The Rôle of Domain Engineering in Software Development

Why Current Requirements Engineering is Flawed

Scotland October 2009

Dines Bjørner

Fredsvej 11, DK-2840 Holte, Danmark

E–Mail: bjorner@gmail.com, URL: www.imm.dtu.dk/˜db

September 1, 2009: 09:12

Summary
General

- We introduce the notion of domain descriptions (D) in order to ensure
  - ⋆ that software (S) is right and
  - ⋆ is the right software,
  - ⋆ that is, that it is correct with respect to written requirements (R)
  - ⋆ and that it meets customer expectations (D).
- That is, before software can be designed (S)
- we must make sure we understand the requirements (R),
- and before we can express the requirements
- we must make sure that we understand the application domain (D):
  - ⋆ the area of activity of the users of the required software,
  - ⋆ before and after installment of such software.

- We shall outline what we mean by
  - ⋆ informal, narrative
  - ⋆ and formal domain descriptions,
- and how one can systematically —
  - ⋆ albeit not (in fact: never) automatically —
  - ⋆ go from domain descriptions to requirements prescriptions.
- As it seems that domain engineering is a relatively new discipline
  - ⋆ within software engineering
  - ⋆ we shall mostly focus on domain engineering and discuss its necessity.
- The paper will show some formulas
  - ⋆ but they are really not meant to be read,
  - ⋆ let alone understood.

- They are merely there to bring home the point:
  - ⋆ Professional software engineering,
  - ⋆ like other professional engineering branches
  - ⋆ rely on and use mathematics.
- And it is all very simple to learn and practise anyway !
- We end this talk with, to some, perhaps, controversial remarks:
  - ⋆ Requirements engineering, as pursued today,
    - ◇ researched, taught and practised,
  - ⋆ is outdated, is thus fundamentally flawed.
- We shall justify this claim.

The Software Development Dogma

The Dogma

- The dogma is this:
  - ⋆ Before software can be designed
  - ⋆ we must understand the requirements.
  - ⋆ Before requirements can be finalised
  - ⋆ we must have understood the domain.
- We assume that the audience knows what is meant by
  - ⋆ software design and
  - ⋆ requirements.
- But what do we mean by “the domain” ?

[The Software Development Dogma]

What Do We Mean by 'Domain' ?

- By a domain we shall loosely understand an 'area' of
  - ⋆ natural or
  - ⋆ human

  activity, or both,

- where the 'area' is "well-delineated" such as, for example,
  - ⋆ for physics:
    - ◇ mechanics or
    - ◇ electricity or
    - ◇ chemistry or
    - ◇ hydrodynamics;
  - ⋆ or for an infrastructure component:
    - ◇ banking,
    - ◇ railways,
    - ◇ hospital health-care,
    - ◇ "the market":
      - ∘ consumers,
      - ∘ retailers,
      - ∘ wholesalers,
      - ∘ producers and
      - ∘ the distribution chain.

By a *domain* we shall thus, less loosely, understand

- a universe of discourse, small or large, a structure
  - ⋆ (i) of **simple entities**, that is, of "things", individuals, particulars
    - ◇ some of which are designated as state components;
  - ⋆ (ii) of **functions**, say over entities,
    - ◇ which when applied become possibly state-changing actions of the domain;
  - ⋆ (iii) of **events**,
    - ◇ possibly involving entities, occurring in time and
    - ◇ expressible as predicates over single or pairs of (before/after) states; and
  - ⋆ (iv) of **behaviours**,
    - ◇ sets of possibly interrelated sequences of actions and events.

[The Software Development Dogma]

Dialectics

- Now, let's get this "perfectly" straight !
  - ⋆ Can we develop software requirements without understanding the domain ?
  - ⋆ Well, how much of the domain should we understand ?
  - ⋆ And how well should we understand it ?

[The Software Development Dogma, Dialectics]

- Can we develop software requirements without understanding the domain ?
  - ⋆ No, of course we cannot !
  - ⋆ But we, you, do develop software for hospitals (railways, banks) without understanding health-care (transportation, the financial markets) anyway !
  - ⋆ In other engineering disciplines professionalism is ingrained:
    - ◇ Aeronautics engineers understand the domain of aerodynamics;
    - ◇ naval architects (i.e., ship designers) understand the domain of hydrodynamics;
    - ◇ telecommunications engineers understand the domain of electromagnetic field theory;
    - ◇ and so forth.

- Well, how much of the domain should we understand ?
  - ⋆ A basic answer is this:
    - ◇ enough for us to understand formal descriptions of such a domain.
  - ⋆ This is so in classical engineering:
    - ◇ Although the telecommunications engineer has not herself researched and made mathematical models of electromagnetic wave propagation in the form of Maxwell's equations:
      - ∘ Gauss's Law for Electricity,
      - ∘ Gauss's Law for Magnetism,
      - ∘ Faraday's Law of Induction,
      - ∘ Ampéres Law:

$$\oint \vec{E} \cdot d\vec{A} = \frac{q}{\varepsilon_0} \qquad \oint \vec{B} \cdot d\vec{A} = 0 \qquad \oint \vec{E} \cdot d\vec{s} = -\frac{d\Phi_B}{dt} \qquad \oint \vec{B} \cdot d\vec{s} = \mu_0 i + \frac{1}{c^2}\frac{\partial}{\partial t}\int \vec{E} \cdot d\vec{A}$$

    - ◇ the telecommunications engineer certainly understands these laws.

- And how well should we understand it ?
  - ⋆ Well, enough, as an engineer, to manipulate the formulas,
  - ⋆ to further develop these for engineering calculations.

Conclusion

- It is about time that software engineers
  - ⋆ consult precise descriptions, including formalisations,
  - ⋆ and establish, themselves or by consultants, such descriptions,
  - ⋆ of the application domains for software.
- These domain models may have to be developed by computing scientists.
- Software engineers then "transform" these into
  - ⋆ requirements prescriptions
  - ⋆ and software designs.

The Triptych of Software Development

- We recall the dogma:
  - ⋆ before software can be designed
  - ⋆ we must understand the requirements.
  - ⋆ Before requirements can be finalised
  - ⋆ we must have understood the domain.

- We conclude from that, that an “ideal” software development proceeds, in three major development phases, as follows:
- **Domain engineering**: The results of domain engineering include a domain model: a description,
  - ⋆ both informal, as a precise narrative,
  - ⋆ and formal, as a specification.
- The domain is described **as it is.**

- **Requirements engineering**: The results of requirements engineering include a requirements model: a prescription,
  - ⋆ both informal, as a precise narrative,
  - ⋆ and formal, as a specification.
- The requirements are described **as we would like the software to be,**
- and the requirements must be clearly related to the domain description.

- **Software design**: The results of software design include
  - ⋆ executable code
  - ⋆ and all documentation that goes with it.
- The software design specification must be **correct with respect to the requirements.**

Technicalities: An Overview

Domain Engineering

- Below we outline techniques of domain engineering. But just as a preview:
  - ⋆ Based on extensive domain acquisition and analysis
  - ⋆ an informal and a formal domain model is established, a model which is centered around sub-models of:
    - ◇ intrinsics,
    - ◇ supporting technologies,
    - ◇ mgt. and org.,
    - ◇ rules and regulations,
    - ◇ script [or contract] languages and
    - ◇ human behaviours,

    which are then
  - ⋆ validated and verified.

Requirements Engineering

- Below we outline techniques of requirements engineering. But just as a preview:
  - ⋆ Based on presentations of the domain model to requirements stakeholders
  - ⋆ requirements can now be “derived” from the domain model and as follows:
    - ⋄ First a **domain requirements** model:
      - ∘ **projection**,
      - ∘ **instantiation**,
      - ∘ **determination**,
      - ∘ **extension** and
      - ∘ **fitting** of several, separate domain requirements models;
    - ⋄ then an **interface requirements** model,
    - ⋄ and finally a **machine requirements** model.
  - ⋆ These are simultaneously verified and validated
  - ⋆ and the feasibility and satisfiability of the emerging model is checked.
- We show only the briefly explained specifications of an example “derivation” of (and in this case only of, and then only some aspects of) domain requirements.

Software Design

- We do not cover techniques of software design in detail — so only this summary.
  - ⋆ From the requirements prescription one develops,
    - ⋄ in stages and steps of transformation ("refinement"),
    - ⋄ first the system architecture,
    - ⋄ then the program (code) organisation (structure),
    - ⋄ and then, in further steps of development,
      - ◦ the component design,
      - ◦ the module design and
      - ◦ the code.
  - ⋆ These stages and step can be verified, model checked and tested with respect
    - ⋄ to the previous phase of requirements prescription,
    - ⋄ respectively the previous software design stages and steps.
- One can then assert that the $\mathcal{S}$oftware design is correct with respect to the $\mathcal{R}$equirements in the context of the assumptions expressed about the $\mathcal{D}$omain:

$$\mathcal{D}, \ \mathcal{S} \models \mathcal{R}$$

Domain Engineering

- We shall focus only on the actual modelling, thus omitting any treatment of
  - ⋆ the preparatory administrative and informative work,
  - ⋆ the identification of and liaison with domain stakeholders,
  - ⋆ the domain acquisition and analysis, and
  - ⋆ the establishment of a domain terminology (document).
- So we go straight to the descriptive work.
  - ⋆ We first illustrate the ideas of modelling domain phenomena and concepts in terms of simple entities, operations, events and behaviours,
  - ⋆ then we model the domain in terms of domain facets.
- Also, at then end, we do not have time and paper space for any treatment of domain verification, domain validations and the establishment of a domain theory.

[Domain Engineering]

Simple Entities, Operations, Events and Behaviours

- Without discussing our specification ontology,
  - ⋆ that is, the principles according to which we view the world around us,
- we just present the decomposition of phenomena and concepts into
  - ⋆ simple entities,
  - ⋆ operations,
  - ⋆ events and
  - ⋆ behaviours.
- All of these are "first class citizens", that is, are entities.
- We now illustrate examples of each of these ontological categories.

Simple Entities

- A *simple entity* is something that has a distinct, separate existence, though it need not be a material existence, to which we apply functions.
- With simple entities we associate attributes, i.e., properties modelled as types and values.
- Simple entities can be considered
  - ⋆ either continuous
  - ⋆ or discrete,
    - ◇ and, if discrete
      - ∘ then either atomic
      - ∘ or composite.

- It is the observer (that is, the specifier) who decides whether to consider a simple entity to be atomic or composite.
- Atomic entities cannot meaningfully be decomposed into sub-entities, but atomic entities may be analysed into (Cartesian) "compounds" of properties, that is, attributes. Attributes have name, type and value.
- Composite entities can be meaningfully decomposed into sub-entities, which are entities.
- The composition of sub-entities into a composite entity "reveals" the, or a mereology of the composite entity: that is, how it is "put together".

Example 1: Transport Entities: Nets, Links and Hubs — Narrative

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) identifiers (which can be observed from the hub [respectively link]).

Example 2: Transport Entities: Nets, Links and Hubs — Formalisation

type

1 H, L,

2 N = H-**set** × L-**set**

axiom

2 ∀ (hs,ls):N · **card** hs≥2 ∧ **card** hs≥1

type

3 HI, LI

value

4a obs_HI: H → HI, obs_LI: L → LI

axiom

4b ∀ h,h′:H, l,l′:L · h≠h′⇒obs_HI(h)≠obs_HI(h′) ∧ l≠l′⇒obs_LI(l)≠obs_LI(l′)

Operations

- By an *operation* we shall understand something which when *applied* to some entities, called the *arguments* of the operation, *yields* an entity, called the *result* of the operation application (also referred to as the operation invocation).
  - ⋆ Operations have signatures, that is, can be grossly described by the Cartesian type of its arguments and the possibly likewise compounded type of its results.
  - ⋆ Operations may be total over their argument types, or may be just partial. We shall consider some acceptable operations as "never terminating" processes.
  - ⋆ We shall, for the sake of consistency, consider all operation invocations as processes (terminating or non-terminating), and shall hence consider all operationdefinitions as also designating process definitions.

- We shall also use the term **function** to mean the same as the term operation.
- By a ***state*** we shall loosely understand a collection of one or more simple entities whose value may change.
- By an ***action*** we shall understand an operation application which applies to and/or yields a state.

Example 3: Link Insertion Operation

5. 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.

6. From a net one can remove a link. The removal command specifies

a link identifier.

type

5 Insert == Ins(s_ins:Ins)
5 Ins = 2xHubs | 1x1nH | 2nHs
5(a 2xHubs == 2oldH(s_hi1:HI,s_l:L,s_hi2:HI)
5(b 1x1nH == 1oldH1newH(s_hi:HI,s_l:L,s_h:H)
5(c 2nHs == 2newH(s_h1:H,s_l:L,s_h2:H)

axiom

5(d $\forall$ 2oldH(hi′,l,hi″):Ins $\cdot$ hi′$\neq$hi″ $\wedge$ obs_LIs(l)={hi′,hi″} $\wedge$
$\forall$ 1old1newH(hi,l,h):Ins $\cdot$ obs_LIs(l)={hi,obs_HI(h)} $\wedge$
$\forall$ 2newH(h′,l,h″):Ins $\cdot$ obs_LIs(l)={obs_HI(h′),obs_HI(h″)}

7. 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}.

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

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

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

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

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

   9.1–.2 : the two connecting hubs are updated to reflect the new link,

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

int_Insert(op)(hs,ls) $\equiv$
$\star_i$ **case** op **of**
7 2newH(h′,l,h″) $\rightarrow$ (hs $\cup$ {h′,h″},ls $\cup$ {l}),

```
8     1oldH1newH(hi,l,h) →
8.1       let h′ = aLI(xtr_H(hi,hs),obs_LI(l)) in
8.2       (hs\{xtr_H(hi,hs)}∪{h,h′},ls ∪{l}) end,
9     2oldH(hi′,l,hi″) →
9.1       let hsδ = {aLI(xtr_H(hi′,hs),obs_LI(l)),
9.2                  aLI(xtr_H(hi″,hs),obs_LI(l))} in
9.3       (hs\{xtr_H(hi′,hs),xtr_H(hi″,hs)}∪ hsδ,ls ∪{l}) end
⋆j  end     ⋆k  pre pre_int_Insert(op)(hs,ls)
```

Events

- Informally, by an ***event*** we shall loosely understand the occurrence of "something" that may either trigger an action, or is triggered by an action, or alter the course of a behaviour, or a combination of these.
- An ***event*** can be characterised by
  - ⋆ a predicate, p and
  - ⋆ a pair of ("before") and ("after") of pairs of
    - ◇ states and
    - ◇ times:
    - ◇ $p((t_b,\sigma_b),(t_a,\sigma_a))$.
  - ⋆ Usually the time interval $t_a - t_b$
  - ⋆ is of the order $t_a \simeq (t_b) + \delta_{\text{tiny}}$.

Example 4: Transport Events

- (i) A link, for some reason "ceases to exist"; for example:
  - ⋆ a bridge link falls down,
  - ⋆ or a level road link is covered by a mud slide,
  - ⋆ or a road tunnel is afire,
  - ⋆ or a link is blocked by some vehicle accident.
- (ii) A vehicle enters or leaves the net.
- (iii) A hub is saturated with vehicles.

Behaviours

- By a ***behaviour*** we shall informally understand a strand of (sets of) actions and events.
  - ⋆ In the context of domain descriptions we shall speak of behaviours
  - ⋆ whereas, in the context of requirements prescriptions and software designs we shall use the term processes.
- By a ***behaviour*** we, more formally, understand a sequence, q
  - ⋆ of actions
  - ⋆ and/or events

$q_1, q_2, \ldots, q_i, q_{i+1}, \ldots, q_n$

- such that the state
  - ⋆ resulting from one such action, q_i,
  - ⋆ or in which some event, q_i, occurs,
- becomes the state in which the next action or event, q_{i+1},
  - ⋆ if it is an action, is effected,
  - ⋆ or, if it is an event, is the event state.

Example 5: Transport: Traffic Behaviour

10. There are further undefined vehicles.
11. Traffic is a discrete function from a 'Proper subset of Time' to pairs of nets and vehicle positions.
12. Vehicles positions is a discrete function from vehicles to vehicle positions.

type

10 Veh

11 TF = Time $\overrightarrow{m}$ (N × VehPos)

12 VehPos = Veh $\overrightarrow{m}$ Pos

13. There are positions, and a position is either on a link or in a hub.
    (a) A hub position is indicated just by a triple: the identifier of the hub in question, and a pair of (from and to) link identifiers, namely of links connected to the identified hub.
    (b) A link position is identified by a quadruplet: The identifier of the link, a pair of hub identifiers (of the link connected hubs), designating a direction, and a real number, properly between 0 and 1, denoting the relative offset from the from hub to the to hub.

type

13 Pos = HPos | LPos

13(a) HPos == hpos(s_hi:HI,s_fli:LI,s_tli:LI)

13(b) LPos == lpos(s_li:HI,s_fhi:LI,s_tli:LI,s_offset:Frac)

13(b) Frac = {|r:**Real**·0<r<1|}

[Domain Engineering]

Domain Facets

- By a **domain facet** we mean
  - ⋆ one amongst a finite set of generic ways
  - ⋆ of analysing a domain:
  - ⋆ a view of the domain,
  - ⋆ such that the different facets cover conceptually different views,
  - ⋆ and such that these views together cover the domain
- We shall postulate the following domain facets:
  - ⋆ intrinsics,
  - ⋆ support technologies,
  - ⋆ management & organisation,
  - ⋆ rules & regulations,
  - ⋆ script languages [contract languages] and
  - ⋆ human behaviour.
- Each facet covers simple entities, operations, events and behaviours.
- We shall now illustrate these.

[Domain Engineering, Domain Facets]

Intrinsics

- By **domain intrinsics** we mean
  - ⋆ those phenomena and concepts of a domain which are basic to any of the other facets (listed earlier and treated, in some detail, below),
  - ⋆ with such domain intrinsics initially covering at least one specific, hence named, stakeholder view.

Example 6: Intrinsics, I

- The links, hubs, hence the nets,
- and the identifiers of links and hubs
- are intrinsic phenomena, respectively concepts.

- So are:

Example 7: Intrinsics, II

14. From any link of a net one can observe the two hubs to which the link is connected.
    (a) We take this 'observing' to mean the following: From any link of a net one can observe the two distinct identifiers of these hubs.
15. From any hub of a net one can observe the one or more links to which are connected to the hub.
    (a) Again: by observing their distinct link identifiers.
16. Extending Item 14: the observed hub identifiers must be identifiers of hubs of the net to which the link belongs.
17. Extending Item 15: the observed link identifiers must be identifiers of links of the net to which the hub belongs.

value
14a obs_HIs: L → HI-**set**,
15a obs_LIs: H → LI-**set**,
axiom
14b ∀ l:L • **card** obs_HIs(l)=2 ∧
15b ∀ h:H • **card** obs_LIs(h)≥1 ∧
∀ (hs,ls):N •
14(a) ∀ h:H • h ∈ hs ⇒ ∀ li:LI • li ∈ obs_LIs(h) ⇒
∃ l′:L • l′ ∈ ls ∧ li=obs_LI(l′) ∧ obs_HI(h) ∈ obs_HIs(l′) ∧
15(a) ∀ l:L • l ∈ ls ⇒
∃ h′,h″:H • {h′,h″}⊆hs ∧ obs_HIs(l)={obs_HI(h′),obs_HI(h″)}
16 ∀ h:H • h ∈ hs ⇒ obs_LIs(h) ⊆ iols(ls)
17 ∀ l:L • l ∈ ls ⇒ obs_HIs(h) ⊆ iohs(hs)
value
iohs: H-**set** → HI-**set**, iols: L-**set** → LI-**set**
iohs(hs) ≡ {obs_HI(h)|h:H•h ∈ hs}
iols(ls) ≡ {obs_LI(l)|l:L•l ∈ ls}

[Domain Engineering, Domain Facets]

Support Technologies

- By **domain support technologies** we mean
  - ⋆ ways and means of concretesing
  - ⋆ certain observed (abstract or concrete) phenomena or
  - ⋆ certain conceived concepts
  - ⋆ in terms of (possibly combinations of)
    - ◇ human work,
    - ◇ mechanical,
    - ◇ hydro mechanical,
    - ◇ thermo-mechanical,
    - ◇ pneumatic,
    - ◇ aero-mechanical,
    - ◇ electro-mechanical,
    - ◇ electrical,
    - ◇ electronic,
    - ◇ telecommunication,
    - ◇ photo/opto-electric,
    - ◇ chemical, etc.

  (possibly computerised) sensor, actuator tools.

- In this example of a support technology
  - ⋆ we shall illustrate an abstraction
  - ⋆ of the kind of semaphore signalling
  - ⋆ one encounters at road intersections, that is, hubs.
- The example is indeed an abstraction:
  - ⋆ we do not model the actual "machinery"
    - ◇ of road sensors,
    - ◇ hub-side monitoring & control boxes, and
    - ◇ the actuators of the green/yellow/red sempahore lamps.
  - ⋆ But, eventually, one has to,
  - ⋆ all of it,
  - ⋆ as part of domain modelling.

[Domain Engineering, Domain Facets, Support Technologies]

Example 8: Hub Sempahores

- To model signalling we need to model hub and link states.
- A hub (link) state is the set of all traversals that the hub (link) allows.
  - ⋆ A hub traversal is a triple of identifiers:
    - ⋄ of the link from where the hub traversal starts,
    - ⋄ of the hub being traversed, and
    - ⋄ of the link to where the hub traversal ends.
  - ⋆ A link traversal is a triple of identifiers:
    - ⋄ of the hub from where the link traversal starts,
    - ⋄ of the link being traversed, and
    - ⋄ of the hub to where the link traversal ends.
  - ⋆ A hub (link) state space is the set of all states that the hub (link) may be in.
  - ⋆ A hub (link) state changing operation can be designated by
    - ⋄ the hub and a possibly new hub state (the link and a possibly new link state).

type
  $L\Sigma' = L\_Trav$-**set**
  $L\_Trav = (HI \times LI \times HI)$
  $L\Sigma = \{| \, lnk\sigma{:}L\Sigma' \bullet syn\_wf\_L\Sigma\{lnk\sigma\} \, |\}$

$\text{H}\Sigma' = \text{H\_Trav-}\textbf{set}$
$\text{H\_Trav} = (\text{LI} \times \text{HI} \times \text{LI})$
$\text{H}\Sigma = \{|\ \text{hub}\sigma\text{:H}\Sigma' \bullet \text{wf\_H}\Sigma\{\text{hub}\sigma\}\ |\}$
$\text{H}\Omega = \text{H}\Sigma\text{-}\textbf{set},\ \text{L}\Omega = \text{L}\Sigma\text{-}\textbf{set}$

value

$\text{obs\_L}\Sigma\text{: L} \rightarrow \text{L}\Sigma,\ \text{obs\_L}\Omega\text{: L} \rightarrow \text{L}\Omega$
$\text{obs\_H}\Sigma\text{: H} \rightarrow \text{H}\Sigma,\ \text{obs\_H}\Omega\text{: H} \rightarrow \text{H}\Omega$

axiom

$\forall\ \text{h:H} \bullet \text{obs\_H}\Sigma(\text{h}) \in \text{obs\_H}\Omega(\text{h}) \wedge \forall\ \text{l:L} \bullet \text{obs\_L}\Sigma(\text{l}) \in \text{obs\_L}\Omega(\text{l})$

value

$\text{chg\_H}\Sigma\text{: H} \times \text{H}\Sigma \rightarrow \text{H},\ \text{chg\_L}\Sigma\text{: L} \times \text{L}\Sigma \rightarrow \text{L}$
$\text{chg\_H}\Sigma(\text{h,h}\sigma)\ \textbf{as}\ \text{h}'$
$\quad \textbf{pre}\ \text{h}\sigma \in \text{obs\_H}\Omega(\text{h})\ \textbf{post}\ \text{obs\_H}\Sigma(\text{h}')\text{=h}\sigma$
$\text{chg\_L}\Sigma(\text{l,l}\sigma)\ \textbf{as}\ \text{l}'$
$\quad \textbf{pre}\ \text{l}\sigma \in \text{obs\_L}\Omega(\text{h})\ \textbf{post}\ \text{obs\_H}\Sigma(\text{l}')\text{=l}\sigma$

- Well, so far we have indicated that there is an operation that can change hub and link states.
- But one may debate whether those operations shown are really examples of a support technology. (That is, one could equally well claim that they remain examples of intrinsic facets.)
- We may accept that and then ask the question:

- ⋆ How to effect the described state changing functions ?
- ⋆ In a simple street crossing a semaphore does not instantaneously change from red to green in one direction while changing from green to red in the cross direction.
- ⋆ Rather there is are intermediate sequences of, for example, not necessarily synchronised green/yellow/red and red/yellow/green states to help avoid vehicle crashes and to prepare vehicle drivers.

- Our "solution" is to modify the hub state notion.

type
  Colour == red | yellow | green
  X = LI×HI×LI×Colour [crossings **of** a hub]
  HΣ = X-**set** [hub states]
value
  obs_HΣ: H → HΣ, xtr_Xs: H → X-**set**
  xtr_Xs(h) ≡
    {(li,hi,li′,c)|li,li′:LI,hi:HI,c:Colour•{li,li′}⊆obs_LIs(h)∧hi=obs_HI(h)}
axiom
  ∀ n:N,h:H • h ∈ obs_Hs(n) ⇒ obs_HΣ(h)⊆xtr_Xs(h) ∧
    ∀ (li1,hi2,li3,c),(li4,hi5,li6,c′):X •
      {(li1,hi2,li3,c),(li4,hi5,li6,c′)}⊆obs_HΣ(h) ∧

$li1=li4 \wedge hi2=hi5 \wedge li3=li6 \Rightarrow c=c'$

- We consider the colouring, or any such scheme, an aspect of a support technology facet.
- There remains, however, a description of how the technology that supports the intermediate sequences of colour changing hub states.
- We can think of each hub being provided with a mapping from pairs of "stable" (that is non-yellow coloured) hub states ($h\sigma_i$,$h\sigma_f$) to well-ordered sequences of intermediate "un-stable' (that is yellow coloured) hub states
  - ⋆ paired with some time interval information
  - ⋆ $\langle(h\sigma', t\delta'), (h\sigma'', t\delta''), \ldots, (h\sigma'^{\cdots\prime}, t\delta'^{\cdots\prime})\rangle$
  - ⋆ and so that each of these intermediate states can be set,
  - ⋆ according to the time interval information,[1]
  - ⋆ before the final hub state ($h\sigma_f$) is set.

type
  TI [time interval]
  Signalling = (HΣ × TI)*
  Sema = (HΣ × HΣ) $\overrightarrow{m}$ Signalling
value
  obs_Sema: H → Sema, chg_HΣ: H × HΣ → H, chg_HΣ_Seq: H × HΣ → H

```
chg_HΣ(h,hσ) as h′ pre hσ ∈ obs_HΩ(h) post obs_HΣ(h′)=hσ
chg_HΣ_Seq(h,hσ) ≡
   let sigseq = (obs_Sema(h))(obs_Σ(h),hσ) in sig_seq(h)(sigseq) end

sig_seq: H → Signalling → H
sig_seq(h)(sigseq) ≡
   if sigseq=⟨⟩ then h else
   let (hσ,tδ) = hd sigseq in
   let h′ = chg_HΣ(h,hσ); wait tδ;
   sig_seq(h′)(tl sigseq) end end end
```

[Domain Engineering, Domain Facets]

Management and Organisation

Management

- By **domain management** we mean people
  - ⋆ (i) who determine, formulate and thus set standards (cf. rules and regulations, a later lecture topic) concerning
    - ◇ strategic, tactical and operational decisions;
  - ⋆ (ii) who ensure that these decisions are passed on to (lower) levels of management, and to "floor" staff;
  - ⋆ (iii) who make sure that such orders, as they were, are indeed carried out;
  - ⋆ (iv) who handle undesirable deviations in the carrying out of these orders cum decisions;
  - ⋆ and (v) who "backstop" complaints from lower management levels and from floor staff.

Organisation

- By **domain organisation** we mean
  - ⋆ the structuring of management and non-management staff levels;
  - ⋆ the allocation of
    - ◇ strategic, tactical and operational concerns
    - ◇ to within management and non-management staff levels;
  - ⋆ and hence the “lines of command”:
    - ◇ who does what and
    - ◇ who reports to whom —
      - ◦ administratively and
      - ◦ functionally.

Examples

Example 9: Bus Transport Management & Organisation

- On Slides 69–75 we illustrate what is there called a contract language.
  - ⋆ "Programs" in that language are either contracts or are orders to perform the actions permitted or obligated by contracts.
  - ⋆ The language in question is one of managing bus traffic on a net.
  - ⋆ The **management & organisation** of bus traffic involves
    - ◇ contractors issuing contracts,
    - ◇ contractees acting according to contracts,
    - ◇ busses (owned or leased) by contractees,
    - ◇ and the bus traffic on the (road) net.
  - ⋆ Contractees, i.e., bus operators,

- ◇ `"start"` buses according to a contract timetable,
- ◇ `"cancel"` buses if and when deemed necessary,
- ◇ `"insert"` rush-hour and other buses if and when deemed necessary,
- ◇ and, acting as contractors, `"sub-contract"` sub-contractees to operate bus lines,
  - ○ for example, when the issuing contractor is not able to operate these bus lines,
  - ○ i.e., not able to fulfill contractual obligations,
  - ○ due to unavailability of buses or staff.

- Clearly the programs of bus contract languages
  - ⋆ are "executed" according to **management** decisions
  - ⋆ and the sub-contracting "hierarchy" reflects **organisational** facets.

[Domain Engineering, Domain Facets]

Rules and Regulations

- Human stakeholders act in the domain, whether
  - ⋆ clients,
  - ⋆ workers,
  - ⋆ managers,
  - ⋆ suppliers,
  - ⋆ regulatory authorities,
  - ⋆ or other.
- Their actions are guided and constrained by rules and regulations.
- These are sometimes implicit, that is, not "written down".
- But we can talk about rules and regulations as if they were explicitly formulated.

- The main difference between rules and regulations is that
  - ⋆ rules express properties that must hold and
  - ⋆ regulations express state changes that must be effected if rules are observed broken.
- Rules and regulations are directed
  - ⋆ not only at human behaviour
  - ⋆ but also at expected behaviours of support technologies.
- Rules and regulations are formulated
  - ⋆ by enterprise staff, management or workers,
  - ⋆ and/or by business and industry associations,
    - ◇ for example in the form of binding or guiding
    - ◇ national, regional or international standards,
  - ⋆ and/or by public regulatory agencies.

Domain Rules

- By a **domain rule** we mean
  - some text
  - which prescribes how people or equipment
  - are expected to behave when dispatching their duty,
  - respectively when performing their functions.

Domain Regulations

- By a **domain regulation** we mean
  - some text
  - which prescribes what remedial actions are to be taken
  - when it is decided that a rule has not been followed according to its intention.

[Domain Engineering, Domain Facets, Rules and Regulations]

Two Informal Examples

Example 10: Trains at Stations: Available Station Rule and Regulation

- Rule:
  - ⋆ In China the arrival and departure of trains at, respectively from, railway stations is subject to the following rule:
  - ⋆ *In any three-minute interval at most one train may either arrive to or depart from a railway station.*
- Regulation:
  - ⋆ *If it is discovered that the above rule is not obeyed*, then there is some regulation which prescribes administrative or legal management and/or staff action, as well as some correction to the railway traffic.

Example 11: Trains Along Lines: Free Sector Rule and Regulation

- Rule:
  - ⋆ In many countries railway lines (between stations) are segmented into blocks or sectors. The purpose is to stipulate that if two or more trains are moving along the line, then:
  - ⋆ *There must be at least one free sector (i.e., without a train) between any two trains along a line.*
- Regulation:
  - ⋆ *If it is discovered that the above rule is not obeyed*, then there is some regulation which prescribes administrative or legal management and/or staff action, as well as some correction to the railway traffic.

A Formal Example

- We shall develop the above example (11, Slide 56) into a partial, formal specification.
- That is, not complete, but "complete enough" for the reader to see what goes on.

Example 12: Continuation of Example 11 Slide 56

- We start by analysing the text of the rule and regulation.
  - ⋆ The rule text: *There must be at least one free sector (i.e., without a train) between any two trains along a line.* contains the following terms:
    - ◇ free (a predicate),
    - ◇ sector (an entity),
    - ◇ train (an entity) and
    - ◇ line (an entity).
- We shall therefore augment our formal model to reflect these terms.
- We start by modelling
  - ⋆ sectors and sector descriptors,
  - ⋆ lines and train position descriptors,
  - ⋆ trains, and
  - ⋆ the predicate free.


```
type
  Sect′ = H × L × H,
  SectDescr = HI × LI × HI
  Sect = {|(h,l,h′):Sect′ • obs_HIs(l)={obs_HI(h),obs_HI(h′)}|}
  SectDescr =  {|(hi,li,hi′):SectDescr′ •
                ∃ (h,l,j′):Sect•obs_HIs(l)={obs_HI(h),obs_HI(h′)}|}
  Line′ = Sect*,
  Line = {|line:Line′•wf_Line(line)|}
  TrnPos′ = SectDescr*
  TrnPos = {|trnpos′:TrnPos′•∃ line:Line•conv_Line_to_TrnPos(line)=trnpos′|}
value
  wf_Line: Line′ → Bool
  wf_Line(line) ≡
    ∀ i:Nat • {i,i+1}⊆inds(line) ⇒
      let (_,l,h)=line(i),(h′,l′,_)=line(i+1) in h=h′ end
  conv_Line_to_TrnPos: Line → TrnPos
  conv_Line_to_TrnPos(line) ≡
    ⟨(obs_HI(h),obs_LI(l),obs_HI(h′))|1≤i≤len line∧line(i)=(h,l,h′)⟩
```

value
 lines: N → Line-**set**
 lines(hs,ls) ≡
  let lns = {⟨(h,l,h′)⟩|h,h′:H,l:L•proper_line((h,l,h′),(hs,ls))}
   ∪ {ln⁀ln′|ln,l′:Line•{ln,ln′}⊆lns∧adjacent(ln,ln′)} **in**
  lns **end**

 adjacent: Line × Line → **Bool**
 adjacent((_,l,h),(h′,l′,_)) ≡ h=h′
  pre {obs_LI(l),obs_LI(l′)}⊆ obs_LIs(h)

type
 $TF = T \underset{m}{\rightarrow} (N \times (TN \underset{m}{\rightarrow} TrnPos))$

value
 wf_TF: TF → **Bool**

$$
\begin{array}{l}
\text{wf\_TF}(tf) \equiv \\
\quad \forall\ t{:}T \cdot t \in \textbf{dom}\ tf \Rightarrow \\
\qquad \textbf{let}\ ((hs,ls),trnposs) = tf(t)\ \textbf{in} \\
\qquad \forall\ trn{:}TN \cdot trn \in \textbf{dom}\ trnposs \Rightarrow \\
\qquad\quad \exists\ line{:}Line \cdot line \in lines(hs,ls) \wedge \\
\qquad\qquad trnposs(trn) = \text{conv\_Line\_to\_TrnPos}(line)\ \textbf{end}
\end{array}
$$

- Nothing prevents two or more trains from occupying overlapping train positions.
- They have “merely” – and regrettably – crashed. But such is the domain.
- So **wf_TF(tf)** is not part of an axiom of traffic, merely a desirable property.

value

$$
\begin{array}{l}
\text{has\_free\_Sector}{:}\ TN \times T \to TF \to \textbf{Bool} \\
\text{has\_free\_Sector}(trn,(hs,ls),t)(tf) \equiv \\
\quad \textbf{let}\ ((hs,ls),trnposs) = tf(t)\ \textbf{in} \\
\quad (trn \notin \textbf{dom}\ trnposs \vee (tn \in \textbf{dom}\ trnposs(t) \wedge \\
\quad\ \exists\ ln{:}Line \cdot ln \in lines(hs,ls) \wedge \\
\qquad \text{is\_prefix}(trnposs(trn),ln))(hs,ls)) \wedge \\
\qquad \sim\exists\ trn'{:}TN \cdot trn' \in \textbf{dom}\ trnposs \wedge trn' \neq trn \wedge
\end{array}
$$

$$
\begin{array}{l}
\quad\quad \text{trnposs(trn}'\text{)=conv\_Line\_to\_TrnPos(}\langle\text{follow\_Sect(ln)(hs,ls)}\rangle\text{)} \\
\quad \mathbf{end} \\
\quad \mathbf{pre}\ \text{exists\_follow\_Sect(ln)(hs,ls)}
\end{array}
$$

$$
\begin{array}{l}
\text{is\_prefix: Line} \times \text{Line} \rightarrow \text{N} \rightarrow \mathbf{Bool} \\
\text{is\_prefix(ln,ln}'\text{)(hs,ls)} \equiv \exists\ \text{ln}'':\text{Line} \bullet \text{ln}'' \in \text{lines(hs,ls)} \wedge \text{ln} ^\frown \text{ln}''\text{=ln}'
\end{array}
$$

$$
\begin{array}{l}
\text{exists\_follow\_Sect: Line} \rightarrow \text{Net} \rightarrow \mathbf{Bool} \\
\text{exists\_follow\_Sect(ln)(hs,ls)} \equiv \\
\quad \exists\ \text{ln}':\text{Line}\bullet\text{ln}' \in \text{lines(hs,ls)} \wedge \text{ln} ^\frown \text{ln}' \in \text{lines(hs,ls)} \\
\quad \mathbf{pre}\ \text{ln} \in \text{lines(hs,ls)} \\
\text{follow\_Sect: Line} \rightarrow \text{Net} \stackrel{\sim}{\rightarrow} \text{Sect} \\
\text{follow\_Sect(ln)(hs,ls)} \equiv \\
\quad \mathbf{let}\ \text{ln}':\text{Line}\bullet\text{ln}' \in \text{lines(hs,ls)} \wedge \text{ln} ^\frown \text{ln}' \in \text{lines(hs,ls)}\ \mathbf{in}\ \mathbf{hd}\ \text{ln}'\ \mathbf{end} \\
\quad \mathbf{pre}\ \text{line} \in \text{lines(hs,ls)} \wedge \text{exists\_follow\_Sect(ln)(hs,ls)}
\end{array}
$$

- We doubly recursively define a function free_sector_rule(tf)(r).

- tf is that part of the traffic which has yet to be "searched" for non-free sectors.
  - ⋆ Thus tf is "counted" up from a first time t till the traffic tf is empty.
  - ⋆ That is, we assume a finite definition set tf .
- r is like a traffic but without the net.
  - ⋆ Initially r is the empty traffic.
  - ⋆ r is "counted" up from "earliest" cases of trains with no free sector ahead of them.
- The recursion stops, for a given time when
  - ⋆ there are no more train positions to be "searched" for that time;
  - ⋆ and when the "to-be-searched" traffic is empty.

type
  $\mathrm{TNPoss} = \mathrm{T} \ \overrightarrow{m}\ (\mathrm{TN} \rightarrow \mathrm{TrnPos})$
value
  free_sector_rule: TF × TF → TNPoss
  free_sector_rule(tf)(r) ≡
    if tf=[ ] **then** r **else**

```
let t:T•t ∈ dom tf∧smallest(t)(tf) in
let ((hs,ls),trnposs)=tf(t) in
if trnposs=[ ] then free_sector_rule(tf\{t})(r) else
let tn:TN•tn ∈ dom trnposs in
if exists_follow_Sect(trnposs(tn))(hs,ls)∧∼has_free_Sector(tn,(hs,ls),t)(tf)
  then
    let r′ = if t ∈ dom r then r else r ∪ [ t↦[ ] ] end in
    free_sector_rule(tf†[ t↦((hs,ls),trnposs\{tn}) ])
                (r†[ t↦r(t)∪[ tn↦trnposs(tn) ] ]) end
  else
    free_sector_rule(tf†[ t↦((hs,ls),trnposs\{trn}) ])(r)
end end end end end end

smallest(t)(tf) ≡ ∼∃ t′:T• t′isin dom tf∧t′<t pre t ∈ dom tf
```

[Domain Engineering, Domain Facets]

Script Languages [Contract Languages]

- By a **domain script language** we mean
  - ⋆ the definition of a set of licenses and actions
  - ⋆ where these licenses when issued
  - ⋆ and actions when performed have morally obliging power.
- By a **domain contract language**
  - ⋆ a domain script language whose licenses and actions have legally binding power,
  - ⋆ that is, their issuance and their invocation may be contested in a court of law.

[Domain Engineering, Domain Facets, Script Languages [Contract Languages]]

A Script Language

- Some common, visual forms of bus timetables are shown in Fig. 4.1.

Figure 4.1: Some bus timetables: Spain, India and Norway

[Domain Engineering, Domain Facets, Script Languages [Contract Languages], A Script Language]

Example 13: Narrative Syntax of a Bus Timetable Script Language

18. 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.
19. A TimeTable associates to Bus Line Identifiers a set of Journies.
20. Journies are designated by a pair of a BusRoute and a set of BusRides.
21. 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.
22. A set of BusRides associates, to each of a number of Bus Identifiers a Bus Schedule.
23. 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.
24. 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.
25. A Fraction is a **Real** properly between 0 and 1.
26. The Journies must be well_formed in the context of some net.

[Domain Engineering, Domain Facets, Script Languages [Contract Languages], A Script Language]

Example 14: Formal Syntax of a Bus Timetable Script Language

type

18. T, BLId, BId
19. TT = BLId $\overrightarrow{m}$ Journies
20. Journies′ = BusRoute × BusRides
21. BusRoute = BusStop × BusStop* × BusStop
22. BusRides = BId $\overrightarrow{m}$ BusSched
23. BusSched = T × T* × T
24. BusStop == mkBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
25. Frac = {|r:**Real**·0<r<1|}
26. Journies = {|j:Journies′·∃ n:N · wf_Journies(j)(n)|}

[Domain Engineering, Domain Facets, Script Languages [Contract Languages], A Script Language]

Example 15: Semantics of a Bus Timetable Script Language

type
  Bus
value
  obs_X: Bus → X
type
  BusTraffic = $T \xrightarrow[m]{} (N \times (BusNo \xrightarrow[m]{} (Bus \times BPos)))$
  BPos = atHub | onLnk | atBS
  atHub == mkAtHub(s_fl:LIs_hi:HI,s_tl:LI)
  onLnk == mkOnLnk(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
  atBSt == mkAtBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
  Frac = {|r:**Real**·0<r<1|}
value
  gen_BusTraffic: TT → BusTraffic-**infset**
  gen_BusTraffic(tt) **as** btrfs
    post ∀ btrf:BusTraffic · btrf ∈ btrfs ⇒ on_time(btrf)(tt)

A Contract Language

- We shall, as for the timetable script, just hint at a contract language.

Example 16: Informal Syntax of Bus Transport Contracts

- An example contract can be 'schematised':

  con_id: **contractor** corn **contracts contractee** ceen
  to perform operations `"start","cancel","insert","subcontract"`
  with respect to bus timetable tt.

[Domain Engineering, Domain Facets, Script Languages [Contract Languages], A Contract Language]

Example 17: Formal Syntax of a Bus Transport Contracts

type
CId, CNm
Contract = CId × CNm × CNm × Body
Body = Op-**set** × TT
Op == ″conduct″ | ″cancel″ | ″insert″ | ″subcontract″

an example contract:

(cid,cor,cee,({″start″,″cancel″,″insert″,″subcontract″},tt))

[Domain Engineering, Domain Facets, Script Languages [Contract Languages], A Contract Language]

Example 18: Informal Syntax of a Bus Transport Actions

- Example actions can be schematised:

(a) cid: **start bus ride** (blid,bid) **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 69) shown earlier is almost like an action; here is the action form:

(d) cid: **contractee** cee **is granted a license** cid′
to perform operations {"start","cancel","insert",subcontract"}
with respect to timetable tt′.

[Domain Engineering, Domain Facets, Script Languages [Contract Languages], A Contract Language]

Example 19: Formal Syntax of a Bus Transport Actions

type

Action = CNm × CId × (SubLic | SmpAct) × Time
SmpAct = Start | Cancel | Insert
DoRide == 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:ConId,s_cee:CNm,s_body:(s_ops:Op-**set**,s_tt:TT))

examples:

(a) (cee,cid,mkRid(blid,id),t)
(b) (cee,cid,mkCan(blid,id),t)
(c) (cee,cid,mkIns(blid,id),t)
(d) (cee,cid,mkCon(cid′,({″`start`″,″`cancel`″,″`insert`″,″`subcontract`″},tt′),t))

where: cid′ = generate_ConId(cid,cee,t)

Example 20: Semantics of a Bus Transport Contract Language: States

type

$$
\begin{aligned}
&\text{Body} = \text{Op-}\mathbf{set} \times \text{TT}\\
&\text{Con}\Sigma = \text{RcvCon}\Sigma \times \text{SubCon}\Sigma \times \text{CorBus}\Sigma\\
&\text{RcvCon}\Sigma = \text{CNm} \underset{m}{\rightarrow} (\text{CId} \underset{m}{\rightarrow} (\text{Body} \times \text{TT}))\\
&\text{SubCon}\Sigma = \text{CNm} \underset{m}{\rightarrow} (\text{CId} \underset{m}{\rightarrow} \text{Body})\\
&\text{BusNo}\\
&\text{Bus}\Sigma = \text{FreeBuses}\Sigma \times \text{ActvBuses}\Sigma \times \text{BusHists}\Sigma\\
&\text{FreeBuses}\Sigma = \text{BusStop} \underset{m}{\rightarrow} \text{BusNo-}\mathbf{set}\\
&\text{ActvBuses}\Sigma = \text{BusNo} \underset{m}{\rightarrow} \text{BusInfo}\\
&\text{BusInfo} = \text{BLId} \times \text{BId} \times \text{CId} \times \text{CNm} \times \text{BusTrace}\\
&\text{BusHists}\Sigma = \text{Bno} \underset{m}{\rightarrow} \text{BusInfo}^*\\
&\text{BusTrace} = (\text{Time} \times \text{BusStop})^*\\
&\text{CorBus}\Sigma = \text{CNm} \underset{m}{\rightarrow} (\text{CId} \underset{m}{\rightarrow} ((\text{BLId} \times \text{BId}) \underset{m}{\rightarrow} (\text{BNo} \times \text{BusTrace})))\\
&\text{AllBs} = \text{CNm} \underset{m}{\rightarrow} \text{BusNo-}\mathbf{set}
\end{aligned}
$$

[Domain Engineering, Domain Facets, Script Languages [Contract Languages], A Contract Language]

Example 21: Semantics of a Bus Transport Contract Language: Constants and Functions

value

cns:CNm-**set**, busnos:BNo-**set**, ibσ:IBΣs=CNm $\overrightarrow{m}$ BusΣ,
rcor,icee:CNm • rcor $\notin$ cns$\wedge$icee $\in$ cns, itr:BusTraffic,
rcid:ConId, iops:Op-**set**={"`subcontract`"}, itt:TT, t_0:Time
allbs:AllBs • **dom** allbs=cns $\cup$ {rcor}$\wedge\cup$ **rng** allbs=busnos,
icon:Contract=(rcid,rcor,icee,(iops,itt)),
icσ:ConΣ=([icee $\mapsto$ [rcid $\mapsto$ [icee $\mapsto$ icon]]]
$\cup$ [cee $\mapsto$ [ ] | cee:CNm • cee $\in$ cnms\{icee}],[ ],[ ]),

system: **Unit** $\rightarrow$ **Unit**
system() $\equiv$
cntrcthldr(icee)(ilσ(icee),ibσ(icee))
‖(‖{**cntrcthldr**(cee)(ilσ(cee),ibσ(cee))|cee:CNm•cee $\in$ cns\{icee}})
‖(‖{**bus_ride**(b,cee)(rcor,"`nil`")
| cee:CNm,b:BusNo•cee $\in$ **dom** allbs $\wedge$ b $\in$ allbs(cee)})
‖**time_clock**(t_0) ‖ **bus_traffic**(itr)

[Domain Engineering, Domain Facets, Script Languages [Contract Languages], A Contract Language]

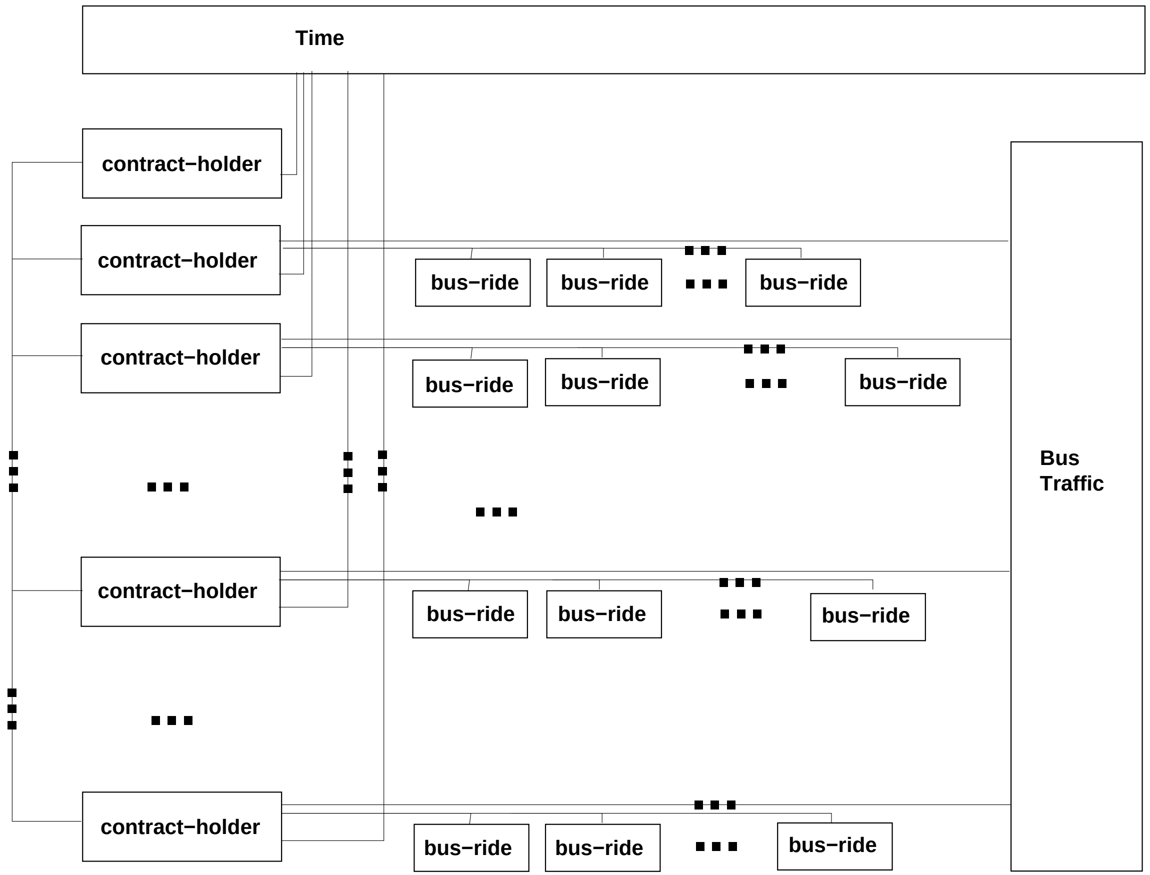

Figure 4.2: An organisation

- The thin lines of Fig. 4.2 denote communication "channels".

[Domain Engineering, Domain Facets]

Human Behaviour

- By **human behaviour** we mean any of a quality spectrum of carrying out assigned work:
  - ⋆ from **careful, diligent** and **accurate**,

  via

  - ⋆ **sloppy** dispatch, and
  - ⋆ **delinquent** work,

  to

  - ⋆ outright **criminal** pursuit.

[Domain Engineering, Domain Facets, Human Behaviour]

Example 22: A Diligent Operation

- The int_Insert operation of Slide 30
  - ⋆ was expressed without stating necessary pre-conditions:

27. 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.
28. 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.
29. If the command is of the form 2oldH(hi′,l,hi′) 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 hi″ must be the identifier of a(nother) hub in hs.
30. 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.

31. 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 !).

value

28′ pre_int_Insert: Ins → N → **Bool**
28″ pre_int_Insert(Ins(op))(hs,ls) ≡
⋆2 s_l(op)∉ ls ∧ obs_LI(s_l(op)) ∉ iols(ls) ∧
case op **of**
29 2oldH(hi′,l,hi″) → {hi′,hi″}⊆iohs(hs),
30 1oldH1newH(hi,l,h) → hi ∈ iohs(hs)∧h∉ hs∧obs_HI(h)∉ iohs(hs),
31 2newH(h′,l,h″) → {h′,h″}∩ hs={}∧{obs_HI(h′),obs_HI(h″)}∩ iohs(hs)={}
end

- These must be **carefully** expressed and adhered to
- in order for staff to be said to carry out the link insertion operation **accurately**.

[Domain Engineering, Domain Facets, Human Behaviour]

Example 23: A Sloppy via Delinquent to Criminal Operation

- We replace systematic checks (∧) with partial checks (∨), etcetera,
- and obtain various degrees of **sloppy** to **delinquent**, or even **criminal** behaviour.

```
value
 28′  pre_int_Insert: Ins → N → Bool
 28″  pre_int_Insert(Ins(op))(hs,ls) ≡
 ⋆2     s_l(op)∉ ls ∧ obs_LI(s_l(op)) ∉ iols(ls) ∧
        case op of
 29        2oldH(hi′,l,hi″) → hi′ ∈ iohs(hs)∨hi″isin iohs(hs),
 30        1oldH1newH(hi,l,h) → hi ∈ iohs(hs)∨h∉ hs∨obs_HI(h)∉ iohs(hs),
 31        2newH(h′,l,h″) → {h′,h″}∩ hs={}∨{obs_HI(h′),obs_HI(h″)}∩ iohs(hs)={}
        end
```

[Domain Engineering, Domain Facets]

Dialectics

- So now you should have a practical and technical "feel" for domain engineering:
  - ⋆ What it takes to express a domain model.
- But there is lots' more: We have not shown you
  - ⋆ (i) the rôle of domain stakeholders:
    - ◇ (i.1) how to identify them,
    - ◇ (i.2) how to involve them and
    - ◇ (i.3) how they help validate resulting domain descriptions.
  - ⋆ (ii) the domain (ii.1) knowledge acquisition and (ii.2) analysis processes,
  - ⋆ (ii) the domain (ii.1) model verification and (ii.2) validation and processes, and
  - ⋆ (iii) the domain theory R&D process.

- Can we agree that we cannot,
  - ⋆ as professional software engineers,
  - ⋆ start on gathering requirements,
  - ⋆ let alone prescribing these
  - ⋆ before we have understood the domain ?
- Can we agree that, “ideally”, we must therefore
  - ⋆ first R&D the domain model
  - ⋆ before we can embark on any requirements prescription process ?
- By “ideally” we mean the following:
  - ⋆ Ideally domain engineering should fully precede requirements engineering,
  - ⋆ but for many practical reasons we must co-develop domain descriptions “hand-in-hand” with requirements prescriptions.
  - ⋆ And that is certainly feasible, when done with care.
  - ⋆ So we shall, for years assume this to be the case.

[Domain Engineering, Domain Facets]

Pragmatics

- While the software industry "humps along":
  - ⋆ co-developing domain descriptions and requirements
  - ⋆ with their clients, or, for COTS, with their marketing departments,
- private and public research centres should and will embark on
  - ⋆ large scale (5–8 manyears/year),
  - ⋆ long range projects (5–8 year)
  - ⋆ foundational research and development (R&D) of

infrastructure component domain models of

- **the financial service industry:**
  - banking (all forms);
  - insurance (all forms);
  - portfolio management;
  - securities trading:
    - brokers,
    - traders,
    - commodities and
    - stock etc. exchanges;
- **transportation:**
  - road,
  - rail,
  - air, and
  - sea;
- **healthcare:**
  - physicians,
  - hospitals,
  - clinics,
  - pharmacies, etc.;
- **"the market":**
  - consumers,
  - retailers,
  - wholesalers, and
  - the supply chain;
- **etcetera.**

Requirements Engineering

- We cannot possibly,
  - ⋆ within the confines of a seminar talk
  - ⋆ and a reasonably sized paper
- cover, however superficially,
  - ⋆ both informal
  - ⋆ and formal

examples of requirements engineering.

- Instead we shall just briefly mention the major stages and sub-stages of requirements modeling:
  - ⋆ **Domain Requirements:** those which can be expressed sôlely using terms from the domain description;
  - ⋆ **Interface Requirements:** those which can be expressed using terms both from the domain description and from IT; and
  - ⋆ **Machine Requirements:** those which can be expressed sôlely using terms from IT.

IEEE Definition of Requirements

- ⋆ By IT requirements we understand (cf. IEEE Standard 610.12):
  - ◇ *"A condition or capability needed by a user to solve a problem or achieve an objective on a computing machine".*

- By computing **machine** we shall understand a, or the, combination of computer (etc.) **hardware** and **software** that is the target for, or result of the required computing systems development.

[Requirements Engineering]

Domain Requirements

Domain Requirements

- By *domain requirements*
  - ⋆ we mean such which can be expressed
  - ⋆ sôlely using terms from the domain description

- To construct the domain requirements
  - ⋆ the domain engineer
  - ⋆ together with the various groups of requirements stakeholder

  "apply" the following "domain-to-requirements" operations to a copy of the domain description:

  - ⋆ **projection**,
  - ⋆ **instantiation**,
  - ⋆ **determination**,
  - ⋆ **extension** and
  - ⋆ **fitting**.

- First we briefly charaterise these.

[Requirements Engineering, Domain Requirements]

The Domain-to-Requirements Operations

- The 'domain-to-requirements' operations cannot be automated.
- They increasingly "turn" the copy of the domain description into a domain requirements prescription.

Projection

removes all the domain phenomena and concepts for which the customer does not need IT support.

Simple Linear Road: Projection

Our requirements is for a simple road: a linear sequence of links and hubs:

type
  N, L, H, LI, HI
value
  obs_Hs: N → H-**set**, obs_Ls: N → L-**set**
  obs_HI: H → HI, obs_LI: L → LI
  obs_HIs: L → HI-**set**, obs_LIs: H → LI-**set**
axiom
  See Items 14–17 Pages 39–39

Instantiation

makes a number of entities: ***simple, operations, events and behaviours***, less abstract, more concrete.

Simple Linear Road: Instantiation

The linear sequence consists of eaxtly 34 links.

```
type
  H, L,
  N′ = H × (L × H)*
  N″ = {|n:N′•wf(n)|}
value
  wf_N″: N′ → Bool
  wf_N″(h,(l,h)^lhl) ≡
    len lhl = 33 ∧
    obs_Hl(l)=obs_Hl(h) ∧
    ∀ i,j:Nat • {i,i+1,j}⊆inds lhl ⇒
      let (li,hi)=lhl(i),(li′,hi′)=lhl(i+1),(lj,hj)=lhl(j) in
      h≠hi∧i≠j⇒li≠lj∧hi≠hj∧
      obs_Hls(li′)={obs_Hl(hi),obs_Hl(hi′)}∧
      obs_Lls(hi)∩ obs_Ll(li)≠{}∧obs_Lls(hi′)∩ obs_Ll(li′)≠{} end
  obs_N: N″ → N
  obs_N(h,lhl) ≡
    ({h}∪{hi|(hi,li):(L×H)•(hi,li)∈ elems lhl},
          {li|(hi,li):(L×H)•(hi,li)∈ elems lhl})
```

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

Determination

makes the emerging requirements entities more determinate.

Simple Linear Road: Determination

All links and all non-end hubs are open in both directions; we leave end-hub states undefined — but see below, under ‘Extension’.

```
type
  LΣ = (HI×HI)-set, LΩ
  HΣ = (LI×LI)-set, HΩ
value
  obs_LΩ: L → LΩ
  obs_HΩ: H → HΩ
axiom
  ∀ (h,⟨(l1,h2)⟩⌢lhl):N″ •
    obs_LΣ(l1)={obs_HI(h),obs_HI(h2)}∧
    ∀ i,i+1:Nat • {i,i+1}⊆inds lhl ⇒
      let (li,hi)=lhl(i),(li′,hi′)=lhl(i+1),(lj,hj)=lhl(j) in
      obs_LΩ(li′)={{(obs_HI(hi),obs_HI(hi′)),(obs_HI(hi′),obs_HI(hi))}}∧
      obs_HΩ(hi)={{(obs_LI(li),obs_LI(li′)),(obs_LI(li′),obs_LI(li))}} end
```

The last two lines of the axiom express that links are always open two ways and that hubs are always open for through traffic.

Extension

introduces new, computable entities that were not possible in the non-IT domain.

Simple Linear Road: Extension

We extend the model of linear roads by introducing the concept of a Hub-Plaza: this is an area "around" each hub from where and into where there is always access onto, respectively from the hub:

type
  HP, HPI
  $H\Sigma' = (LI \times LI)$-**set** $\cup$ $(LI \times HPI)$-**set** $\cup$ $(HPI \times LI)$-**set**
  $H\Omega' = H\Sigma'$-**set**
value
  obs_HΩ': H $\rightarrow$ HΩ'
  obs_HP: H $\rightarrow$ HP
  obs_HPI: HP $\rightarrow$ HPI
axiom
  $\forall$ h,h′:H • h$\neq$h′ $\Rightarrow$ obs_HP(h)$\neq$obs_HP(h′)$\wedge$obs_HPI(obs_HP(h))$\neq$obs_HPI(obs_HP(h′))
  $\forall$ (h,(l,h)^lhl):N″ •
    $\forall$ i,j:**Nat** • {i,i+1,j}$\subseteq$**inds** lhl $\Rightarrow$
      let (li,hi)=lhl(i),(li′,hi′)=lhl(i+1),(lj,hj)=lhl(j) **in**
      obs_HΩ'(h)={{(obs_LI(l),obs_HPI(obs_HP(h))),(obs_HPI(obs_HP(h)),obs_LI(l))}}
      $\forall$ i,i+1:**Nat** • {i,i+1}$\subseteq$**inds** lhl $\Rightarrow$
        let (_,hi)=lhl(i),(_,hi′)=lhl(i+1),(_,hj)=lhl(j) **in**
        obs_HΩ'(hi)={{(obs_LI(li),obs_LI(li′)),(obs_LI(li′),obs_LI(li)),
          (obs_HPI(obs_HP(hi)),obs_LI(li)),(obs_HPI(obs_HP(hi)),obs_LI(li′))
          (obs_LI(li),obs_HPI(obs_HP(hi))),(obs_LI(li′),obs_HPI(obs_HP(hi)))}}
        end end

The obs_HΩ' lines of the axiom with respect to that of 'Determination' express plaza access.

Fitting

merges the domain requirements prescription with those of other IT developments.

• • •

The domain requirements examples are necessarily "microscopic".
The very briefly outlined domain requirements methodology has many fascinating aspects.

[Requirements Engineering]

Interface Requirements

Interface Requirements

- By *interface requirements*
  - ⋆ we mean such which those which can be expressed using terms
  - ⋆ from both the domain description and from IT,
  - ⋆ that is, terminology of hardware and of software.

- When phenomena and concepts of the domain
  - ⋆ are also to be represented by the machine,
  - ⋆ these phenomena and concepts are said to be **shared** between the domain and the machine;
  - ⋆ the requirements therefore need be expressed both
    - ⋄ in terms of phenomena and concepts of the domain and
    - ⋄ in terms of phenomena and concepts of the machine.

[Requirements Engineering, Interface Requirements]

Shared Phenomena and Concepts

- A shared phenomenon or concept is either
  - ⋆ a simple entity,
  - ⋆ an operation,
  - ⋆ an event or
  - ⋆ a behaviour.

- **Shared simple entities** need
  - ⋆ to be initially input to the machine and
  - ⋆ their machine representation need to be
  - ⋆ regularly, perhaps real-time refreshed.
- **Shared operations** need
  - ⋆ to be interactively performed by
  - ⋆ human or other agents of the domain
  - ⋆ and by the machine.

- **Shared events** are shared in the sense that
  - ⋆ their occurrence in the domain (in the machine)
  - ⋆ must be made known to the machine (to the domain).
- **Shared behaviours** need
  - ⋆ to occur in the domain and in the machine
  - ⋆ by alternating means,
  - ⋆ that is, a protocol need be devised.

- For each of these four kinds of interface requirements
  - ⋆ the reqs. engineers work with the reqs. stakeholders
  - ⋆ to determine the properties of these forms of sharing.
- These interface requirements are then narrated and formalised.
- They are always "anchored" in specific items of the domain description.

• • •

The very briefly outlined interface requirements methodology has many fascinating aspects.

[Requirements Engineering]

Machine Requirements

Machine Requirements

- By *machine requirements*
  - ⋆ we mean those which can be expressed
  - ⋆ sôlely using terms from the machine,
  - ⋆ that is, terminology of hardware and of software.

- We shall not cover any principles or techniques for developing machine requirements,
- but shall just list the very many issues that must be captured by a machine requirements.

- Performance
  - ⋆ Storage
  - ⋆ Time
  - ⋆ Software Size
- Dependability
  - ⋆ Accessibility
  - ⋆ Availability
  - ⋆ Reliability
  - ⋆ Robustness
  - ⋆ Safety
  - ⋆ Security
- Maintenance
  - ⋆ Adaptive
  - ⋆ Corrective
  - ⋆ Perfective
  - ⋆ Preventive
- Platform (P)
  - ⋆ Development P
  - ⋆ Demonstration P
  - ⋆ Execution P
  - ⋆ Maintenance P
- Documentation Requirements
- Other Requirements

- The machine requirements are usually not so easily, formalised, if at all, with today's specification language tools.
- Extra great care must therefore be exerted in their narration.
- Some formal modelling calculations, like fault (tree) analysis, can be made in order to justify quantitative requirements.

Why “Current” Requirements Engineering (RE) is Flawed

- Current, conventional requirements engineering has no scientific basis.
  - ⋆ The requirements engineering sketched in this paper starts with a domain model.
  - ⋆ The domain model provides the scientific basis.
  - ⋆ “Derivation” of domain and interface requirement provides a further scientific basis.
  - ⋆ The fact that the requirements engineering models advocated in this paper also are formalised provides a final scientific basis.

[**Why "Current" Requirements Engineering is Flawed**]

- The separation of concerns:
  - ⋆ (the formalised) domain model, in-and-by-itself, and
  - ⋆ the (the formalised) requirements projection, instantiation, determination, extension and fitting operations

  provide a basis for scientific analysis.
- Current, conventional RE does not have these bases.
- If we are to pursue Software Engineering in a professionally responsible manner then requirements engineering must be pursued in a scientifically responsible manner.

Conclusion

Summary — A Wrap Up

- We have illustrated the triptych concept:
  - ⋆ from domains via requirements to software.
- We spent most time on domain engineering.
- We just sketched major requirements engineering concepts.
- And we assumed you know how to turn formal requirements into correct software designs !

[Conclusion]

Dialectics

- So, are we clear on this:
  - ⋆ (i) that we must understand the domain before we express the requirements;
  - ⋆ (ii) that we can "derive" major parts of the requirements prescription from the domain description;
  - ⋆ (iii) that domains are far more "stable" than requirements;
  - ⋆ (iv) that prescribing requirements with no prior domain description is thoroughly unsound;
  - ⋆ (v) that describing [prescribing] domains [requirements] both informally (narratives) and formally (formal specifications) helps significantly towards consistent specifications; and
  - ⋆ (vi) that we must therefore embrace the triptych:
  - ⋆ from domains via requirements to software.

[Conclusion, Dialectics]

Implication: Theory-work

- So, get on with it !
- Pick up one or another of the new
  - ⋆ domain engineering ideas:
    - ◇ business processes,
    - ◇ domain theories,
    - ◇ facets,
    - ◇ etc.,

  or the new
  - ⋆ requirements engineering ideas:
    - ◇ projection,
    - ◇ determination,
    - ◇ fitting,
    - ◇ instantiation,
    - ◇ extension and
- research them, write papers about it.

[Conclusion, Dialectics]

Implication: Engineering-work — Extrovert Applications

- But do it in connection with real life, actual domains:
  - ⋆ banking,
  - ⋆ insurance,
  - ⋆ stock exchange and brokerage,
  - ⋆ hospitalisation,
  - ⋆ bus & tax transport,
  - ⋆ rail transport,
  - ⋆ container line shipping,
  - ⋆ etcetera.
- That is, "build" some impressive domain theories !

[Conclusion, Dialectics]

Implication: Engineering-work — Introspective Applications

- By introspective applications we mean such as providing software for, or such as
  - ⋆ the Internet,
  - ⋆ the Web,
  - ⋆ operating systems
  - ⋆ database management,
  - ⋆ data communication,
  - ⋆ etcetera, etcetera,
- Also these are lack proper domain descriptions.

[Conclusion]

For More on Domain and Requirements Engineering

- For details on domain and requirements engineering we refer to:

  Software Engineering:
  Vol. 3: Domains, Requirements and Software Design, XXX+766 pages.
  Texts in Theoretical Computer Science, EATCS Series, 2006 Springer

  and the upcoming book:

  From Domain to Requirements,
  The Triptych Approach to Software Engineering

- This book (draft) has been and is the basis for lectures at
  - ⋆ (i) Univ. Henri Poincaré/INRIA, Nancy, France, Oct.-Dec. 2007;
  - ⋆ (ii) Techn. Univ. of Graz, Austria, Nov.-Dec. 2008;
  - ⋆ (iii) Univ. of Saarland, Germany, March 2009;
  - ⋆ (iv) Univ. of Tokyo, Japan, Fall (Nov.-Dec.) 2009

[Conclusion]

For More on Extrovert Applications

We refer to some indicative Internet-based reports — from:
www.imm.dtu.dk/~db/

- air traffic: brisbane.pdf and airtraffic.pdf;
- container line industry: container-paper.pdf;
- the 'Market': themarket.pdf;
- IT security: 5lectures/it-system-security-ISO.pdf;
- oil industry and pipelines: de-p.pdf and pipeline.pdf;
- railways: www.railwaydomain.org/;
- transportation (in general): tseb.pdf;
- logistics: logistics.pdf
- et cetera.

[Conclusion]

Software Engineering Archeology

- In general I would prefer to see precise domain models of
  - ⋆ the Internet,
  - ⋆ the Web,
  - ⋆ 'Cloud Computing',
  - ⋆ Windows Vista,
  - ⋆ Linux and
  - ⋆ idealised SQL
- as the basis for
  - ⋆ requirements and
  - ⋆ software
- that claim that they are "based" on
  - ⋆ the Internet,
  - ⋆ the Web,
  - ⋆ 'Cloud Computing',
  - ⋆ Windows Vista,
  - ⋆ Linux and/or
  - ⋆ SQL.
- Here is clearly a fascination engineering task.
- I see the **Internet** as an instantiation of **'Cloud Computing'**.

[Conclusion]

For More on Research Topics

- A number of research topics of domain theory has been outlined in:

  Domain Theory: Practice and Theories, Discussion of Possible Research Topics. In *ICTAC'2007*, volume 4701 of *Lecture Notes in Computer Science (eds. J.C.P. Woodcock et al.)*, pages 1–17, Heidelberg, September 2007. Springer.

- Excursions in 'Philosophy of Informatics' are covered in:

  On Mereologies in Computing Science. Festschrift for Tony Hoare, Springer UK, History of Computing (ed. Bill Roscoe), 2009

  An Emerging Domain Science – A Rôle for Stanisław Leśniewski's Mereology and Bertrand Russell's Philosophy of Logical Atomism. Higher-order and Symbolic Computation, Fall 2009

Acknowledgements

- Thanks to Alan Bundy for inviting me to Edinburgh.