \times S \rightarrow N \xrightarrow{\sim} N$
  110    prepare_and_engage(w,s)(n) ≡
  110(a)      let r:R · ⟨w⟩⌢r⌢⟨s⟩ ∈ routes(n) in
  110(b)      action_sequence(⟨w⟩⌢r⌢⟨s⟩)(len⟨w⟩⌢r⌢⟨s⟩)(n) end
  110      pre ∃ r:R · ⟨w⟩⌢r⌢⟨s⟩ ∈ routes(n)

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

Emergency Actions

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

■ End of Example 55

C.4. Behaviours: A Semantic Model of a Class of Mereologies

- The model of mereology (Slides 387–347) given earlier focused on the following simple entities (i) the assemblies, (ii) the units and (iii) the connectors.
- To assemblies and units we associate CSP processes, and
- to connectors we associate a CSP channels,
- one-by-one.
- The connectors form the mereological attributes of the model.

C.4.1. Channels

- The CSP channels,
  – are each “anchored” in two parts:
  – if a part is a unit then in “its corresponding” unit process, and
  – if a part is an assembly then in “its corresponding” assembly process.
- From a system assembly we can extract all connector identifiers.
- They become indexes into an array of channels.
  – Each of the connector channel identifiers is mentioned
  – in exactly two unit or assembly processes.

```
value
  s:S
  kis:KI-set = xtr∀KIs(s)
type
  ChMap = AUI →m KI-set
value
  cm:ChMap = [obs_AUI(p)↦obs_KIs(p)|p:P·p ∈ xtr_Ps(s)]
channel
  ch[i|i:KI·i ∈ kis] MSG
```

C.4.2. Process Definitions

```
value
  system: S → Process
  system(s) ≡ assembly(s)

  assembly: a:A→in,out {ch[cm(i)]|i:KI·i ∈ cm(obs_AUI(a))} process
  assembly(a) ≡
    𝓜𝒜(a)(obs_AΣ(a)) ‖
    ‖ {assembly(a′)|a′:A·a′ ∈ obs_Ps(a)} ‖
    ‖ {unit(u)|u:U·u ∈ obs_Ps(a)}
  obs_AΣ: A → AΣ

  𝓜𝒜: a:A→AΣ→in,out {ch[cm(i)]|i:KI·i ∈ cm(obs_AUI(a))} process
  𝓜𝒜(a)(aσ) ≡ 𝓜𝒜(a)(A𝓕(a)(aσ))

  A𝓕: a:A → AΣ → in,out {ch[em(i)]|i:KI·i ∈
  cm(obs_AUI(a))}×AΣ
```

```
  unit: u:U → in,out {ch[cm(i)]|i:KI·i ∈ cm(obs_UI(u))} process
  unit(u) ≡ 𝓜𝒰(u)(obs_UΣ(u))
  obs_UΣ: U → UΣ

  𝓜𝒰: u:U → UΣ → in,out {ch[cm(i)]|i:KI·i ∈ cm(obs_UI(u))} process
  𝓜𝒰(u)(uσ) ≡ 𝓜𝒰(u)(U𝓕(u)(uσ))

  U𝓕: U → UΣ → in,out {ch[em(i)]|i:KI · i ∈ cm(obs_AUI(u))}  UΣ
```

C.4.3. Mereology, Part III

- (See Sect. on page 410 for Mereology, Part II.)
- A little more meaning has been added to the notions of parts and connections.
- The within and adjacent to relations between parts (assemblies and units) reflect a phenomenological world of geometry, and
- the connected relation between parts (assemblies and units)
  - reflect both physical and conceptual world understandings:
    - ∗ physical world in that, for example, radio waves cross geometric “boundaries”, and
    - ∗ conceptual world in that ontological classifications typically reflect lattice orderings where *overlaps* likewise cross geometric “boundaries”.

C.4.4. Discussion

C.4.4.1. Partial Evaluation

- The assembly function "first" "functions" as a compiler.
- The 'compiler' translates an assembly structure into three process expressions:
  - the $\mathcal{M}_\mathcal{A}(a)(a\sigma)$ invocation,
  - the parallel composition of assembly processes, a', one for each sub-assembly of a, and
  - the parallel composition of unit processes, one for each unit of assembly a —
  - with these three process expressions "being put in parallel".
  - The recursion in assembly ends when a sub-...-assembly consists of no sub-sub-...-assemblies.
- Then the compiling task ends and the many generated $\mathcal{M}_\mathcal{A}(a)(a\sigma)$ and $\mathcal{M}_\mathcal{U}(u)(u\sigma)$ process expressions are invoked.

C.4.4.2. Generalised Channel Processes

- That completes our 'contribution':
  - A mereology of systems has been given
  - a syntactic explanation, Sect. 2,
  - a semantic explanation, Sect. 5 and
  - their relationship to classical mereologies.

End of Lecture 12: MEREOLOGY

Start of Lecture 13: Domain Actions

D. Domain Actions BLANK

D.1. Definitions

D.2. Examples

D.3. Research Challenge

End of Lecture 13: Domain Actions

Start of Lecture 14: Domain Events

E. Domain Events BLANK

E.1. Definitions

E.2. Examples

E.3. Research Challenge

(E. **Domain Events** BLANK E.3. **Research Challenge**)

End of Lecture 14: Domain Events

(E. **Domain Events** BLANK E.3. **Research Challenge**)

Start of Lecture 15: Domain Behaviour

F Domain Behaviours BLANK

F.1. Definitions

(F. **Domain Events** BLANK F.1. **Definitions**)

F.2. Examples

(F. **Domain Events** BLANK F.2. **Examples**)

F.3. Research Challenge

End of Lecture 15: Domain Behaviour

Start of Lecture 16: A Specification Ontology

G. A Specification Ontology

G.1. Description Ontology Versus Ontology Description

- According to Wikipedia: *Ontology is the philosophical study of*
  - (i) *the nature of being, existence or reality in general,*
  - (ii) *as well as of the basic categories of being and their relations.*
- An earlier lecture emphasized the need for describing domain phenomena and concepts.
- This section puts forward a description ontology:
  - (i) *which "natures of being, existence or reality"* and
  - (ii) *which "categories of being and their relations".*

  which we shall apply in the description of domain phenomena and concepts.

- Yes, we do know that
  - the term 'description ontology' can easily be confused with 'ontology description' —
  - a term used very much in two computing related communities:
    - ∗ AI (artificial intelligence) and
    - ∗ WWW (World Wide Web).
  - These communities use the term 'ontology' as we use the term 'domain'.
- By [domain] 'description ontology' we shall mean
  - a set of notions
  - that are used in
  - describing a domain.
- So the ontology is one of the description language
- not of the domain that is being described.

G.2. Categories, Predicates and Observers for Describing Domains

- It is not the purpose of this talk to motivate the categories, predicates and observer functions for describing phenomena and concepts.
- This is done elsewhere.
- Instead we shall more-or-less postulate one approach to the analysis of domains.
- We do so by postulating a number of meta-categories, meta-predicates and meta-observer functions.
- They characterise those non-meta categories, predicates and observer functions that the domain engineer cum researcher is suggested to make use of.

G.2.1. The Hypothetical Nature of Categories, Predicates and Observers

- In the following we shall postulate some categories,
  - that is, some meta-types:

categories

$\mathbb{ALPHA}, \mathbb{BETA}, \mathbb{GAMMA}$

- What such a clause as the above means
  - is that we postulate that there are such categories of "things" (phenomena and concepts)
  - in the world of domains.

- That is,
  - there is no proof that such "things" exists.
  - It is just our way of modelling domains.
  - If that way is acceptable to other domain science researchers, fine.
  - In the end,
    - ∗ which we shall never reach,
    - ∗ those aspects of a, or the domain science,
    - ∗ may "survive".
    - ∗ If not, not !

G.2.2. Predicates and Observers

- With the categories just introduced we then go on to postulate some predicate and observer functions.
- For example:

predicate signatures

is_$\mathbb{ALPHA}$: "Things" $\rightarrow$ **Bool**
is_$\mathbb{BETA}$: "Things" $\rightarrow$ **Bool**
is_$\mathbb{GAMMA}$: "Things" $\rightarrow$ **Bool**

observer signatures

obs_$\mathbb{ALPHA}$: "Things" $\stackrel{\sim}{\rightarrow}$ $\mathbb{ALPHA}$
obs_$\mathbb{BETA}$: $\mathbb{ALPHA}$ $\stackrel{\sim}{\rightarrow}$ $\mathbb{BETA}$
obs_$\mathbb{GAMMA}$: $\mathbb{ALPHA}$ $\stackrel{\sim}{\rightarrow}$ $\mathbb{GAMMA}$

- So we are "fixing" a logic !

- The "Things" clause is a reference to the domain under scrutiny.
  - Some 'things' in that domain are of category $\mathbb{ALPHA}$, or $\mathbb{BETA}$, or $\mathbb{GAMMA}$.
  - Some are not.
- It is then postulated that
  - from such things of category $\mathbb{ALPHA}$
  - one can observe things of categories $\mathbb{BETA}$ or $\mathbb{GAMMA}$.
- Whether this is indeed the case,
  - i.e., that one can observe these things
  - is a matter of conjecture, not of proof.

G.2.3. Meta-Conditions

- Finally we may sometimes postulate the existence of a meta-axiom:

meta condition:
Predicates over $\mathbb{ALPHA}$, $\mathbb{BETA}$ and $\mathbb{GAMMA}$

- Again,
  - the promulgation of such logical meta-expressions
  - are just conjectures,
  - not the expression of "eternal" truths.

G.2.4. Discussion

- So, all in all, we suggest four kinds of meta-notions:
  - categories,
  - is_Category predicates,
  - obs_Property) predicates,
  - obs_Category observers
  - obs_Attribute observers, and
  - meta-conditions (axiom-like predicates).

- The
  - **category [type]** A, B, ...,
  - is_A, is_B, ...
  - obs_A, obs_B, ...
  - **meta-condition [axiom]** predicate

  notions derive from McCarthy's *analytic syntax* 1962 paper.
- In that paper McCarthy also suggested a *synthetic syntax* constructor function: mk_A, ...
  - At present, we find no need to introduce this *synthetic syntax* constructor function.
  - A basic reason for this is that we are not constructing domain phenomena.
  - The reason McCarthy (and computing science) needed the *synthetic syntax* constructor functions is that software is constructed.

G.2.5. Entities

- What we shall describe is what we shall refer to as entities.
- In other words, there is a category and meta-logical predicate

  ENTITY, is_ENTITY.
- The is_ENTITY predicate applies to "whatever" in the domain, whether an entity or not, and "decides", i.e., is postulated to analyse whether that "thing" is an entity or not:

predicate signature:
is_ENTITY: "Thing" → **Bool**
meta condition:
$\forall$ e:ENTITY · is_ENTITY(e)

• • •

- By introducing the predicate is_ENTITY we have put the finger on what this section is all about, namely
  - *"what exists ?"* and
  - *"what can be described ?"*
- We are postulating a description ontology.
  - It may not be an adequate one.
  - It may have flaws.
  - But, for the purposes of raising some issues of epistemological and ontological nature, it is adequate.

G.2.6. Entity Categories

- We postulate four entity categories:

category:
SIMPLE_ENTITY, ACTION, EVENT, BEHAVIOUR

- Simple entities are **phenomena** or **concepts**.
- **Simple entity phenomena** are the things we can point to, touch and see. They are manifest.
- Other phenomena, for example those we can hear, smell, taste, or measure by physics (including chemistry) apparatus are properties (attributes) of simple entity phenomena.
- **Concepts** are abstractions about phenomena and/or other concepts.
- A subset of simple domain entities form a **state**.

- **Actions** are the result of applying functions to simple domain entities and changing the **state**.
- **Events** are **state** changes that satisfy a predicate on the 'before' and 'after states'.
- **Behaviours** are sets of sequences (of sets of) actions and events.

category:
$\mathbb{ENTITY} = \mathbb{SIMPLE\_ENTITY} \cup \mathbb{ACTION} \cup \mathbb{EVENT} \cup \mathbb{BEHAVIOUR}$

- With each of the four categories there is a predicate:

predicate signature:
is_$\mathbb{SIMPLE_ENTITY}$ "Thing" → **Bool**
is_$\mathbb{ACTION}$ "Thing" → **Bool**
is_$\mathbb{EVENT}$ "Thing" → **Bool**
is_$\mathbb{BEHAVIOUR}$ "Thing" → **Bool**

- Each of the above four predicates require that their argument t:"Thing" satisfies:

is_$\mathbb{ENTITY}$(t)

- The ∪ "union" is inclusive:

meta condition:
∀ t:"Thing"•is_$\mathbb{ENTITY}$(t) ⇒
is_$\mathbb{SIMPLE_ENTITY}$(t) ∨ is_$\mathbb{ACTION}$(t) ∨ is_$\mathbb{EVENT}$(t) ∨ is_$\mathbb{BEHAVIOUR}$(t)

G.2.7. Simple Entities

- We postulate
  - that there are **atomic** simple entities,
  - that there are [therefrom distinct] **composite** simple entities,
  - and that a simple entity is indeed either atomic or composite.
- That
  - atomic simple entities cannot meaningfully be described as consisting of proper other simple entities, but that
  - composite simple entities indeed do consist of proper other simple entities.

- That is:

category:
$\mathbb{SIMPLE\_ENTITY} = \mathbb{ATOMIC} \cup \mathbb{COMPOSITE}$
observer signature:
is_$\mathbb{ATOMIC}$: $\mathbb{SIMPLE\_ENTITY}$ → **Bool**
is_$\mathbb{COMPOSITE}$: $\mathbb{SIMPLE\_ENTITY}$ → **Bool**
meta condition:
$\mathbb{ATOMIC} \cap \mathbb{COMPOSITE} = \{\}$
∀ s: "Things":$\mathbb{SIMPLE\_ENTITY}$ ·
is_$\mathbb{ATOMIC}$(s) ≡ ∼is_$\mathbb{COMPOSITE}$(s)

G.2.8. Discrete and Continuous Entities

- We postulate two forms of $\mathbb{SIMPLE\_ENTITIES}$:
  - $\mathbb{DISCRETE}$, such as a railroad net, a bank, a pipeline pump, and a securities instrument, and
  - $\mathbb{CONTINUOUS}$, such as oil and gas, coal and iron ore, and beer and wine.

category:
$\mathbb{SIMPLE\_ENTITY} = \mathbb{DISCRETE\_SIMPLE\_ENTITY} \cup \mathbb{CONTINUOUS\_SIMPLE\_EN}$
predicate signatures:
is_$\mathbb{DISCRETE_SIMPLE_ENTITY}$: $\mathbb{SIMPLE\_ENTITY}$ → **Bool**
is_$\mathbb{CONTINUOUS_SIMPLE_ENTITY}$: $\mathbb{SIMPLE\_ENTITY}$ → **Bool**
meta condition:
[is it desirable to impose the following]
∀ s:$\mathbb{SIMPLE\_ENTITY}$ ·
is_$\mathbb{DISCRETE_SIMPLE_ENTITY}$(s) ≡ ∼$\mathbb{CONTINUOUS_SIMPLE_ENTITY}$(s) ?

G.2.9. Attributes

- Simple entities are characterised by their attributes:
  - attributes have name, are of type and has some value;
  - no two (otherwise distinct) attributes of a simple entity has the same name.

category:
$\mathbb{ATTRIBUTE}, \mathbb{NAME}, \mathbb{TYPE}, \mathbb{VALUE}$
observer signature:
obs_$\mathbb{ATTRIBUTE}$s: $\mathbb{SIMPLE\_ENTITY} \to \mathbb{ATTRIBUTE}$**-set**
obs_$\mathbb{NAME}$: $\mathbb{ATTRIBUTE} \to \mathbb{NAME}$
obs_$\mathbb{TYPE}$: $\mathbb{ATTRIBUTE} \times \mathbb{NAME} \to \mathbb{TYPE}$
obs_$\mathbb{VALUE}$: $\mathbb{ATTRIBUTE} \times \mathbb{NAME} \to \mathbb{VALUE}$
meta condition:
$\forall$ s:$\mathbb{SIMPLE\_ENTITY}$ ·
  $\forall$ a,a′:$\mathbb{ATTRIBUTE}$ · {a,a′}⊆obs_$\mathbb{ATTRIBUTE}$s(s)
   $\wedge$ a≠a′ ⇒ obs_$\mathbb{NAME}$(a)≠obs_$\mathbb{NAME}$(a′)

- Examples of attributes of atomic simple entities are:
  - (i) A pipeline pump usually has the following attributes: `maximum pumping capacity`, `current pumping capacity`, `whether for oil or gas`, `diameter (of pipes to which the valve connects)`, etc.
  - (ii) Attributes of a person usually includes `name`, `gender`, `birth date`, `central registration number`, `address`, `marital state`, `nationality`, etc.

- Examples of attributes of composite simple entities are:
  - (iii) A railway system usually has the following attributes: `name of system`, `name of geographic areas of location of rail nets and stations`, `whether a public or a private company` `whether fully, partly or not electrified`, etc.
  - (iv) Attributes of a bank usually includes: `name of bank`, `name of geographic areas of location of bank branch offices`, `whether a commercial portfolio bank or a high street, i.e., demand/deposit bank`, etc.
- We do not further define what we mean by attribute names, types and values.

G.2.10. Atomic Simple Entities: Attributes

- Atomic simple entities are characterised only by their attributes.

G.2.11. Composite Simple Entities: Attributes, Sub-entities and Mereolog

- Composite simple entities are characterised by three properties:
  - (i) their **attributes**,
  - (ii) a proper set of one or more **sub-entities** (which are simple entities) and
  - (iii) a **mereology** of these latter, that is,
    - ∗ how they relate to one another, i.e.,
    - ∗ how they are composed.

G.2.11.1. Sub-entities

- Proper sub-entities,
  - that is simple entities properly contained, as immediate **parts** of a composite simple entity,

  can be observed (i.e., can be postulated to be observable):

observer signature:
obs_SIMPLE_ENTITIES: COMPOSITE → SIMPLE_ENTITY-**set**

G.2.11.2. Mereology, Part IV

- Mereology is the theory of **part-hood** relations:
  - of the relations of part to whole
  - and the relations of **part** to **part** within a **whole**.
- Suffice it to suggest some mereological structures:
  - **Set Mereology:** The individual sub-entities of a composite entity are “un-ordered” like elements of a set.
    * The obs_SIMPLE_ENTITIES function yields the set elements.

  predicate signature:
  is_SET: COMPOSITE → **Bool**

- **Cartesian Mereology:** The individual sub-entities of a composite entity are “ordered” like elements of a Cartesian (grouping).
  * The function obs_ARITY yields the arity, 2 or more, of the simple Cartesian entity.
  * The function obs_CARTESIAN yields the Cartesian composite simple entity.

predicate signature:
  is_CARTESIAN: COMPOSITE → **Bool**
observer signatures:
  obs_ARITY: COMPOSITE $\xrightarrow{\sim}$ **Nat**
    pre: obs_ARITY(s) is_CARTESIAN(s)
  obs_CARTESIAN: COMPOSITE $\xrightarrow{\sim}$
        SIMPLE_ENTITY × ... × SIMPLE_ENTITY
    pre obs_CARTESIAN(s): is_CARTESIAN(s)
meta condition:
  ∀ c:SIMPLE_ENTITY·
    is_COMPOSITE(c)∧is_CARTESIAN(c) ⇒
      obs_SIMPLE_ENTITIES(c) = **elements of** obs_CARTESIAN(c)
     ∧ **cardinality of** obs_SIMPLE_ENTITIES(c) = obs_ARITY(c)

* We just postulate the **elements of** and the **cardinality of** meta-functions.

– **List Mereology:** The individual sub-entities of a composite entity are "ordered" like elements of a list (i.e., a sequence).

∗ Where Cartesians are fixed arity sequences,

∗ lists are variable length sequences.

```
predicate signature:
  is_LIST: COMPOSITE → Bool
observer signatures:
  obs_LIST: COMPOSITE ~→ list of SIMPLE_ENTITY
    pre is_LIST(s): is_COMPOSITE(s)
  obs_LENGTH: COMPOSITE ~→ Nat
    pre is_LIST(s): is_COMPOSITE(s)
meta condition:
  ∀ s:SIMPLE_ENTITY·
    is_COMPOSITE(s)∧is_LIST(s) ⇒
      obs_SIMPLE_ENTITIES(s) = elements of obs_LIST(s)
```

∗ We also just postulate the **list of** and the **elements of** meta-functions.

– **Map Mereology:** The individual sub-entities of a map are "indexed" by unique definition set elements.

– Thus we can speak of pairings of unique map definition set element identifications and their not necessarily distinct range set elements.

∗ By a map we shall therefore understand

· a function, with a finite definition set,

· from distinct definition set elements

· to not necessarily distinct range elements,

· such that the pairs of (definition set,range) elements,

· which are all simple entities,

· can be characterised by a predicate.

∗ It is this,

· the finiteness of maps

· and the (potential, but often un-expressed) predicate,

which 'distinguishes' maps from functions.

∗ Let us refer to the map as being of **category** MAP.

∗ Let us refer to the definition set elements of a map as being the DEFINITION_SET of the MAP.

∗ Let us refer to the range elements of such a map as being the RANGE of the MAP.

∗ No two definition set elements of a map, to repeat, are the same.

∗ Given a definition set element, s, of a map, m, one can obtain its IMAGE of the RANGE of m.

```
predicate signature:
  is_MAP: COMPOSITE → Bool
observer signatures:
  obs_MAP: COMPOSITE ~→ MAP
    pre obs_MAP(c): is_MAP(c)
  obs_DEFINITION_SET: MAP → SIMPLE_ENTITY-set
    pre obs_MAP(c): is_MAP(c)
  obs_RANGE: MAP → SIMPLE_ENTITY-set
    pre obs_MAP(c): is_MAP(c)
  obs_IMAGE: MAP × SIMPLE_ENTITY ~→ SIMPLE_ENTITY
    pre obs_IMAGE(m,d): is_MAP(m) ∧ d ∈ obs_DEFINITION_SET(m)
meta condition:
  ∀ m:SIMPLE_ENTITY·
    is_COMPOSITE(m)∧is_MAP(m) ⇒
      obs_SIMPLE_ENTITIES(m) =
        {(d,obs_IMAGE(c,d))|d:SIMPLE_ENTITY·d ∈ obs_DEFINITION_SET(m)
```

- ∗ Given that we can postulate "an existence" of
  - · the obs_DEFINITION_SET and
  - · the obs_RANGE

  observer functions
- ∗ we can likewise postulate a category

```
category
  MAP = map of SIMPLE_ENTITY into ENTITY
observer signatures
  obs_DEF_SET: MAP → set of SIMPLE_ENTITY
  obs_RNG: MAP → set of ENTITY
meta condition
  ∀ m:MAP •
    obs_DEF_SET(m) = obs_DEFINITION_SET(m)
   ∧ obs_RNG(m) = obs_RANGE(m)
```

- ∗ Here, again, the **map of ... into ...** is further unexplained.
- ∗ We shall not pursue the notions of DEF_SET and RNG further.

– **Graph Mereology:** The individual sub-entities of a composite entity are "ordered" like elements of a graph, i.e., a net, of elements.

- ∗ Trees and lattices are just special cases of graphs.
- ∗ Any (immediate) sub-entity of a composite simple entity of GRAPH mereology may be related to any number of (not necessarily other) (immediate) sub-entities of that same composite simple entity GRAPH in a number of ways:
  - · it may immediately PRECEDE,
  - · or immediate SUCCEED
  - · or be BIDIRECTIONALLY_LINKED

  with these (immediate) sub-entities of that same composite simple entity.
- ∗ In the latter case
  - · some sub-entities PRECEDE a SIMPLE_ENTITY of the GRAPH,
  - · some sub-entities SUCCEED a SIMPLE_ENTITY of the GRAPH,
  - · some both.

```
predicate signature:
  is_GRAPH: COMPOSITE → Bool
observer signatures:
  obs_GRAPH: COMPOSITE ~→ GRAPH
      pre obs_GRAPH(g): is_GRAPH(g)
  obs_PRECEDING_SIMPLE_ENTITIES:
     COMPOSITE × SIMPLE_ENTITY → SIMPLE_ENTITY-set
      pre obs_PRECEDING_SIMPLE_ENTITIES(c,s):
            is_GRAPH(c) ∧ s ∈ obs_SIMPLE_ENTITIES(c)
  obs_SUCCEEDING_SIMPLE_ENTITIES:
     COMPOSITE × SIMPLE_ENTITY → SIMPLE_ENTITY-set
      pre obs_PRECEDING_SIMPLE_ENTITIES(c,s):
            is_GRAPH(c) ∧ s ∈ obs_SIMPLE_ENTITIES(c)
meta condition:
  ∀ c:SIMPLE_ENTITY • is_COMPOSITE(c) ∧ is_GRAPH(c)
     ⇒ let ss = SIMPLE_ENTITIES(c) in
        ∀ s′:SIMPLE_ENTITY • s′ ∈ ss
          ⇒ obs_PRECEDING_SIMPLE_ENTITIES(c)(s′) ⊆ ss
            ∧ obs_SUCCEEDING_SIMPLE_ENTITIES(c)(s′) ⊆ ss
      end
```

G.2.12. Discussion

- Given a "thing", s, which satisfies is_SIMPLE_ENTITY(s),
  - – the domain engineer can now systematically analyse this "thing"
  - – using any of the predicates
    - ∗ is_ATOMIC(s),
    - ∗ is_COMPOSITE(s),
    - ∗ is_SET(s),
    - ∗ is_CARTESIAN(s),
    - ∗ is_LIST(s),
    - ∗ is_MAP(s),
    - ∗ is_GRAPH(s),
    - ∗ etcetera.

    and observer functions sketched above.

- Given any SIMPLE_ENTITY
  - the domain engineer can now analyse it to find out whether it is
    - an ATOMIC or
    - a COMPOSITE

    entity.
  - An, in either case, the domain engineer can analyse it to find out about its ATTRIBUTES.
  - If the SIMPLE_ENTITY is COMPOSITE
    - then its SIMPLE_ENTITIES and
    - their MEREOLOGY

    can be additionally ascertained.

- In summery:
  - If ATOMIC then ATTRIBUTES can be analysed.
  - If COMPOSITE then
    - ATTRIBUTES,
    - SIMPLE_ENTITIES and
    - MEREOLOGY

    can be analysed.

G.2.13. Actions

- By a STATE we mean a set of one or more SIMPLE_ENTITIES.
- By an ACTION we shall understand
  - the application
  - of a FUNCTION
  - to (a set of, including the state of) SIMPLE_ENTITIES
  - such that a STATE change occurs.
- We postulate that the domain engineer can indeed decide,
- that is, conjecture,
- whether a “thing”, which is an ENTITY
- is an ACTION.

category:
ACTION, FUNCTION, STATE
predicate signature:
is_ACTION: ENTITY → **Bool**

- Given an ENTITY of category ACTION one can observe, i.e., conjecture
  - the FUNCTION (being applied),
  - the ARGUMENT CARTESIAN of SIMPLE_ENTITIES to which the FUNCTION is being applied, and
  - the resulting change STATE change.
- Not all elements of the CARTESIAN ARGUMENT are SIMPLE STATE ENTITIES.

category:

STATE = SIMPLE_ENTITY
FUNCTION = SIMPLE_ENTITY × STATE → STATE
ARGUMENT = {|s:SIMPLE_ENTITY·is_CARTESIAN(s)|}

observer signatures:

obs_ACTION: ENTITY → ACTION
obs_FUNCTION: ACTION → FUNCTION
obs_ARGUMENT: ACTION → ARGUMENT
obs_INPUT_STATE: ACTION → STATE
obs_RESULT_STATE: ACTION → STATE

G.2.13.1. "Half-way" Discussion[15]

- The domain engineer cum researcher makes decisions
- as to the modelling of the domain.
- Choices as to whether a "thing" is an entity,
- and, if so, whether it is a simple entity, an action, an event or a behaviour,
- those choices represent abstractions and approximations.

[15] "Halfway": after simple entities and actions and before events and behaviours.

G.2.14. Events

- By an EVENT we shall understand
  - A pair, (σ,σ'), of STATEs,
  - a STIMULUS, s,
  - (which is like a FUNCTION of an ACTION),
  - and an EVENT PREDICATE, $p:\mathcal{P}$,
  - such that $p(\sigma,\sigma')(s)$,

  yields **true**.
- The difference between an ACTION and an EVENT is two things:
  - the EVENT ACTION need not originate within the analysed DOMAIN, and
  - the EVENT PREDICATE is trivially satisfied by most ACTIONs which originate within the analysed DOMAIN.

- Examples of events, that is, of predicates are:
  - a bank goes "bust" (e.g., looses all its monies, i.e., bankruptcy),
  - a bank account becomes negative,
  - (unexpected) stop of gas flow and
  - iron ore mine depleted.
- Respective stimuli of these events could be:
  - (massive) loan defaults,
  - a bank client account is overdrawn,
  - pipeline breakage, respectively
  - over-mining.

- We postulate that the domain engineer from an EVENT can observe
  - the STIMULUS,
  - the BEFORE_STATE,
  - the AFTER_STATE and
  - the EVENT_PREDICATE.
- As said before: the domain engineer cum researcher can decide on these abstractions, these approximations.

```
category:
  STIMULUS = SIMPLE_ENTITY × STATE → STATE
  𝒫 = STATE × STATE → Bool
observer signatures:
  obs_STIMULUS: EVENT → STIMULUS
  obs_BEFORE_STATE: EVENT → STATE
  obs_AFTER_STATE: EVENT → STATE
  obs_EVENT_PREDICATE: EVENT → 𝒫
meta condition:
  ∀ e:EVENT ·
    ∃ s:STIMULUS ·
      INPUT_STATE(e) = BEFORE_STATE(s) ∧
      RESULT_STATE(e) = AFTER_STATE(s) ∧
      ∃ p:𝒫 ·p(s)(INPUT_STATE(e),RESULT_STATE(e))
```

G.2.15. Behaviours

- By a BEHAVIOUR we shall understand
  - a set of sequences of ACTIONs and EVENTs
  - such that some EVENTs in two or more such sequences
    - have their STATEs and PREDICATEs
    - express, for example, mutually exclusive
    - synchronisation and communication EVENTs
    - between these sequences
    - which are each to be considered as simple SEQUENTIAL_BEHAVIOURs.
  - Other forms than mutually exclusive synchronisation and communication EVENTs,
  - that "somehow link" two or more behaviours,
  - can be identified.

- We may think of the mutually exclusive synchronisation and communication EVENTs
- as being designated simply by their PREDICATEs
  - such as, for example, in CSP:

```
type A, B, C, D, M
channel ch M
value
  f: A → out ch C
  g: B → in ch D
  f(a) ≡ ... point ℓ_f:ch!e ...
  g(b) ≡ ... point ℓ_g:ch? ...
```

  - Here the zero capacity buffer communication channel, ch,
  - express mutual exclusivity,
  - and the output/input clauses: ch!e and ch?
  - express synchronisation and communication.

- The predicate is here, in the `CSP` schema, "buried" in
  - the simultaneous occurrence
  - behaviour f having "reached point" **point** ℓ_f and
  - behaviour g having "reached point" **point** ℓ_g.

- We abstract
  - from the orderly example of synchronisation and communication given above and
  - introduce a further un-explained notion of behaviour (synchronisation and communication) BEHAVIOUR INTERACTION LABELs
  - and allow BEHAVIOURs to now just be sets of sequences of
    - ACTIONs and
    - BEHAVIOUR INTERACTION LABELs.
  - such that any one simple sequence has unique labels.

- We can classify some BEHAVIOURs.
  - (i) SIMPLE SEQUENTIAL BEHAVIOURs
    - are sequences of ACTIONs.
  - (ii) SIMPLE CONCURRENT BEHAVIOURs
    - are sets of SIMPLE SEQUENTIAL BEHAVIOURs.
  - (iii) COMMUNICATING CONCURRENT BEHAVIOURs
    - are sets of sequences of
      - ACTIONs and
      - BEHAVIOUR INTERACTION LABELs.
    - We say that two or more such COMMUNICATING CONCURRENT BEHAVIOURs SYNCHRONISE & COMMUNICATE when all distinct BEHAVIOURs "sharing" a (same) label have all reached that label.

- Many other composite behaviours can be observed.
- For our purposes it suffice with having just identified the above.
- SIMPLE_ENTITIES, ACTIONs and EVENTs can be described without reference to time.
- BEHAVIOURs, in a sense, take place over time.[16]

[16]If it is important that ACTIONs take place over time, that is, are not instantaneous, then we can just consider ACTIONs as very simple SEQUENTIAL_BEHAVIOURs not involving EVENTs.

- It will bring us into a rather long discourse
- if we are to present some predicates, observer functions and axioms concerning behaviours — along the lines such predicates, observer functions and axioms were present, above, for SIMPLE_ENTITIES, ACTIONs and EVENTs.
- We refer instead to Johan van Benthems seminal work on the `The Logic of Time`.
- In addition, more generally, we refer to A.N. Prior's and McTaggart's works.
- The paper by Wayne D. Blizard proposes an axiom system for time-space.

G.2.16. Mereology, Part V

G.2.16.1. Compositionality of Entities

- Simple entities — when composite — are said to exhibit a mereology.
- Thus composition of simple entities imply a mereology.
- We discussed mereologies of behaviours: simple sequential, simple concurrent, communicating concurrent, etc.
- Above we did not treat actions and events as potentially being composite.
- But we now relax that seeming constraint.
- There is, in principle, nothing that prevents actions and events from exhibiting mereologies.

- An action, still instantaneous, can,
  - for example, "fork" into a number of concurrent actions, all instantaneous, on "disjoint" parts of a state;
  - or an instantaneous action can "dribble" (not little-by-little, but one-after-the-other. still instantaneously) into several actions as if a simple sequential behaviour, but instantaneous.

- Two or more events
  - can occur simultaneously:
    - ∗ two or more (up to four, usually) people become grandparents
    - ∗ when a daughter of theirs give birth to their first grandchild;
  - or an event can — again a "dribble" (not little-by-little, but instantaneously) — "rapidly" sequence through a number of instantaneous sub-events (with no intervening time intervals):
    - ∗ A bankruptcy events
    - ∗ immediately causes the bankruptcy of several enterprises
    - ∗ which again causes the immediate bankruptcy of several employes,
    - ∗ etcetera.

- The problems of compositionality of entities,
  - whether simple, actions, events or behaviours,
  - is was studied, initially, in [Bjørner and Eir. 2008]

. Impossibility of Definite Mereological Analysis of Seemingly Composite E

- It would be nice if there was a more-or-less obvious way of "deciphering" the mereology of an entity.
- In the many • (bulleted) items above (cf. Set, Cartesian, List, Map, Graph) we may have left the impression with the listener that is a more-or-less systematic way of uncovering the mereology of a composite entity.
- That is not the case: there is no such obvious way.
- It is a matter of both discovery and choice between seemingly alternative mereologies, and it is also a matter of choice of abstraction.

G.3. What Exists and What Can Be Described ?

- In the previous section we have suggested
  - a number of *categories*[17] of entities,

[17]Some categories: ENTITY, SIMPLE_ENTITY, ACTION, EVENT, BEHAVIOUR, ATOMIC, COMPOSITE, DISCRETE, CONTINUOUS, ATTRIBUTE, NAME, TYPE, VALUE, SET, CARTESIAN, LIST, MAP, GRAPH, FUNCTION, STATE, ARGUMENT, STIMULUS, EVENT_PREDICATE, BEFORE_STATE, AFTER_STATE, SEQUENTIAL_BEHAVIOUR, BEHAVIOUR_INTERACTION_LABEL, SIMPLE_SEQUENTIAL_BEHAVIOUR, SIMPLE_CONCURRENT_BEHAVIOUR, COMMUNICATING_CONCURRENT_BEHAVIOUR, etc.

- a number of *predicate*[18] and *observer*[19] functions and
- a number of *meta conditions* (i.e., axioms).

- These concepts and their relations to one-another,
  - suggest an ontology for describing domains.
- It is now very important that we understand these
  - categories,
  - predicates,
  - observers and
  - axioms

  properly.

[18]Some predicates: is_ENTITY, is_SIMPLE_ENTITY, is_ACTION, is_EVENT, is_BEHAVIOUR, is_ATOMIC, is_COMPOSITE, is_DISCRETE_SIMPLE_ENTITY, is_CONTINUOUS_SIMPLE_ENTITY, is_SET, is_CARTESIAN, is_LIST, is_MAP, is_GRAPH, etc.

[19]Some observers: obs_SIMPLE_ENTITY, obs_ACTION, obs_EVENT, obs_BEHAVIOUR, obs_ATTRIBUTE, obs_NAME, obs_TYPE, obs_VALUE, obs_SET, obs_CARTESIAN, obs_ARITY, obs_LIST, obs_LENGTH, obs_DEFINITION_SET, obs_RANGE, obs_IMAGE, obs_GRAPH, obs_PRECEDING_SIMPLE_ENTITIES, obs_SUCCEEDING_SIMPLE_ENTITIES, obs_MEREOLOGY, obs_INPUT_STATE, obs_ARGUMENT, obs_RESULT_STATE, obs_STIMULUS, obs_EVENT_PREDICATE, obs_BEFORE_STATE, obs_AFTER_STATE, etc.

G.3.1. Description Versus Specification Languages

- Footnotes 17–19 (Slide 518) summarised a number of main concepts of an ontology for describing domains.
- The categories and predicate and observer function signatures are not part of a formal language for descriptions.
- The identifiers used for these categories are intended to denote the real thing, classes of entities of a domain.
- In a philosophical discourse about describability of domains one refers to the real things.
- That alone prevents us from devising a formal specification language for giving (syntax and) semantics to a specification, in that language, of what these (Footnote 17–19) identifiers mean.

G.3.2. Formal Specification of Specific Domains

- Once we have decided to describe a specific domain
- then we can avail ourselves of using one or more of a set of formal specification languages.
- But such a formal specification does not give meaning to identifiers of the categories and predicate and observer functions;
- they give meaning to very specific subsets of such categories and predicate and observer functions.
- And the domain specification now ascribes, not the real thing, but usually some form of mathematical structures as models of the specified domain.

G.3.3. Formal Domain Specification Languages

- There are, today, 2009, a large number of formal specification languages.
  - Some or textual, some are diagrammatic.
  - The textual specification languages are like mathematical expressions, that is: linear text, often couched in an abstract "programming language" notation.
  - The diagrammatic specification languages provide for the specifier to draw two-dimensional figures composed from primitives.
- Both forms of specification languages have precise mathematical meanings, but the linear textual ones additionally provide for proof rules.

- Examples of textual, formal specification languages are
  - `Alloy`: model-oriented,
  - `B, Event-B`: model-oriented,
  - `CafeOBJ`: property-oriented (algebraic),
  - `CASL`: property-oriented (algebraic),
  - `DC (Duration Calculus)`: temporal logic,
  - `RAISE, RSL`: property and model-oriented,
  - `TLA+`: temporal logic and sets,
  - `VDM, VDM-SL`: model-oriented and
  - `Z`: model-oriented.
- `DC` and `TLA+` are often used in connection with either a model-oriented specification languages or just plain old discrete mathematics notation !

- But the model-oriented specification languages mentioned above do not succinctly express concurrency.
- The diagrammatic, formal specification languages, listed below, all do that:
  - `Petri Nets`,
  - `Message Sequence Charts (MSC)`,
  - `Live Sequence Charts (LSC)` and
  - `Statecharts`.

G.3.4. Discussion: "Take-it-or-leave-it !"

- With the formal specification languages,
  - not just those listed above,
  - but with any conceivable formal specification language,
    - ∗ the issue is:
      - · you can basically only describe using that language
      - · what it was originally intended to specify,
    - ∗ and that, usually, was to specify software !
- If, in the real domain you find phenomena or concepts,
  - which it is somewhat clumsy
  - and certainly not very abstract
  - or, for you, outright impossible,
- to describe, then, well, then you cannot formalise them !

G.3.4.1.

-
-
-
-

G.3.4.2.

-
-
-
-

End of Lecture 16: A Specification Ontology

Start of Lecture 17: Domain Intrinsics

H. Domain Intrinsics BLANK

H.1. Delineation

H.2. Examples

(H. **Domain Intrinsics** BLANK H.2. **Examples**)

H.3. Research Challenges

(H. **Domain Intrinsics** BLANK H.3. **Research Challenges**)

End of Lecture 17: Domain Intrinsics

(H. **Domain Intrinsics** BLANK H.3. **Research Challenges**)

Start of Lecture 18: Domain Support Technologies

I. Domain Support Technologies

I.1. Definition

- By a support technology of a domain we shall understand
  - either of a set of (one or more) alternative
  - **entities, functions, events** and **behaviours**
  - which "implement" an intrinsic phenomenon or concept.
- Thus for some intrinsic phenomenon or concept
  - there might be a technology
  - which supports that phenomenon or concept.

I.2. Examples

Example 56 – Railway Switches (I): We give a rough sketch description of possible rail unit switch technologies.

- In "ye olde" days, rail switches were "thrown" by manual labour, i.e., by railway staff assigned to and positioned at switches.
- With the advent of reasonably reliable mechanics, pulleys and levers[20] and steel wires, switches were made to change state by means of "throwing" levers in a cabin tower located centrally at the station (with the lever then connected through wires etc., to the actual switch).
- This partial mechanical technology then emerged into electromechanics, and cabin tower staff was "reduced" to pushing buttons.
- Today, groups of switches, either from a station arrival point to a station track, or from a station track to a station departure point, are set and reset by means also of electronics, by what is known as

interlocking (for example, so that two different routes cannot be open in a station if they cross one another).

■ End of Example 56

[20] http://en.wikipedia.org/wiki/Lever

- It must be stressed that Example 56 is just a rough sketch.
- In a proper narrative description the software (cum domain) engineer must describe, in detail, the subsystem of electronics, electromechanics and the human operator interface (buttons, lights, sounds, etc.).
- An aspect of supporting technology includes recording the state-behaviour in response to external stimuli.
- We give an example.

Example 57 – Railway Switches (II): Figure 13 indicates a way of formalising this aspect of a supporting technology.

- Figure 13 intends to model the probabilistic (erroneous and correct) behaviour of a switch when subjected to settings (to switched (s) state) and re-settings (to direct (d) state).
- A switch may go to the switched state from the direct state when subjected to a switch setting s with probability psd.

(I. Domain Support Technologies I.2. Examples)

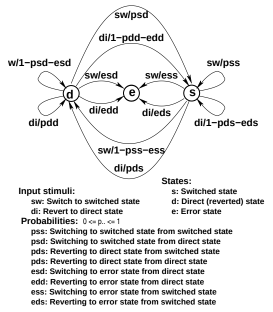

sw/psd
di/1−pdd−edd
w/1−psd−esd
sw/esd
sw/ess
sw/pss
d
e
s
di/edd
di/eds
di/pdd
di/1−pds−eds
sw/1−pss−ess
di/pds
States:
s: Switched state
d: Direct (reverted) state
e: Error state
Input stimuli:
sw: Switch to switched state
di: Revert to direct state
Probabilities: 0 <= p.. <= 1
pss: Switching to switched state from switched state
psd: Switching to switched state from direct state
pds: Reverting to direct state from switched state
pds: Reverting to direct state from direct state
esd: Switching to error state from direct state
edd: Reverting to error state from direct state
ess: Switching to error state from switched state
eds: Reverting to error state from switched state

Figure 13: Probabilistic state switching

(I. Domain Support Technologies I.2. Examples)

Another example shows another aspect of support technology:

- Namely that the technology must guarantee certain of its own behaviours,
- so that software designed to interface with this technology,
- together with the technology, meets dependability requirements.

(I. Domain Support Technologies I.2. Examples)

Example 58 – Sampling Behaviour of Support Technologies:

- Let us consider **i**ntrinsic **A**ir **T**raffic as a continuous function ($\rightarrow$)
- from **T**ime to **F**light **L**ocations:

type
  T, F, L
  iAT = T $\rightarrow$ (F $\overrightarrow{m}$ L)

- But what is observed, by some support technology,
- is not a continuous function, but a discrete sampling (a map $\overrightarrow{m}$):

  sAT = T $\overrightarrow{m}$ (F $\overrightarrow{m}$ L)

- There is a support technology, say in the form of **radar**
- which "observes" the intrinsic traffic and delivers the sampled traffic:

value
  radar: iAT $\rightarrow$ sAT

- But even the radar technology is not perfect.
- Its positioning of flights follows some probabilistic or statistical pattern:

type
  P = {| r:**Real** · 0≤r≤1 |}
  ssAT = P $\overrightarrow{m}$ sAT-**infset**
value
  radar′: iAT $\xrightarrow{\sim}$ ssAT

- The radar technology will, with some probability produce either of a set of samplings,
- and with some other probability some other set of samplings, etc.

■ End of Example 58

I.3. Support Technology Quality Control, a Sketch

- How can we express that a given technology delivers a reasonable support ?
- One approach is to postulate
  - intrinsic and technology states (or observed behaviours), Θ_i, Θ_s,
  - a support technology τ
  - and a "**close**ness" predicate:

```
type
  Θ_i, Θ_s
value
  $\tau$: Θ_i → P $\underset{m}{\rightarrow}$ Θ_s-**infset**
  close: Θ_i × Θ_s → **Bool**
```

- and then require that an experiment can be performed

- which validates the support technology.

- The experiment is expressed by the following axiom:

```
value
  p_threshhold:P
axiom
  ∀ $\theta$_i:Θ_i ·
    let p$\theta$_ss = $\tau$($\theta$_i) in
    ∀ p:P · p>p_threshhold ⇒
      $\theta$_s:Θ_s · $\theta$_s ∈ p$\theta$_ss(p) ⇒ close($\theta$_i,$\theta$_s) end
```

I.4. Research Challenges $\boxed{\text{BLANK}}$

End of Lecture 18: Domain Support Technologies

Start of Lecture 19: Domain Rules and Regulations

J. Domain Rules and Regulations

J.1. Definitions

- By a **rule** we understand
  - a **syntactic** piece of text whose **meaning**
  - apply in any pair of actual present and potential next **states** of the enterprise,
  - and then evaluates to either **true** or **false**:
  - **the rule has been obeyed, or the rule has been (or will be, or might be) broken.**
- By a **regulation** we understand
  - a **syntactic** piece of text whose **meaning**, for example,
  - apply in **states** of the enterprise where a rule has been broken,
  - and when applied in such states will **change the state**,
  - **that is, "remedy" the "breaking of a rule".**

J.2. Abstraction of Rules and Regulations

- **Sti**muli are introduced in order to capture the possibility of rule-breaking next states.
- **Rul**es
- **Reg**ulations
- Θ
- To each of the three syntactic notions: **Sti**muli, **Rul**es and **Reg**ulations there is a **meaning**, i.e., a semantic function from syntax to semantics.
- The meaning, **STI**, of **Sti**muli, are state transitions, that is, a stimulus provokes a state change.
- The meaning, **RUL**, of a **Rul**e, is a predicate over a before (stimulus) state and an after (stimulus) state.

- The meaning, **REG**, of a **Reg**ulation, is another transition, intended to replace stimula transitions whose **Rul**e predicate does not hold, that is, the regulation transition shall lead to an after state for which the rule now holds.

type
  Sti, Rul, Reg, Θ
  RulReg = Rul $\times$ Reg
  STI = $\Theta \rightarrow \Theta$
  RUL = $(\Theta \times \Theta) \rightarrow$ **Bool**
  REG = $\Theta \rightarrow \Theta$
value
  meaning: Sti $\rightarrow$ STI
  meaning: Rul $\rightarrow$ RUL
  meaning: Reg $\rightarrow$ REG
  valid: Sti $\times$ Rul $\rightarrow \Theta \rightarrow$ **Bool**
  valid(sti,rul)$\theta \equiv$ (meaning(rul))(θ,meaning(sti)θ)

axiom
  $\forall$ sti:Sti,(rul,reg):RulReg,θ:Θ $\cdot$ $\sim$valid(sti,rul)$\theta \Rightarrow$ meaning(rul)(θ,meaning(r

J.2.1. Quality Control of Rules and Regulations

- The axiom above presents us with a guideline
  - for checking the suitability of (pairs of) rules and regulations
  - in the context of stimuli:
    - $*$ for every proposed pair of rules and regulations
    - $*$ and for every conceivable stimulus
    - $*$ check whether the stimulus might cause a breaking of the rule
    - $*$ and, if so, whether the regulation
    - $*$ will restore the system to an acceptable state.

J.2.2. Research Challenges

- The above sketched a quality control procedure for ‘stimuli, rules and regulations’.
- It left out the equally important ‘monitoring’ aspects.
- Here is a research challenge:
  - Develop experimentally two or three distinct models of domains involving distinct sets of rules and regulations.
  - Then propose and study concrete implementations of procedures for quality monitoring and control of ‘stimuli, rules and regulations’.

End of Lecture 19: Domain Rules and Regulations

Start of Lecture 20: Domain Scripts

K. Domain Scripts, Licenses and Contracts

K.1. Domain Scripts

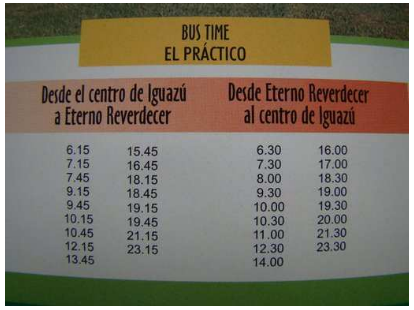
By a **domain script** we shall understand

- a structured text
- which can be interpreted as a set of rules ("in disguise").

Example 59 – Timetables

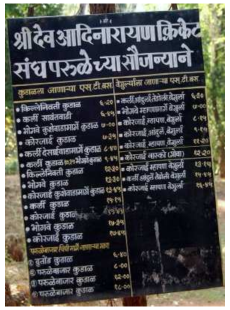
- We shall view timetables as scripts.
- In on the present and next slides (550–571) we shall
  - first narrate and formalise the **syntax**, including the well-formedness of timetable scripts,
  - then we consider the **pragmatics** of timetable scripts,
    - ∗ including the bus routes prescribed by these journey descriptions and

    - ∗ timetables marked with the status of its currently active routes, and
  - finally we consider the **semantics** of timetable, that is, the traffic they denote.
- In Example. ?? on contracts for bus traffic, we shall assume the timetable scripts of this part of the lecture on scripts.



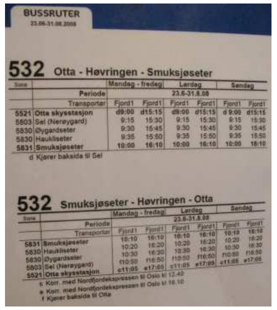
BUS TIME
EL PRÁCTICO

Desde el centro de Iguazú a Eterno Reverdecer		Desde Eterno Reverdecer al centro de Iguazú	
6.15	15.45	6.30	16.00
7.15	16.45	7.30	17.00
7.45	18.15	8.00	18.30
9.15	18.45	9.30	19.00
9.45	19.15	10.00	19.30
10.15	19.45	10.30	20.00
10.45	21.15	11.00	21.30
12.15	23.15	12.30	23.30
13.45		14.00	



श्री देव आदिनारायण क्रिकेट संघ परुळे च्या सौजन्याने

कुडाळला जाणाऱ्या एस.टी.बस		वेंगुर्ल्याला जाणाऱ्या एस.टी.बस	
• किल्लेनिवती कुडाळ	६-२०	• कर्ली आंदुर्ले तेंडोली वेंगुर्ला	६-४०
• कर्ली सावंतवाडी	६-४५	• भोगवे म्हापणमार्गे वेंगुर्ला	७-००
• भोगवे कुशेवाडामार्गे कुडाळ	७-००	• कोरजाई म्हापण वेंगुर्ला	८-१५
• कोरजाई कुडाळ	७-३५	• कोरजाई आंदुर्ले वेंगुर्ला	९-५०
• कर्ली देसाईवाडामार्गे कुडाळ	८-४०	• कोरजाई म्हापण वेंगुर्ला	१२-२०
• कर्ली कुडाळ [illegible] भोगवे कुडाळ	९-४५	• कोरजाई वास्को (गोवा)	१२-२०
• किल्लेनिवती कुडाळ	१२-३०	• कोरजाई म्हापण वेंगुर्ला	१३-१५
• भोगवे कुडाळ	१३-३०	• कर्ली आंदुर्ले तेंडोली वेंगुर्ला	१५-४५
• कोरजाई कुशेवाडामार्गे कुडाळ	१३-४५	• कोरजाई म्हापण वेंगुर्ला	१६-४५
• कर्ली कुडाळ	१४-१५		
• कोरजाई कुडाळ	[illegible]		
• भोगवे कुडाळ	१७-३५		
• कोरजाई कुडाळ	१९-४५		
परुळेबाजार [illegible] जाणाऱ्या [illegible]			
① बुतोंड कुडाळ	६-४०		
② परुळेबाजार कुडाळ	८-००		
③ परुळेबाजार कुडाळ	१२-००		
④ परुळेबाजार कुडाळ	१८-००		



BUSSRUTER
23.06-31.08.2008

532 Otta - Høvringen - Smuksjøseter

Sone		Mandag - fredag		Lørdag		Søndag	
	Periode	23.6-31.8.08					
	Transportør	Fjord1	Fjord1	Fjord1	Fjord1	Fjord1	Fjord1
5521	Otta skysstasjon	d9:00	d15:15	d9:00	d15:15	d 9:00	d15:15
5803	Sel (Nerøygard)	9:15	15:30	9:15	15:30	9:15	15:30
5830	Øygardseter	9:30	15:45	9:30	15:45	9:30	15:45
5830	Haukliseter	9:35	15:50	9:35	15:50	9:35	15:50
5831	Smuksjøseter	10:00	16:10	10:00	16:10	10:00	16:10

d Kjører baksida til Sel

532 Smuksjøseter - Høvringen - Otta

Sone		Mandag - fredag		Lørdag		Søndag	
	Periode	23.6-31.8.08					
	Transportør	Fjord1	Fjord1	Fjord1	Fjord1	Fjord1	Fjord1
5831	Smuksjøseter	10:10	16:10	10:10	16:10	10:10	16:10
5830	Haukliseter	10:20	16:20	10:20	16:20	10:20	16:20
5830	Øygardseter	10:30	16:30	10:30	16:30	10:30	16:30
5803	Sel (Nerøygard)	f10:50	f16:50	f10:50	f16:50	f10:50	f16:50
5521	Otta skysstasjon	c11:05	e17:05	c11:05	e17:05	c11:05	e17:05

c Korr. med Nordfjordekspressen til Oslo kl 12.40
e Korr. med Nordfjordekspressen til Oslo kl 18.10
f Kjører baksida til Otta

Figure 14: Some bus timetables: Italy, India and Norway

⊕ *The Syntax of Timetable Scripts* ⊕

113. Time is a concept covered earlier. Bus lines and bus rides have unique names (across any set of time tables). Hub and link identifiers, HI, LI, were treated from the very beginning.
114. A TimeTable associates to Bus Line Identifiers a set of Journies.
115. Journies are designated by a pair of a BusRoute and a set of BusRides.
116. A BusRoute is a triple of the Bus Stop of origin, a list of zero, one or more intermediate Bus Stops and a destination Bus Stop.
117. A set of BusRides associates, to each of a number of Bus Identifiers a Bus Schedule.
118. A Bus Schedule a triple of the initial departure Time, a list of zero, one or more intermediate bus stop Times and a destination arrival Time.
119. A Bus Stop (i.e., its position) is a Fraction of the distance along a link (identified by a Link Identifier) from an identified hub to an identified hub.
120. A Fraction is a **Real** properly between 0 and 1.

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

type

113. $\mathrm{T}, \mathrm{BLId}, \mathrm{BId}$
114. $\mathrm{TT} = \mathrm{BLId} \underset{m}{\rightarrow} \mathrm{Journies}$
115. $\mathrm{Journies}' = \mathrm{BusRoute} \times \mathrm{BusRides}$
116. $\mathrm{BusRoute} = \mathrm{BusStop} \times \mathrm{BusStop}^* \times \mathrm{BusStop}$
117. $\mathrm{BusRides} = \mathrm{BId} \underset{m}{\rightarrow} \mathrm{BusSched}$
118. $\mathrm{BusSched} = \mathrm{T} \times \mathrm{T}^* \times \mathrm{T}$
119. $\mathrm{BusStop} == \mathrm{mkBS}(\mathrm{s\_fhi{:}HI}, \mathrm{s\_ol{:}LI}, \mathrm{s\_f{:}Frac}, \mathrm{s\_thi{:}HI})$
120. $\mathrm{Frac} = \{|\mathrm{r}{:}\mathbf{Real}\cdot 0<\mathrm{r}<1|\}$
121. $\mathrm{Journies} = \{|\mathrm{j}{:}\mathrm{Journies}'\cdot\exists\ \mathrm{n}{:}\mathrm{N}\cdot \mathrm{wf\_Journies}(\mathrm{j})(\mathrm{n})|\}$

- The free n in $\exists$ n:N · wf_Journies(j)(n) is the net given in the license.

⊕ *Well-formedness of Journies* ⊕

122. A set of journies is well-formed
123. if the bus stops are all different,
124. if a defined notion of a bus line is embedded in some line of the net, and
125. if all defined bus trips (see below) of a bus line are commensurable.

value

```
122. wf_Journies: Journies → N → Bool
122. wf_Journies((bs1,bsl,bsn),js)(hs,ls) ≡
123.   diff_bus_stops(bs1,bsl,bsn) ∧
124.   is_net_embedded_bus_line(⟨bs1⟩⌢bsl⌢⟨bsn⟩)(hs,ls) ∧
125.   commensurable_bus_trips((bs1,bsl,bsn),js)(hs,ls)
```

126. The bus stops of a journey are all different
127. if the number of elements in the list of these equals the length of the list.

value

```
126. diff_bus_stops: BusStop × BusStop* × BusStop → Bool
126. diff_bus_stops(bs1,bsl,bsn) ≡
127.   card elems ⟨bs1⟩⌢bsl⌢⟨bsn⟩ = len ⟨bs1⟩⌢bsl⌢⟨bsn⟩
```

- We shall refer to the (concatenated) list (⟨bs1⟩⌢bsl⌢⟨bsn⟩ = **len** ⟨bs1⟩⌢bsl⌢⟨bsn⟩) of all bus stops as the bus line.

128. To explain that a bus line is embedded in a line of the net
129. let us introduce the notion of all lines of the net, lns,
130. and the notion of projecting the bus line on link sector descriptors.
131. For a bus line to be embedded in a net then means that there exists a line, ln, in the net, such that a compressed version of the projected bus line is amongst the set of projections of that line on link sector descriptors.

value

```
128. is_net_embedded_bus_line: BusStop* → N → Bool
128. is_net_embedded_bus_line(bsl)(hs,ls)
129.   let lns = lines(hs,ls),
130.       cbln = compress(proj_on_links(bsl)(elems bsl)) in
131.   ∃ ln:Line • ln ∈ lns ∧ cbln ∈ projs_on_links(ln) end
```

132. Projecting a list (*) of BusStop descriptors (mkBS(hi,li,f,hi′)) onto a list of Sector Descriptors ((hi,li,hi′))
133. we recursively unravel the list from the front:
134. if there is no front, that is, if the whole list is empty, then we get the empty list of sector descriptors,
135. else we obtain a first sector descriptor followed by those of the remaining bus stop descriptors.

```
value
132. proj_on_links: BusStop* → SectDescr*
132. proj_on_links(bsl) ≡
133.   case bsl of
134.     ⟨⟩ → ⟨⟩,
135.     ⟨mkBS(hi,li,f,hi′)⟩⌢bsl′ → ⟨(hi,li,hi′)⟩⌢proj_on_links(bsl′)
135.   end
```

136. By compression of an argument sector descriptor list we mean a result sector descriptor list with no duplicates.
137. The compress function, as a technicality, is expressed over a diminishing argument list and a diminishing argument set of sector descriptors.
138. We express the function recursively.
139. If the argument sector descriptor list an empty result sector descriptor list is yielded;
140. else
141. if the front argument sector descriptor has not yet been inserted in the result sector descriptor list it is inserted else an empty list is "inserted"

142. in front of the compression of the rest of the argument sector descriptor list.

```
136. compress: SectDescr* → SectDescr-set → SectDescr*
137. compress(sdl)(sds) ≡
138.   case sdl of
139.     ⟨⟩ → ⟨⟩,
140.     ⟨sd⟩⌢sdl′ →
141.       (if sd ∈ sds then ⟨sd⟩ else ⟨⟩ end)
142.       ⌢compress(sdl′)(sds\{sd}) end
```

- In the last recursion iteration (line 142.)
  - the continuation argument sds\{sd}
  - can be shown to be empty: {}.

143. We recapitulate the definition of lines as sequences of sector descriptions.
144. Projections of a line generate a set of lists of sector descriptors.
145. Each list in such a set is some arbitrary, but ordered selection of sector descriptions.

type
143. $\text{Line}' = (\text{HI}\times\text{LI}\times\text{HI})^*$ **axiom** ... **type** Line = ...
value
144. $\text{projs\_on\_links: Line} \rightarrow \text{Line}'\textbf{-set}$
144. $\text{projs\_on\_links(ln)} \equiv$
145. $\{\langle\text{isl(i)}|\text{i}:\langle 1..\textbf{len}\ \text{isl}\rangle\rangle|\text{isx}:\textbf{Nat-set}\cdot\text{isx}\subseteq\textbf{inds}\ \text{ln}\wedge\text{isl}=\text{sort(isx)}\}$

146. sorting a set of natural numbers into an ordered list, isl, of these is expressed by a post-condition relation between the argument, isx, and the result, isl.
147. The result list of (arbitrary) indices must contain all the members of the argument set;
148. and "earlier"elements of the list must precede, in value, those of "later" elements of the list.

value
146. $\text{sort: } \textbf{Nat-set} \rightarrow \textbf{Nat}^*$
146. sort(isx) **as** isl
147. $\quad$ **post card** $\text{isx} = \text{lsn isl} \wedge \text{isx} = \textbf{elems}\ \text{isl} \wedge$
148. $\qquad \forall\ \text{i}:\textbf{Nat}\cdot\{\text{i,i+1}\}\subseteq\textbf{inds}\ \text{isl} \Rightarrow \text{isl(i)}<\text{isl(i+1)}$

149. The bus trips of a bus schedule are commensurable with the list of bus stop descriptions if the following holds:
150. All the intermediate bus stop times must equal in number that of the bus stop list.
151. We then express, by case distinction, the reality (i.e., existence) and timeliness of the bus stop descriptors and their corresponding time descriptors – and as follows.
152. If the list of intermediate bus stops is empty, then there is only the bus stops of origin and destination, and they must be exist and must fit time-wise.

153. If the list of intermediate bus stops is just a singleton list, then the bus stop of origin and the singleton intermediate bus stop must exist and must fit time-wise. And likewise for the bus stop of destination and the the singleton intermediate bus stop.
154. If the list is more than a singleton list, then the first bus stop of this list must exist and must fit time-wise with the bus stop of origin.
155. As for Item 154 but now with respect to last, resp. destination bus stop.
156. And, finally, for each pair of adjacent bus stops in the list of intermediate bus stops
157. they must exist and fit time-wise.

```
value
149. commensurable_bus_trips: Journies → N → Bool
149. commensurable_bus_trips((bs1,bsl,bsn),js)(hs,ls)
150.   ∀ (t1,til,tn):BusSched·(t1,til,tn)∈ rng js∧len til=len bsl∧
151.     case len til of
152.       0 → real_and_fit((t1,t2),(bs1,bs2))(hs,ls),
153.       1 → real_and_fit((t1,til(1)),(bs1,bsl(1)))(hs,ls)∧fit((til(1),t2),(bsl(1),
154.       _ → real_and_fit((t1,til(1)),(bs1,bsl(1)))(hs,ls)∧
155.           real_and_fit((til(len til),t2),(bsl(len bsl),bsn))(hs,ls)∧
156.           ∀ i:Nat·{i,i+1}⊆inds til ⇒
157.             real_and_fit((til(i),til(i+1)),(bsl(i),bsl(i+1)))(hs,ls) end
```

158. A pair of (adjacent) bus stops exists and a pair of times, that is the time interval between them, fit with the bus stops if the following conditions hold:
159. All the hub identifiers of bus stops must be those of net hubs (i.e., exists, are real).
160. There exists links, l, l′, for the identified bus stop links, li, li′,
161. such that these links connect the identified bus stop hubs.
162. Finally the time interval between the adjacent bus stops must approximate fit the distance between the bus stops
163. The distance between two bus stops is a loose concept as there may be many routes, short or long, between them.
164. So we leave it as an exercise to the student to change/augment the description, in order to be able to ascertain a plausible measure of

distance.

165. The approximate fit between a time interval and a distance must build on some notion of average bus velocity, etc., etc.
166. So we leave also this as an exercise to the student to complete.

```
158. real_and_fit: (T×T)×(BusStop×BusStop) → N → Bool
158. real_and_fit((t,t′),(mkBS(hi,li,f,hi′),mkBS(hi″,li′,f′,hi‴)))(hs,ls) ≡
159.   {hi,hi′,hi″,hi‴}⊆his(hs)∧
160.   ∃ l,l′:L·{l,l′}⊆ls∧(obs_LI(l)=li∧obs(l′)=li′)∧
161.     obs_HIs(l)={hi,hi′}∧obs_HIs(l′)={hi″,hi‴}∧
162.   afit(t′−t)(distance(mkBS(hi,li,f,hi′),mkBS(hi″,li′,f′,hi‴))(hs,ls))

163. distance: BusStop × BusStop → N → Distance
164. distance(bs1,bs2)(n) ≡ ... [left as an exercise !] ...

165. afit: TI → Distance → Bool
166.   [ time interval fits distance between bus stops ]
```

K.2. Domain Licenses and Contracts

By a **domain license** we shall understand

- a right or permission granted in accordance with law
- by a competent authority
  - to engage in some business or occupation,
  - to do some act,
  - or to engage in some transaction
- which
  - but for such license
- would be unlawful Merriam Webster On-line.

By a **domain contract** we shall understand

- very much the same thing as a license:
- a binding agreement between two or more persons or parties —
- one which is legally enforceable.

- The concepts of licenses and licensing express relations between
  - *actors* (licensors (the authority) and licensees),
  - *simple entities* (artistic works, hospital patients, public administration and citizen documents) and
  - *operations* (on simple entities), and as performed by actors.
- By issuing a license to a licensee, a licensor wishes to express and enforce certain permissions and obligations:
  - which operations
  - on which entities
  - the licensee is allowed (is licensed, is permitted) to perform.
- As such a license denotes a possibly infinite set of allowable behaviours.

- We shall consider four kinds of entities:
  - (i) digital recordings of artistic and intellectual nature:
    - ∗ music, movies, readings (“audio books”), and the like,
  - (ii) patients in a hospital:
    - ∗ as represented also by their patient medical records,
  - (iii) documents related to public government:
    - ∗ citizen petitions, law drafts, laws, administrative forms, letters between state and local government adminsitrators and between these and citizens, court verdicts, etc., and
  - (iv) bus timetables,
    - ∗ as part of contracts for a company to provide bus servises.

- The *permissions* and *obligations* issues are:
  - (i) for the owner (agent) of some intellectual property to be paid (i.e., an *obligation*) by users when they perform *permitted* operations (rendering, copying, editing, sub-licensing) on their works;
  - (ii) for the patient to be professionally treated — by medical staff who are basically *obliged* to try to cure the patient;
  - (iii) for public administrators and citizens to enjoy good governance: transparency in law making (national parliaments and local prefectures and city councils), in law enforcement (i.e., the daily administration of laws), and law interpretation (the judiciary) — by agents who are basically *obliged* to produce certain documents while being *permitted* to consult (i.e., read, perhaps copy) other documents;
  - (iv) for citizens to enjoy timely and reliable bus services and the

local government to secure adequate price-performance standards.

Example 60 – A Health Care License Language

- Citizens
  - go to hospitals
  - in order to be treated for some calamity (disease or other),
  - and by doing so these citizens become patients.
- At hospitals patients, in a sense, issue a request to be treated with the aim of full or partial restitution.
- This request is directed at medical staff, that is,
  - the patient authorises medical staff to perform a set of actions upon the patient.
  - One could claim, as we shall, that the patient issues a license.

⊕ *Patients and Patient Medical Records* ⊕

- So patients and their attendant patient medical records (PMRs) are the main entities, the "works" of this domain.
- We shall treat them synonymously: PMRs as surrogates for patients.
- Typical actions on patients — and hence on PMRs — involve
  - admitting patients,
  - interviewing patients,
  - analysing patients,
  - diagnosing patients,
  - planning treatment for patients,
  - actually treating patients, and,
  - under normal circumstance, to finally release patients.

⊕ *Medical Staff* ⊕

- Medical staff may request ('refer' to)
  - other medical staff to perform some of these actions.
  - One can conceive of describing action sequences (and 'referrals') in the form of hospitalisation (not treatment) plans.
  - We shall call such scripts for licenses.

⊕ *Professional Health Care* ⊕

- The issue is now,
  - given that we record these licenses,
  - their being issued and being honoured,
  - whether the handling of patients at hospitals
    - ∗ follow,
    - ∗ or does not follow

    properly issued licenses.

⊕ *A Notion of License Execution State* ⊕

- In the context of the Artistic License Language licensees could basically perform licensed actions in any sequence and as often as they so desired.
  - There were, of course, some obvious constraints.
    - ∗ Operations on local works could not be done before these had been created — say by copying.
    - ∗ Editing could only be done on local works and hence required a prior action of, for example, copying a licensed work.
- In the context of hospital health care most of the actions can only be performed if the patient has reached a suitable state in the hospitalisation.
- We refer to Fig. 15 on the following page for an idealised hospitalisation plan.

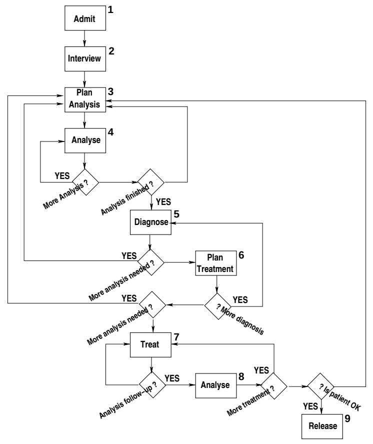


Figure 15: An example hospitalisation plan. States: {1,2,3,4,5,6,7,8,9}

- We therefore suggest
  - to join to the licensed commands
  - an indicator which prescribe the (set of) state(s) of the hospitalisation plan in which the command action may be performed.
- Two or more medical staff may now be licensed
  - to perform different (or even same !) actions
  - in same or different states.
  - If licensed to perform same action(s) in same state(s) —
  - well that may be "bad license programming" if and only if it is bad medical practice !
- One cannot design a language and prevent it being misused!

⊕ *The License Language* ⊕

- The syntax has two parts.
  - One for licenses being issued by licensors.
  - And one for the actions that licensees may wish to perform.

type

0. Ln, Mn, Pn
1. License = Ln × Lic
2. Lic == mkLic(staff1:Mn,mandate:ML,pat:Pn)
3. ML == mkML(staff2:Mn,to_perform_acts:CoL-**set**)
4 CoL = Cmd | ML | Alt
5. Cmd == mkCmd(σs:Σ-**set**,stmt:Stmt)
6 Alt == mkAlt(cmds:Cmd-**set**)
7. Stmt = **admit** | **interview** | **plan-analysis** | **do-analysis** | **diagnose** | **plan-treatment** | **treat** | **transfer** | **release**

- The above syntax is correct RSL.
- But it is decorated!
- The subtypes {|**boldface keyword**|} are inserted for readability.
- (0.) Licenses, medical staff and patients have names.
- (1.) Licenses further consist of license bodies (Lic).
- (2.) A license body names the licensee (Mn), the patient (Pn), and,
- (3.) through the "mandated" licence part (ML), it names the licensor (Mn) and which set of commands (C) or (o) implicit licenses (L, for CoL) the licensor is mandated to issue.
- (4.) An explicit command or licensing (CoL) is either a command (Cmd), or a sub-license (ML) or an alternative.

- (5.) A command (Cmd) is a state-labelled statement.
- (3.) A sub-license just states the command set that the sub-license licenses.
  - As for the Artistic License Language the licensee
  - chooses an appropriate subset of commands.
  - The context "inherits" the name of the patient.
  - But the sub-licensee is explicitly mandated in the license!
- (6.) An alternative is also just a set of commands.
  - The meaning is that
    - ∗ either the licensee choose to perform the designated actions
    - ∗ or, as for ML, but now freely choosing the sub-licensee,
    - ∗ the licensee (now new licensor) chooses to confer actions to other staff.

- (7.) A statement is either
  - an admit,
  - an interview,
  - a plan analysis,
  - an analysis,
  - a diagnose,
  - a plan treatment,
  - a treatment,
  - a transfer, or
  - a release

  directive
- Information given in the patient medical report
  - for the designated state
  - inform medical staff as to the details
  - of analysis, what to base a diagnosis on, of treatment, etc.

8. Action = Ln × Act
9. Act = Stmt | SubLic
10. SubLic = mkSubLic(sublicensee:Ln,license:ML)

- (8.) Each action actually attempted by a medical staff refers to the license, and hence the patient name.
- (9.) Actions are either of
  - an admit,
  - an interview,
  - a plan analysis,
  - an analysis,
  - a diagnose,
  - a plan treatment,
  - a treatment,
  - a transfer, or
  - a release

  actions.

- Each individual action is only allowed in a state σ
  - if the action directive appears in the named license
  - and the patient (medical record) designates state σ.
- (10.) Or an action can be a sub-licensing action.
  - Either the sub-licensing action that the licensee is attempting is explicitly mandated by the license (4. ML),
  - or is an alternative one thus implicitly mandated (6.).
  - The full sub-license, as defined in (1.–3.) is compiled from contextual information.

Example 61 – A Public Administration License Language

⊕ *The Three Branches of Government* ⊕

- By public government we shall,
  - following Charles de Secondat, baron de Montesquieu (1689–1755),
  - understand a composition of three powers:
    * the law-making (legislative),
    * the law-enforcing and
    * the law-interpreting

    parts of public government.
- Typically
  - national parliament and local (province and city) councils are part of law-making government,
  - law-enforcing government is called the executive (the administration),

– and law-interpreting government is called the judiciary [system] (including lawyers etc.).

⊕ *Documents* ⊕

- A crucial means of expressing public administration is through *documents*.
- We shall therefore provide a brief domain analysis of a concept of documents.
- (This document domain description also applies
  – to patient medical records and,
  – by some "light" interpretation, also to artistic works —

  insofar as they also are documents.)

- Documents are
  – *created*,
  – *edited* and
  – *read*;
- and documents can be
  – *copied*,
  – *distributed*,
  – the subject of *calculations* (interpretations) and be
  – *shared* and
  – *shredded*

  .

⊕ *Document Attributes* ⊕

- With documents one can associate, as attributes of documents, the *actors* who
  – created,
  – edited,
  – read,
  – copied,
  – distributed
    ∗ (to whom distributed),
  – shared,
  – performed calculations and
  – shredded

  documents.
- With these operations on documents,
- and hence as attributes of documents one can, again conceptually,
- associate the

– *location* and
– *time*

of these operations.

⊕ *Actor Attributes and Licenses* ⊕

- With actors (whether agents of public government or citizens)
  - one can associate the *authority* (i.e., the *rights*)
  - these actors have with respect to performing actions on documents.
- We now intend to express these *authorisations as licenses*.

⊕ *Document Tracing* ⊕

- An issue of public government is
  - whether citizens and agents of public government act in accordance with the laws —
  - with actions and laws reflected in documents
  - such that the action documents enables a trace from the actions to the laws "governing" these actions.
- We shall therefore assume that every document can be traced
  - back to its law-origin
  - as well as to all the documents any one document-creation or -editing was based on.

⊕ *A Document License Language* ⊕

- The syntax has two parts.
  - One for licenses being issued by licensors.
  - And one for the actions that licensees may wish to perform.

```
type
0. Ln, An, Cfn
1. L        == Grant | Extend | Restrict | Withdraw
2. Grant    == mkG(license:Ln,licensor:An,granted_ops:Op-set,licensee:An
3. Extend   == mkE(licensor:An,licensee:An,license:Ln,with_ops:Op-set)
4. Restrict == mkR(licensor:An,licensee:An,license:Ln,to_ops:Op-set)
5. Withdraw == mkW(licensor:An,licensee:An,license:Ln)
6. Op       == Crea|Edit|Read|Copy|Licn|Shar|Rvok|Rlea|Rtur|Calc|Shrd
```

```
type
7.   Dn, DCn, UDI
8.   Crea  == mkCr(dn:Dn,doc_class:DCn,based_on:UDI-set)
9.   Edit  == mkEd(doc:UDI,based_on:UDI-set)
10.  Read  == mkRd(doc:UDI)
11.  Copy  == mkCp(doc:UDI)
12a. Licn  == mkLi(kind:LiTy)
12b. LiTy  == grant | extend | restrict | withdraw
13.  Shar  == mkSh(doc:UDI,with:An-set)
14.  Rvok  == mkRv(doc:UDI,from:An-set)
15.  Rlea  == mkRl(dn:Dn)
16.  Rtur  == mkRt(dn:Dn)
17.  Calc  == mkCa(fcts:CFn-set,docs:UDI-set)
18.  Shrd  == mkSh(doc:UDI)
```

- (0.) The are names of licenses (Ln), actors (An), documents (UDI), document classes (DCn) and calculation functions (Cfn).
- (1.) There are four kinds of licenses: granting, extending, restricting and withdrawing.
- (2.) Actors (licensors) grant licenses to other actors (licensees).
  - An actor is constrained to always grant distinctly named licenses.
  - No two actors grant identically named licenses.
  - A set of operations on (named) documents are granted.

- (3.–5.) Actors who have issued named licenses may extend, restrict or withdraw the license rights (wrt. operations, or fully).
- (6.) There are nine kinds of operation authorisations. Some of the next explications also explain parts of some of the corresponding actions (see (16.–24.).
- (7.) There are names of documents (Dn), names of classes of documents (DCn), and there are unique document identifiers (UDI).

- (8.) **Creation** results in an initially void document which is
  - not necessarily uniquely named (dn:Dn) (but that name is uniquely associated with the unique document identifier created when the document is created)
  - typed by a document class name (dcn:DCn) and possibly
  - based on one or more identified documents (over which the licensee (at least) has reading rights).
  - We can presently omit consideration of the document class concept.
  - "based on" means that the initially void document contains references to those (zero, one or more) documents.
  - The "based on" documents are moved from licensor to licensee.

- (9.) **Editing** a document
  - may be based on “inspiration” from, that is, with reference to a number of other documents (over which the licensee (at least) has reading rights).
  - What this “be based on” means is simply that the edited document contains those references. (They can therefore be traced.)
  - The “based on” documents are moved from licensor to licensee
    * if not already so moved as the result of the specification of other authorised actions.

- (10.) **Reading** a document
  - only changes its “having been read” status.
  - The read document, if not the result of a copy, is moved from licensor to licensee — if not already so moved as the result of the specification of other authorised actions.

- (11.) **Copying** a document
  - increases the document population by exactly one document.
  - All previously existing documents remain unchanged except that the document which served as a master for the copy has been so marked.
  - The copied document is like the master document except that the copied document is marked to be a copy.
  - The master document, if not the result of a create or copy, is moved from licensor to licensee
    * if not already so moved as the result of the specification of other authorised actions.

- (12a.) A licensee can **sub-license** (sL) certain operations to be performed by other actors.
- (12b.) The granting, extending, restricting or withdrawing permissions,
  - cannot name a license (the user has to do that),
  - do not need to refer to the licensor (the licensee issuing the sub-license),
  - and leaves it open to the licensor to freely choose a licensee.
  - The licensor (the licensee issuing the sub-license) must choose a unique license name.

- (13.) A document can be **shared**
  - between two or more actors.
  - One of these is the licensee, the others are implicitly given read authorisations.
  - (One could think of extending, instead the licensing actions with a **shared** attribute.)
  - The shared document, if not the result of a create and edit or copy, is moved from licensor to licensee — if not already so moved as the result of the specification of other authorised actions.
  - Sharing a document does not move nor copy it.

- (14.) Sharing documents can be **revoked**. That is, the reading rights are removed.
- (15.) The **release** operation:
  - if a licensor has authorised a licensee to create a document
  - (and that document, when created got the unique document identifier udi:UDI)
  - then that licensee can **release** the created, and possibly edited document (by that identification)
  - to the licensor, say, for comments.
  - The licensor thus obtains the master copy.

- (16.) The **return** operation:
  - if a licensor has authorised a licensee to create a document
  - (and that document, when created got the unique document identifier udi:UDI)
  - then that licensee can **return** the created, and possibly edited document (by that identification)
  - to the licensor — "for good"!
  - The licensee relinquishes all control over that document.

- (17.) Two or more documents can be subjected to any one of a set of permitted **calculation** functions.
  - These documents, if not the result of a creates and edits or copies, are moved from licensor to licensee —
  - if not already so moved as the result of the specification of other authorised actions.
  - Observe that there can be many calculation permissions, over overlapping documents and functions.
- (18.) A document can be **shredded**.
  - It seems pointless to shred a document if that was the only right granted wrt. document.

17. Action = Ln × Clause
18. Clause = Cre | Edt | Rea | Cop | Lic | Sha | Rvk | Rel | Ret | Cal | Shr
19. Cre == mkCre(dcn:DCn,based_on_docs:UID-**set**)
20. Edt == mkEdt(uid:UID,based_on_docs:UID-**set**)
21. Rea == mkRea(uid:UID)
22. Cop == mkCop(uid:UID)
23. Lic == mkLic(license:L)
24. Sha == mkSha(uid:UID,with:An-**set**)
25. Rvk == mkRvk(uid:UID,from:An-**set**)
25. Rev == mkRev(uid:UID,from:An-**set**)
26. Rel == mkRel(dn:Dn,uid:UID)
27. Ret == mkRet(dn:Dn,uid:UID)
28. Cal == mkCal(fct:Cfn,over_docs:UID-**set**)
29. Shr == mkShr(uid:UID)

- A clause elaborates to a state change and usually some value.
- The value yielded by elaboration of the above
  - create, copy, and calculation clauses
  - are **unique document identifiers**.
  - These are chosen by the “system”.

- (17.) Actions are **tagged** by the name of the license
  - with respect to which their authorisation and document names has to be checked.
  - No action can be performed by a licensee
  - unless it is so authorised by the named license,
  - both as concerns the operation (create, edit, read, copy, license, share, revoke, calculate and shred)
  - and the documents actually named in the action.
  - They must have been mentioned in the license,
  - or, created or copies of downloaded (and possibly edited) documents or copies of these — in which cases operations are inherited.

- (19.) A licensee may **create** documents if so licensed —
  - and obtains all operation authorisations to this document.
- (20.) A licensee may **edit** “downloaded” (edited and/or copied) or created documents.
- (21.) A licensee may **read** “downloaded” (edited and/or copied) or created and edited documents.
- (22.) A licensee may (conditionally) **copy** “downloaded” (edited and/or copied) or created and edited documents.
  - The licensee decides which name to give the new document, i.e., the copy.
  - All rights of the master are inherited to the copy.

- (23.) A licensee may **issue licenses**
  - of the kind permitted.
  - The licensee decides whether to do so or not.
  - The licensee decides
    - to whom,
    - over which, if any, documents,
    - and for which operations.
  - The licensee looks after a proper ordering of licensing commands:
    - first grant,
    - then sequences of zero, one or more either extensions or restrictions,
    - and finally, perhaps, a withdrawal.

- (24.) A “downloaded” (possibly edited or copied) document may (conditionally) be **shared** with one or more other actors.
  - Sharing, in a digital world, for example,
  - means that any edits done after the opening of the sharing session,
  - can be read by all so-granted other actors.

- (25.) Sharing may (conditionally) be **revoked**, partially or fully, that is, wrt. original “sharers”.
- (26.) A document may be **released**.
  - It means that the licensor who originally requested
  - a document (named dn:Dn) to be created
  - now is being able to see the results —
  - and is expected to comment on this document
  - and eventually to re-license the licensee to further work.

- (27.) A document may be **returned**.
  - It means that the licensor who originally requested
  - a document (named dn:Dn) to be created
  - is now given back the full control over this document.
  - The licensee will no longer operate on it.

- (28.) A license may (conditionally) apply any of a licensed set of **calculation functions**
  - to "downloaded" (edited, copied, etc.) documents,
  - or can (unconditionally) apply any of a licensed set of calculation functions
  - to created (etc.) documents.
  - The result of a calculation is a document.
  - The licensee obtains all operation authorisations to this document (— as for created documents).
- (29.) A license may (conditionally) **shred** a "downloaded" (etc.) document.

Example 62 – A Bus Services Contract Language

- In a number of steps
  - ('A Synopsis',
  - 'A Pragmatics and Semantics Analysis', and
  - 'Contracted Operations, An Overview')
- we arrive at a sound basis from which to formulate the narrative.
  - We shall, however, forego such a detailed narrative.
  - Instead we leave that detailed narrative to the student.
  - (The detailed narrative can be "derived" from the formalisation.)

⊕ *A Synopsis* ⊕

- Contracts obligate transport companies to deliver bus traffic according to a timetable.
- The timetable is part of the contract.
- A contractor may sub-contract (other) transport companies to deliver bus traffic according to timetables that are sub-parts of their own timetable.
- Contractors are either public transport authorities or contracted transport companies.
- Contracted transport companies may cancel a subset of bus rides provided the total amount of cancellations per 24 hours for each bus line does not exceed a contracted upper limit.
- The cancellation rights are spelled out in the contract.
- A sub-contractor cannot increase a contracted upper limit for can-

cellations above what the sub-contractor was told (in its contract) by its contractor.

- Etcetera.

⊕ A Pragmatics and Semantics Analysis ⊕

- The "works" of the bus transport contracts are two:
  - the timetables and, implicitly,
  - the designated (and obligated) bus traffic.
- A bus timetable appears to define one or more bus lines,
  - with each bus line giving rise to one or more bus rides.
- Nothing is (otherwise) said about regularity of bus rides.
- It appears that bus ride cancellations must be reported back to the contractor.
  - And we assume that cancellations by a sub-contractor is further reported back also to the sub-contractor's contractor.
  - Hence eventually that the public transport authority is notified.

- Nothing is said, in the contracts, such as we shall model them,
  - about passenger fees for bus rides
  - nor of percentages of profits (i.e., royalties) to be paid back from a sub-contractor to the contractor.
- So we shall not bother, in this example, about transport costs nor transport subsidies.
- The opposite of cancellations appears to be 'insertion' of extra bus rides,
  - that is, bus rides not listed in the time table,
  - but, perhaps, mandated by special events
  - We assume that such insertions must also be reported back to the contractor.

- We assume concepts of acceptable and unacceptable bus ride delays.
  - Details of delay acceptability may be given in contracts,
    - ∗ but we ignore further descriptions of delay acceptability.
    - ∗ but assume that unacceptable bus ride delays are also to be (iteratively) reported back to contractors.
- We finally assume that sub-contractors cannot (otherwise) change timetables.
  - (A timetable change can only occur after, or at, the expiration of a license.)
- Thus we find that contracts have definite period of validity.
  - (Expired contracts may be replaced by new contracts, possibly with new timetables.)

⊕ Contracted Operations, An Overview ⊕

- So these are the operations that are allowed by a contractor according to a contract:
  - (i) *start:* to perform, i.e., to start, a bus ride (obligated);
  - (ii) *cancel:* to cancel a bus ride (allowed, with restrictions);
  - (iii) *insert:* to insert a bus ride; and
  - (iv) *subcontract:* to sub-contract part or all of a contract.

⊕ *Syntax* ⊕

- We treat separately,
  - the syntax of contracts (for a schematised example see Slide 628) and
  - the syntax of the actions implied by contracts (for schematised examples see Slide 632).

Contracts

- An example contract can be 'schematised':

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

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

167. contracts, contractors and sub-contractors have unique identifiers CId, CNm, CNm.
168. A contract has a unique identification, names the contractor and the sub-contractor (and we assume the contractor and sub-contractor names to be distinct). A contract also specifies a contract body.
169. A contract body stipulates a timetable and the set of operations that are mandated or allowed by the contractor.
170. An Operation is either a `"start"` (i.e., start a bus ride), a bus ride `"cancel"`lation, a bus ride `"insert"`, or a `"subcontract"`ing operation.

type
167. CId, CNm
168. Contract = CId × CNm × CNm × Body
169. Body = Op-**set** × TT
170. Op == "`start`" | "`cancel`" | "`insert`" | "`subcontract`"

An abstract example contract:

$(\mathrm{cid},\mathrm{cnm}_i,\mathrm{cnm}_j,(\{\texttt{"start"},\texttt{"cancel"},\texttt{"insert"},\texttt{"sublicense"}\},\mathrm{tt}))$

Actions

- Concrete example actions can be schematised:

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

- The schematised license (Slide 84) shown earlier is almost like an action; here is the action form:

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

K.2.0.1. Actions

- All actions are being performed by a sub-contractor in a context which defines
  - that sub-contractor cnm,
  - the relevant net, say n,
  - the base contract, referred here to by cid (from which this is a sublicense), and
  - a timetable tt of which tt′ is a subset.
- contract name cnm′ is new and is to be unique.
- The subcontracting action can (thus) be simply transformed into a contract as shown on Slide 84.

```
type
  Action = CNm × CId × (SubCon | SmpAct) × Time
  SmpAct = Start | Cancel | Insert
  Conduct == mkSta(s_blid:BLId,s_bid:BId)
  Cancel == mkCan(s_blid:BLId,s_bid:BId)
  Insert = mkIns(s_blid:BLId,s_bid:BId)
  SubCon == mkCon(s_cid:CId,s_cnm:CNm,s_body:(s_ops:Op-set,s_tt:TT))

examples:
  (a) (cnm,cid,mkSta(blid,id),t)
  (b) (cnm,cid,mkCan(blid,id),t)
  (c) (cnm,cid,mkIns(blid,id),t)
  (d) (cnm,cid,mkCon(cid′,({"conduct","cancel","insert","sublicense"},tt′),t

where: cid′ = generate_CId(cid,cnm,t)      See Item/Line 173 on page 638
```

- We observe that
  - the essential information given in the start, cancel and insert action prescriptions is the same;
  - and that the RSL record-constructors (mkSta, mkCan, mkIns) make them distinct.

Uniqueness and Traceability of Contract Identifications

171. There is a "root" contract name, rcid.
172. There is a "root" contractor name, rcnm.

value
171 rcid:CId
172 rcnm:CNm

- All other contract names are derived from the root name.
- Any contractor can at most generate one contract name per time unit.
- Any, but the root, sub-contractor obtains contracts from other sub-contractors, i.e., the contractor. Eventually all sub-contractors, hence contract identifications can be referred back to the root contractor.

173. Such a contract name generator is a function which given a contract identifier, a sub-contractor name and the time at which the new contract identifier is generated, yields the unique new contract identifier.
174. From any but the root contract identifier one can observe the contract identifier, the sub-contractor name and the time that "went into" its creation.

value
173 gen_CId: CId × CNm × Time → CId
174 obs_CId: CId $\xrightarrow{\sim}$ CIdL [**pre** obs_CId(cid):cid≠rcid]
174 obs_CNm: CId $\xrightarrow{\sim}$ CNm [**pre** obs_CNm(cid):cid≠rcid]
174 obs_Time: CId $\xrightarrow{\sim}$ Time [**pre** obs_Time(cid):cid≠rcid]

175. All contract names are unique.

axiom
175 ∀ cid,cid′:CId•cid≠cid′⇒
175 obs_CId(cid)≠obs_CId(cid′) ∨ obs_CNm(cid)≠obs_CNm(cid′)
175 ∨ obs_LicNm(cid)=obs_CId(cid′)∧obs_CNm(cid)=obs_CNm(cid′)
175 ⇒ obs_Time(cid)≠obs_Time(cid′)

176. Thus a contract name defines a trace of license name, sub-contractor name and time triple, “all the way back” to “creation”.

```
type
  CIdCNmTTrace = TraceTriple*
  TraceTriple == mkTrTr(CId,CNm,s_t:Time)
value
176  contract_trace: CId → LCIdCNmTTrace
176  contract_trace(cid) ≡
176    case cid of
176      rcid → ⟨⟩,
176      _ → contract_trace(obs_LicNm(cid))⌢⟨obs_TraceTriple(cid)⟩
176    end

176  obs_TraceTriple: CId → TraceTriple
176  obs_TraceTriple(cid) ≡
```

```
176    mkTrTr(obs_CId(cid),obs_CNm(cid),obs_Time(cid))
```

- The trace is generated in the chronological order: most recent contract name generation times last.
- Well, there is a theorem to be proven once we have outlined the full formal model of this contract language:
- namely that time entries in contract name traces increase with increasing indices.

```
theorem
  ∀ licn:LicNm ·
    ∀ trace:LicNmLeeNmTimeTrace · trace ∈ license_trace(licn) ⇒
       ∀ i:Nat · {i,i+1}⊆inds trace ⇒ s_t(trace(i))<s_t(trace(i+1))
```

⊕ *Execution State* ⊕

Local and Global States

- Each sub-contractor has an own local state and has access to a global state.
- All sub-contractors access the same global state.
- The global state is the bus traffic on the net.
- There is, in addition, a notion of running-state. It is a meta-state notion.
  - The running state “is made up” from the fact that
  - there are n sub-contractors, each communicating, as contractors,
  - over channels with other sub-contractors.
- The global state is distinct from sub-contractor to sub-contractor – no sharing of local states between sub-contractors.

- We now examine, in some detail, what the states consist of.

Global State

- The net is part of the global state (and of bus traffics).
- We consider just the bus traffic.

177. Bus traffic is a modelled as a discrete function from densely positioned time points to a pair of the (possibly dynamically changing) net and the position of busses. Bus positions map bus numbers to the physical entity of busses and their position.
178. A bus is positioned either
179. at a hub (coming from some link heading for some link), or
180. on a link, some fraction of the distance from a hub towards a hub, or
181. at a bus stop, some fraction of the distance from a hub towards a hub.

```
type
119.  BusStop == mkBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)

177.  BusTraffic = T →m (N × (BusNo →m (Bus × BPos)))
178.  BPos = atHub | onLnk | atBS
179.  atHub == mkAtHub(s_fl:LI,s_hi:HI,s_tl:LI)
180.  onLnk == mkOnLnk(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
181.  atBSt == mkAtBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
      Frac = {|f:Real·0<f<1|}
```

- We shall consider BusTraffic (with its Net) to reflect the global state.

Local Sub-contractor Contract States: Semantic Types

- A sub-contractor state contains, as a state component, the zero, one or more contracts
  - that the sub-contractor has received and
  - that the sub-contractor has sublicensed.

```
type
  Body = Op-set × TT
  LicΣ = RcvLicΣ×SubLicΣ×LorBusΣ
  RcvLicΣ = LorNm →m (LicNm →m (Body×TT))
  SubLicΣ = LeeNm →m (LicNm →m Body)
  LorBusΣ ... [ see "Local sub-contractor Bus States: Semantic T
```

- (Recall that LorNm and LeeNm are the same.)

Local Sub-contractor Bus States: Semantic Types

- The sub-contractor state further contains a bus status state component which records
  - which buses are free, FreeBusΣ, that is, available for dispatch, and where "garaged",
  - which are in active use, ActvBusΣ, and on which bus ride, and a bus history for that bus ride,
  - and histories of all past bus rides, BusHistΣ.
  - A trace of a bus ride is a list of zero, one or more pairs of times and bus stops.
  - A bus history, BusHistory, associates a bus trace to a quadruple of bus line identifiers, bus ride identifiers, contract names and sub-contractor name.

```
type
  BusNo
  BusΣ = FreeBusesΣ × ActvBusesΣ × BusHistsΣ
  FreeBusesΣ = BusStop m→ BusNo-set
  ActvBusesΣ = BusNo m→ BusInfo
  BusInfo = BLId×BId×LicNm×LeeNm×BusTrace
  BusHistsΣ = Bno m→ BusInfo*
  BusTrace = (Time×BusStop)*
  LorBusΣ = LeeNm m→ (LicNm m→ ((BLId×BId) m→ (BNo×BusTrace)))
```

- A bus is identified by its unique number (i.e., registration) plate (BusNo).
- The two components are modified whenever a bus is commissioned into action or returned from duty, that is, twice per bus ride.

Local Sub-contractor Bus States: Update Functions

```
value
 update_BusΣ: Bno×(T×BusStop) → ActBusΣ → ActBusΣ
 update_BusΣ(bno,(t,bs))(actσ) ≡
   let (blid,bid,licn,leen,trace) = actσ(bno) in
   actσ†[ bno↦(licn,leen,blid,bid,trace⌢⟨(t,bs)⟩) ] end
   pre bno ∈ dom actσ

 update_FreeΣ_ActΣ:
   BNo×BusStop→BusΣ→BusΣ
 update_FreeΣ_ActΣ(bno,bs)(freeσ,actvσ) ≡
   let (_,_,_,_,trace) = actσ(b) in
   let freeσ′ = freeσ†[ bs ↦ (freeσ(bs))∪{b} ] in
   (freeσ′,actσ\{b}) end end
   pre bno ∉ freeσ(bs) ∧ bno ∈ dom actσ
```

```
update_LorBusΣ:
   LorNm×LicNm×lee:LeeNm×(BLId×BId)×(BNo×Trace)
     →LorBusΣ→out {l_to_l[ leen,lorn ]|lorn:LorNm·lorn ∈ leenms\{leen}}
update_LorBusΣ(lorn,licn,leen,(blid,bid),(bno,tr))(lbσ) ≡
  l_to_l[ leenm,lornm ]!Licensor_BusHistΣMsg(bno,blid,bid,libn,leen,tr) ;
  lbσ†[ leen↦(lbσ(leen))†[ licn↦((lbσ(leen))(licn))†[ (blid,bid)↦(bno,trace)
  pre leen ∈ dom lbσ ∧ licn ∈ dom (lbσ(leen))

update_ActΣ_FreeΣ:
  LeeNm×LicNm×BusStop×(BLId×BId)→BusΣ→BusΣ×BNo
update_ActΣ_FreeΣ(leen,licn,bs,(blid,bid))(freeσ,actvσ) ≡
  let bno:Bno · bno ∈ freeσ(bs) in
  ((freeσ\{bno},actvσ ∪ [ bno↦(blid,bid,licnm,leenm,⟨⟩) ]),bno) end
  pre bs ∈ dom freeσ ∧ bno ∈ freeσ(bs) ∧ bno ∉ dom actvσ ∧ [ bs exists
```

Constant State Values

- There are a number of constant values, of various types, which characterise the “business of contract holders”. We define some of these now.

182. For simplicity we assume a constant net — constant, that is, only with respect to the set of identifiers links and hubs. These links and hubs obviously change state over time.
183. We also assume a constant set, leens, of sub-contractors. In reality sub-contractors, that is, transport companies, come and go, are established and go out of business. But assuming constancy does not materially invalidate our model. Its emphasis is on contracts and their implied actions — and these are unchanged wrt. constancy or variability of contract holders.

184. There is an initial bus traffic, tr.
185. There is an initial time, t_0, which is equal to or larger than the start of the bus traffic tr.
186. To maintain the bus traffic “spelled out”, in total, by timetable tt one needs a number of buses.
187. The various bus companies (that is, sub-contractors) each have a number of buses. Each bus, independent of ownership, has a unique (car number plate) bus number (BusNo).
These buses have distinct bus (number [registration] plate) numbers.
188. We leave it to the student to define a function which ascertain the minimum number of buses needed to implement traffic tr.

value
182. net : N,
183. leens : LeeNm-**set**,
184. tr : BusTraffic, **axiom** wf_Traffic(tr)(net)
185. t_0 : T · $t_0 \geq$ **min dom** tr,
186. min_no_of_buses : **Nat** · necessary_no_of_buses(itt),
187. busnos : BusNo-**set** · **card** busnos ≥ min_no_of_buses
188. necessary_no_of_buses: TT → **Nat**

189. To “bootstrap” the whole contract system we need a distinguished contractor, named init_leen, whose only license originates with a “ghost” contractor, named root_leen (o, for outside [the system]).
190. The initial, i.e., the distinguished, contract has a name, root_licn.
191. The initial contract can only perform the `"sublicense"` operation.
192. The initial contract has a timetable, tt.
193. The initial contract can thus be made up from the above.

value

189. $\text{root\_leen},\text{init\_ln} : \text{LeeNm} \cdot \text{root\_leen} \notin \text{leens} \wedge \text{initi\_leen} \in \text{leens},$
190. $\text{root\_licn} : \text{LicNm}$
191. $\text{iops} : \text{Op-}\mathbf{set} = \{\text{"}\mathtt{sublicense}\text{"}\},$
192. $\text{itt} : \text{TT},$
193. $\text{init\_lic}:\text{License} = (\text{root\_licn},\text{root\_leen},(\text{iops},\text{itt}),\text{init\_leen})$

Initial Sub-contractor Contract States

type

$\text{InitLic}\Sigma\text{s} = \text{LeeNm} \xrightarrow[m]{} \text{Lic}\Sigma$

value

$\text{il}\sigma\text{:Lic}\Sigma\text{=}([\ \text{init\_leen} \mapsto [\ \text{root\_leen} \mapsto [\ \text{iln} \mapsto \text{init\_lic}\ ]\ ]\ ]$
$\quad \cup [\ \text{leen} \mapsto [\,]\ |\ \text{leen:LeeNm} \cdot \text{leen} \in \text{leenms}\backslash\{\text{init\_leen}\}\ ],[\,],[\,])$

Initial Sub-contractor Bus States

194. Initially each sub-contractor possesses a number of buses.
195. No two sub-contractors share buses.
196. We assume an initial assignment of buses to bus stops of the free buses state component and for respective contracts.
197. We do not prescribe a “satisfiable and practical” such initial assignment (ibσs).
198. But we can constrain ibσs.
199. The sub-contractor names of initial assignments must match those of initial bus assignments, allbuses.
200. Active bus states must be empty.
201. No two free bus states must share buses.

202. All bus histories are void.

```
type
194. AllBuses′ = LeeNm →m BusNo-set
195. AllBuses = {|ab:AllBuses′·∀ {bs,bs′}⊆rng ab∧bns≠bns′⇒bns ∩ bns′={
196. InitBusΣs = LeeNm →m BusΣ
value
195. allbuses:Allbuses · dom allbuses = leenms ∪ {root_leen} ∧ ∪ rng allbu

196. ibσs:InitBusΣs
197. wf_InitBusΣs: InitBusΣs → Bool
198. wf_InitBusΣs(iσs) ≡
199.   dom iσs = leenms ∧
200.   ∀ (_,abσ,_):BusΣ·(_,abσ,_) ∈ rng iσs ⇒ abσ=[ ] ∧
201.   ∀ (fbiσ,abiσ),(fbjσ,abjσ):BusΣ ·
201.     {(fbiσ,abiσ),(fbjσ,abjσ)}⊆rng iσs
201.       ⇒ (fbiσ,actiσ)≠(fbjσ,actjσ)
```

```
201.         ⇒ rng fbiσ ∩ rng fbjσ = {}
202.          ∧ actiσ=[ ]=actjσ
```

⊕ *Communication Channels* ⊕

- The running state is a meta notion. It reflects the channels over which
  - contracts are issued;
  - messages about committed, cancelled and inserted bus rides are communicated, and
  - fund transfers take place.

Sub-Contractor↔Sub-Contractor Channels

- Consider each sub-contractor (same as contractor) to be modelled as a behaviour.
- Each sub-contractor (licensor) behaviour has a unique name, the LeeNm.
- Each sub-contractor can potentially communicate with every other sub-contractor.
- We model each such communication potential by a channel.
- For n sub-contractors there are thus $n \times (n-1)$ channels.

```
channel { l_to_l[ fi,ti ] | fi:LeeNm,ti:LeeNm · {fi,ti}⊆leens ∧ fi≠ti } LLMSG
type LLMSG = ...
```

Sub-Contractor↔Bus Channels

- Each sub-contractor has a set of buses. That set may vary.
- So we allow for any sub-contractor to potentially communicate with any bus.
- In reality only the buses allocated and scheduled by a sub-contractor can be "reached" by that sub-contractor.

channel { l_to_b[l,b] | l:LeeNm,b:BNo · l ∈ leens ∧ b ∈ busnos } LBMSG
type LBMSG = ...

Sub-Contractor↔Time Channels

- Whenever a sub-contractor wishes to perform a contract operation
- that sub-contractor needs know the time.
- There is just one, the global time, modelled as one behaviour: time_clock.

channel { l_to_t[l] | l:LeeNm · l ∈ leens } LTMSG
type LTMSG = ...

Bus↔Traffic Channels

- Each bus is able, at any (known) time to ascertain where in the traffic it is.
- We model bus behaviours as processes, one for each bus.
- And we model global bus traffic as a single, separate behaviour.

channel { b_to_tr[b] | b:BusNo · b ∈ busnos } LTrMSG
type BTrMSG == reqBusAndPos(s_bno:BNo,s_t:Time) | (Bus×BusPos)

Buses↔Time Channel

- Each bus needs to know what time it is.

channel { b_to_t[b] | b:BNo · b ∈ busnos } BTMSG
type BTMSG ...

⊕ *Run-time Environment* ⊕

:

- So we shall be modelling the transport contract domain as follows:
  - As for behaviours we have this to say.
    - There will be n sub-contractors. One sub-contractor will be initialised to one given license.
    - Each sub-contractor is modelled, in RSL, as a CSP-like process.
    - With each sub-contractor, l_i, there will be a number, b_i, of buses. That number may vary from sub-contractor to sub-contractor.
    - There will be b_i channels of communication between a sub-contractor and that sub-contractor's buses, for each sub-contractor.
    - There is one global process, the traffic. There is one channel of communication between a sub-contractor and the traffic. Thus there are n such channels.

- As for operations, including behaviour interactions we assume the following.
  - All operations of all processes are to be thought of as instantaneous, that is, taking nil time !
  - Most such operations are the result of channel communications
    - either just one-way notifications,
    - or inquiry requests.
  - Both the former (the one-way notifications) and the latter (inquiry requests) must not be indefinitely barred from receipt, otherwise holding up the notifier.
  - The latter (inquiry requests) should lead to rather immediate responses, thus must not lead to dead-locks.

⊕ *The System Behaviour* ⊕

- The system behaviour starts by establishing a number of

  – licenseholder – and – bus_ride

  behaviours and the single

  – time_clock – and – bus_traffic

  behaviours

```
value
 system: Unit → Unit
 system() ≡
  licenseholder(init_leen)(ilσ(init_leen),ibσ(init_leen))
  ∥ (∥ {licenseholder(leen)(ilσ(leen),ibσ(leen))
     | leen:LeeNm·leen ∈ leens\{init_leen}})
  ∥ (∥ {bus_ride(b,leen)(root_lorn,"nil")
     | leen:LeeNm,b:BusNo ·leen ∈ dom allbuses ∧ b ∈ allbuses(leen)})
  ∥ time_clock(t₀) ∥ bus_traffic(tr)
```

- The initial licenseholder behaviour states are individually initialised
  - with basically empty license states and
  - by means of the global state entity bus states.
- The initial bus behaviours need no initial state.
- Only a designated licenseholder behaviour is initialised
  - to a single, received license.

⊕ *Semantic Elaboration Functions* ⊕

The Licenseholder Behaviour

203. The licenseholder behaviour is a sequential, but internally non-deterministic behaviour.
204. It internally non-deterministically ($\lceil\rceil$) alternates between
    (a) performing the licensed operations (on the net and with buses),
    (b) receiving information about the whereabouts of these buses, and informing contractors of its (and its subsub-contractors') handling of the contracts (i.e., the bus traffic), and
    (c) negotiating new, or renewing old contracts.

203. **licenseholder**: LeeNm $\rightarrow$ (Lic$\Sigma\times$BusΣ) $\rightarrow$ **Unit**
204. **licenseholder**(leen)(licσ,busσ) $\equiv$
204.   **licenseholder**(leen)((**lic_ops**$\lceil\rceil$**bus_mon**$\lceil\rceil$**neg_licenses**)(leen)(licσ,bu

The Bus Behaviour

205. Buses ply the network following a timed bus route description.
    A timed bus route description is a list of timed bus stop visits.
206. A timed bus stop visit is a pair: a time and a bus stop.
207. Given a bus route and a bus schedule one can construct a timed bus route description.
    (a) The first result element is the first bus stop and origin departure time.
    (b) Intermediate result elements are pairs of respective intermediate schedule elements and intermediate bus route elements.
    (c) The last result element is the last bus stop and final destination arrival time.
208. Bus behaviours start with a "nil" bus route description.

type
205. TBR = TBSV*
206. TBSV = Time $\times$ BusStop
value
207. conTBR: BusRoute $\times$ BusSched $\rightarrow$ TBR
207. conTBR((dt,til,at),(bs1,bsl,bsn)) $\equiv$
207(a)) $\langle$(dt,bs1)$\rangle$
207(b)) $\frown$ $\langle$(til[i],bsl[i])|i:**Nat**·i:$\langle$1..**len** til$\rangle\rangle$
207(c)) $\frown$ $\langle$(at,bsn)$\rangle$
    pre: **len** til = **len** bsl
type
208. BRD == "`nil`" | TBR

209. The bus behaviour is here abstracted to only communicate with some contract holder, time and traffic,
210. The bus repeatedly observes the time, t, and its position, po, in the traffic.
211. There are now four case distinctions to be made.
212. If the bus is idle (and a a bus stop) then it waits for a next route, brd′ on which to engage.
213. If the bus is at the destination of its journey then it so informs its owner (i.e., the sub-contractor) and resumes being idle.
214. If the bus is ‘en route’, at a bus stop, then it so informs its owner and continues the journey.
215. In all other cases the bus continues its journey

```
value
209. bus_ride: leen:LeeNm × bno:Bno → (LicNm × BRD) →
209.     in,out l_to_b[leen,bno], in,out b_to_tr[bno], in b_to_t[bno] Unit
209. bus_ride(leen,bno)(licn,brd) ≡
210.   let t = b_to_t[bno]? in
210.   let (bus,pos) = (b_to_tr[bno]!reqBusAndPos(bno,t) ; b_to_tr[bno]?) in
211.   case (brd,pos) of
212.     (″nil″,mkAtBS(_,_,_,_)) →
212.           let (licn,brd′) = (l_to_b[leen,bno]!reqBusRid(pos);l_to_b[leen,bno]?) in
212.           bus_ride(leen,bno)(licn,brd′) end
213.     (⟨(at,pos)⟩,mkAtBS(_,_,_,_)) →
213s            l_to_b[l,b]!BusΣMsg(t,pos);
213           l_to_b[l,b]!BusHistΣMsg(licn,bno);
213           l_to_b[l,b]!FreeΣ_ActΣMsg(licn,bno) ;
213            bus_ride(leen,bno)(ilicn,″nil″),
214.     (⟨(t,pos),(t′,bs′)⟩⌢brd′,mkAtBS(_,_,_,_)) →
214s            l_to_b[l,b]!BusΣMsg(t,pos) ;
214            bus_ride(licn,bno)(⟨(t′,bs′)⟩⌢brd′),
```

```
215.      _ → bus_ride(leen,bno)(licn,brd) end end end
```

- In formula line 210 of **bus_ride** we obtained the **bus**.
- But we did not use “that” bus !
- We we may wish to record, somehow, number of passengers alighting and boarding at bus stops, bus fees paid, one way or another, etc.
- The **bus**, which is a time-dependent entity, gives us that information.
- Thus we can revise formula lines 213s and 214s:

```
Simple:  213s  l_to_b[l,b]!BusΣMsg(pos);
Revised: 213r  l_to_b[l,b]!BusΣMsg(pos,bus_info(bus));

Simple:  214s  l_to_b[l,b]!BusΣMsg(pos);
Revised: 214r  l_to_b[l,b]!BusΣMsg(pos,bus_info(bus));
```

type

```
Bus_Info = Passengers × Passengers × Cash × ...
value
  bus_info: Bus → Bus_Info
  bus_info(bus) ≡ (obs_alighted(bus),obs_boarded(bus),obs_till(bus),...)
```

The Global Time Behaviour

216. The **time_clock** is a never ending behaviour — started at some time t_0.
217. The time can be inquired at any moment by any of the licenseholder behaviours and by any of the bus behaviours.
218. At any moment the **time_clock** behaviour may not be inquired.
219. After a skip of the clock or an inquiry the **time_clock** behaviour continues, non-deterministically either maintaining the time or advancing the clock!

```
value
216. time_clock: T →
216.     in,out {l_to_t[leen] | leen:LeeNm · leen ∈ leenms}
216.     in,out {b_to_t[bno] | bno:BusNo · bno ∈ busnos}  Unit
216. time_clock:(t) ≡
218.   (skip ⌈⌉
217.   (⌈⌉{l_to_t[leen]? ; l_to_t[leen]!t | leen:LeeNm·leen ∈ leens})
217.   ⌈⌉ (⌈⌉{b_to_t[bno]? ; b_to_t[bno]!t | bno:BusNo·bno ∈ busnos})) ;
219.   (time_clock:(t) ⌈⌉ time_clock(t+δ_t))
```

The Bus Traffic Behaviour

220. There is a single **bus_traffic** behaviour. It is, “mysteriously”, given a constant argument, “the” traffic, **tr**.
221. At any moment it is ready to inform of the position, **bps(b)**, of a bus, **b**, assumed to be in the traffic at time **t**.
222. The request for a bus position comes from some bus.
223. The bus positions are part of the traffic at time **t**.
224. The **bus_traffic** behaviour, after informing of a bus position reverts to “itself”.

```
value
220. bus_traffic: TR → in,out {b_to_tr[ bno ]|bno:BusNo•bno ∈ busnos} U
220. bus_traffic(tr) ≡
222.   ⌷ { let reqBusAndPos(bno,time) = b_to_tr[ b ]? in assert b=bno
221.       if time ∉ dom tr then chaos else
223.       let (_,bps) = tr(t) in
221.       if bno ∉ dom tr(t) then chaos else
221.       b_to_tr[ bno ]!bps(bno) end end end end | b:BusNo•b ∈ busnos}
224.   bus_traffic(tr)
```

License Operations

225. The lic_ops function models the contract holder choosing between and performing licensed operations.
226. To perform any licensed operation the sub-contractor needs to know the time and
227. must choose amongst the four kinds of operations that are licensed.
    - The choice function, which we do not define, makes a basically non-deterministic choice among licensed alternatives.
    - The choice yields the contract number of a received contract and,
    - based on its set of licensed operations,
    - it yields either a simple action or a sub-contracting action.
228. Thus there is a case distinction amongst four alternatives.

229. This case distinction is expressed in the four lines identified by: 229.
230. All the auxiliary functions, besides the action arguments, require the same state arguments.

```
value
225. lic_ops: LeeNm → (LicΣ×BusΣ) → (LicΣ×BusΣ)
225. lic_ops(leen)(licσ,busσ) ≡
226.   let t = (time_channel(leen)!req_Time;time_channel(leen)?) in
227.   let (licn,act) = choice(licσ)(busσ)(t) in
228.   (case act of
229.      mkCon(blid,bid)  → cndct(licn,leenm,t,act),
229.      mkCan(blid,bid)  → cancl(licn,leenm,t,act),
229.      mkIns(blid,bid)  → insrt(licn,leenm,t,act),
229.      mkLic(leenm′,bo) → sublic(licn,leenm,t,act) end)(licσ,busσ) end e

     cndct,cancl,insert: SmpAct→(LicΣ×BusΣ)→(LicΣ×BusΣ)
     sublic: SubLic→(LicΣ×BusΣ)→(LicΣ×BusΣ)
```

Bus Monitoring

- Like for the **bus_ride** behaviour we decompose the **bus_mon**itoring behaviour into two behaviours.
  - The **local_bus_mon**itoring behaviour monitors the buses that are commissioned by the sub-contractor.
  - The **licensor_bus_mon**itoring behaviour monitors the buses that are commissioned by sub-contractors sub-contractd by the contractor.

```
value
 bus_mon: l:LeeNm → (LicΣ×BusΣ)
        → in {l_to_b[l,b]|b:BNo•b ∈ allbuses(l)} (LicΣ×BusΣ)
 bus_mon(l)(licσ,busσ) ≡
   local_bus_mon(l)(licσ,busσ) ⌈⌉ licensor_bus_mon(l)(licσ,busσ)
```

231. The **local_bus_mon**itoring function models all the interaction between a contract holder and its despatched buses.
232. We show only the communications from buses to contract holders.
233.
234.
235.
236.
237.
238.
239.
240.
241.

```
231. local_bus_mon: leen:LeeNm → (LicΣ×BusΣ)
232.        → in {l_to_b[leen,b]|b:BNo•b ∈ allbuses(l)} (LicΣ×BusΣ)
231. local_bus_mon(leen)(licσ:(rlσ,slσ,lbσ),busσ:(fbσ,abσ)) ≡
233.    let (bno,msg) = ⌈⌉{(b,l_to_b[l,b]?)|b:BNo•b ∈ allbuses(leen)} in
237.    let (blid,bid,licn,lorn,trace) = abσ(bno) in
234.    case msg of
235.      BusΣMsg(t,bs) →
239.        let abσ′ = update_BusΣ(bno)(licn,leen,blid,bid)(t,bs)(abσ) in
239.        (licσ,(fbσ,abσ′,histσ)) end,
241.      BusHistΣMsg(licn,bno) →
241.        let lbσ′ =
241.          update_LorBusΣ(obs_LorNm(licn),licn,leen,(blid,bid),(b,trace))
241.        l_to_l[leen,obs_LorNm(licn)]!Licensor_BusHistΣMsg(licn,leen,bn
241.        ((rlσ,slσ,lbσ′),busσ) end
240.      FreeΣ_ActΣMsg(licn,bno) →
```

```
241.        let (fbσ′,abσ′) = update_FreeΣ_ActΣ(bno,bs)(fbσ,abσ) in
241.        (licσ,(fbσ′,abσ′)) end
241.    end end end
```

242.

243.

244.

245.

246.

```
242. licensor_bus_mon: lorn:LorNm → (LicΣ×BusΣ)
242.       → in {l_to_l[lorn,leen]|leen:LeeNm•leen ∈ leenms\{lorn}} (LicΣ×
242. licensor_bus_mon(lorn)(licσ,busσ) ≡
242.    let (rlσ,slσ,lbhσ) = licσ in
242.    let (leen,Licensor_BusHistΣMsg(licn,leen″,bno,blid,bid,tr))
                       = [] {(leen′,l_to_l[lorn,leen′]?)|leen′:LeeNm•leen′ ∈ leenms\{l
242.    let lbhσ′ =
242.       update_BusHistΣ(obs_LorNm(licn),licn,leen″,(blid,bid),(bno,trace
242.    l_to_l[leenm,obs_LorNm(licnm)]!Licensor_BusHistΣMsg(b,blid,bid,l
242.    ((rlσ,slσ,lbhσ′),busσ)
242.    end end end
```

License Negotiation

247.

248.

249.

250.

251.

252.

253.

254.

255.

256.

257.

258.

247.
248.
249.
250.
251.
252.
253.
254.
255.
256.
257.

The Conduct Bus Ride Action

259. The conduct bus ride action prescribed by (ln,mkCon(bli,bi,t′) takes place in a context and shall have the following effect:
   (a) The action is performed by contractor li and at time t. This is known from the context.
   (b) First it is checked that the timetable in the contract named ln does indeed provide a journey, j, indexed by bli and (then) bi, and that that journey starts (approximately) at time t′ which is the same as or later than t.
   (c) Being so the action results in the contractor, whose name is "embedded" in ln, receiving notification of the bus ride commitment.

   (d) Then a bus, selected from a pool of available buses at the bust stop of origin of journey j, is given j as its journey script, whereupon that bus, as a behaviour separate from that of sub-contractor li, commences its ride.
   (e) The bus is to report back to sub-contractor li the times at which it stops at en route bus stops as well as the number (and kind) of passengers alighting and boarding the bus at these stops.
   (f) Finally the bus reaches its destination, as prescribed in j, and this is reported back to sub-contractor li.
   (g) Finally sub-contractor li, upon receiving this 'end-of-journey' notification, records the bus as no longer in actions but available at the destination bus stop.

259.
259(a))
259(b))
259(c))
259(d))
259(e))
259(f))
259(g))

The Cancel Bus Ride Action

260. The cancel bus ride action prescribed by (ln,mkCan(bli,bi,t′) takes place in a context and shall have the following effect:
    (a) The action is performed by contractor li and at time t. This is known from the context.
    (b) First a check like that prescribed in Item 259(b)) is performed.
    (c) If the check is OK, then the action results in the contractor, whose name is "embedded" in ln, receiving notification of the bus ride cancellation.
    That's all !

260.
260(a))
260(b))
260(c))

The Insert Bus Ride Action

261. The insert bus ride action prescribed by (ln,mkIns(bli,bi,t′) takes place in a context and shall have the following effect:
    (a) The action is performed by contractor li and at time t. This is known from the context.
    (b) First a check like that prescribed in Item 259(b)) is performed.
    (c) If the check is OK, then the action results in the contractor, whose name is "embedded" in ln, receiving notification of the new bus ride commitment.
    (d) The rest of the effect is like that prescribed in Items 259(d))–259(g)).

261.
261(a))
261(b))
261(c))
261(d))

The Contracting Action

262. The subcontracting action prescribed by (ln,mkLic(li′,(pe′,ops′,tt′))) takes place in a context and shall have the following effect:
   (a) The action is performed by contractor li and at time t. This is known from the context.
   (b) First it is checked that timetable tt is a subset of the timetable contained in, and that the operations ops are a subset of those granted by, the contract named ln.
   (c) Being so the action gives rise to a contract of the form (ln′,li,(pe′,ops′,tt′),li′ ln′ is a unique new contract name computed on the basis of ln, li, and t. li′ is a sub-contractor name chosen by contractor li. tt′ is a timetable chosen by contractor li. ops′ is a set of operations likewise chosen by contractor li.

   (d) This contract is communicated by contractor li to sub-contractor li′.
   (e) The receipt of that contract is recorded in the license state.
   (f) The fact that the contractor has sublicensed part (or all) of its obligation to conduct bus rides is recorded in the modified component of its received contracts.

262.
262(a))
262(b))
262(c))
262(d))
262(e))
262(f))

⊕ *Discussion* ⊕

•
•
•
•

K.3. Principles $\boxed{\text{BLANK}}$

K.4. Discussion $\boxed{\text{BLANK}}$

K.5. Research Challenges $\boxed{\text{BLANK}}$

End of Lecture 20: Domain Scripts

Start of Lecture 21: Domain Management and Organisation

L. Domain Management and Organisation

L.1. Definition

- By the **management** of an enterprise we shall understand
  - a (possibly **stratified**, see 'organisation' next) set of enterprise staff (behaviours, processes)
  - **authorised** to perform certain **functions**
  - not allowed performed by other enterprise staff
  - and where such functions involve monitoring and controlling other enterprise staff.
- By **organisation** of an enterprise we shall understand
  - the **stratification** (partitioning) of enterprise staff with
  - each **partition** endowed with a set of **authorised functions** and with
  - **communication interfaces** defined between partitions, i.e., between behaviours (processes).

L.2. An Abstraction of Management Functions

```
type
 E
value
 stra_mgt, tact_mgt, oper_mgt, wrkr, merge: $E \times E \times E \times E \rightarrow E$
 p: $E^* \rightarrow$ Bool
 mgt: $E \rightarrow E$
 mgt(e) $\equiv$
   let $e' = \text{stra\_mgt}(e,e'',e''',e'''')$,
      $e'' = \text{tact\_mgt}(e,e',e''',e'''')$,
      $e''' = \text{oper\_mgt}(e,e',e'',e'''')$,
      $e'''' = \text{wrkr}(e,e',e'',e''')$ in
   if $p(e,e'',e''',e'''')$
      then skip
      else mgt(merge($e,e',e'',e'''$))
```

```
   end end
```

L.3. Research Challenges

- We made no explicit references to such "business school of administration" "BA101" topics as 'strategic' and 'tactical' management.
- Contemplate the types of entities and signatures of functions related to executive, strategic, tactical and operational management and organisation matters given on Slide 92.
- Come up with better or other proposals, and/or attempt clear,
  - but not necessarily computable predicates
  - which (help) determine whether an operation
    - (above they are alluded to as 'stra' and 'tact')
  - is one of strategic or of tactical concern.

End of Lecture 21: Domain Management and Organisation

Start of Lecture 22: Human Behaviour

M. Domain Human Behaviour BLANK

M.1. Definitions

(M. Domain Human Behaviour BLANK M.1. Definitions)

M.2. A Formal Characterisation of Human Behaviour

(M. Domain Human Behaviour BLANK M.2. A Formal Characterisation of Human Behaviour)

M.3. Examples

(M. Domain Human Behaviour BLANK M.3. Examples)

M.4. Research Challenge

(M. Domain Human Behaviour BLANK M.4. Research Challenge)

End of Lecture 22: Human Behaviour

Start of Lecture 23: Domain Requirements Projection

N. Domain Requirements Projection BLANK

N.1. Definitions

N.2. Examples

N.3. Research Challenge

End of Lecture 23: Domain Requirements Projection

Start of Lecture 24: Domain Requirements Instantiation

O. Domain Requirements Instantiation BLANK

O.1. Definitions

O.2. Examples

O.3. Research Challenge

End of Lecture 24: Domain Requirements Instantiation

Start of Lecture 25: Domain Requirements Determination

P. Domain Requirements Determination BLANK

P.1. Definitions

P.2. Examples

P.3. Research Challenge

End of Lecture 25: Domain Requirements Determination

Start of Lecture 26: Domain Requirements Extension

Q. Domain Requirements Extension $\boxed{\text{BLANK}}$

Q.1. Definitions

Q.2. Examples

Q.3. Research Challenge

End of Lecture 26: Domain Requirements Extension

Start of Lecture 27: Domain Requirements Fitting

R. Domain Requirements Fitting BLANK

R.1. Definitions

R.2. Examples

R.3. Research Challenge

(R. **Domain Requirements Fitting** $\boxed{\text{BLANK}}$ R.3. **Research Challenge**)

$\boxed{\textbf{End of Lecture 27: Domain Requirements Fitting}}$