Software
Engineering 1

Software
Engineering 2

Software
Engineering 3

WELCOME

Domain Science & Engineering
A Precursor for Requirements Engineering

Dines Bjørner

DTU Informatics

September 5, 2012: 11:29

Lecture Schedule

• Lecture 1:	09:00–09:40 + 09:50–10:30
• Lecture 2:	11:00–11:40 + 11:50–12:30
• Lecture 3:	14:00–14:40 + 14:50–15:30
• Lecture 4:	16:00–16:40 + 16:50–17:30

Lecture 1: 9:00–9:40 + 9:50–10:30
Introduction and Main Example

Summary

- This seminar covers
  - ◈ a **new science & engineering of domains** as well as
  - ◈ a **new foundation for software development**.

We treat the latter first.

- Instead of commencing with requirements engineering,
  - whose pursuit may involve repeated,
  - but unstructured forms of domain analysis,
  - we propose a predecessor phase of domain engineering.
- That is, we single out domain analysis as an activity to be pursued prior to requirements engineering.

- In emphasising **domain engineering** as a predecessor phase
  - we, at the same time, introduce a number of facets
  - that are **not present,** we think,
  - in current software engineering studies and practices.
- **One facet** is **the construction of separate domain descriptions.**
  - Domain descriptions are void of any reference to requirements
  - and encompass the **modelling** of **domain phenomena**
  - without regard to their being computable.

- **Another facet** is **the pursuit of domain descriptions as a free-standing activity.**
  - In this seminar we emphasize domain description development need not lead to software development.
  - This gives a new meaning to business process engineering, and should lead to
    - a deeper understanding of a domain
    - and to possible non-IT related business process re-engineering of areas of that domain.

- In this seminar we shall investigate
  - a method for analysing domains,
  - for constructing domain descriptions
  - and some emerging scientific bases.

- **Our contribution** to **domain analysis** is
  - that we view domain analysis
  - as a variant of formal concept analysis [Wille:ConceptualAnalysis1999],
    - a contribution which can be formulated by the "catch phrase"
    - **domain entitities and their qualities form Galois connections**,
  - and further contribute with a methodology of
  - necessary corresponding principles and techniques of domain analysis.

- Those corresponding principles and techniques hinge on our view of domains as having the following **ontology**.
  - There are the **entities** that we can describe and then there is "the rest" which we leave un-described.
  - We **analyse** entities into
    - **endurant entities** and
    - **perdurant entities** ,

    that is,
    - **parts** and **materials** as **endurant entities** and
    - **discrete actions**, **discrete events** and **behaviours** as **perdurant entities** , respectively.
- Another way of looking at **entities** is as
  - **discrete entities** , or as
  - **continuous entities**.

- We also contribute to the analysis of discrete endurants in terms of the following notions:
  - part types and material types,
  - part unique identifiers,
  - part mereology and
  - part attributes and material attributes and
  - **material laws**.
- Of the above we point to the introduction, into computing science and software engineering of the notions of
  - materials and
  - continuous behaviours

  as novel.

- The example formalisations are expressed in
  - `RAISE` [RaiseMethod] (with [TheSEBook1wo,TheSEBook2wo,TheSEBook3eps] being a rather comprehensive monograph cum textbook),
- but could as well have been expressed in, for example,
  - `Alloy` [alloy],
  - `Event B` [JRAbrial:TheBBooks] ,
  - `VDM` [e:db:Bj78bwo,e:db:Bj82b,JohnFitzgerald+PeterGormLarsen] or
  - `Z` [m:z:jd+jcppw96].

1. Introduction

- This is primarily a **methodology** paper.
- By a **method**$_\delta$ we shall understand
  - a set of **principles**
  - for **selecting** and **applying**
  - a number of **techniques** and **tools**
  - in order to **analyse** a **problem**
  - and **construct** an **artefact**.
- By **methodology**$_\delta$ we shall understand
  - the study and knowledge about methods.

- This seminar contributes to
  - ◈ the study and knowledge
  - ◈ of software engineering development methods.
- Its contributions are those of suggesting and exploring
  - ◈ **domain engineering** and
  - ◈ domain engineering as a basis for **requirements engineering**.
- We are not saying
  - ◈ *“thou must develop software this way”*,
- but we do suggest
  - ◈ that since it is possible
  - ◈ and makes sense to do so
  - ◈ it may also be wise to do so.

1.1. Domains: Some Definitions

- By a **domain**$_\delta$ we shall here understand
  - ◈ an area of human activity
  - ◈ characterised by observable phenomena:
    - ⦾ **entities**
      - ∗ whether **endurants** (manifest **parts** and **materials**)
      - ∗ or **perdurants** (**actions**, **events** or **behaviours**),
    - ⦾ whether
      - ∗ **discrete** or
      - ∗ **continuous**;
    - ⦾ and of their **properties**.

Example: 1 **Some Domains** Some examples are:

- air traffic,
- airport,
- banking,
- consumer market,
- container lines,
- fish industry,
- health care,
- logistics,
- manufacturing,
- pipelines,
- securities trading,
- transportation
- etcetera. ■

1.1.1. Domain Analysis

- By **domain analysis**$_\delta$ we shall understand
  - ◈ an inquiry into the domain,
  - ◈ its **entities**
  - ◈ and their **properties**.

Example: 2 A Container Line Analysis.

We omit enumerating entity properties.

- *parts:*
  - container,
  - vessel,
  - terminal port, etc.;
- *actions:*
  - container loading,
  - container unloading,
  - vessel arrival in port, etc.;
- *events:*
  - container falling overboard;
  - container afire;
  - etc.;
- *behaviour:*
  - vessel voyage,
  - across the seas,
  - visiting ports, etc.

Length of a container is a container *property*.
Name of a vessel is a vessel *property*.
Location of a container terminal port is a port *property*.

1.1.2. Domain Descriptions

- By a domain description$_\delta$ we shall understand
  - a narrative description
  - tightly coupled (say line-number-by-line-number)
  - to a formal description.
- To develop a domain description
  requires a thorough amount of domain analysis.

Example: 3 A Transport Domain Description.

- *Narrative:*
  - a transport net, n:N,
    consists of an aggregation of hubs, hs:HS,
    which we "concretise" as a set of hubs, H-**set**, and
    an aggregation of links, ls:LS, that is, a set L-**set**,
- *Formalisation:*
  - **type** N, HS, LS, Hs = H-**set**, Ls = L-**set**, H, L
    value
    obs_HS: N→HS,
    obs_LS: N→LS.
    obs_Hs: HS→H-**set**,
    obs_Ls: LS→L-**set**.

■

1.1.3. Domain Engineering

- By **domain engineering**$_\delta$ we shall understand
  - the **engineering** of a domain description,
  - that is,
    - the rigorous construction of domain descriptions, and
    - the further analysis of these, creating **theories of domains**.

- The size, structure and complexity of interesting domain descriptions is usually such as to put a special emphasis on engineering:
  - the management and organisation of several, typically 5–6 collaborating domain describers,
  - the ongoing check of description quality, completeness and consistency, etcetera.

1.1.4. Domain Science

- By domain science$_\delta$ we shall understand
  - two things:
    - the general study and knowledge of
      - how to create and handle domain descriptions
      - (a general theory of domain descriptions)

      and
    - the specific study and knowledge of a particular domain.
  - The two studies intertwine.

1.2. The Triptych of Software Development

- We suggest a "dogma":
  - before **software** can be **designed** one must understand[1] the **requirements**; and
  - before **requirements** can be expressed one must understand[2] the **domain**.
- We can therefore view software development as ideally proceeding in three (i.e., `TripTych`) phases:
  - an initial phase of **domain engineering**, followed by
  - a phase of **requirements engineering**, ended by
  - a phase of **software design**.

[1]Or maybe just: have a reasonably firm grasp of
[2]See previous footnote!

- In the **domain engineering phase** ($\mathcal{D}$)
  - a domain is analysed, described and "theorised",
  - that is, the beginnings of a **specific domain theory** is established.
- In the **requirements engineering phase** ($\mathcal{R}$)
  - a **requirements prescription** is constructed —
  - significant fragments of which are "derived",
  - systematically, from the **domain description**.
- In the **software design phase** ($\mathcal{S}$)
  - a **software design**
  - is derived, systematically, rigorously or formally,
  - from the **requirements prescription**.
- Finally the $\mathcal{S}$oftware is proven correct with respect to the $\mathcal{R}$equirements under assumption of the $\mathcal{D}$omain: $\mathcal{D}, \mathcal{S} \models \mathcal{R}$.

- By a **machine**$_\delta$ we shall understand the **hardware** and **software** of a target, i.e., a required **IT system**.
- In `[dines:ugo65:2008,psi2009,Kiev:2010ptI]` we indicate how one can "derive" significant parts of requirements from a suitably comprehensive domain description – basically as follows.
  - **Domain projection**: from a domain description one **project**s those areas that are to be somehow manifested in the software.
  - **Domain initialisation**: for that resulting projected requirements prescription one **initialise**s a number of part types as well as action and behaviour definitions, from less **abstract** to more **concrete**, specific types, respectively definitions.

  - **Domain determination:** hand-in-hand with domain initialisation a[n interleaved] stage of making values of types less **non-deterministic**, i.e., more **deterministic**, can take place.
  - **Domain extension:** Requirements often arise in the context of new business processes or technologies either placing old or replacing human processes in the domain. Domain extension is now the 'enrichment' of the domain requirements, so far developed, with the description of these new business processes or technologies.
  - Etcetera.
- The result of this part of "**requirements derivation**" is the **domain requirements**.

- A set of domain-to-requirements operators similarly exists for constructing **interface requirements**
  - from the domain description and,
  - independently, also from knowledge of the **machine**
  - for which the required IT system is to be developed.
- We illustrate the **techniques** of **domain requirements** and **interface requirements** in Sect. 8.
- Finally **machine requirements** are “derived”
  - from just the knowledge of the **machine**,
  - that is,
    - the target hardware and
    - the software system tools for that hardware.

- When you review this section ('A Triptych of Software Development')
  - then you will observe how 'the domain'
  - predicates both the requirements
  - and the software design.
- For a specific domain one may develop
  - many (thus related) requirements
  - and from each such (set of) requirements
  - one may develop many software designs.
- We may characterise this multitude of domain-predicated requirements and designs as a **product line** [dines-maurer].
- You may also characterise domain-specific developments as representing another 'definition' of **domain engineering**.

1.3. Issues of Domain Science & Engineering

- We specifically focus on the following issues of domain science &[3] engineering:
  - (i) which are the “things” to be described[4],
  - (ii) how to analyse these “things” into description structures[5],
  - (iii) how to describe these “things” informally and formally,
  - (iv) how to further structure descriptions[6], and a further study of
  - (v) **mereology**[7].

[3]When we put ‘&’ between two terms that the compound term forms a whole concept.

[4]**endurants** [manifest entities henceforth called **parts** and **materials**] and **perdurants** [**actions**, **events**, **behaviours**]

[5]**atomic** and **composite**, **unique identifiers**, **mereology**, **attributes**

[6]*intrinsics, support technology, rules & regulations, organisation & management, human behaviour* etc.

[7]the study and knowledge of parts and relations of parts to other parts and a “whole”.

1.4. Structure of Paper

- First (Sect. 1) we introduce the problem. And that was done above.
- Then, in (Sects. 4–6)
  - we bring a rather careful analysis of
  - the concept of the **observable, manifest phenomena**
  - that we shall refer to as **entities**.
- We strongly think that these sections of this seminar
  - brings, to our taste, a simple and elegant
  - reformulation of what is usually called *"data modelling"*,
  - in this case for domains —
  - but with major aspects applicable as well to
  - **requirements development** and **software design**.

- That analysis focuses on
  - **endurant entities**, also called **parts** and **materials**,
    - those that can be observed at no matter what time,
    - i.e., entities of substance or continuant, and
  - **perdurant entities**: **action**, **event** and **behaviour** entities, those
    - that occur,
    - that happen,
    - that, in a sense, are accidents.

- **We think** that this "decomposition" of the "data analysis" problem into
  - discrete parts and continuous materials,
  - atomic and composite parts,
  - their unique identifiers and mereology, and
  - their attributes
  - **is novel**,
  - and differs from past practices in domain analysis.

- In Sect. 7 we suggest
  - for each of the entity categories
    - parts,
    - materials,
    - actions,
    - events and
    - behaviours,
  - a calculus of meta-functions:
    - analytic functions,
      - that guide the domain description developer
      - in the process of selection,

      and
    - so-called discovery functions,
      - that guide that person
      - in “generating” appropriate domain description texts, informal and formal.

- The domain description calculus is to be thought of
  - as directives to the domain engineer,
  - mental aids that help a team of domain engineers
  - to steer it simply through the otherwise daunting task
  - of constructing a usually large domain description.
- Think of the calculus
  - as directing
  - a human calculation
  - of domain descriptions.
- Finally the domain description calculus section
  - suggests a number of laws that the
  - domain description process ought satisfy.

- In Sect. 8 we bring a brief survey of the kind of **requirements engineering**
  - that one can now pursue based on a reasonably comprehensive **domain description**.
  - We show how one can systematically, but not automatically
  - "derive" significant fragments
    - of **requirements prescriptions**
    - from **domain descriptions**.

- The formal descriptions will here be expressed in the RAISE [RaiseMethod] Specification Language, RSL.
- We otherwise refer to [TheSEBook1wo].
- Appendix C of the tutorial notes brings a short primer, mostly on the syntactic aspects of RSL.
- But other model-oriented formal specification languages can be used with equal success; for example:
  - Alloy [alloy],
  - Event B [JRAbrial:TheBBooks] ,
  - VDM [e:db:Bj78bwo,e:db:Bj82b,JohnFitzgerald+PeterGormLarsen] and
  - Z [m:z:jd+jcppw96].

2. The Main Example – Example 3: Road Traffic System

- The main example presents a terse narrative and formalisation of a road traffic domain.
  - Since the example description conceptually covers also major aspects of
    - railroad nets,
    - shipping nets, and
    - air traffic nets,
  - we shall use such terms as hubs and links to stand for
    - road (or street) intersection and road (or street) segments,
    - train stations and rail lines,
    - harbours and shipping lanes, and
    - airports and air lanes.

2.1. Parts

2.1.1. Root Sorts

- The domain,
  - the stepwise unfolding of
  - whose description is
  - to be exemplified,

  is that of a composite traffic system
  - with a road net,
  - with a fleet of vehicles
  - of whose individual position on the road net we can speak, that is, monitor.

1. We analyse the composite traffic system into
   a a composite road net,
   b a composite fleet (of vehicles), and
   c an atomic monitor.

```
type
1.       Δ
1(a).    N
1(b).    F
1(c).    M
value
1(a).    obs_N: Δ → N
1(b).    obs_F: Δ → F
1(c).    obs_M: Δ → M
```

2.1.2. Sub-domain Sorts and Types

2. From the road net we can observe

   a a composite part, HS, of road (i.e., street) intersections (hubs) and

   b an composite part, LS, of road (i.e., street) segments (links).

type
2. HS, LS
value
2(a). **obs_**HS: N $\to$ HS
2(b). **obs_**LS: N $\to$ LS

3. From the fleet sub-domain, F, we observe a composite part, VS, of vehicles

type
3. VS
value
3. $\underline{\textbf{obs_}}$VS: F $\rightarrow$ VS

4. From the composite sub-domain VS we observe

   a the composite part Vs, which we concretise as a set of vehicles

   b where vehicles, V, are considered atomic.

type
4(a). Vs = V-**set**
4(b). V
value
4(a). **obs_**Vs: VS → V-**set**

- The "monitor" is considered atomic; it is an abstraction of the fact that
  - we can speak of the positions of each and every vehicle on the net
  - without assuming that we can indeed pin point these positions
  - by means of for example sensors.

2.1.3. Further Sub-domain Sorts and Types

- We now analyse the sub-domains of HS and LS.

5. From the hubs aggregate we decide to observe
   a the concrete type of a set of hubs,
   b where hubs are considered atomic; and
6. from the links aggregate we decide to observe
   a the concrete type of a set of links,
   b where links are considered atomic;

```
type
5(a). Hs = H-set
6(a). Ls = L-set
5(b). H
6(b). L
value
5. obs_Hs: HS → H-set
6. obs_Ls: LS → L-set
```

- We have no composite parts left to further analyse into parts
  - whether they be again composite
  - or atomic.
- That is,
  - at various, what we shall refer to as, **domain indexes**
  - we have discovered the following part types:

$\langle\Delta\rangle$:	N, F, M	$\langle\Delta, HS\rangle$:	Hs, H
$\langle\Delta, N\rangle$:	HS, LS	$\langle\Delta, LS\rangle$:	Ls, L
$\langle\Delta, F\rangle$:	VS	$\langle\Delta, VS\rangle$:	Vs, V

  - Thus we have ended up with atomic parts.

2.2. Properties

- Parts are distinguished by their properties:
  - the types and
  - the values

  of these.
- We consider three kinds of properties:
  - unique identifiers,
  - mereology and
  - attributes.

2.2.1. Unique Identifications

7. We decide the following:

   a each hub has a unique hub identifier,

   b each link has a unique link identifier and

   c each vehicle has a unique vehicle identifier.

type

7(a). HI

7(b). LI

7(c). VI

value

7(a). <u>**uid**_</u>H: H → HI

7(b). <u>**uid**_</u>L: L → LI

7(c). <u>**uid**_</u>V: V → VI

2.2.2. Mereology

2.2.2.1 Road Net Mereology

- By *mereology* we mean the study, knowledge and practice of understanding parts and part relations.

8. Each link is connected to exactly two hubs, that is,

   a from each link we can observe its mereology, that is, the identities of these two distinct hubs,

   b and these hubs must be of the net of the link;

9. and each hub is connected to zero, one or more links, that is,

   a from each hub we can observe its mereology, that is, the identities of these links,

   b and these links must be of the net of the hub.

value
8(a). $\underline{\textbf{mereo\_}}\mathrm{L}$: L $\rightarrow$ HI-**set**, **axiom** $\forall$ l:L·**card** $\underline{\textbf{mereo\_}}\mathrm{L}(l)=2$
axiom
8(b). $\forall$ n:N,l:L,hi:HI · l $\in$ $\underline{\textbf{obs\_}}\mathrm{Ls}(\underline{\textbf{obs\_}}\mathrm{LS}(n)) \wedge hi \in \underline{\textbf{mereo\_}}\mathrm{L}(l)$
8(b). $\Rightarrow \exists$ h:H·h $\in \underline{\textbf{obs\_}}\mathrm{Hs}(\underline{\textbf{obs\_}}\mathrm{HS}(n)) \wedge \underline{\textbf{uid\_}}\mathrm{H}(h)=hi$
value
9(a). $\underline{\textbf{mereo\_}}\mathrm{H}$: H $\rightarrow$ LI-**set**
axiom
9(b). $\forall$ n:N,h:H,li:LI · h $\in \underline{\textbf{obs\_}}\mathrm{Hs}(\underline{\textbf{obs\_}}\mathrm{HS}(n)) \wedge li \in \underline{\textbf{mereo\_}}\mathrm{H}(h)$
9(b). $\Rightarrow \exists$ l:L·l $\in \underline{\textbf{obs\_}}\mathrm{Ls}(\underline{\textbf{obs\_}}\mathrm{LS}(n)) \wedge \underline{\textbf{uid\_}}\mathrm{L}(l)=li$

2.2.2.2 Fleet of Vehicles Mereology

- In the traffic system that we are building up
  - there are no relations to be expressed between vehicles,
  - only between vehicles and the (single and only) monitor.
- Thus there is no mereology needed for vehicles.

2.2.3. Attributes

- We shall model attributes of
  - links,
  - hubs and
  - vehicles.
- The composite parts,
  - aggregations of hubs, HS and Hs,
  - aggregations of links, LS and Ls and
  - aggregations of vehicles, VS and Vs,

  also have attributes, but we shall omit modelling them here.

2.2.3.1 Attributes of Links

10. The following are attributes of links.

  a Link states, lσ:LΣ, which we model as possibly empty sets of pairs of distinct identifiers of the connected hubs.

  - A link state expresses the directions that are open to traffic across a link.

  b Link state spaces, lω:LΩ which we model as the set of link states.

  - A link state space expresses the states that a link may attain across time.

  c Further link attributes are length, location, etcetera.

- Link states are usually dynamic attributes
- whereas
  - link state spaces,
  - link length and
  - link location (usually some curvature rendition)

  are considered static attributes.

type
10(a). $\mathrm{L\Sigma} = (\mathrm{HI} \times \mathrm{HI})$**-set**
axiom
10(a). $\forall\ \mathrm{l\sigma{:}L\Sigma} \cdot 0 \leq$ **card** $\mathrm{l\sigma} \leq 2$
value
10(a). $\underline{\textbf{attr\_}}\mathrm{L\Sigma}$: $\mathrm{L} \to \mathrm{L\Sigma}$
axiom
10(a). $\forall\ \mathrm{l{:}L} \cdot$ **let** $\{\mathrm{hi,hi'}\}=\underline{\textbf{mereo\_}}\mathrm{L(l)}$ **in** $\underline{\textbf{attr\_}}\mathrm{L\Sigma(l)}\subseteq\{(\mathrm{hi,hi'}),(\mathrm{hi',hi})\}$ **end**
type
10(b). $\mathrm{L\Omega} = \mathrm{L\Sigma}$**-set**
value
10(b). $\underline{\textbf{attr\_}}\mathrm{L\Omega}$: $\mathrm{L} \to \mathrm{L\Omega}$
axiom
10(b). $\forall\ \mathrm{l{:}L} \cdot$ **let** $\{\mathrm{hi,hi'}\}=\underline{\textbf{mereo\_}}\mathrm{L(l)}$ **in** $\underline{\textbf{attr\_}}\mathrm{L\Sigma(l)}\in \underline{\textbf{attr\_}}\mathrm{L\Omega(l)}$ **end**
type
10(c). LOC, LEN, ...
value
10(c). $\underline{\textbf{attr\_}}\mathrm{LOC}$: $\mathrm{L} \to \mathrm{LOC}$, $\underline{\textbf{attr\_}}\mathrm{LEN}$: $\mathrm{L} \to \mathrm{LEN}$, ...

2.2.3.2 Attributes of Hubs

11. The following are attributes of hubs:
   a Hub states, hσ:HΣ, which we model as possibly empty sets of pairs of identifiers of the connected links.
      - A hub state expresses the directions that are open to traffic across a hub.
   b Hub state spaces, hω:HΩ which we model as the set of hub states.
      - A hub state space expresses the states that a hub may attain across time.
   c Further hub attributes are location, etcetera.

- Hub states are usually dynamic attributes
- whereas
   - hub state spaces and
   - hub location

  are considered static attributes.

type
11(a). $\mathrm{H\Sigma} = (\mathrm{LI} \times \mathrm{LI})$**-set**
value
11(a). **attr**_HΣ: $\mathrm{H} \rightarrow \mathrm{H\Sigma}$
axiom
11(a). $\forall$ h:H $\cdot$ **attr**_HΣ(h)$\subseteq\{(\mathrm{li},\mathrm{li}')|\mathrm{li},\mathrm{li}':\mathrm{LI}\cdot\{\mathrm{li},\mathrm{li}'\}\subseteq$**mereo**_H(h)$\}$
type
11(b). $\mathrm{H\Omega} = \mathrm{H\Sigma}$**-set**
value
11(b). **attr**_HΩ: $\mathrm{H} \rightarrow \mathrm{H\Omega}$
axiom
11(b). $\forall$ h:H $\cdot$ **attr**_HΣ(h) $\in$ **attr**_HΩ(h)
type
11(c). LOC, ...
value
11(c). **attr**_LOC: $\mathrm{L} \rightarrow \mathrm{LOC}$, ...

2.2.3.3 Attributes of Vehicles

12. Dynamic attributes of vehicles include
    a position
      i. at a hub (about to enter the hub — referred to by the link it is coming from, the hub it is at and the link it is going to, all referred to by their unique identifiers or
      ii. some fraction "down" a link (moving in the direction from a from hub to a to hub — referred to by their unique identifiers)
      iii. where we model fraction as a real between 0 and 1 included.
    b velocity, acceleration, etcetera.
13. All these vehicle attributes can be observed.

type

12(a). $\mathrm{VP} = \mathrm{atH} \mid \mathrm{onL}$

12((a))i. $\mathrm{atH} :: \mathrm{fli{:}LI} \times \mathrm{hi{:}HI} \times \mathrm{tli{:}LI}$

12((a))ii. $\mathrm{onL} :: \mathrm{fhi{:}HI} \times \mathrm{li{:}LI} \times \mathrm{frac{:}FRAC} \times \mathrm{thi{:}HI}$

12((a))iii. $\mathrm{FRAC} = \mathbf{Real}$, $\mathbf{axiom}\ \forall\ \mathrm{frac{:}FRAC} \cdot 0 \leq \mathrm{frac} \leq 1$

12(b). VEL, ACC, ...

value

13. $\underline{\mathbf{attr\_}}\mathrm{VP{:}V{\rightarrow}VP}$, $\underline{\mathbf{attr\_}}\mathrm{onL{:}V{\rightarrow}onL}$, $\underline{\mathbf{attr\_}}\mathrm{atH{:}V{\rightarrow}atH}$

13. $\underline{\mathbf{attr\_}}\mathrm{VEL{:}V{\rightarrow}VEL}$, $\underline{\mathbf{attr\_}}\mathrm{ACC{:}V{\rightarrow}ACC}$

2.2.3.4 Vehicle Positions

14. Given a net, n:N, we can define the possibly infinite set of potential vehicle positions on that net, vps(n).
    a vps(n) is expressed in terms of the links and hubs of the net.
    b vps(n) is the
    c union of two sets:
       i. the potentially[8] infinite set of "on link" positions
       ii. for all links of the net

       and

       i. the finite set of "at hub" positions
       ii. for all hubs in the net.

[8]The 'potentiality' arises from the nature of FRAC. If fractions are chosen as, for example, 1/5'th, 2/5'th, ..., 4/5'th, then there are only a finite number of "on link" vehicle positions. If instead fraction are arbitrary infinitesimal quantities, then there are infinitely many such.

```
value
14.      vps: N → VP-infset
14(b).   vps(n) ≡
14(a).       let ls=obs_Ls(obs_LS(n)), hs=obs_Hs(obs_HS(n)) in
14((c))i.      { onL(fhi,uid(l),f,thi) | fhi,thi:HI,l:L,f:FRAC ·
14((c))ii.             l ∈ ls ∧ {fhi,thi}=mereo_L(l) }
14(c).        ∪
14((c))i.      { atH(fli,uid_H(h),tli) | fli,tli:LI,h:H ·
14((c))ii.             h ∈ hs ∧ {fli,tli}⊆mereo_H(h) }
14(a).       end
```

- Given a net and a finite set of vehicles
  - ⟐ we can distribute these over the net, i.e., assign initial vehicle positions,
  - ⟐ so that no two vehicles "occupy" the same position, i.e., are "crashed" !
- Let us call the non-deterministic assignment function, i.e., a relation, for vpr.

15. vpm:VPM is a bijective map from vehicle identifiers to (distinct) vehicle positions.
16. vpr has the obvious signature.
17. vpr(vs)(n) is defined in terms of
18. a non-deterministic selection, vpa, of vehicle positions, and
19. a non-deterministic assignment of these vehicle positions to vehicle identifiers —
20. being the resulting distribution.

```
type
15.  $\mathrm{VPM}' = \mathrm{VI} \xrightarrow[m]{} \mathrm{VP}$
15.  $\mathrm{VPM} = \{|\ \mathrm{vpm}{:}\mathrm{VPM}' \cdot \mathbf{card\ dom}\ \mathrm{vpm} = \mathbf{card\ rng}\ \mathrm{vpm}\ |\}$
value
16.  $\mathrm{vpr}{:}\ \mathrm{V}\text{-}\mathbf{set} \times \mathrm{N} \to \mathrm{VMP}$
17.  $\mathrm{vpr(vs)(n)} \equiv$
18.      $\mathbf{let}\ \mathrm{vpa}{:}\mathrm{VP}\text{-}\mathbf{set} \cdot \mathrm{vpa} \subseteq \mathrm{vps(vs)(n)} \wedge \mathbf{card}\ \mathrm{vpa} = \mathrm{vard\ vs}\ \mathbf{in}$
19.      $\mathbf{let}\ \mathrm{vpm}{:}\mathrm{VPM} \cdot \mathbf{dom}\ \mathrm{vpm} = \mathrm{vps} \wedge \mathbf{rng}\ \mathrm{vpm} = \mathrm{vpa}\ \mathbf{in}$
20.      $\mathrm{vpm}\ \mathbf{end\ end}$
```

2.3. Definitions of Auxiliary Functions

21. From a net we can extract all its link identifiers.
22. From a net we can extract all its hub identifiers.

value

21. xtr_LIs: N → LI-**set**
21. xtr_LIs(n) ≡ {**uid**_L(l)|l:L•l ∈ **obs**_Ls(**obs**_LS(n))}
22. xtr_HIs: N → HI-**set**
22. xtr_HIs(n) ≡ {**uid**_H(l)|h:H•h ∈ **obs**_Hs(**obs**_HS(n))}

23. Given a link identifier and a net get the link with that identifier in the net.
24. Given a hub identifier and a net get the hub with that identifier in the net.

value

26. $\text{get\_H: HI} \rightarrow \text{N} \overset{\sim}{\rightarrow} \text{H}$
26. $\text{get\_H(hi)(n)} \equiv \iota\, \text{h:H}\cdot\text{h} \in \underline{\textbf{obs\_}}\text{Hs(}\underline{\textbf{obs\_}}\text{HS(n))}\wedge\underline{\textbf{uid\_}}\text{H(h)=hi}$
26. $\quad\textbf{pre}\text{: hi} \in \text{xtr\_HIs(n)}$

26(a). $\text{get\_L: LI} \rightarrow \text{N} \overset{\sim}{\rightarrow} \text{L}$
26(a). $\text{get\_L(li)(n)} \equiv \iota\, \text{l:L}\cdot\text{l} \in \underline{\textbf{obs\_}}\text{Ls(}\underline{\textbf{obs\_}}\text{LS(n))}\wedge\underline{\textbf{uid\_}}\text{L(l)=li}$
26(a). $\quad\textbf{pre}\text{: hl} \in \text{xtr\_LIs(n)}$

- The $\iota\, \mathsf{a{:}A}\cdot\mathcal{P}(\mathsf{a})$ expression
  - yields the unique value $\mathsf{a{:}A}$
  - which satisfies the predicate $\mathcal{P}(\mathsf{a})$.
  - If none, or more than one exists then the function is undefined.

2.4. Some Derived Traffic System Concepts

2.4.1. Maps

25. A road map is an abstraction of a road net. We define one model of maps below.

    a A road map, RM, is a finite definition set function, M, (a specification language map) from
      - hub identifiers (the source hub)
      - to (such finite definition set) functions
      - from link identifiers
      - to hub identifiers (the target hub).

type

25(a). $\mathrm{RM}' = \mathrm{HI} \underset{m}{\rightarrow} (\mathrm{LI} \underset{m}{\rightarrow} \mathrm{HI})$

- If a hub identifier in the source or an rm:RM maps into the empty map then the "corresponding" hub is "isolated": has no links emanating from it.

26. These road maps are subject to a well-formedness criterion.

  a The target hubs must be defined also as source hubs.

  b If a link is defined from source hub (referred to by its identifier) shi via link li to a target hub thi, then, vice versa, link li is also defined from source thi to target shi.

```
type
26.    RM = {| rm:RM′ · wf_RM(rm) |}
value
26.    wf_RM: RM′ → Bool
26.    wf_RM(rm) ≡
26(a).        ∪ { rng(rm(hi))|hi:HI·hi ∈ dom rm } ⊆ dom rm
26(b).     ∧ ∀ shi:HI·shi ∈ dom rm ⇒
26(b).          ∀ li:LI · li ∈ dom rm(shi) ⇒
26(b).             li ∈ dom rm((rm(shi))(li)) ∧ (rm((rm(shi))(li)))(li)=shi
```

27. Given a road net, n, one can derive “its” road map.
    a Let hs and ls be the hubs and links, respectively of the net n.
    b Every hub with no links emanating from it is mapped into the empty map.
    c For every link identifier uid_L(l) of links, l, of ls and every hub identifier, hi, in the mereology of l
    d hi is mapped into a map from uid_L(l) into hi’
    e where hi’ is the other hub identifier of the mereology of l.

```
value
27.     derive_RM: N → RM
27.     derive_RM(n) ≡
27(a).       let hs = obs_Hs(obs_HS(n)), ls = obs_Ls(obs_LS(n)) in
27(b).       [ hi ↦ [ ] | hi:HI · ∃ h:H · h ∈ hs ∧ mereo_H(h) = {} ] ∪
27(d).       [ hi ↦ [ uid_L(l) ↦ hi′
27(e).                          | hi′:HI · hi′ = mereo_L(l)\{hi} ]
27(c).                | l:L,hi:HI · l ∈ ls ∧ hi ∈ mereo_L(l) ] end
```

- **Theorem:** If the road net, n, is well-formed then wf_RM(derive_RM(n)).

2.4.2. Traffic Routes

28. A traffic route, tr, is an alternating sequence of hub and link identifiers such that

    a li:LI is in the mereology of the hub, h:H, identified by hi:HI, the predecessor of li:LI in route r, and

    b hi':HI, which follows li:LI in route r, is different from hi, and is in the mereology of the link identified by li.

```
type
28.    R′ = (HI|LI)*
28.    R = {| r:R′ • ∃ n:N • wf_R(r)(n) |}
value
28.    wf_R: R′ → N → Bool
28.    wf_R(r)(n) ≡
28.        ∀ i:Nat • {i,i+1}⊆inds r ⇒
28(a).          is_HI(r(i)) ⇒ is_LI(r(i+1)) ∧ r(i+1) ∈ mereo_H(get_H(r(i))(n)),
28(b).          is_LI(r(i)) ⇒ is_HI(r(i+1)) ∧  r(i+1) ∈ mereo_L(get_L(r(i))(n))
```

29. From a well-formed road map (i.e., a road net) we can generate the possibly infinite set of all routes through the net.

   a **Basis Clauses:**

      i. The empty sequence of identifiers is a route.

      ii. The one element sequences of link and hub identifiers of links and hubs of a road map (i.e., a road net) are routes.

      iii. If hi maps into some li in rm then $\langle$hi,li$\rangle$ and $\langle$li,hi$\rangle$ are routes of the road map (i.e., of the road net).

   b **Induction Clause:**

      i. Let r$\widehat{\ }\langle$i$\rangle$ and $\langle$i′$\rangle\widehat{\ }$r′ be two routes of the road map.

      ii. If the identifiers i and i′ are identical, then r$\widehat{\ }\langle$i$\rangle\widehat{\ }$r′ is a route.

   c **Extremal Clause:**

      i. Only such routes that can be formed from a finite number of applications of the above clauses are routes.

```
value
29.       gen_routes: M → Routes-infset
29.       gen_routes(m) ≡
29((a))i.      let rs = {⟨⟩}
29((a))ii.          ∪ {⟨li,hi⟩,⟨hi,li⟩|li:LI,hi:HI·...}
29((b))i.           ∪ {let r⌢⟨li⟩,⟨li′⟩⌢r′:R · {r⌢⟨li⟩,⟨li′⟩⌢r′}⊆rs,
29((b))i.                   r″⌢⟨hi⟩,⟨hi′⟩⌢r‴:R · {r″⌢⟨hi⟩,⟨hi′⟩⌢r‴}⊆rs in
29((b))ii.               r⌢⟨li⟩⌢r′,r″⌢⟨hi⟩⌢r‴ end} in
29((c))i.      rs end
```

2.4.2.1 Circular Routes

30. A route is circular if the same identifier occurs more than once.

value

30. is_circular_route: R $\rightarrow$ **Bool**

30. is_circular_route(r) $\equiv \exists$ i,j:**Nat** $\cdot$ $\{i,j\} \subseteq$ **inds** r $\land$ $i \neq j \Rightarrow r(i)=r(j)$

2.4.2.2 Connected Road Nets

31. A road net is connected if there is a route from any hub (or any link) to any other hub or link in the net.

```
31.  is_conn_N: N → **Bool**
31.  is_conn_N(n) ≡
31.      let m = derive_RM(n) in
31.      let rs = gen_routes(m) in
31.      ∀ i,i′:(LI|HI) · {i,i′}⊆xtr_LIs(n)∪ xtr_HIs(n)
31.         ∃ r:R · r ∈ rs ∧ r(1)=i ∧ r(len r)=i′ end end
```

2.4.2.3 Set of Connected Nets of a Net

32. The set, cns, of connected nets of a net, n, is
    a the smallest set of connected nets, cns,
    b whose hubs and links together "span" those of the net n.

```
value
32.     conn_Ns: N → N-set
32.     conn_Ns(n) as cns
32(a).      pre: true
32(b).      post: conn_spans_HsLs(n)(cns)
32(a).              ∧ ∼∃ kns:N-set · card kns < card cns
32(a).                  ∧ conn_spans_HsLs(n)(kns)
```

```
32(b).  conn_spans_HsLs: N → N → Bool
32(b).  conn_spans_HsLs(n)(cns) ≡
32(b).      ∀ cn:N•cn ∈ cns ⇒ is_connected_N(n)(cn)
32(b).    ∧ let (hs,ls) = (obs_Hs(obs_HS(n)),obs_Ls(obs_LS(n))),
32(b).              chs = ∪{obs_Hs(obs_HS(cn))|cn ∈ cns},
32(b).              cls = ∪{obs_Ls(obs_LS(cn))|cn ∈ cns} in
32(b).        hs = chs ∧ ls = cls end
```

2.4.2.4 Route Length

33. The length attributes of links can be

 a added and subtracted,

 b multiplied by reals to obtain lengths,

 c divided to obtain fractions,

 d compared as to whether one is shorter than another, etc., and

 e there is a “zero length” designator.

value

33(a). $+,- : \mathrm{LEN} \times \mathrm{LEN} \rightarrow \mathrm{LEN}$

33(b). $* : \mathrm{LEN} \times \mathbf{Real} \rightarrow \mathrm{LEN}$

33(c). $/ : \mathrm{LEN} \times \mathrm{LEN} \rightarrow \mathbf{Real}$

33(d). $<,\leq,=,\neq,\geq,> : \mathrm{LEN} \times \mathrm{LEN} \rightarrow \mathbf{Bool}$

33(e). $\ell_0 : \mathrm{LEN}$

34. One can calculate the length of a route.

```
value
34.   length: R → N → LEN
34.   length(r)(n) ≡
34.       case r of:
34.           ⟨⟩ → $\ell_0$,
34.           ⟨si⟩⌢r′ →
34.               is_LI(si)→attr_LEN(get_L(si)(n))+length(r′)(n)
34.               is_HI(si)→length(r′)(n)
34.       end
```

2.4.2.5 Shortest Routes

35. There is a predicate, is_R, which,
   a given a net and two distinct hub identifiers of the net,
   b tests whether there is a route between these.

value

35. $\text{is\_R: N} \rightarrow (\text{HI}\times\text{HI}) \rightarrow \textbf{Bool}$
35. $\text{is\_R(n)(fhi,thi)} \equiv$
35(a). $\quad\text{fhi} \neq \text{thi} \wedge \{\text{fht,thi}\}\subseteq\text{xtr\_HIs(n)}$
35(b). $\quad\wedge \exists\, \text{r:R} \cdot \text{r} \in \text{routes(n)} \wedge \textbf{hd}\ \text{r} = \text{fhi} \wedge \text{r}(\textbf{len}\ \text{r}) = \text{thi}$

36. The shortest between two given hub identifiers
    a is an acyclic route, r,
    b whose first and last elements are the two given hub identifiers
    c and such that there is no route, r′ which is shorter.

value

36. shortest_route: N → (HI×HI) → R

36(a). shortest_route(n)(fhi,thi) **as** r

36(b). **pre**: pre_shortest_route(n)(fhi,thi)

36(c). **post**: pos__shortest_route(n)(r)(fhi,thi)

36(b). pre_shortest_route: $N \rightarrow (HI{\times}HI) \rightarrow$ **Bool**
36(b). pre_shortest_route(n)(fhi,thi) $\equiv$
36(b). is_R(n)(fhi,thi) $\wedge$ fhi$\neq$thi $\wedge$ $\{$fhi,thi$\}\subset$xtr_HIs(n)

36(c). pos_shortest_route: $N \rightarrow R \rightarrow (HI{\times}HI) \rightarrow$ **Bool**
36(c). pos_shortest_route(n)(r)(fhi,thi) $\equiv$
36(c). r $\in$ routes(n)
36(c). $\wedge \sim\exists\, r'$:R $\cdot$ $r' \in$ routes(n) $\wedge$ length(r') $<$ length(r)

2.5. States

- There are different notions of state. In our example these are some of the states:
  - ◈ the road net composition of hubs and links;
  - ◈ the state of a link, or a hub; and
  - ◈ the vehicle position.

2.6. Actions

- An action is what happens when a function invocation changes, or potentially changes a state.
- Examples of traffic system actions are:
  - insertion of hubs,
  - insertion of links,
  - removal of hubs,
  - removal of links,
  - setting of hub state (hσ),
  - setting of link state (lσ),
  - moving a vehicle along a link,
  - moving a vehicle from a link to a hub and
  - moving a vehicle from a hub to a link.

37. The **insert** action applies to a net and a hub and conditionally yields an updated net.
    a The condition is that there must not be a hub in the "argument" net with the same unique hub identifier as that of the hub to be inserted and
    b the hub to be inserted does not initially designate links with which it is to be connected.
    c The updated net contains all the hubs of the initial net "plus" the new hub.
    d and the same links.

value

37. ins_H: N → H $\overset{\sim}{\rightarrow}$ N
37. ins_H(n)(h) **as** n′, **pre**: pre_ins_H(n)(h), **post**: post_ins_H(n)(h)

37(a). pre_ins_H(n)(h) ≡
37(a). ∼∃ h′:H · h′ ∈ **obs**_Hs(n) ∧ **uid**_HI(h)=**uid**_HI(h′)
37(b). ∧ **mereo**_H(h) = {}

37(c). post_ins_H(n)(h)(n′) ≡
37(c). **obs**_Hs(n) ∪ {h} = **obs**_Hs(n′)
37(d). ∧ **obs**_Ls(n) = **obs**_Ls(n′)

2.7. Events

- By an **event** we understand
  - a state change
  - resulting indirectly from an unexpected application of a function,
  - that is, that function was performed "surreptitiously".
- Events can be characterised by a pair of (before and after) states, a predicate over these and, optionally, a **time** or **time interval**.
- Events are thus like actions:
  - change states,
  - but are usually
    - either caused by "previous" actions,
    - or caused by "an outside action".

38. Link disappearance is expressed as a predicate on the "before" and "after" states of the net. The predicate identifies the "missing" ℓink (!).

39. Before the disappearance of link ℓ in net n

 a the hubs h' and h'' connected to link ℓ

 b were connected to links identified by $\{l'_1, l'_2, \ldots, l'_p\}$ respectively $\{l''_1, l''_2, \ldots, l''_q\}$

 c where, for example, l'_i, l''_j are the same and equal to **uid_**$\Pi(\ell)$.

```
38.  link_dis: N × N → Bool
38.  link_dis(n,n′) ≡
38.      ∃ ℓ:L • pre_link_dis(n,ℓ) ⇒ post_link_dis(n,ℓ,n′)
39.  pre_link_dis: N × L → Bool
39.  pre_link_dis(n,ℓ) ≡ ℓ ∈ obs_Ls(n)
```

40. After link ℓ disappearance there are instead
    a two separate links, ℓ_i and ℓ_j, "truncations" of ℓ
    b and two new hubs h''' and h''''
    c such that ℓ_i connects h' and h''' and
    d ℓ_j connects h'' and h'''';
    e Existing hubs h' and h'' now have mereology
       i. $\{l'_1, l'_2, \ldots, l'_p\} \setminus \{\mathsf{uid\_}\Pi(\ell)\} \cup \{\mathsf{uid\_}\Pi(\ell_i)\}$ respectively
       ii. $\{l''_1, l''_2, \ldots, l''_q\} \setminus \{\mathsf{uid\_}\Pi(\ell)\} \cup \{\mathsf{uid\_}\Pi(\ell_j)\}$
41. All other hubs and links of n are unaffected.

42. We shall "explain" *link disappearance* as the combined, instantaneous effect of

   a first a **remove link** "event" where the **removed link** connected **hubs** hi_j and hi_k;

   b then the **insertion** of two new, "fresh" **hubs**, h_α and h_β;

   c "followed" by the **insertion** of two new, "fresh" links $\mathsf{l}_{j\alpha}$ and $\mathsf{l}_{k\beta}$ such that

      i. $\mathsf{l}_{j\alpha}$ connects hi_j and h_α and

      ii. $\mathsf{l}_{k\beta}$ connects hi_k and $\mathsf{h}_{k\beta}$

```
value
42.        post_link_dis(n,$\ell$,n$'$) $\equiv$
42.                let h_a,h_b:H $\cdot$
42.                    let {li_a,li_b}=mereo_L($\ell$) in
42.                    (get_H(li_a)(n),get_H(li_b)(n)) end in
42(a).          let n$''$        = rem_L(n)(uid_L($\ell$)) in
42(b).          let h$_\alpha$,h$_\beta$:H  $\cdot$  {h$_\alpha$,h$_\beta$}$\cap$obs_Hs(n)={} in
42(b).          let n$'''$       = ins_H(n$''$)(h$_\alpha$) in
42(b).          let n$''''$      = ins_H(n$'''$)(h$_\beta$) in
42(c).          let l$_{j\alpha}$,l$_{k\beta}$:L  $\cdot$  {l$_{j\alpha}$,l$_{k\beta}$}$\cap$obs_Ls(n)={}
42(c).            $\wedge$ mereo_L(l$_{j\alpha}$) = {uid_H(h_a),uid_H(h$_\alpha$)}
42(c).            $\wedge$ mereo_L(l$_{k\beta}$) = {uid_H(h_b),uid_H(h$_\beta$)} in
42((c))i.       let n$'''''$     = ins_L(n$''''$)(l$_{j\alpha}$) in
42((c))ii.      n$'$ = ins_L(n$'''''$)(l$_{k\beta}$) end end end end end end end
```

2.8. Behaviours

2.8.1. Traffic

2.8.1.1 Continuous Traffic

- For the road traffic system
  - perhaps the most significant example of a behaviour
  - is that of its traffic

43. the continuous time varying discrete positions of vehicles, vp:VP[9],
44. where time is taken as a dense set of points.

type

44. $c\mathbb{T}$

43. $cRTF = c\mathbb{T} \rightarrow (V \xrightarrow[m]{} VP)$

[9] For VP see Item 12(a) on Slide 56.

2.8.1.2 Discrete Traffic

- We shall model, not continuous time varying traffic, but

45. discrete time varying discrete positions of vehicles,
46. where time can be considered a set of linearly ordered points.

46. $\mathrm{d}\mathbb{T}$
45. $\mathrm{dRTF} = \mathrm{d}\mathbb{T} \underset{m}{\rightarrow} (\mathrm{V} \underset{m}{\rightarrow} \mathrm{VP})$

47. The road traffic that we shall model is, however, of vehicles referred to by their unique identifiers.

type
47. $\mathrm{RTF} = \mathrm{d}\mathbb{T} \underset{m}{\rightarrow} (\mathrm{VI} \underset{m}{\rightarrow} \mathrm{VP})$

2.8.1.3 Time: An Aside

- We shall take a rather simplistic view of time `[wayne.d.blizard.90,mctaggart-t0,prior68,J.van.Benthem.Log`

48. We consider $\mathtt{d}\mathbb{T}$, or just $\mathbb{T}$, to stand for a totally ordered set of time points.
49. And we consider $\mathbb{TI}$ to stand for time intervals based on $\mathbb{T}$.
50. We postulate an infinitesimal small time interval δ.
51. $\mathbb{T}$, in our presentation, has lower and upper bounds.
52. We can compare times and we can compare time intervals.
53. And there are a number of “arithmetics-like” operations on times and time intervals.

type

48. $\mathbb{T}$
49. $\mathbb{TI}$

value

50. δ:$\mathbb{TI}$
51. $\mathbb{MIN},\mathbb{MAX}$: $\mathbb{T} \rightarrow \mathbb{T}$
51. $<,\leq,=,\geq,>$: $(\mathbb{T}\times\mathbb{T})|(\mathbb{TI}\times\mathbb{TI}) \rightarrow$ **Bool**
52. $-$: $\mathbb{T}\times\mathbb{T} \rightarrow \mathbb{TI}$
53. $+$: $\mathbb{T}\times\mathbb{TI},\mathbb{TI}\times\mathbb{T} \rightarrow \mathbb{T}$
53. $-,+$: $\mathbb{TI}\times\mathbb{TI} \rightarrow \mathbb{TI}$
53. $*$: $\mathbb{TI}\times$**Real** $\rightarrow \mathbb{TI}$
53. $/$: $\mathbb{TI}\times\mathbb{TI} \rightarrow$ **Real**

54. We postulate a global **clock** behaviour which offers the current time.

55. We declare a channel **clk_ch**.

value
54. clock: $\mathbb{T} \rightarrow$ **out** clk_ch **Unit**
54. clock(t) $\equiv$ ... clk_ch!t ... clock(t $\sqcap$ t+δ)
channnel
55. clk_ch:$\mathbb{T}$

2.8.2. Globally Observable Parts

- There is given

56. a net, **n:N**,

57. a set of vehicles, **vs:V-set**, and

58. a monitor, **m:M**.

- The **n:N**, **vs:V-set** and **m:M** are observable from the road traffic system domain.

value

56. $n{:}N = \underline{\mathbf{obs\_}}N(\Delta)$
56. $ls{:}L\textbf{-set} = \underline{\mathbf{obs\_}}Ls(\underline{\mathbf{obs\_}}LS(n)),\ hs{:}H\textbf{-set} = \underline{\mathbf{obs\_}}Hs(\underline{\mathbf{obs\_}}HS(n)),$
56. $lis{:}LI\textbf{-set} = \{\underline{\mathbf{uid\_}}L(l) | l{:}L{\cdot}l \in ls\},\ his{:}HI\textbf{-set} = \{\underline{\mathbf{uid\_}}H(h) | h{:}H{\cdot}h \in hs\}$
57. $vs{:}V\textbf{-set} = \underline{\mathbf{obs\_}}Vs(\underline{\mathbf{obs\_}}VS(\underline{\mathbf{obs\_}}F(\Delta))),\ vis{:}V\textbf{-set} = \{\underline{\mathbf{uid\_}}V(v) | v{:}V{\cdot}v$
58. $m{:}\underline{\mathbf{obs\_}}M(\Delta)$

2.8.3. Road Traffic System Behaviours

59. Thus we shall consider our road traffic system, **rts**, as
    a the concurrent behaviour of a number of vehicles and,
      to “observe”, or, as we shall call it, to monitor their movements,
    b the **mon**itor behaviour, based on
    c the monitor and its unique identifier,
    d an initial vehicle position map, and
    e an initial starting time.

value

59(c). $\text{mi:MI} = \underline{\textbf{uid\_}}(\text{m})$

59(d). $\text{vpm:VPM} = \text{vpr(vs)(n)}$

59(e). $\text{t}_0\text{:T} = \text{clk\_ch?}$

59. $\text{rts}() =$

59(a). $\quad \| \; \{\text{veh}(\underline{\textbf{uid\_}}\text{V(v))(v)(vpm(}\underline{\textbf{uid\_}}\text{V(v)))} \mid \text{v:V}\cdot\text{v} \in \text{vs}\}$

59(b). $\quad \| \; \text{mon(mi)(m)([} \text{t}_0 \mapsto \text{vpm])}$

- where the "extra" monitor argument
  - records the discrete road traffic, RTF,
  - initially set to the singleton map from an initial start time, t_0 to the initial assignment of vehicle positions.

2.8.4. Channels

- In order for the monitor behaviour to assess the vehicle positions
  - these vehicles communicate their positions
  - to the monitor
  - via a vehicle to monitor channel.
- In order for the monitor to time-stamp these positions
  - it must be able to "read" a clock.

60. Thus we declare a set of channels indexed by the unique identifiers of vehicles and communicating vehicle positions.

channel

60. $\{vm\_ch[mi,vi]|vi:VI\cdot vi \in vis\}:VP$

2.8.5. Behaviour Signatures

61. The road traffic system behaviour, rts, takes no arguments; and "behaves", that is, continues forever.
62. The vehicle behaviours are indexed by the unique identifier, uid_V(v):VI, the vehicle part, v:V and the vehicle position; offers communication to the monitor behaviour; and behaves "forever".
63. The monitor behaviour takes monitor part, m:M, as argument and also the discrete road traffic, drtf:dRTF; the behaviour otherwise runs forever.

value

61. rts: **Unit** $\rightarrow$ **Unit**

62. veh: vi:VI $\rightarrow$ v:V $\rightarrow$ VP $\rightarrow$ **out** vm_ch[vi],mi:MI **Unit**

63. mon: mi:MI $\rightarrow$ m:M $\rightarrow$ dRTF $\rightarrow$ **in** $\{$vm_ch[mi,vi]$|$vi:VI$\cdot$vi $\in$ vis$\}$,clk_c

2.8.6. The Vehicle Behaviour

64. A **vehicle** process

- is indexed by the unique vehicle identifier vi:VI,
- the vehicle "as such", v:V and
- the vehicle position, vp:VPos.

The vehicle process communicates

- with the **monitor** process on channel vm[vi]
- (sends, but receives no messages), and
- otherwise evolves "in[de]finitely" (hence **Unit**).

65. We describe here an abstraction of the vehicle behaviour **at** a **Hub** (**hi**).
    a Either the vehicle remains at that hub informing the monitor,
    b or, internally non-deterministically,
        i. moves onto a link, **tli**, whose “next” hub, identified by **thi**, is obtained from the mereology of the link identified by **tli**;
        ii. informs the monitor, on channel **vm[vi]**, that it is now on the link identified by **tli**,
        iii. whereupon the vehicle resumes the vehicle behaviour positioned at the very beginning (**0**) of that link,
    c or, again internally non-deterministically,
    d the vehicle “disappears — off the radar” !

```
65.        veh(vi)(v)(vp:atH(fli,hi,tli)) ≡
65(a).            vm_ch[ mi,vi ]!vp ; veh(vi)(v)(vp)
65(b).         ⊓
65((b))i.         let {hi′,thi}=mereo_L(get_L(tli)(n)) in assert: hi′=hi
65((b))ii.        vm_ch[ mi,vi ]!onL(tli,hi,0,thi) ;
65((b))iii.       veh(vi)(v)(onL(tli,hi,0,thi)) end
65(c).         ⊓
65(d).            stop
```

66. We describe here an abstraction of the vehicle behaviour **on** a **Link** (**ii**). Either
    a the vehicle remains at that link position informing the monitor,
    b or, internally non-deterministically,
    c if the vehicle's position on the link has not yet reached the hub,
        i. then the vehicle moves an arbitrary increment δ along the link informing the monitor of this, or
        ii. else, while obtaining a "next link" from the mereology of the hub (where that next link could very well be the same as the link the vehicle is about to leave),
            A. the vehicle informs the monitor that it is now at the hub identified by **thi**,
            B. whereupon the vehicle resumes the vehicle behaviour positioned at that hub.
67. or, internally non-deterministically,
68. the vehicle "disappears — off the radar" !

```
64.          veh(vi)(v)(vp:onL(fhi,li,f,thi)) ≡
66(a).              vm_ch[ mi,vi ]!vp ; veh(vi)(v)(vp)
66(b).          ⊓
66(c).              if f + $\delta$<1
66((c))i.               then vm_ch[ mi,vi ]!onL(fhi,li,f+$\delta$,thi) ;
66((c))i.                    veh(vi)(v)(onL(fhi,li,f+$\delta$,thi))
66((c))ii.              else let li′:LI•li′ ∈ mereo_H(get_H(thi)(n)) in
66((c))iiA.                  vm_ch[ mi,vi ]!atH(li,thi,li′);
66((c))iiB.                  veh(vi)(v)(atH(li,thi,li′)) end end
67.             ⊓
68.              stop
```

2.8.7. The Monitor Behaviour

69. The monitor behaviour evolves around the attributes of an own "state", m:M, a table of traces of vehicle positions, while accepting messages about vehicle positions and otherwise progressing "in[de]finitely".

70. Either the monitor "does own work"

71. or, internally non-deterministically accepts messages from vehicles.

   a A vehicle position message, vp, may arrive from the vehicle identified by vi.

   b That message is appended to that vehicle's movement trace,

   c whereupon the monitor resumes its behaviour —

   d where the communicating vehicles range over all identified vehicles.

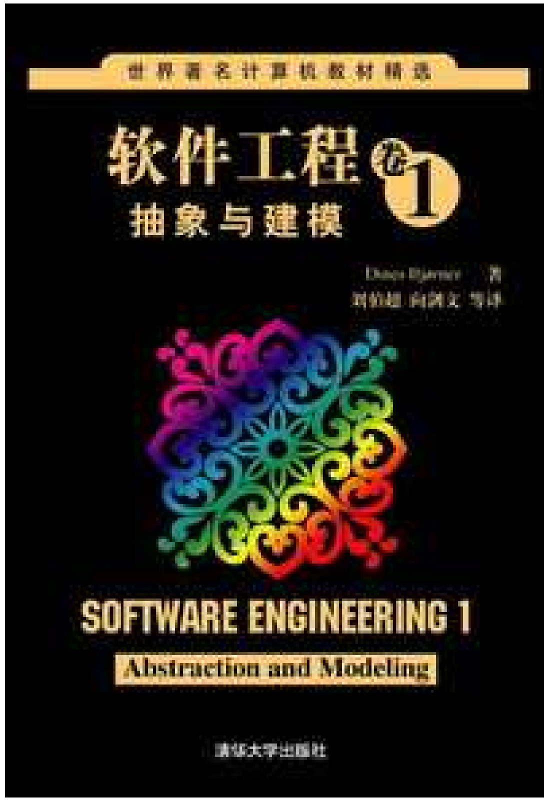
```
69.     mon(mi)(m)(rtf) ≡
70.             mon(mi)(own_mon_work(m))(rtf)
71.         ⊓
71(a).          ⊓ { let ((vi,vp),t) = (vm_ch[ mi,vi ]?,clk_ch?) in
71(b).                let rtf′ = rtf † [ t ↦ rtf(max dom rtf) † [ vi ↦ vp ] ] in
71(c).                mon(mi)(m)(rtf′) end
71(d).                end | vi:VI · vi ∈ vis }
```

```
70.     own_mon_work: M → dRTF → M
```

- We do not describe the clock behaviour by other than stating that it continually offers the current time on channel **clkm_ch**. ■

Springer
SOFTWARE ENGINEERING 3
DOMAINS, REQUIREMENTS AND SOFTWARE DESIGN
软件工程 领域、需求与软件设计 卷3
Dines Bjørner 著
CHINA-PUB
清华大学出版社

See You in 30 Minutes — Thanks !



Welcome Back — Thanks !

Lecture 2: 11:00–11:40 + 11:50–12:30
Domains, Discrete Endurants

3. Domains

3.1. Delineations

We characterise a number of terms.

3.1.0.1 Domain

- By a **domain**$_\delta$ we shall here understand
  - ⟐ an area of human activity
  - ⟐ characterised by observable phenomena:
    - ⚭ **entities**
      - ∗ whether **endurants** (manifest **parts** and **materials**)
      - ∗ or **perdurants** (**actions**, **events** or **behaviours**),
    - ⚭ whether
      - ∗ **discrete** or
      - ∗ **continuous**;
    - ⚭ and of their **properties**.

3.1.0.2 Domain Phenomena

- By a **domain phenomenon**$_\delta$ we shall understand
  - something that can be observed by the **human senses**
  - or by **equipment** based on laws of physics and chemistry.
- Those phenomena that can be observed by
  - the human eye or
  - touched, for example, by human hands,
  - we call **part**s and **material**s.
- Those phenomena that can be observed of parts and materials
  - can usually be measured
  - and we call them **properties** of these **part**s and those **material**s.

3.1.0.3 Domain Entity

- By a **domain entity**$_\delta$ we shall understand
  - a **manifest domain phenomenon** or
  - a **domain concept**, i.e., an **abstraction**,
  - derived from a **domain entity**.
- The distinction between
  - a **manifest domain phenomenon** and
  - a **concept** thereof, i.e., a **domain concept**,

  is important.
- Really, what we describe are the **domain concepts** derived
  - from **domain phenomena** or
  - from other **domain concepts**.

3.1.0.4 Endurant Entity

- We distinguish between
  - endurants and
  - perdurants.
- From Wikipedia:
  - *By an* **endurant**$_\delta$ *(also known as a* **continuant**$_\delta$ *or a* **substance**$_\delta$*) we shall understand an entity*
    - *that can be observed, i.e., perceived or conceived,*
    - *as a complete concept,*
    - *at no matter which given snapshot of time.*
  - *Were we to freeze time*
    - *we would still be able to observe the entire endurant.*

3.1.0.5 Perdurant Entity

- From Wikipedia:
  - ⟐ *Perdurant: Also known as occurrent, accident or happening.*
  - ⟐ *Perdurants are those entities for which only a fragment exists if we look at them at any given snapshot in time.*
  - ⟐ *When we freeze time we can only see a fragment of the perdurant.*
  - ⟐ *Perdurants are often what we know as processes, for example 'running'.*
  - ⟐ *If we freeze time then we only see a fragment of the running, without any previous knowledge one might not even be able to determine the actual process as being a process of running.*
  - ⟐ *Other examples include an activation, a kiss, or a procedure.*

3.1.0.6 Discrete Endurant

- We distinguish between
  - discrete endurants and
  - continuous endurants.
- By a **discrete endurant**$_\delta$, that is, a **part**, we shall understand something which is
  - separate or distinct in form or concept,
  - consisting of distinct or separate parts.

3.1.0.7 Continuous Endurant

- By a **continuous endurant**$_\delta$, that is, a **material**, we shall understand an **endurant** whose spatial characteristics are
  - prolonged, without interruption,
  - in an unbroken spatial series or pattern.

3.1.0.8 Domain Parts and Materials

- By a part$_\delta$ we mean
  - ◈ a discrete endurant,
  - ◈ a manifest entity which is fixed in shape and extent.
- By a material$_\delta$
  - ◈ a continuous endurant,
  - ◈ a manifest entity which typically varies in shape and extent.

3.1.0.9 Domain Analysis

- By domain analysis$_\delta$ we shall understand an examination of a domain,
  - its entities,
  - their possible composition,
  - properties
  - and relations between entities,

3.1.0.10 Domain Description

- By a domain description$_\delta$ we shall understand
  - a narrative description
  - tightly coupled (say line-number-by-line-number)
  - to a formal description.

3.1.0.11 Domain Engineering

- By **domain engineering**$_\delta$ we shall understand
  - the **engineering** of a domain description,
  - that is,
    - the rigorous construction of domain descriptions, and
    - the further analysis of these, creating **theories of domains**[10], etc.

[10]Section (Slides 36–105) is an example of the basis for a theory of road traffic systems.

3.1.0.12 Domain Science

- By domain science$_\delta$ we shall understand
  - two things:
    - the general study and knowledge of
      - how to create and handle domain descriptions
      - (a general theory of domain descriptions)

      and
    - the specific study and knowledge of a particular domain.
  - The two studies intertwine.

3.1.0.13 Values & Types

- By a **value**$_\delta$ we mean some **mathematical quantity**.
- By a **type**$_\delta$ we mean
  - a largest set of **value**s,
  - each characterised by the same predicate,
  - such that there are no other **value**s,
  - not members of the set,
  - but which still satisfy that predicate.
- We do not give examples here of the kind of **type predicate**s that may characterise **type**s.

- When we observe a domain we observe **instances** of **entities**;
- but when we describe those instances
  - (which we shall call **value**s)
  - we describe, not the values,
  - but their **type** and **properties**:
    - parts and materials have **type**s and **value**s;
    - actions, events and behaviours, all, have **type**s and **value**s, namely as expressed by their **signature**s; and
    - actions, events and behaviours have **properties**, namely as expressed by their **function definition**s.
- Values are phenomena and types are concepts thereof.

3.1.0.14 Discrete Perdurant

- By a **discrete perdurant**$_\delta$ we shall understand
  - a perdurant
  - which we consider as taking place instantaneously,
  - in no time,
  - or where whatever time interval it may take to complete
  - is considered immaterial.

3.1.0.15 Continuous Perdurant

- By a **continuous perdurant**$_\delta$ we shall understand a **perdurant** whose temporal characteristics are likewise
  - ◈ prolonged, without interruption,
  - ◈ in an unbroken temporal series or pattern.

3.1.0.16 Extensionality

- By extensionality$_\delta$ `Merriam-Webster`[11] means
  - *“something which relates to, or is marked by extension,”*
  - *“that is, concerned with objective reality”*.
- Our use basically follows this characterisation:
  - We think of extensionality as a syntactic notion,
  - one that characterises an exterior appearance or form
- We shall therefore think of
  - part types and material types
  - whether parts are atomic or composite, and
  - how composite parts are composed

as extensional features.

[11]Extensionality. Merriam-Webster.com. 2011, http://www.merriam-webster.com (16 August 2012).

3.1.0.17 Intentionality

- By intentionality$_\delta$ `Merriam-Webster`[12] means:
  - *"done by intention or design"*,
  - *"intended"*,
  - *"of or relating to epistemological intention"*,
  - *"having external reference"*.
- Our use basically follows this characterisation:
  - we think of intentionality as a semantic notion,
  - one that characterises an intention.
- We shall therefore think of
  - part attributess and material attributes

  as intentional features.

[12]Intentionality. Merriam-Webster.com. 2011, http://www.merriam-webster.com (16 August 2012).

3.2. Formal Analysis of Entities

3.2.1. Theory

- This section is a transcription of
  - Ganter & Wille's [Wille:ConceptualAnalysis1999] *Formal Concept Analysis, Mathematical Foundations*, the 1999 edition, Pages 17–18.

Some Notation:

- By $\mathcal{E}$ we shall understand the type of entities;
- by $\mathbb{E}$ we shall understand a value of type $\mathcal{E}$;
- by $\mathcal{Q}$ we shall understand the type of qualities;
- by $\mathbb{Q}$ we shall understand a value of type $\mathcal{Q}$;
- by $\mathcal{E}$**-set** we shall understand the type of sets of entities;
- by $\mathbb{ES}$ we shall understand a value of type $\mathcal{E}$**-set**;
- by $\mathcal{Q}$**-set** we shall understand the type of sets of qualities; and
- by $\mathbb{QS}$ we shall understand a value of type $\mathcal{Q}$**-set**.

Definition: 1 Formal Context:

- A formal context$_\delta$ $\mathbb{K} := (\mathbb{ES}, \mathbb{I}, \mathbb{QS})$ consists of two sets;
  - $\mathbb{ES}$ of entities,
  - $\mathbb{QS}$ of qualities, and a
  - relation $\mathbb{I}$ between $\mathbb{E}$ and $\mathbb{Q}$. ■

- To express that $\mathbb{E}$ is in relation $\mathbb{I}$ to a Quality $\mathbb{Q}$ we write
  - $\mathbb{E} \cdot \mathbb{I} \cdot \mathbb{Q}$, which we read as
  - *"entity $\mathbb{E}$ **has** quality $\mathbb{Q}$"*.

- Example endurant entities are
  - a specific vehicle,
  - another specific vehicle,
  - etcetera;
  - a specific street segment (link),
  - another street segment,
  - etcetera;
  - a specific road intersection (hub),
  - another specific road intersection,
  - etcetera,
  - a monitor.

  One can also list perdurant entities.

- Example endurant entity qualities are
  - has mobility,
  - has possible velocity,
  - has possible acceleration,
  - has length,
  - has location,
  - has traffic state,
  - can vehicles be sensed,
  - etcetera.

  One can also list perdurant entity qualities.

Definition: 2 Qualities Common to a Set of Entities:

- For any subset, $s\mathbb{ES} \subseteq \mathbb{ES}$, of entities we can define

$$\begin{aligned}
&\mathcal{DQ} : \mathcal{E}\textbf{-set} \to (\mathcal{E}\textbf{-set}\times\mathcal{I} \times \mathcal{Q}\textbf{-set}) \to \mathcal{Q}\textbf{-set}\\
&\mathcal{DQ}(s\mathbb{ES})(\mathbb{ES},\mathbb{I},\mathbb{QS}) \equiv \{\mathbb{Q} \mid \mathbb{Q}{:}\mathcal{Q}, \mathbb{E}{:}\mathcal{E} \cdot \mathbb{E}{\in}s\mathbb{ES} \wedge \mathbb{E} \cdot \mathbb{I} \cdot \mathbb{Q}\}\\
&\textbf{pre}\text{: } s\mathbb{ES} \subseteq \mathbb{ES}
\end{aligned}$$

"the set of qualities common to entities in s$\mathbb{ES}$*".* ■

Definition: 3 Entities Common to a Set of Qualities:

- For any subset, $s\mathbb{QS} \subseteq \mathbb{QS}$, of qualities we can define

$$\begin{aligned}
&\mathcal{DE}\text{: } \mathcal{Q}\textbf{-set} \to (\mathcal{E}\textbf{-set}\times\mathcal{I} \times \mathcal{Q}\textbf{-set}) \to \mathcal{E}\textbf{-set}\\
&\mathcal{DE}(s\mathbb{QS})(\mathbb{ES},\mathbb{I},\mathbb{QS}) \equiv \{\mathbb{E} \mid \mathbb{E}{:}\mathcal{E},\ \mathbb{Q}{:}\mathcal{Q} \cdot \mathbb{Q}{\in}s\mathbb{Q} \wedge \mathbb{E} \cdot \mathbb{I} \cdot \mathbb{Q}\ \},\\
&\textbf{pre}\text{: } s\mathbb{QS} \subseteq \mathbb{QS}
\end{aligned}$$

"the set of entities which have all qualities in s$\mathbb{Q}$*".* ■

Definition: 4 Formal Concept:

- A **formal concept**$_\delta$ of a **context** $\mathbb{K}$ is a pair:
  - $(s\mathbb{Q}, s\mathbb{E})$ where
    - $\mathcal{DQ}(s\mathbb{E})(\mathbb{E}, \mathbb{I}, \mathbb{Q}) = s\mathbb{Q}$ and
    - $\mathcal{DE}(s\mathbb{Q})(\mathbb{E}, \mathbb{I}, \mathbb{Q}) = s\mathbb{E}$;
  - $s\mathbb{Q}$ is called the **intent**$_\delta$ of $\mathbb{K}$ and $s\mathbb{E}$ is called the **extent**$_\delta$ of $\mathbb{K}$. ■
- Now comes the "crunch":
  - *In the `TripTych` domain analysis*
  - *we strive to find formal concepts*
  - *and, when we think we have found one,*
  - *we assign a type to it.*

- In mathematical terms it turns out that **formal concepts** are **Galois connections**.
- We can, in other words, characterise **domain analysis** to be the "hunting" for **Galois connections**.
- Or, even more "catchy":
  - **domain types**,
  - whether they be **endurant entity types**
  - or they be **perdurant entity signatures**
  - are **Galois connections**.

• • •

- The entities referred to by $\mathbb{E}$
  - ⟐ are the domain entities that we shall deal with in this seminar,
- and the qualities referred to by $\mathbb{Q}$
  - ⟐ are the mereologies and attributes of discrete endurant entities
  - ⟐ and the signatures of actions, events and behaviours of discrete perdurant entities;
  - ⟐ with these terms becoming clearer as we progress through this seminar.

● ● ●

- Earlier in this section, two signatures were expressed as
  - ◈ $\mathcal{DQ}$: $\mathcal{E} \to \mathcal{K} \to \mathcal{Q}$ and
  - ◈ $\mathcal{DE}$: $\mathcal{Q} \to \mathcal{K} \to \mathcal{E}$
- The “switch” between using $\mathcal{K}$ for types and $\mathbb{K}$ for values of that type is “explained”:
  - ◈ $\mathcal{K}$ is the Cartesian type: $\mathcal{E} \times \mathcal{I} \times \mathcal{Q}$, and
  - ◈ $\mathbb{K} = (\mathbb{E}, \mathbb{I}, \mathbb{Q})$ is a value of that type.

3.2.2. Practice

-
-
-
-

3.3. Discussion

- The crucial characterisation (above) is that of **domain entity** (Slide 108).
  - It is pivotal since all we describe are domain entities.
  - If we get the characterisation wrong we get everything wrong!
  - What might get the characterisation, or its interpretation, wrong is the interpretation of **domain entities**:
    - *"those phenomena that can be observed by*
      - *the human eye or*
      - *touched, for example, by human hands,"*

    and
    - *"manifest domain phenomena or*
    - *domain concepts, i.e., abstractions,*
    - *derived from a domain entities".*

- The whole thing hinges of
  - *what can be described,*
  - *what constitutes a description and*
  - *when is a text a bona fide description.*
- Another set of questions are
  - *of what we have chosen to constitute entities*
  - *which should we describe,*
  - *which not ?*

- Philosophers have dealt with these questions.
  - ❖ Recent writings are
    `[Badiou1988,BarrySmith1993,ChrisFox2000]` and
    `[CasatiVarzi2010,HenryLaycock2011,WilsonScpall2012]`.
  - ❖ Going back in time we find
    `[LeonardGoodman1940,Kripke1980,BowmanLClarke81]`.
  - ❖ Among the classics we mention
    `[Russell1905,Russell1922,RudolfCarnap1928,StanislawLesniewksi1927-1`

- We shall only indirectly contribute to this philosophical discussion
  - and do so by presenting the material of this paper;
  - having studied, over the years, fragments of the above cited publications
  - we have concluded with the suggestions of this paper:
    - following the principles, techniques and tools presented here
    - can lead the **domain engineer** to
    - a large class of **domain descriptions**s,
    - large enough for our "immediate future" needs !
- We shall, in the conclusion, return to the questions of
  - what can be described,
  - what constitutes a description and
  - when is a text a bona fide description ?

4. Discrete Endurant Entities

- For pragmatics reasons we structure our treatment of discrete endurant domain entities as follows:
  - First we treat the extensional aspects of parts,
  - then their properties: the intentional aspects.
- One could claim that when we say "first parts"
  - we mean fist: a syntactic analysis of parts
  - into atomic and composite parts,
  - etcetera;
- and when we say "then their properties"
  - we mean: then a partial semantic analysis,
  - something which "throws" light over parts,
  - since parts really are distinguishable only through their properties.

4.1. Parts

4.1.1. What is a Part ?

- By a part$_\delta$ we mean an observable manifest endurant.

Discussion:

- We use the term ‘part’ where others use different terms, for example,
  - ‘individual’,
  - ‘particular’,
  - ‘unit’,
  - ‘object’,
  - ‘thing’,
  - or other.

Example: 5 Parts.

- Example **parts** have their **types** defined in the items as follows:
  - N Item 1(a) Slide 38,
  - F Item 1(b) Slide 38,
  - M Item 1(c) Slide 38,
  - HS Item 2(a) Slide 39,
  - LS Item 2(b) Slide 39,
  - VS Item 3 Slide 40,
  - Vs Item 4(a) Slide 41,
  - V Item 4(b) Slide 41,
  - Hs Item 5 Slide 44,
  - Ls Item 6 Slide 44,
  - H Item 5(a) Slide 44,
  - L Item 6(b) Slide 44.

4.1.2. Classes of "Same Kind" Parts

- We repeat:
  - the **domain describer** does not describe instances of parts,
  - but seeks to describe classes of parts of the same kind.
- Instead of the term 'same kind' we shall use either the terms
  - **part sort** or
  - **part type**.
- By a **same kind class of parts**$_\delta$, that is a **part sort** or **part type** we shall mean
  - a class all of whose members, i.e., **parts**,
  - enjoy "exactly" the same **properties**
  - where a **property** is expressed as a **proposition**.

Example: 6 Part Properties. We continue Example 4.

- Examples of part properties are:
  - *has unique identity*,
  - *has mereology*,
  - *has length*,
  - *has location*,
  - *has traffic movement restriction*,
  - *has position*,
  - *has velocity* and
  - *has acceleration*.

■

4.1.3. A Preview of Part Properties

- For pragmatic reasons we group **endurant properties** into two categories:
  - a group which we shall refer to as **meta properties**:
    - *is discrete*,
    - *is continuous*,
    - *is atomic*,
    - *is composite*,
    - *has observers*,
    - *is sort* and
    - *has concrete type*;
  - and a group which we shall refer to as **part properties**
    - *has unique existence*,
    - *has mereology* and
    - *has attributes*.
- The first group is treated in this section;
- the second group in the next section.

4.1.4. Formal Concept Analysis: Endurants

- The **domain analyser** examines collections of **part**s.
  - ◈ In doing so the **domain analyser** discovers and thus identifies and lists a number of **properties**.
  - ◈ Each of the **part**s examined usually satisfies only a subset of these properties.
  - ◈ The **domain analyser** now groups **part**s into collections
    - ⚭ such that each collection have its **part**s satisfy the same set of **properties**,
    - ⚭ such that no two distinct collections are indexed, as it were, by the same set of **properties**, and
    - ⚭ such that all **part**s are put in some collection.
  - ◈ The **domain analyser** now
    - ⚭ assigns distinct **type name**s (same as **sort name**s)
    - ⚭ to distinct collections.
- That is how we assign **type**s to **part**s.

4.1.5. Part Property Values

- By a **part property value**$_\delta$, i.e., a **property value**$_\delta$ of a **part**, we mean
  - ⟐ the **value**
  - ⟐ associated with an **intentional property**
  - ⟐ of the **part**.

Example: 7 Part Property Values.

- A link, l:L, may have the following **intentional property values**:
  - ⟐ LOCation value *loc_set*,
  - ⟐ LENgth value *123 meters* and
  - ⟐ *mereology* value $\{\kappa_i, \kappa_j\}$. ■

- Two **part**s of the same **type** are different
  - if for at least one of the intentional properties of that **part type**
  - they have different **part property value**s.

slut

Example: 8 Distinct Parts.

- Two links, l_a,l_b:L, may have the following respective **property value**s:
  - LOCation values *loc_set*$_a$, and *loc_set*$_b$,
  - LENgth value *123 meters* and *123 meters*, i.e., the same, and
  - *mereology* values $\{\kappa_i,\kappa_j\}$ and $\{\kappa_m,\kappa_n\}$ where $\{\kappa_i,\kappa_j\} \neq \{\kappa_m,\kappa_n\}$.
- When so, they are distinct, and the cadestral space *loc_set*$_a$ must not share any point with cadestral space *loc_set*$_b$.

4.1.6. Part Sorts

- By an **abstract type**$_\delta$, or a **sort**$_\delta$, we shall understand a type
  - which has been given a name
  - but is otherwise undefined, that is,
    - is a set of values of further undefined quantities `[Milne1990:RSL:SemFound,Milne1990:RSL:ProofTheory]`.
      - where these are given properties
      - which we may express in terms of **axiom**s over sort (including **property**) values.
- All of the above examples are examples of **sort**s.

Example: 9 Part Sorts.

- The discovery of N, F and M was made as a result of examining the domain, Δ, at **domain index** $\langle\Delta\rangle$;
- HS and LS at **domain index** $\langle\Delta$,N$\rangle$;
- Hs and H (Ls and L) at **domain indexes** $\langle\Delta$,HS$\rangle$ ($\langle\Delta$,LS$\rangle$); and
- Vs and V at **domain index** $\langle\Delta$,VS$\rangle$. ■

4.1.7. Atomic Parts

- By an **atomic part**$_\delta$ we mean a part which,
  - in a given context,
  - is deemed *not* to consist of meaningful, separately observable proper **sub-part**s.
- A **sub-part** is a **part**.

Example: 10 Atomic Types.

- We have exemplified the following atomic types:
  - H (Item 5(b) on Slide 43),
  - L (Item 6(b) on Slide 43),
  - V (Item 4(b) on Slide 41) and
  - M (Item 1(c) on Slide 38).
- Implicit tests,
  - at **domain indexes**,
  - by the **domain analyser**,
  - for atomicity

  were performed as follows:
  - for H at $\langle\Delta$, N,HS,Hs,H$\rangle$;
  - for L at $\langle\Delta$, N,LS,Ls,L$\rangle$;
  - for V at $\langle\Delta$, F,VS,Vs,V$\rangle$; and
  - for M at $\langle\Delta$, M$\rangle$. ■

4.1.8. Composite Parts

- By a composite part$_\delta$ we mean *a part which,*
  - *in a given context,*
  - *is deemed to indeed consist of meaningful, separately observable proper sub-parts.*

Example: 11 Composite Types.

- We have exemplified the following composite types:
  - N (Items 2(a)– 2(b) on Slide 39), HS (Item 5 on Slide 43), LS (Item 6 on Slide 43), Hs (Item 5(a) on Slide 43), Ls (Item 6(a) on Slide 43), F (Item 3 on Slide 40), VS (Item 4(a) on Slide 41), Va (Item 4(a) on Slide 41),

  respectively.
- Tests for compostionality of these were implicitly performed;
  - for N at index $\langle\Delta,$ N$\rangle$;
  - for HS and LS at index $\langle\Delta,$ N,HS$\rangle$ and $\langle\Delta,$ N,LS$\rangle$;
  - for Hs and Ls at indexes $\langle\Delta,$ N,HS,Hs$\rangle$ and $\langle\Delta,$ N,LS,Ls$\rangle$;
  - for F at index $\langle\Delta,$ F$\rangle$;
  - for VS at index $\langle\Delta,$ F,VS$\rangle$; and
  - for Vs at index $\langle\Delta,$ F,VS,Vs$\rangle$. ■

4.1.9. Part Observers

- By a part observer$_\delta$ or a material observer$_\delta$ we mean
  - a meta-physical operator$_\delta$ (a meta function),

  72. **obs_**B: P → B

  - that is, one performed by the domain analyser,
  - which "applies" (i.e., who applies it) to a composite part value[13], P,
  - and which yields the sub-part of type B,
  - of the examined part.

[13]or composite part type

- We name these **obs**_erver functions **obs_X** to indicate that they are observing **parts** of **type** X.
- The **obs**_erver functions are not computable.
  - ◈ They can not be mechanised.
  - ◈ Therefore we refer to them as mental.
  - ◈ They can be "implemented" as, for example, follows:

Example: 12 Implementation of Observer Functions.

- I take you around a particular road net, n,say in my town.
- I point out to you, one-by-one,
  all the street intersections, $h_1, h_2, \ldots, h_n$, of that net.
- You "write" them down:
  - as many characteristics as you (and I) can come across,
    - including some choice of **unique identifiers**,
    - their **mereologies**, and
    - **attributes**, "one-by-one".
- In the end we have identified, i.e., visited,
  all the hubs in my town's road net n. ■

Example: 13 Observer Functions.

- We have exemplified the following obs_erver functions:
  - **obs_**N (Item 1(a) on Slide 38),
  - **obs_**F (Item 1(b) on Slide 38),
  - **obs_**M (Item 1(c) on Slide 38),
  - **obs_**HS (Item 2(a) on Slide 39),
  - **obs_**LS (Item 2(b) on Slide 39),
  - **obs_**VS (Item 3 on Slide 40),
  - **obs_**Vs (Item 4(a) on Slide 41),
  - **obs_**Hs (Item 5 on Slide 44) and
  - **obs_**Ls (Item 6 on Slide 44),

where we list their “definitions”, not their many uses. 🟩

4.1.10. Part Types

- By a **concrete type**$_\delta$ we shall understand a type, T,
  - which has been given both a name
  - and a defining type expression of, for example the form
    - T = A-**set**,
    - T = A-**infset**,
    - T = A$\times$B$\times \cdots \times$C,
    - T = A*,
    - T = A$^\omega$,
    - T = A $\xrightarrow[m]{}$B,
    - T = A$\rightarrow$B,
    - T = A$\overset{\sim}{\rightarrow}$B, or
    - T = A$|$B$|\cdots|$C.
  - where A, B, ..., C are **type names** or **type expressions**.

Example: 14 Concrete Types.

- Example **concrete part types** were exemplified in
  - Vs = V-**set**: Item 4(a) on Slide 41,
  - Hs = H-**set**: Item 5(a) Slide 44,
  - Ls = L-**set**: Item 6(a) Slide 44. ■

Example: 15 Has Composite Types.

- The discovery of concrete types were done as follows:
  - for HS, Hs = H-**set** at ⟨Δ,N,HS⟩,
  - for LS, Ls = L-**set** at ⟨Δ,N,LS⟩, and
  - for VS, Vs = V-**set** at ⟨Δ,F,VS⟩. ■

4.2. Part Properties

- (I) By a **property**[14] we mean a pair
  - a (finite) collection of one or more propositions.
- (II) By an **endurant property**
  - a **property** which holds of an **endurant** —
  - which we *model* as a *pair* of a **type** and a **value** (of that type)[15].
- (III) By a **perdurant property**$_\delta$ we shall mean
  - a **property** which holds of an **perdurant** —
  - which we, as a minimum, *model* as a *pair* of a **perdurant name** and a **function type**,
  - that is, as a **function signature**.

[14] By saying 'a property' we definitely mean to distinguish our use of the term from one which refers to legal property such as physical (land) or intangible (legal rights) property.

[15] The **type value** may be a singleton, or lie within a range of discrete values, or lie within a range of continuous values. The ranges may be finite or may be infinite.

- **Property Value Scales:**
  - ⟐ With **intentional properties** we associate a **property value scale**.
  - ⟐ By a **property value scale**$_\delta$ of a **part type** we shall mean
    - ⚭ a **value range** that **parts** of that **type**
    - ⚭ will have their **property values** range over.

Example: 16 **Property Value Scales.** We continue Example 4.

- The mereology **property value scale**$_\delta$ for hubs of a net range over finite sets of link identifiers of that net.
- The mereology **property value scale**$_\delta$ for links of a net range over two element sets of hub identifiers for that net.
- The range of location values for any one hub of a net is restricted to not share any cadestral point with any other hub's location value for that net.

- **Discussion:**
  - The notion of 'property' is central to much philosophical discussion; we mention a few (that we have studied):
    - [Chris Fox: The Ontology of Language: Properties, Individuals and Discourse, 2000],
    - [Simons: Parts – A Study in Ontology, 1987] and
    - [Mellor & Oliver (eds.): Properties].[16]

  Their reading has influenced our work.

[16] A reading of the contents listing of [Mellor & Oliver] reveals an interpretation of *parts and properties:*

I Function and Concept, Gottlob Frege	IX On the Elements of Being: I, Donald C. Williams
II The World of Universals, Bertrand Russell	X The Metaphysic of Abstract Particulars, Keith Campbell
III On our Knowledge of Universals, Bertrand Russell	XI Tropes, Chris Daly
IV Universals, F. P. Ramsey	XII Properties, D. M. Armstrong
V On What There Is, W. V. Quine	XIII Modal Realism at Work: Properties, David Lewis
VI Statements about Universals, Frank Jackson	XIV New Work for a Theory of Universals, David Lewis
VII 'Ostrich Nominalism'|'Mirage Realism', Michael Devitt	XV Causality and Properties, Sydney Shoemaker
VIII Against 'Ostrich' Nominalism, D. M. Armstrong	XVI Properties and Predicates, D. H. Mellor.

- ⟐ The notion of ‘property’ is also central to the recent notion of **concept analysis** [Ganter and Wille: Formal Concept Analysis — Mathematical Foundations, 1999].
  - ⦶ Here the term **concept** is understood as *a property of a part*.
  - ⦶ There is no associated type and value notions such as we have expressed in (II) on Slide 157 and Footnote 15 on Slide 157.
  - ⦶ We shall have more to say about the relations between our concept of **domain analysis** and Will & Ganter’s **concept analysis**
    - ∗ starting on Slide 124 and
    - ∗ in Item (iii) starting on Slide 461.
- We shall now unravel our ‘Property Theory’[17] of parts.

[17]— with apologies to `[Turner:1990,Turner:1992,ChrisFox2000]`.

- We see three categories of **part properties**:
  - ◈ **unique identifiers**,
  - ◈ **mereology** and
  - ◈ (general) **attributes**.
- Each and every **part** has **unique existence**
  — which we model through **unique identifiers**.
- **Parts** relate (somehow) to other **parts**, that is, **mereology**
  — which we model a relations between **unique identifiers**.
- And **parts** usually have other, additional properties
  which we shall refer to as **attributes**
  — which we model as pairs of **attribute types** and **attribute values**.

4.2.1. Unique Identifiers

Example: 17 Unique Identifier Functions.

- We have only exemplified the following **unique identifier** meta-functions and types:
  - **uid_**H, HI Item 7(a) on Slide 47,
  - **uid_**L, LI Item 7(b) on Slide 47 and
  - **uid_**V, VI Item 7(c) on Slide 47.
- We did not find a need for defining **unique identifier** meta-functions for N, F, M, HS, Hs, LS, Ls, VS, and Vs. ■

4.2.1.1 A Dogma of Unique Existence

- We take, as a dogma, that
  - every two parts whose **intentional property values** differ for at least one property,
  - other than their **unique identifiers**,
  - are distinct and
  - thus have distinct **unique identifiers**.

4.2.1.2 A Simplification on Specification of Intentional Properties

- So we make a simplification in our treatment of intentional part properties
  - By postulating distinct unique identifiers
  - we are forcing distinctness of parts
  - and can dispense with,
    - that is, do not have to explicitly ascribe such intentional properties
    - whose associated values would then have to differ in order to guarantee distinctness of parts,

4.2.1.3 Discussion

- Parts have unique existence.
  - Whether they be spatial or conceptual.
  - Two **manifest parts** cannot overlap spatially.
  - A part is a **conceptual part** if it is an abstraction of a part.
  - Two **conceptual parts** are identical
    - if they have identical properties,
    - that is, abstract the same manifest part,
    - otherwise they are distinct.
  - We shall therefore associate with each part
    - a **unique identifier**,
    - whether we may need to refer to that property or not.
  - There are only **manifest parts** and **conceptual parts**.

4.2.1.4 The uid_P Operator

- More specifically we postulate, for every **part**, p:P, a meta-function:

73. **uid_**P: P $\rightarrow$ Π

- where Π is the type of the **unique identifiers** of **parts** p:P.

- In practice
  - we "construct" the **unique identifier type name** for parts of type P by "suffixing" I to P, and
  - we explicitly "postulate define" the meta-function shown in Item 73 on the facing slide.
- How is the uid_PI meta-function "implemented" ?
  - Well, for a **domain description** it suffices to postulate it.
  - If we later were to develop software in support of the described domain, then there are many ways of "implementing" the uid_PIs.

4.2.1.5 Constancy of Unique Identifiers — Some Dogmas

- We postulate the following dogmas:
  - ◈ parts may be "added" to or "removed" from a domain;
  - ◈ parts that are "added" to a domain have **unique identifiers** that are not identifiers of any other part of the history of the domain;
  - ◈ parts that are "removed" from a domain will not have their identifiers reused should parts subsequently be "added" to the domain; and
  - ◈ domains do not allow for the changing (**update**) of **unique identifier values**.

4.2.2. Mereology

- **Mereology:** By mereology$_\delta$ (Greek: $\mu\epsilon\rho o\varsigma$) we shall understand the study and knowledge about
  - the theory of ***part-hood*** relations:
    - of the relations of ***part*** to ***whole*** and
    - the relations of ***part*** to ***part*** within a ***whole***.

- In the following please observe the type font distinctions:
  - *part*, etc., and
  - part (etc.).
- In the above definition of the term mereology
  - we have used the terms
    - *part-hood,*
    - *part* and
    - *whole*
  - in a more general sense than we use the term part.

- In this the “more general sense”
  - we interpret ***part*** to include,
    - besides what the term **part** covers in this seminar,
    - also concepts, abstractions, derived from the concept of **part**.

- That is, by ***part*** we mean
  - not only **manifest phenomena**
  - but also **intangible phenomena**
    - that may be **abstract models** of **parts**,
    - or may be (further) **abstract models** of ***parts***.

Example: 18 Manifest and Conceptual Parts. We refer to Example 4.

- A net, **n:N** (Item 1(a) on Slide 38), is a manifest **part**
- whereas a map, **rm:RM** (Item 26 on Slide 65), is a ***part***. ■

4.2.2.1 Extensional and Intentional Part Relations

- Henceforth we shall "merge" the two terms
  - ⟐ *part* and
  - ⟐ part

  into one meaning.
- So henceforth the term **part** shall refer to
  - ⟐ both **manifest**, **tangible** and **discrete endurants**
  - ⟐ and to **abstraction**s of these.

- We are forced to do so by necessity.
  - Instead of describing the **manifest phenomena**
  - we are describing conceptual models of these;
- that is,
  - instead of describing **manifest parts**
  - we are describing their **part types** and **part properties**.

- Thus we choose "mereology" to model relations between both
  - **parts** and
  - ***parts***.
- We can thus distinguish between two kinds of such relations:
  - **extensional part relations** which typically are **spatial relations** between **manifest parts** and
  - **intentional part relations** which typically are **conceptual relations** between **abstract parts**.

- **Extensional relations** between **manifest parts** are of the kind:
  - one **part**, p:P, is *"adjacent to"* (*"physically neighbouring"*) another **part**, q:Q,
  - one **part**, p:P, is *"embedded within"* (*"physically surrounded by"*) another **part**, q:Q, and
  - one **part**, p:P, *"overlaps with"* another **part**, q:Q.
- We model these relations, "equivalently", as follows:
  - in the mereology of p, **mereo**_P(p), there is a reference, **uid**_Q(q), to q, and
  - in the mereology of q, **mereo**_Q(q), there is a reference, **uid**_P(p), to p.

- **Intentional relations** between **abstractions** are of the kind:
  - **part p:P**
    - has an **attribute**
    - whose **value**
    - always stand in a certain relation
      - (for example, a copy of a fragment or the whole)
  - to another **part q:Q**'s "corresponding" **attribute value**.

Example: 19 Shared Route Maps and Bus Time Tables. We continue and we extend Example 4.

- The 'Road Transport Domain' of Example 4
  - has its **fleet** of **vehicle**s be that of a metropolitan city's busses
  - which ply some of the **route**s according to the city **road map** (i.e., the **net**) and
  - according to a **bus time table** — which we leave undefined.

- We can now re-interpret the **road traffic monitor** to represent a **coordinating bus traffic authority**, `CBTA`.
  - `CBTA` is now the "new" **monitor**, i.e., is a **part**.
  - Two of its **attribute**s are:
    - a **metropolitan area road map** and
    - a **metropolitan area bus time table**
  - **Vehicles** are now **busses**
    - and each **bus**
      - follows a route of the **metropolitan area road map**
      - of which it has a copy, as a **vehicle attribute**,
      - "shared" with `CBTA`;
    - each **bus** additionally
      - runs according to the **metropolitan area bus time table**
      - of which it has a copy, as a **vehicle attribute**,
      - "shared" with `CBTA`. ■

- We model these **attribute value relations**, " equivalently", as above:
  - ◈ in the mereology of p, **mereo**_P(p), there is a reference, **uid**_Q(q), to q, and
  - ◈ in the mereology of q, **mereo**_Q(q), there is a reference, **uid**_P(p), to p.

Example: 20 Monitor and Vehicle Mereologies. We continue Example 19 on Slide 177.

74. **value** **mereo**_M: VI-**set**

75. **type** MI

76. **value** **uid**_M: M $\rightarrow$ MI

77. **value** **mereo**_V: V $\rightarrow$ MI ■

4.2.2.2 Unique Part Identifier Mereologies

- To express a unique part identifier mereology
  - assumes that the related parts
  - have been endowed, say explicitly,
  - with unique part identifiers.,
  - say of unique identifier types
  - $\Pi_j, \Pi_k, \ldots, \Pi_\ell$.
- A mereology meta function is now postulated:

78. **value** **mereo_**P: P $\rightarrow$ $(\Pi_j \mid \Pi_k \mid \ldots \mid \Pi_\ell)$**-set**,

  - or of some such signature,
  - one which applies to parts, p:P,
  - and yields unique identifiers
  - of other, “the related”, parts —
  - where these “other parts” can be of any part type,
  - including P.

Example: 21 Road Traffic System Mereology.

- We have exemplified unique part identifier mereologies for
  - ◈ hubs, mereo_H Item 8(a) on Slide 48 and
  - ◈ links, mereo_L Item 9(a) on Slide 48. ■

Example: 22 Pipeline Mereology. This is a somewhat lengthy example from a domain now being exemplified.

- We start by narrating a pipeline domain of pipelines and pipeline units.

79. A pipeline consists of pipeline units.
80. A pipeline unit is either
    - a a well unit output connected to a pipe or a pump unit;
    - b a pipe, a pump or a valve unit input and output connected to two distinct pipeline units other than a well;
    - c a fork unit input connected to a pipeline unit other than a well and output connected to two pipeline units other than wells and sinks;
    - d a join unit input connected to two pipeline units other than wells and output connected to a a pipeline unit other than a sink; and
    - e a sink unit input connected to a valve.

```
type
79.   PL
value
79.   obs_Us: PL → U-set
type
80.   U = WeU | PiU | PuU | VaU | FoU | JoU | SiU
value
80.   uid_U: U → UI
80.   mereo_U: U → UI-set × UI-set
80.   i_mereo_U,o_mereo_U: U → UI-set
80.   i_UIs(u) ≡ let (ius,_) = mereo_U(u) in ius end
80.   o_UIs(u) ≡ let (_,ous) = mereo_U(u) in ous end
axiom
    ∀ pl:PL,u:U • u ∈ obs_Us(pl) ⇒
80(a).   is_WeU(u) → card i_UIs(u)=0 ∧ card o_UIs(u)=1,
80(b).   (is_PiU|is_PuU|is_VaU)(u) → card i_UIs(u)=1=card o_UIs(u),
80(c).   is_FoU(u) → card i_UIs(u)=1 ∧ card o_UIs(u)=2,
80(d).   is_JoU(u) → card i_UIs(u)=2 ∧ card o_UIs(u)=1,
80(e).   is_SiU(u) → card i_UIs(u)=1 ∧ card o_UIs(u)=0
```

- The UI "typed" **value** and **axiom** Items 80 "reveal" the mereology of pipelines. 🟩

4.2.2.3 Concrete Part Type Mereologies

- Let A_i and B_j, for suitable i, j denote distinct part types and let $\mathsf{B}_j\mathsf{I}$
- Let there be the following concrete type definitions:

type

$$
\begin{aligned}
&a_1{:}A_1 = bs{:}B_1\textbf{-set}\\
&a_2{:}A_2 = bc{:}B_{2_1} \times B_{2_2} \times \ldots \times B_{2_n}\\
&a_3{:}A_3 = bl{:}{B_3}^*\\
&a_4{:}A_4 = bm{:}BI_4 \;\overrightarrow{m}\; B_4
\end{aligned}
$$

- The above part type definitions can be interpreted mereologically:
  - Part $\mathsf{a}{:}\mathsf{A}_1$ has sub-parts b_{1_i},b_{1_2},...,b_{1_m}:B_1 of bs parthood related to just part $\mathsf{a}{:}\mathsf{A}_1$.
  - Parts $\mathsf{a}{:}\mathsf{A}_2$ has sub-parts b_{2_1},b_{2_2},...,b_{2_m}:B_2 of bc parthood related only to parts $\mathsf{a}{:}\mathsf{A}_1$
  - Parts $\mathsf{a}{:}\mathsf{A}_3$ has sub-parts b_{3_i}, for all indices i of the list $\mathsf{b}\ell$, parthood related to parts $\mathsf{a}{:}\mathsf{A}_3$, and to part $\mathsf{b}_{3_{i-1}}$ and part $\mathsf{b}_{3_{i+1}}$, for $1<i<\textbf{len}\,\mathsf{b}\ell$ by being “neighbours” and also to other b_{3_j} if the index j is known to b_{3_i} for $i{\neq}j$.
  - Parts $\mathsf{a}{:}\mathsf{A}_4$ have all parts $\mathsf{bm}(\mathsf{bi}_j)$ for index bi_j in the definition set $\textbf{dom}\,\mathsf{bm}$, be parthood related to $\mathsf{a}{:}\mathsf{A}_4$ and to other such $\mathsf{bm}{:}\mathsf{B}_4$ parts if they know their indexes.

Example: 23 A Container Line Mereology. This example brings yet another domain into consideration.

81. Two parts, sets of container vessels, **CV-set**, and sets of container terminal ports, **CTP-set**, are crucial to container lines, CL.
82. Crucial parts of container vessels and container terminal ports are their structures of *bays*, bs:BS.
83. A bay structure consists of an indexed set of *bay*s.
84. A *bay* consists of an indexed set of *rows*
85. A *row* consists of an index set of *stacks*.
86. A *stack* consists of a linear sequence of *container*s.

type
81. CP, CVS, CTPS
value
81. <u>**obs_**</u>CVS: CL $\rightarrow$ CVS
81. <u>**obs_**</u>CTPS: CL $\rightarrow$ CTPS
type
81. CVS = CV-**set**
81. CTPS = CTP-**set**
value
82. <u>**obs_**</u>BS: (CV|CTP) $\rightarrow$ BS
type
83. BI, BS, B = BI $\overrightarrow{m}$ B
value
84. <u>**obs_**</u>RS: B $\rightarrow$ RS
type
84. RI, RS, R = RI $\overrightarrow{m}$ R
value
85. <u>**obs_**</u>SS: R $\rightarrow$ SS
type
85. SI, SS, C = SI $\overrightarrow{m}$ S
86. S = C*

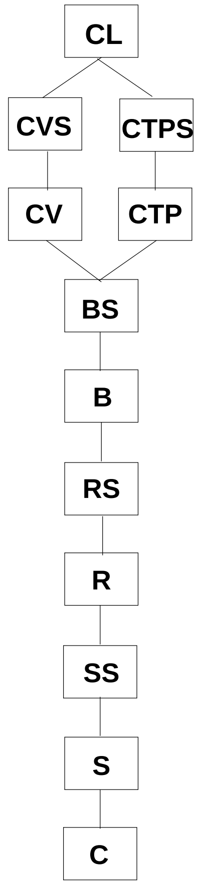


Figure 1: A container line domain index lattice

- In Fig. 1 on the preceding slide is shown a container line domain index lattice.
  - At the top ("root") there is the container line domain type name.
  - Immediately below it are the, in this case, two sub-domains (that we consider), CVS and CTPS.
  - For each of these two there are the corresponding CV and CTP sun-domains.
  - For each of these one can observe the container bays, hence, definition-wise, shared sub-domain.
  - It is then defined in terms of a sequence of increasingly more "narrowly" defined sub-domains.
  - The lattice "ends" with the atomic sub-domain of containers, C.■

4.2.2.4 Variability of Mereologies

- The **mereology** of **parts** (of **type** P) may be
  - a **constant**, i.e., **static**, or
  - a **variable**, i.e., **dynamic**.
- That is, for some, or all, **parts** of a **part type** may need to be **update**d.
  - We express the **update** of a **part mereology** as follows:

    87. **value** <u>**upd_mereo_**</u>P: $(\Pi_i|\Pi_i|\ldots|\Pi_i)$**-set** $\rightarrow$ P $\rightarrow$ P
  - where <u>**upd_mereo_**</u>P($\{\pi_a, \pi_b, \ldots, \pi_c\}$)(p)
  - results in a part p′:P where
    - all **part properties** of p′
      - other than its **mereology**
      - are as they "were" in p
      - but the **mereology** of p′ is $\{\pi_a, \pi_b, \ldots, \pi_c\}$.

Example: 24 Insert Link. We continue Example 4, Item 42 on Slide 87:

- In the post_link_dis predicate we referred to the undefined link insert function, ins_L.
- We now define that function:

88. The insert_Link action applies to a net, n, and a link, l,
89. and yields a new net, n′.

90. The conditions for a successful insertion are
    - a that the link, l, is not in the links of net n,
    - b that the **unique identifier** of l is not in the set of **unique identifiers** of the net n, and
    - c that the **mereology** of link l has been prepared to be, i.e., is the two element set of **unique identifiers** of two hubs in net n.
91. The result of a successful insertion is
    - a that the links of the new net, n′, are those of the previous net, n, "plus" link l;
    - b that the hubs, "originally" h_a,h_b, connected by l, are only mereo-logically updated to each additional include the **unique identifier** of l; and
    - c that all other hubs of n and n′ are unchanged.

```
88.    ins_L: N → L → N
89.    ins_L(n)(l) as n′
90.       pre:
90(a).           l ∉ obs_Ls(obs_LS(n))
90(b).         ∧ uid_L(l) ∉ in xtr_LIs(n)
90(c).         ∧ mereo_L(l) ⊆ xtr_HIs(n)
91.       post:
91(a).           obs_Ls(obs_LS(n′))=obs_Ls(obs_LS(n))∪{l}
91.          ∧ let {hi_a,hi_b}=mereo_L(l) in
91.            let {h_a,h_b}={get_H(hi_a)(n),get_H(hi_b)(n)} in
91(b).           get_H(hi_a)(n′)=upd_mereo_H(h_a)(mereo_H(h_a)∪{uid_L(l)})
91(b).         ∧ get_H(hi_b)(n′)=upd_mereo_H(h_b)(mereo_H(h_b)∪{uid_L(l)})
91(c).         ∧ obs_Hs(obs_HS(n))=obs_Hs(obs_HS(n))\{hi_a,hi_b}∪{h_a′,h_b′} end end
```

- As for the very many other **function definitions** in this seminar
  - we illustrate one form of **function definition annotations**,
  - and not always consistently the same “style”.
- We do not pretend that our **function definitions**
  - are novel, let alone a contribution of this seminar;
  - instead we rely on the listener
  - having learnt, more laboriously than we this seminar can muster,
  - an appropriate **function definition narrative style**. ■

● ● ●

4.2.3. Attributes

- **Attribute:** By a part attribute$_\delta$ we mean
  - ⟐ a part property
    - ⊚ other than part unique identifier and
    - ⊚ part mereology,
  - ⟐ and its associated attribute property value.

Example: 25 Road Transport System Part Attributes. We have exemplified, Example 4, a number of part attribute observation functions:

- attr_LΣ Item 10(a) on Slide 52,
- attr_LΩ Item 10(b) on Slide 52,
- attr_LOC, attr_LEN Item 10(c) on Slide 52,
- attr_HΣ Item 11(a) on Slide 54,
- attr_HΩ Item 11(b) on Slide 54,
- attr_LOC Item 11(c) on Slide 54,
- attr_VP, attr_onL, attr_atH, attr_VEL and attr_ACC Item 13 on Slide 56. ■

4.2.3.1 Stages of Attribute Analysis

- There are four facets to deciding upon part attributes:
  - (i) determining on which attributes to focus;
  - (ii) selecting appropriate **attribute type names**,
    (**viz.**, LΣ, LΩ, HΣ, HΩ, LEN, LOC, VP, atH, onL, VEL and ACC);
  - (iii) determining whether an **attribute type** is
    - a **static attribute type** (having **constant value**)
      (**viz.**, LEN, LOC), or
    - a **dynamic attribute type** (having **variable values**))
      (**viz.**, LΣ, LΩ, HΣ, HΩ, VP, atH, onL, VEL, ACC);

    and
  - (iv) deciding upon possible **concrete type definitions** for (some of) those **attribute types**
    (**viz.**, LΣ, LΩ, HΣ, HΩ, VP, atH, onL).

Example: 26 **Static and Dynamic Attributes.** Continuing Example 4 we have:

- Dynamic attributes:
  - ◈ LΣ Item 10(a) on Slide 52;
  - ◈ HΣ Item 11(a) on Slide 54;
  - ◈ VP, atH, onL Items 12(a)–12((a))ii on Slide 56; and
  - ◈ VEL and ACC both Item 13 on Slide 56.
- All other attributes are considered static. ■

Example: 27 Concrete Attribute Types. From Example 4:

- $\mathsf{L\Sigma}$=($\mathsf{HI}\times\mathsf{HI}$) Item 10(a) on Slide 52,
- $\mathsf{L\Omega}$=$\mathsf{L\Sigma}$-**set** Item 10(b) on Slide 52,
- $\mathsf{H\Sigma}$=($\mathsf{LI}\times\mathsf{LI}$)-**set** Item 11(a) on Slide 54 and
- $\mathsf{H\Omega}$=$\mathsf{H\Sigma}$-**set** Item 11(b) on Slide 54. ■

4.2.3.2 The attr_A Operator

- To observe a **part attribute** we therefore describe
  - the attribute observer signature

    92. **attr**_A: P → A,
  - where P is the **part type** being examined for **attribute**s, and
  - A is one of the chosen **attribute type names**.
- The "hunt" for
  - **part attribute**s, i.e., **attribute type**s,
  - the resulting **attribute function signatures** and
  - the chosen **concrete attribute types**

  is crucial for achieving successful **domain descriptions**.

4.2.3.3 Variability of Attributes

- Static attributes are constants.
- Dynamic attributes are variables.
- To express the update of any one specific **dynamic attribute**value we use the meta-operator:

93. **value** <u>**upd_attr**</u>_A: A → P → P

- where <u>**upd_attr**</u>_A(a)(p) results in a part p′:P where
  - all **part properties** of p′
    - other than its the **attribute value** for **attribute** A
      - are as they "were" in p
    - but the **attribute value** for **attribute** A is a.

Example: 28 Setting Road Intersection Traffic Lights. We refer to Example 4, Items 11(a) (HΣ) and 11(b) (HΩ) on Slide 55.

- The intent of the hub state model
  (a hub state as a set of pairs of unique link identifiers) is
  - that it expresses the possibly empty set of allowed hub traversals,
  - from a link incident upon the hub
  - to a link emanating from that hub.

94. In order to "change" a hub state the **set_hub_state** action is performed,
95. It takes a hub and a hub state and yields a changed hub.
    The argument hub state must be in the state space of the hub.
    The result of setting the hub state is that the resulting hub has the argument state as its (updated) hub state.

value
94. set_hub_state: $\mathrm{H} \to \mathrm{H}\Sigma \to \mathrm{H}$
95. set_hub_state(h)(hσ) $\equiv$ $\underline{\textbf{upd\_attr\_}}\mathrm{H}\Sigma(\mathrm{h})(\mathrm{h}\sigma)$
95. **pre**: h$\sigma \in \underline{\textbf{attr_}}\mathrm{H}\Omega(\mathrm{h})$

- The hub state has not changed if $\underline{\textbf{attr\_}}\mathrm{H}\Sigma(\mathrm{h}) = \mathrm{h}\sigma$. ■

4.2.4. Properties and Concepts

- Some remarks are in order.

4.2.4.1 Inviolability of Part Properties

- Given any **part** p of **type** P
  - ◈ one cannot "remove" any one of its properties
  - ◈ and still expect the the **part** to be of **type** P.
- Properties are what "makes" parts.
- To put the above remark in "context"
  let us review Ganter & Wille's **formal concept analysis**
  [Ganter & Wille: Formal Concept Analysis, 1999].

4.2.4.2 Ganter & Wille: Formal Concept Analysis

- This review is based on [Ganter & Wille: Formal Concept Analysis, 1999].

◈ $\boxed{\text{TO BE WRITTEN}}$

4.2.4.3 The Extensionality of Part Attributes

- $\boxed{\text{TO BE WRITTEN}}$

4.2.5. Properties of Parts

- The properties of **parts** and **materials** are fully captured by
  - (i) the **unique part identifiers**,
  - (ii) the **part mereology** and
  - (iii) the full set of**part attributes** and **material attributes**
- We therefore postulate a **property function**
  - when when applied to a **part** or a **material**
  - yield this triplet, (i–iii), of properties
  - in a suitable structure.

type

$\quad \text{Props} = \{|\text{PI}|\textbf{nil}|\} \times \{|(\text{PI-}\textbf{set}\times\ldots\times\text{PI-}\textbf{set})|\textbf{nil}|\} \times \text{Attrs}$

value

$\quad \text{props: Part}|\text{Material} \rightarrow \text{Props}$

- where
  - Part stands for a **part type**,
  - Material stands for a **material type**,
  - PI stand for **unique part identifiers** and
  - **PI-set**$\times\ldots\times$**PI-set** for **part mereologies**.
- The {|...|} denotes a proper specification language sub-type and **nil** denotes the empty type.

4.3. States

- By a state_δ we mean
  - ◈ a collection of such **parts**
  - ◈ some of whose **part attribute values** are **dynamic**,
  - ◈ that is, can vary.

Example: 29 A Variety of Road Traffic Domain States. We continue Example 4.

- A link, l:L, constitutes a state by virtue of if its link traffic state lσ:**attr**_LΣ.
- A hub, h:H, constitutes a state by virtue of its
  - ◈ hub traffic state hσ:**attr**_HΣ, and
  - ◈ indepenently, its hub mereology lis:LI-set:**mereo**_H.
- A net, n:N, constitutes a state by virtue of if its link and hub states.
- A monitor, m:M, constitutes a state by virtue of if its vehicle position map vpm:**attr**_VPM. ■

4.4. An Example Domain: Pipelines

- We close this lecture with a "second main example", albeit "smaller", in text size, than Example 4.
- The domain is that of pipelines.
- The reason we bring this example is the following:
  - Not all **domain endurants** are **discrete domain endurants**.
  - Some domains possess **continuous domain endurants**.
  - We shall call them materials.
  - Two such materials are
    - **liquids**, like `oil` (or `petroleum`), and
    - **gaseous**, like `natural gas`.

- The description of such materials-based domains requires
  - ◈ additional description concepts and
  - ◈ new description techniques.
- The examples illustrates these new concepts and techniques
- as do the examples of Sect. 6.1.

Example: 30 Pipeline Units and Their Mereology.

96. A pipeline consists of connected units, u:U.

97. Units have unique identifiers.

98. And units have mereologies, ui:UI:

    a pump, pu:Pu, pipe, pi:Pi, and valve, va:Va, units have one input connector and one output connector;

    b fork, fo:Fo, [join, jo:Jo] units have one [two] input connector[s] and two [one] output connector[s];

    c well, we:We, [sink, si:Si] units have zero [one] input connector and one [zero] output connector.

    d Connectors of a unit are designated by the unit identifier of the connected unit.

    e The auxiliary sel_UIs_in selector funtion selects the unique identifiers of pipeline units providing input to a unit;

    f sel_UIs_out selects unique identifiers of output recipients.

```
type
96. U = Pu | Pi | Va | Fo | Jo | Si | We
97. UI
value
97. uid_U: U → UI
98. mereo_U: U → UI-set × UI-set
98. wf_mereo_U: U → Bool
98. wf_mereo_U(u) ≡
98(a).     let (iuis,ouis) = mereo_U(u) in
98(a).     is_(Pu|Pi|Va)(u) → card iusi = 1 = card ouis,
98(b).     is_Fo(u) → card iuis = 1 ∧ card ouis = 2,
98(b).     is_Jo(u) → card iuis = 2 ∧ card ouis = 1,
98(c).     is_We(u) → card iuis = 0 ∧ card ouis = 1,
98(d).     is_Si(u) → card iuis = 1 ∧ card ouis = 0 end
98(e). sel_UIs_in: U → UI-set
98(e). sel_UIs_in(u) ≡ let (iuis,_)=mereo_U(u) in iuis end
98(f). sel_UIs_out: U → UI-set
98(f). sel_UIs_out(u) ≡ let (_,ouis)=mereo_U(u) in ouis end
```

Example: 31 Pipelines: Nets and Routes.

99. A pipeline net consists of several properly connected pipeline units.

    Example 30 on Slide 210 already described pipeline units.

    Here we shall concentrate on their connectedness, i.e., the wellformednes of pipeline nets.

100. A pipeline net is well-formed if

    a all routes of the net are **acyclic**, and

    b there are a non-empty set of **well-to-sink routes** that connect any well to some sink, and

    c all other routes of the net are **embedded** in the well-to-sink routes

```
type
99.     PLN′
99.     PLN = {| pln:PLN′ · is_wf_PLN(pln) |}
value
99.     obs_Us: PLN → U-set
100.  is_wf_PLN: PLN′ → Bool
100.  is_wf_PLN(pln) ≡
100.            let rs = routes{pln} in
100(b).            well_to_sink_routes(pln)≠{}
100(c).         ∧ embedded_routes(pln) end
```

101. An acyclic route is a route where any element occurs at most once.
102. A well-to-sink route of a net, pln, is a route whose first element designates a well in pln and whose last element designates a sink in pln.
103. One non-empty route, r', is embedded in another route, r if the latter can be expressed as the concatenation of three routes: $\mathsf{r} = \mathsf{r}''\widehat{\ }\mathsf{r}'\widehat{\ }\mathsf{r}'''$ where r'' or r''' may be empty routes ($\langle\rangle$).

```
type
105.     $R' = UI^*$
100(a).  $R = \{r:R' \cdot is\_acyclic(r)\}$
value
100(a).  $is\_acyclic: R \to \mathbf{Bool}$
100(a).  $is\_acyclic(r) \equiv \forall\ i,j:\mathbf{Nat} \cdot i \neq j \wedge \{i,j\} \subseteq \mathbf{inds}\ r \Rightarrow r[i] \neq r[j]$

100(b).  $well\_to\_sink\_routes: PLN \to R\text{-}\mathbf{set}$
100(b).  $well\_to\_sink\_routes(pln) \equiv$
100(b).      $\{r|r:R \cdot r \in routes(pln) \wedge \exists\ we:WE,si:Si \cdot$
100(b).          $\{we,si\} \subseteq \underline{\mathbf{obs\_}}Us(pln) \Rightarrow r[1]=we \wedge r[\mathbf{len}\ r]=si\}$
```

104. One non-empty route, er, is_embedded in another route, r,
   a if there are two indices, i, j, into r
   b such that the sequence of r elements from and including i to and including j is er.

value

```
104.     is_embedded: R × R → **Bool**
104.     is_embedded(er,r) ≡
104(a).        $\exists$ i,j:**Nat**·{i,j}$\subseteq$**inds** r
104(b).           $\Rightarrow$ er = $\langle$r[k]|k:**Nat** · i$\leq$k$\leq$j$\rangle$
104.        **pre**: er$\neq\langle\rangle$
```

105. A route, r, of a pipeline net is a sequence of **unique unit identifiers**, satisfying the following properties:

   a if $\mathsf{r[i]}=\mathsf{ui}_i$ has ui_i designate a unit, u, of the pipeline then $\langle \mathsf{ui}_i \rangle$ is a route of the net;

   b if $\mathsf{r}_i \widehat{\ } \langle \mathsf{ui}_i \rangle$ and $\langle \mathsf{ui}_j \rangle \widehat{\ } \mathsf{r}_j$ are routes of the net

      i. where u_i and u_j are the units (of the net) designated by ui_i and ui_j

      ii. and ui_j is in the output mereology of u_i and ui_i is in the input mereology of u_j

      iii. then $\mathsf{r}_i \widehat{\ } \langle \mathsf{ui}_i \rangle \widehat{\ } \langle \mathsf{ui}_j \rangle \widehat{\ } \mathsf{r}_j$ is a route of the net.

   c Only such routes that can be constructed by a finite number of "applications" of Items 105(a) and 105(b) are routes.

```
105. routes: PLN → R-set
105. routes(pln) ≡
105(a).       let rs = {⟨uid_UI(u)⟩|u:U•u ∈ obs_Us(pln)}
105((b))iii.        ∪ { $r_i$⌢⟨$ui_i$⟩⌢⟨$ui_j$⟩⌢$r_j$
105(b).               | $r_i$⌢⟨$ui_i$⟩,⟨$ui_j$⟩⌢$r_i$:R • {$r_i$⌢⟨$ui_i$⟩,⟨$ui_j$⟩⌢$r_j$}⊆rs
105((b))i.              ∧ let $u_i$,$u_j$:U•{$u_i$,$u_i$}⊆obs_Us(pln)∧$ui_i$=uid_U($u_i$)∧$ui_j$=uid_U($u_j$)
105((b))ii.               in $ui_i$ ∈ iuis($u_j$) ∧ $ui_j$ ∈ ouis($u_i$) end }
105(c).       in rs end
```

- Section 6.1 will continue with several examples
  - Example 43 on Slide 286,
  - Example 44 on Slide 288,
  - Example 45 on Slide 292,
  - Example 46 on Slide 296 and
  - Example 47 on Slide 299

  following up on the two examples of this section.

SOFTWARE ENGINEERING 2
SPECIFICATION OF SYSTEMS AND LANGUAGES
软件工程 系统与语言规约 卷2

SOFTWARE ENGINEERING 3
DOMAINS, REQUIREMENTS AND SOFTWARE DESIGN
软件工程 领域、需求与软件设计 卷3

See You After Lunch: 14:00 — Thanks !

Welcome Back — Thanks !

Lecture 3: 14:00–14:40 + 14:50–15:30
Discrete Perdurant and Contiuous Entities

5. Discrete Perdurant Entities

- From Wikipedia:
  - *Perdurant: Also known as occurrent, accident or happening.*
  - *Perdurants are those entities for which only a fragment exists if we look at them at any given snapshot in time.*
  - *When we freeze time we can only see a fragment of the perdurant.*
  - *Perdurants are often what we know as processes, for example 'running'.*
  - *If we freeze time then we only see a fragment of the running, without any previous knowledge one might not even be able to determine the actual process as being a process of running.*
  - *Other examples include an activation, a kiss, or a procedure.*
- A discrete perdurant$_\delta$ is a perdurant which is a discrete entity.

- We shall consider the following discrete perdurants.
  - actions (Sect. 5.1),
  - events (Sect. 5.2), and
  - discrete behaviours (Sect. 5.3).
- Actions and events
  - occur instantaneously,
  - that is, in time, but taking no time, and to therefore be
    - discrete action$_\delta$s and
    - discrete event$_\delta$s.

5.1. Formal Concept Analysis: Discrete Perdurants

- The domain analyser examines collections of discrete perdurants.
  - ❖ In doing so the domain analyser discovers and thus identifies and lists a number of perdurant properties.
  - ❖ Each of the discrete perdurants examined usually satisfies only a subset of these properties.
  - ❖ The domain analyser now groups discrete perdurant into collections
    - ⚭ such that each collection have its discrete perdurants satisfy the same set of properties,
    - ⚭ such that no two distinct collections are indexed, as it were, by the same set of properties, and
    - ⚭ such that all discrete perdurants are put in some collection.
  - ❖ The domain analyser now
    - ⚭ classify collections as actions, events or behaviours, and
    - ⚭ assign signatures
  - ❖ to distinct collections.
- That is how we assign signatures to discrete perdurants.

5.2. Actions

- By a function$_\delta$ we understand a **mathematical concept**,
  - a thing
  - which when **applied** to a **value**, called its **argument**,
  - **yield**s a **value**, called its **result**.
- A **discrete action**$_\delta$ can be understood as
  - a **function**
  - **invoked** on a **state value**
  - and is one that potentially changes that value.
- Other terms for **action** are
  - **function invocation**$_\delta$ and
  - **function application**$_\delta$.

Example: 32 Transport Net and Container Vessel Actions.

- *Inserting* and *removing* hubs and links in a net are considered actions.
- *Setting* the traffic signals for a hub (which has such signals) is considered an action.
- *Loading* and *unloading* containers from or unto the top of a container stack are considered actions. ■

5.2.1. Abstraction: On Modelling Domain Actions

- We claim that we describe **domain actions**,
  - ◈ but we actually describe functions,
  - ◈ which are “somewhat far removed” from domains.
- So what are we actually claiming ?
  - ◈ We are claiming that there is an **interesting class** of actions
  - ◈ and that they can all be abstracted into one, possibly **non-deterministic function**
  - ◈ whose properties are then claimed to “mimic” those of the actions in the **interesting class**.

5.2.2. Agents: An Aside on Actions

Think'st thou existence doth depend on time?
It doth; but actions are our epochs.
George Gordon Noel Byron,
Lord Byron (1788-1824) Manfred. Act II. Sc. 1.

- *"An action is*
  - *something an agent does*
  - *that was 'intentional under some description'"* [Davidson1980].
- That is, actions are performed by agents.
  - We shall not yet go into any deeper treatment of **agency** or **agent**s. We shall do so later.
    - **Agents** will here, for simplicity, be considered **behaviour**s,
    - and are treated later in this lecture.

- As to the relation between **intention** and **action**
  - we note that Davidson wrote:
    'intentional under some description'
  - and take that as our cue:
    - the agent follows a script,
    - that is, a behaviour description,
    - and invokes actions accordingly,
    - that is, follow, or honours that script.

5.2.3. Action Signatures

- By an **action signature** we understand a quadruple:
  - a **function name**,
  - a **function definition set type expression**,
  - a **total** or **partial function** designator ($\rightarrow$, respectively $\overset{\sim}{\rightarrow}$), and
  - a **function image set type expression**:
    fct_name: A $\rightarrow$ Σ $(\rightarrow|\overset{\sim}{\rightarrow})$ Σ $[\times$ R],

where $(X \mid Y)$ means either X or Y, and $[Z]$ means that for some signatures there may be a Z component meaning that the action also has the effect of "leaving" a type Z value.

Example: 33 Action Signatures: Nets and Vessels.

insert_Hub: $\mathsf{N}\to\mathsf{H}\overset{\sim}{\to}\mathsf{N}$;
remove_Hub: $\mathsf{N}\to\mathsf{HI}\overset{\sim}{\to}\mathsf{N}$;
set_Hub_Signal: $\mathsf{N}\to\mathsf{HI}\overset{\sim}{\to}\mathsf{H}\Sigma\overset{\sim}{\to}\mathsf{N}$
load_Container: $\mathsf{V}\to\mathsf{C}\to\mathsf{StackId}\overset{\sim}{\to}\mathsf{V}$; and
unload_Container: $\mathsf{V}\to\mathsf{StackId}\overset{\sim}{\to}(\mathsf{V}\times\mathsf{C})$. ■

5.2.4. Action Definitions

- There are a number of ways in which to characterise an action.
- One way is to characterise its underlying function
  by a pair of predicates:
  - precondition: a predicate over function arguments — which includes the state, and
  - postcondition: a predicate over function arguments, a proper argument state and the desired result state.
  - If the precondition holds, i.e., is **true**, then the arguments, including the argument state, forms a proper 'input' to the action.
  - If the postcondition holds, assuming that the precondition held, then the resulting state [and possibly a yielded, additional "result" (R)] is as they would be had the function been applied.

Example: 34 Transport Nets Actions.

- In Example 4 we gave an explicit example of an action:
  - ins_H: Items 37–37(d),
- while implicit references to net actions were made in the event predicates
  - link_dis, pre_link_dis: Items 38–39(c),
  - post_link_dis (Items 38–39(c)):
    - rem_L Item 42(a) and
    - ins_L Items 42((c))i–42((c))ii. ■

- What is not expressed, but tacitly assume in the above pre- and post-conditions is
  - that the state, here n, satisfy invariant criteria before (i.e. n) and after (i.e., n') actions,
  - whether these be implied by axioms
  - or by well-formedness predicates.

  over parts.
- This remark applies to any definition of actions, events and behaviours.
- There are other ways of defining functions.
- But the form of these are not germane to the aims of this seminar.

Modelling Actions, I/III

- We refer to the section on Formal Concept Analysis of Discrete Perdurants on Slide 222.
- The **domain describer** has decided that an **entity** is a **perdurant** and is, or represents an **action**: was *"done by an agent and intentionally under some description"* [Davidson1980].
  - ◇ The domain describer has further decided that the observed action is of a class of actions — of the "same kind" — that need be described.
  - ◇ By actions of the 'same kind' is meant that these can be described by the same **function signature** and **function definition**.

Modelling Actions, II/III

- The domain describer must decide on the underlying **function signature**.
  - The **argument type** and the **result type** of the signature are those of either previously identified
    - parts and/or materials,
    - unique part identifiers, and/or
    - attributes.

Modelling Actions, III/III

- Sooner or later the domain describer must decide on the **function definition**.
  - The form must be decided upon.
  - For pre/post-condition forms it appears to be convenient to have developed, "on the side", a **theory of mereology** for the part types involved in the function signature.

5.3. Events

- By an **event**$_\delta$ we understand
  - ⟐ *a state change*
  - ⟐ *resulting indirectly from an unexpected application of a function,*
  - ⟐ *that is, that function was performed "surreptitiously".*
- Events can be characterised by a pair of (before and after) states, a predicate over these and, optionally, a **time** or **time interval**.
- Events are thus like actions:
  - ⟐ change states,
  - ⟐ but are usually
    - either caused by "previous" actions,
    - or caused by "an outside action".

Example: 35 Events.

- *Container vessel:* A container falls overboard sometimes between times t and t'.
- *Financial service industry:* A bank goes bankrupt sometimes between times t and t'.
- *Health care:* A patient dies sometimes between times t and t'.
- *Pipeline system:* A pipe breaks sometimes between times t and t'.
- *Transportation:* A link "disappears" sometimes between times t and t'.

5.3.1. An Aside on Events

- We may observe an event, and
  - then we do so at a specific time or
  - during a specific time interval.
- But we wish to describe,
  - not a specific event
  - but a class of events of “the same kind”.
- In this seminar
  - we therefore do not ascribe
  - **time points** or **time intervals**
  - with the occurrences of events.

5.3.2. Event Signatures

- An event signature$_\delta$
  - *is a predicate signature*
  - *having an event name (evt),*
  - *a pair of state types ($\Sigma \times \Sigma$),*
  - *a total function space operator ($\rightarrow$)*
  - *and a* **Bool***ean type constant:*
  - *evt: (Σ×Σ)* $\rightarrow$ **Bool**.
- Sometimes there may be a good reason
  - for indicating the type, ET, of an event cause value,
  - if such a value can be identified:
  - evt: ET $\times$ $(\Sigma \times \Sigma) \rightarrow$ **Bool**.

5.3.3. Event Definitions

- An event definition$_\delta$ takes the form of
  - *a predicate definition:*
    - *a predicate name and argument list, usually just a state pair,*
    - *an existential quantification*
      - *over some part (of the state) or*
      - *over some dynamic attribute of some part (of the state)*
      - *or combinations of the above*
    - *a pre-condition expression over the input argument(s),*
    - *an implication symbol (⇒), and*
    - *a post-condition expression over the argument(s):*
  - *evt*(σ,σ') $=$ $\exists$ *(ev:ET)* $\bullet$ *pre_evt(ev)*(σ) $\Rightarrow$ *post_evt(ev)*(σ,σ').

There may be variations to the above form.

Example: 36 Road Transport System Event.

- Example 4,
  - Items 38–42((c))ii
  - (Slides 85–88)

  exemplified an event definition.

Modelling Events I/II

- We refer to the section on **Formal Concept Analysis of Discrete Perdurants** on Slide 222.
- The **domain describer** has decided that an **entity** is a **perdurant** and is, or represents an **event**: occurred surreptitiously, that is, was not an action that was *"done by an agent and intentionally under some description"* [Davidson1980].
  - The domain describer has further decided that the observed event is of a class of events — of the "same kind" — that need be described.
  - By events of the 'same kind' is meant that these can be described by the same **predicate function signature** and **predicate function definition**.

Modelling Events, II/II

- First the domain describer must decide on the underlying **predicate function signature**.
  - The **argument type** and the **result type** of the signature are those of either previously identified
    - parts,
    - unique part identifiers, or
    - attributes.
- Sooner or later the domain describer must decide on the **predicate function definition**.
  - For predicate function definitions it appears to be convenient to have developed, "on the side", a **theory of mereology** for the part types involved in the function signature.

5.4. Discrete Behaviours

- We shall distinguish between
  - **discrete behaviours** (this section) and
  - **continuous behaviours**.
- Roughly **discrete behaviours**
  - proceed in discrete (time) steps —
  - where, in this lecture, we omit considerations of time.
  - Each step corresponds to an **action** or an **event** or a time interval between these.
  - **Actions** and **events** may take some (usually inconsiderable time),
  - but the **domain analyser** has decided that it is not of interest to understand what goes on in the domain during that **time** (**interval**).
  - Hence the behaviour is considered discrete.

- Continuous behaviours
  - are **continuous** in the sense of the **calculus** of **mathematical analysis**;
  - to qualify as a **continuous behaviour time** must be an essential aspect of the **behaviour**.
- **Discrete behaviours** can be modelled in many ways, for example using
  - `CSP [Hoare85+2004]`.
  - `MSC [MSCall]`,
  - `Petri Nets [m:petri:wr09]` and
  - `Statechart [Harel87]`.
- We refer to Chaps. 12–14 of `[TheSEBook2wo]`.
- In this seminar we shall use `RSL/CSP`.

5.4.1. What is Meant by ‘Behaviour’ ?

- We give two characterisations of the concept of ‘behaviour’.
  - a “loose” one and
  - a “slanted one.
- A loose characterisation runs as follows:
  - by a **behaviour**$_\delta$ we understand
    - a set of sequences of
    - **action**s, **event**s and **behaviour**s.

- A "slanted" characterisation runs as follows:
  - by a **behaviour**$_\delta$ we shall understand
    - either a **sequential behaviour**$_\delta$ consisting of a possibly infinite sequence of zero or more actions and events;
    - or one or more **communicating behaviour**$_\delta$s whose **output actions** of one behaviour may **synchronise** and **communicate** with **input actions** of another behaviour;
    - or two or more **behaviours** acting either as **internal non-deterministic behaviour**$_\delta$s ($\sqcap$) or as **external non-deterministic behaviour**$_\delta$s ($\square$).

- This latter characterisation of behaviours
  - ⟐ is "slanted" in favour of a CSP, i.e., a **communicating sequential behaviour**, view of behaviours.
  - ⟐ We could similarly choose to "slant" a behaviour characterisation in favour of
    - ⚭ `Petri Nets`, or
    - ⚭ `MSC`s, or
    - ⚭ `Statechart`s, or other.

5.4.2. Behaviour Narratives

- Behaviour narratives may take many forms.
  - A behaviour may best be seen as composed from several interacting behaviours.
    - Instead of narrating each of these,
    - as was done in Example 4,
    - one may proceed by first narrating the interactions of these behaviours.
  - Or a behaviour may best be seen otherwise,
    - for which, therefore, another style of narration may be called for,
    - one that "traverses the landscape" differently.
  - Narration is an art.
  - Studying narrations – and practice – is a good way to learn effective narration.

5.4.3. Channels

- We remind the listener that we are focusing exclusively on domain behaviours.
  - Domain behaviours, as we shall see in Sect. 5.4.6, take their "root" in **parts**.
  - We shall find, even when "parts" take the form of concepts, that these do not "overlap".
    - They may share properties,
    - but we can consider them "disjoint".
  - Hence communication between processes
    - can be thought of as communication between "disjoint parts",
    - and, as such, can be abstracted as taking place
    - in a non-physical medium which we shall refer to as **channels**.

- By a channel$_\delta$ we shall understand
  - *a means of communicating entities*
  - *between [two] behaviours.*
- To express channel communications we, at present, make use of `RSL [RSL]`'s **out**put (ch ! v) / **in**put (ch ?) clauses and **channel** declarations,

  type M
  channel ch M,
  value ch!v, ch?,

- Variations of the above clauses are

  type ChIdx, ChJdx
  channel $\{ch[\,i\,] | i{:}ChIdx \cdot \mathcal{P}(i,...)\}{:}M$, $\{ch[\,i,j\,] | i{:}ChIdx,j{:}ChJdx \cdot \mathcal{P}(i,j,...)\}{:}M$
  value ch[i]!v, ch[i]?, ch[i,j]!v, ch[i,j]?

- where $\mathcal{P}$ is a suitable predicate
  - over channel indices and
  - possibly global domain values.

5.4.4. Behaviour Signatures

- By a **behaviour signature**$_\delta$ we shall understand *a*
  - *a function signature*
  - *augmented by a clause which declares*
    - *the* **in** *channels on which the function accepts inputs and*
    - *the* **out** *channels on which the function offers output.*

value behaviour: A → **in** in_chs **out** out_chs B

- where (i)
  - the form **in** in_chs **out** out_chs
    - may be just **in** in_chs
    - or **out** out_chs
    - or both **in** in_chs **out** out_chs

  that is, **behaviour** accepts input(s), or offers output(s), or both;

value behaviour: A → **in** in_chs **out** out_chs B

- where (ii)
  - A typically is of the forms
    - **Unit** if the behaviour "takes no arguments",
      - that is: behaviour(),

      or
    - PI×P if the behavior is directly based on a part, p:P, for
      - that is: behaviour(uid_P(p),p);

value behaviour: A → **in** in_chs **out** out_chs B

- where (iii)
- in_chs and out_chs are of the form
  - either ch,
  - or $\{ch[i]|i:ChIdx \cdot \mathcal{Q}(i,...)\}$
  - or $\{ch[i,j]|i:ChIdx,j:ChJdx \cdot \mathcal{R}(i,j,...)\}$,

  $\mathcal{Q}$, $\mathcal{R}$ are appropriate predicates; and
- where (iv)
  - either
  - B is
    - either just **Unit** when the behaviour is typically a never-ending (i.e., cyclic) behaviours,
    - or is some result type C.

5.4.5. Behaviour Definitions

- This section is about the basic form of **behaviour function definitions**.
  - We shall only be concerned with behaviours which define **part behaviours**.
  - By a **part behaviour**$_\delta$ we shall understand
    - *a behaviour whose state*
    - *is that of the part for which it is the behaviour.*
- There are basically two cases for which we are interested in the form of the behaviour definition:
  - the **atomic part behaviour**, and
  - the **composite part behaviour**.

5.4.5.1 Atomic Part Behaviours

- Let p:P be an **atomic part** of type P.
- Then the basic form of a cyclic **atomic behaviour definition** is

```
value
    atomic_core_part_behaviour(uid_P(p))(p) ≡
        let p′ = 𝒜(uid_P(p))(p) in
        atomic_core_part_behaviour(uid_P(p))(p′) end
        post: uid_P(p) = uid_P(p′),

    𝒜: PI → P → in ... out ...  P,
```

- where $\mathcal{A}$ usually is a terminating function
  - which synchronises and
  - communicates with other **part behaviours**.

Example: 37 Atomic Part Behaviours.

- Example 4, Sect. 2.8.6 and Sect. 2.8.7 illustrates cyclic atomic behaviours:
  - vehicle at Hub: Items 65–65(d), on Slide 101,
  - vehicle on Link: Items 64–68, on Slide 103 and
  - monitor: Items 69–71(d), on Slide 105.

■

5.4.5.2 Composite Part Behaviours

- Let p:P be an **atomic part** of type P.
- Then the basic form of a cyclic **atomic behaviour definition** is

```
value
    composite_part_behaviour(uid_P(p))(p) ≡
        composite_core_part_behaviour(uid_P(p))(p)
      ∥ { part_behaviour(uid_P(p′))(p′)|p′:P•p′ ∈ obs_(p)}

    core_part_behaviour: PI → P → in ... out ... Unit
    core_part_behaviour(uid_P(p))(p) ≡
        let p′ = 𝒞(uid_P(p))(p) in
        composite_core_part_behaviour(uid_P(p))(p′) end
        post: uid_P(p) = uid_P(p′)

    𝒞: PI → P → in ... out ...  P,
```

- where $\mathcal{C}$ usually is a terminating function
  - which synchronises and
  - communicates with other part behaviours.

Example: 38 Compositional Behaviours.

- Example 4, Sect. 2.8.3
  - illustrated compositionality,
  - cf. Items 59– 59(b) on Slide 95. ■

- The next section
  - illustrates the basic principles
  - that we recommend
  - when modelling behaviours of domains
  - consisting of composite and atomic parts.

5.4.6. A Model of Parts and Behaviours

- How often have you not “confused”, linguistically,
  - the perdurant notion of a train process: progressing from railway station to railway station,
  - with the endurant notion of the train, say as it appears listed in a train time table, or as it is being serviced in workshops, etc.
- There is a reason for that — as we shall now see:
  parts may be considered **syntactic quantities**
  denoting **semantic quantities**.
  - We therefore describe a general model of parts of domains
  - and we show that for each instance of such a model
  - we can ‘compile’ that instance into a `CSP` ‘program’.

- The example additionally has a more general aim,
  - ◈ *namely that of showing*
  - ◈ *that to every mereology (or parts)*
  - ◈ *there is a* λ*-expression*
  - ◈ *here in the form of basically a* `CSP [Hoare85+2004]` *program.*

Example: 39 Syntax and Semantics of Mereology.

5.4.6.1 A Syntactic Model of Parts

106. The *whole* contains a set of *parts*.
107. *Parts* are either *atomic* or *composite*.
108. From *composite parts* one can observe a set of *parts*.
109. All *parts* have *unique identifiers*

type

106. W, P, A, C

107. P = A | C

value

108. $\underline{\textbf{obs}_}$Ps: (W|C) $\to$ P-**set**

type

109. PI

value

109. $\underline{\textbf{uid}\_}\Pi$: P $\to$ Π

110. From a *whole* and from any *part* of that *whole* we can e**xtr**act all contained *parts*.
111. Similarly one can e**xtr**act the *unique identifiers* of all those contained *parts*.
112. Each part may have a *mereology* which may be “empty”.
113. A *mereology*’s *unique part identifiers* must refer to some other parts other than the part itself.

```
value
110. xtr_Ps: (W|P) → P-set
110. xtr_Ps(w) ≡ {xtr_Ps(p)|p:P·p ∈ obs_Ps(p)}
110.     pre: is_W(p)
110. xtr_Ps(p) ≡ {xtr_Ps(p)|p:C·p∈ obs_Ps(p)}∪{p}
110.     pre: is_P(p)
111. xtr_Πs: (W|P) → Π-set
111. xtr_Πs(wop) ≡ {uid_P(p)|p ∈ xtr_Ps(wop)}
112. mereo_P: P → Π-set
axiom
113. ∀ w:W
113.     let ps = xtr_Ps(w) in
113.     ∀ p:P · p ∈ ps · ∀ π:Π · π ∈ mereo_P(p) ⇒ π ∈ xtr_Πs(p) end
```

114. An **attribute map** of a *part* associates with *attribute names*, i.e., *type names*, their *valu*es, whatever they are.
115. From a *part* one can extract its attribute map.
116. Two *parts share attributes* if their respective **attribute map**s share *attribute names*.
117. Two *parts share properties* if the y
    a either *share attributes*
    b or the *unique identifier* of one is in the *mereology* of the other.

```
type
114.    AttrNm, AttrVAL,
114.    AttrMap = AttrNm $\overrightarrow{m}$ AttrVAL
value
115.    attr_AttrMap: P → AttrMap
116.    share_Attributes: P×P → Bool
116.    share_Attributes(p,p′) ≡
116.        dom attr_AttrMap(p) ∩
116.        dom attr_AttrMap(p′) ≠ {}
117.    share_Properties: P×P → Bool
117.    share_Properties(p,p′) ≡
117(a).        share_Attributes(p,p′)
117(b).    ∨ uid_P(p) ∈ mereo_P(p′)
117(b).    ∨ uid_P(p′) ∈ mereo_P(p)
```

5.4.6.2 A Semantics Model of Parts

118. We can define the set of two element sets of *unique identifiers* where
    - one of these is a *unique part identifier* and
    - the other is in the mereology of some other *part*.
    - We shall call such two element "pairs" of *unique identifiers* connectors.
    - That is, a **connector** is a two element set, i.e., "pairs", of *unique identifiers* for which the identified parts share properties.
119. Let there be given a 'whole', w:W.
120. To every such "pair" of *unique identifiers* we associate a *channel*
    - or rather a position in a matrix of *channels* indexed over the "pair sets" of *unique identifiers*.
    - and communicating messages m:M.

```
type
118.  K = Π-set axiom ∀ k:K•card k=2
value
118.  xtr_Ks: (W|P) → K-set
118.  xtr_Ks(wop) ≡
118.      let ps = xtr_Ps(w) in
118.      {{uid_P(p),π}|p:P,π:Π•p∈ ps ∧ ∃ p′:P•p′≠p∧π=uid_P(p′) ∧ uid_P(p)∈uid_P(p′)} end

119.  w:W

120.  channel {ch[ k ]|k:xtr_Ks(w)}:M
```

121. Now the ‘whole’ *behaviour* whole is the parallel composition of *part processes*, one for each of the immediate parts of the *whole*.

122. A *part process* is

   a either an *atomic part process*, atom, if the *part* is an *atomic part*,

   b or it is a *composite part process*, comp, if the *part* is a *composite part*.

121. whole: W $\rightarrow$ **Unit**
121. whole(w) $\equiv$ $\|$ {part(<u>**uid**_</u>P(p))(p) | p:P•p $\in$ xtr_Ps(w)}

122. part: π:Π $\rightarrow$ P $\rightarrow$ **Unit**
122. part(π)(p) $\equiv$
122(a). is_A(p) $\rightarrow$ atom(π)(p),
122(b). _ $\rightarrow$ comp(π)(p)

123. A *composite process*, part, consists of

   a a *composite core process*, comp_core, and

   b the parallel composition of *part processes* one for each *contained part* of part.

.

value

```
123.    comp: $\pi$:$\Pi$ $\rightarrow$ p:P $\rightarrow$ **in**,**out** {ch[{$\pi$,$\pi'$}|{$\pi'$$\in$ **mereo**_P(p)}]} **Unit**
123.    comp($\pi$)(p) $\equiv$
123(a).      comp_core($\pi$)(p) $\|$
123(b).   $\|$ {part(**uid**_P(p$'$))(p$'$) | p$'$:P•p$'$ $\in$ **obs**_Ps(p)}
```

124. An *atomic process* consists of just an *atomic core process*, *atom_core*

124. atom: π:Π $\rightarrow$ p:P $\rightarrow$ **in**,**out** {ch[{π,π'}|{$\pi'$$\in$ <u>**mereo**_</u>P(p)}]} **Unit**
124. atom(π)(p) $\equiv$ atom_core(π)(p)

125. The **core behaviours** both
   a update the **part properties** and
   b recurses with the updated properties,
   c without changing the part identification.

We leave the **update** action undefined.

```
value
125.   core: $\pi$:$\Pi$ $\rightarrow$ p:P $\rightarrow$ in,out {ch[{$\pi$,$\pi'$}|{$\pi'$$\in$ mereo_P(p)}]} Unit
125.   core($\pi$)(p) $\equiv$
125(a).      let p$'$ = update($\pi$)(p)
125(b).      in core($\pi$)(p$'$) end
125(b).      assert: uid_P(p)=$\pi$=uid_P(p$'$)
```

- The model of parts can be said to be a syntactic model.
  - No meaning was “attached” to parts.
- The conversion of parts into `CSP` programs can be said to be a semantic model of parts,
  - one which to every part associates a behaviour
  - which evolves “around” a state
  - which is that of the properties of the part.

6. Continuous Entities

- There are two kinds of **continuous entities**:
  - **material**s (Slides 279–300) and
  - **continuous behaviours** (Slides 301–315).
- By a **material**$_\delta$ we small mean
  - a **continuous endurant**,
  - a **manifest entity** which typically varies in shape and extent.
- By a **continuous behaviour**$_\delta$ we small mean
  - a **continuous perdurant**,
  - which we may think of as a **function**
    - from continuous $\mathbb{T}$ime
    - to some structure, simple or complicated, of
      - **part**s and
      - **material**s.

6.1. Materials

- Let us start with examples of materials.

Example: 40 Materials. Examples of endurant continuous entities are such as

- coal,
- air,
- natural gas,
- grain,
- sand,
- iron ore,
- minerals,
- crude oil,
- solid waste,
- sewage,
- steam and
- water. ■

The above **materials** are either

- **liquid materials** (crude oil, sewage, water),
- **gaseous materials** (air, gas, steam), or
- **granular materials** (coal, grain, sand, iron ore, mineral, or solid waste).

- **Endurant continuous entities**, or **materials** as we shall call them,
  - ⟐ are the **core endurants** of process domains,
  - ⟐ that is, **domains** in which those **materials** *form the basis* for their *"raison d'être"*.

6.1.1. Materials-based Domains

- By a **materials based domain**$_\delta$ we shall mean a **domain**
  - ⟐ *many of whose parts serve to transport materials, and*
  - ⟐ *some of whose actions, events and behaviours serve to monitor and control the part transport of materials.*

Example: 41 Material Processing.

- Oil or gas materials are ubiquitous to pipeline systems — so pipeline systems are oil or gas-based systems.
- Sewage is ubiquitous to waste management systems — so waste management systems are sewage-based systems.
- Water is ubiquitous to systems composed from reservoirs, tunnels and aqueducts which again are ubiquitous to hydro-electric power plants, irrigation systems or water supply utilities — so hydro-electric power plants, irrigation systems and water supply utilities are water-based systems. ■

- Ubiquitous means 'everywhere'.
- A continuous entity, that is, a material
  - is a core material,
  - if it is "somehow related"
  - to one or more parts of a domain.

6.1.2. "Somehow Related" Parts and Materials

- We explain our use of the term "somehow related".

Example: 42 Somehow Related Materials and Parts. With `teletype font` we designate materials and with *slanted font* we imply parts or part processes.

- `Oil` is pumped from *well*s, runs through *pipe*s, is "lifted" by *pump*s, diverted by *fork*s, "runs together" by means of *join*s, and is delivered to *sink*s.
- `Grain` is delivered to silos by trucks, piped through a network of pipes, forks and valves to vessels, etc.
- `Mineral`s are *mined*, *conveyed* by *belt*s to *lorries* or *trains* or *cargo vessels* and finally *deposited*.
- `Iron ore`, for example, is *'conveyed'* into *smelters*, *'roasted'*, *'reduced'* and *'fluxed'*, *'mixed'* with other mineral ores to produce a molten, pure metal, which is then *'collected'* into *ingots*. ■

6.1.3. Material Observers

- When **analysing domains** a key question,
  - in view of the above notion of **core continuous endurants** (i.e., materials)

  is therefore:
  - does the **domain** embody a notion of **core continuous endurants** (i.e., materials);
  - if so, then identify these “early on” in the **domain analysis**.
- Identifying materials —
  - their types and
  - attributes —

  is slightly different from identifying **discrete endurants**, i.e., **parts**.

Example: 43 Pipelines: Core Continuous Endurant. We continue Examples 30 on Slide 210 and 31 on Slide 212.

- The core continuous endurant, i.e., material,
- of (say oil) pipelines is, yes, oil:

```
type
    O material
value
    obs_O: PLN → O
```

- The keyword **material** is a pragmatic. ■

- Materials are "few and far between" as compared to parts,
  - we choose to mark the type definitions which designate materials with the keyword **material**.
  - In contrast, we do not mark the type definitions which designate parts with the keyword **discrete**.

- First we do not associate the notion of atomicity or composition with a material. Materials are continuous.
- Second, amongst the attributes, none have to do with geographic (or cadestral) matters. Materials are moved.
- And materials have no unique identification or mereology. No "part" of a material distinguishes it from other "parts".
- But they do have other attributes when occurring in connection with, that is, related to **parts**, for example,
  - volume or
  - weight.

Example: 44 Pipelines: Parts and Materials. We continue Examples 30 on Slide 210 and 31 on Slide 212.

126. From an oil pipeline system one can, amongst others,
    a observe the finite set of all its pipeline bodies,
    b units are composite and consists of a unit,
    c and the oil, even if presently, at time of observation, empty of oil.
127. Whether the pipeline is an oil or a gas pipeline is an attribute of the pipeline system.
    a The volume of material that can be contained in a unit is an attribute of that unit.
    b There is an auxiliary function which estimates the volume of a given "amount" of oil.
    c The observed oil of a unit must be less than or equal to the volume that can be contained by the unit.

type

126. PLS, B, U, Vol

126. O **material**

value

126(a). $\underline{\textbf{obs}_}$Bs: PLS → B-**set**

126(b). $\underline{\textbf{obs}_}$U: B → U

126(c). $\underline{\textbf{obs}_}$O: B → O

127. $\underline{\textbf{attr}_}$PLS_Type: PLS → {"oil"|"gas"}

127(a). $\underline{\textbf{attr}_}$Vol: U → Vol

127(b). vol: O → Vol

axiom

127(c). $\forall$ pls:PLS,b:B•b $\in$ $\underline{\textbf{obs}_}$Bs(pls)$\Rightarrow$vol($\underline{\textbf{obs}_}$O(b))$\leq$$\underline{\textbf{attr}_}$Vol($\underline{\textbf{obs}_}$U(b))

- Notice how bodies are composite and consists of
  - a discrete, atomic part, the unit, and
  - a material endurant, the oil.
- We refer to Example 45 on Slide 292. 🟩

6.1.4. Material Properties

- These are some of the key concerns in domains focused on materials:
  - ⟐ transport, flows, leaks and losses, and
  - ⟐ input to systems and output from systems,
- Other concerns are in the direction of
  - ⟐ dynamic behaviours of materials focused domains (mining and production), including
  - ⟐ stability, periodicity, bifurcation and ergodicity.
- In this seminar we shall, when dealing with systems focused on materials, concentrate on modelling techniques for
  - ⟐ transport, flows, leaks and losses, and
  - ⟐ input to systems and output from systems.

- Formal specification languages like
  - Alloy [alloy],
  - Event B [JRAbrial:TheBBooks] ,
  - CASL [CoFI:2004:CASL-RM]
  - CafeOBJ [futatsugi2000a],
  - RAISE [RaiseMethod],
  - VDM [e:db:Bj78bwo,e:db:Bj82b,JohnFitzge and
  - Z [m:z:jd+jcppw96]

  do not embody the mathematical calculus notions of
  - continuity, hence do not "exhibit"
  - neither differential equations
  - nor integrals.
- Hence cannot formalise **dynamic systems** within these **formal specification languages**.
- We refer to Sect. 9.3.1 where we discuss these issues at some length.

Example: 45 Pipelines: Parts and Material Properties. We refer to Examples 30 on Slide 210, 31 on Slide 212 and 44 on Slide 288.

128. Properties of pipeline units additionally include such which are concerned with flows (F) and leaks (L) of materials:
    - a current flow of material into a unit input connector,
    - b maximum flow of material into a unit input connector while maintaining laminar flow,
    - c current flow of material out of a unit output connector,
    - d maximum flow of material out of a unit output connector while maintaining laminar flow,
    - e current leak of material at a unit input connector,
    - f maximum guaranteed leak of material at a unit input connector,
    - g current leak of material at a unit input connector,
    - h maximum guaranteed leak of material at a unit input connector,
    - i current leak of material from “within” a unit,
    - j maximum guaranteed leak of material from “within” a unit.

129. There are “the usual” arithmetic and comparison operators of flows and leaks, and there is a smallest detectable (flow and) leak.

type

129. F, L

value

129. $\oplus, \ominus$: $(F|L)\times(F|L) \to (F|L)$
129. $<,\leq,=$: $(F|L)\times(F|L) \to$ **Bool**
129. $\otimes$: $(F|L)\times$**Real** $\to (F|L)$
129. $/$: $(F|L)\times(F|L) \to$ **Real**
129. ℓ_0:L

128(a). $\underline{\textbf{attr\_}}\text{cur\_iF}: U \to UI \to F$
128(b). $\underline{\textbf{attr\_}}\text{max\_iF}: U \to UI \to F$
128(c). $\underline{\textbf{attr\_}}\text{cur\_oF}: U \to UI \to F$
128(d). $\underline{\textbf{attr\_}}\text{max\_oF}: U \to UI \to F$
128(e). $\underline{\textbf{attr\_}}\text{cur\_iL}: U \to UI \to L$
128(f). $\underline{\textbf{attr\_}}\text{max\_iL}: U \to UI \to L$
128(g). $\underline{\textbf{attr\_}}\text{cur\_oL}: U \to UI \to L$
128(h). $\underline{\textbf{attr\_}}\text{max\_oL}: U \to UI \to L$
128(i). $\underline{\textbf{attr\_}}\text{cur\_L}: U \to L$
128(j). $\underline{\textbf{attr\_}}\text{max\_L}: U \to L$

- The maximum flow attributes are static attributes
  and are typically provided by the manufacturer
  as indicators of flows below which laminar flow can be expected.
- The current flow attributes as dynamic attributes.

130. Properties of pipeline materials may additionally include

   a kind of material[18],
   b paraffins,
   c naphtenes,
   d aromatics,
   e asphatics,
   f viscosity,
   g etcetera.

- We leave it to the student to provide the formalisations. ■

[18]For example `Brent Blend` Crude Oil

6.1.5. Material Laws of Flows and Leaks

- It may be difficult or costly, or both
  - to ascertain flows and leaks in materials-based domains.
  - But one can certainly speak of these concepts.
  - This casts new light on **domain modelling**.
  - That is in contrast to
    - incorporating such notions of flows and leaks
    - in **requirements modelling**
  - where one has to show implementability.
- Modelling flows and leaks is important to the modelling of materials-based domains.

Example: 46 Pipelines: Intra Unit Flow and Leak Law. We continue our line of Pipeline System examples (cf. the opening line of Example 45 on Slide 292).

131. For every unit of a pipeline system, except the well and the sink units, the following law apply.

132. The flows into a unit equal

   a the leak at the inputs

   b plus the leak within the unit

   c plus the flows out of the unit

   d plus the leaks at the outputs.

```
axiom
131.  ∀ pls:PLS,b:B\We\Si,u:U ·
131.      b ∈ obs_Bs(pls)∧u=obs_U(b)⇒
131.      let (iuis,ouis) = mereo_U(u) in
132.      sum_cur_iF(iuis)(u) =
132(a).       sum_cur_iL(iuis)(u)
132(b).     ⊕ attr_cur_L(u)
132(c).     ⊕ sum_cur_oF(ouis)(u)
132(d).     ⊕ sum_cur_oL(ouis)(u)
131.      end
```

133. The sum_cur_iF (cf. Item 132) sums current input flows over all input connectors.

134. The sum_cur_iL (cf. Item 132(a)) sums current input leaks over all input connectors.

135. The sum_cur_oF (cf. Item 132(c)) sums current output flows over all output connectors.
136. The sum_cur_oL (cf. Item 132(d)) sums current output leaks over all output connectors.

133. $\mathrm{sum\_cur\_iF}$: UI-**set** $\to$ U $\to$ F
133. $\mathrm{sum\_cur\_iF(iuis)(u)} \equiv \oplus \langle$ **attr**_cur_iF(ui)(u)$|$ui:UI•ui $\in$ iuis$\rangle$
134. $\mathrm{sum\_cur\_iL}$: UI-**set** $\to$ U $\to$ L
134. $\mathrm{sum\_cur\_iL(iuis)(u)} \equiv \oplus \langle$ **attr**_cur_iL(ui)(u)$|$ui:UI•ui $\in$ iuis$\rangle$
135. $\mathrm{sum\_cur\_oF}$: UI-**set** $\to$ U $\to$ F
135. $\mathrm{sum\_cur\_oF(ouis)(u)} \equiv \oplus \langle$ **attr**_cur_iF(ui)(u)$|$ui:UI•ui $\in$ ouis$\rangle$
136. $\mathrm{sum\_cur\_oL}$: UI-**set** $\to$ U $\to$ L
136. $\mathrm{sum\_cur\_oL(ouis)(u)} \equiv \oplus \langle$ **attr**_cur_iL(ui)(u)$|$ui:UI•ui $\in$ ouis$\rangle$
$\oplus$: $(\mathrm{F}\times\mathrm{F})|\mathrm{F}^* \to \mathrm{F} \mid (\mathrm{L}\times\mathrm{L})|\mathrm{L}^* \to \mathrm{L}$

- where $\oplus$ is both an infix and a distributed-fix function which adds flows and or leaks. ■

Example: 47 Pipelines: Inter Unit Flow and Leak Law.

137. For every pair of connected units of a pipeline system the following law apply:
   a the flow out of a unit directed at another unit minus the leak at that output connector
   b equals the flow into that other unit at the connector from the given unit plus the leak at that connector.

```
137.      ∀ pls:PLS,b,b′:B,u,u′:U·
137.        {b,b′}⊆obs_Bs(pls)∧b≠b′∧u′=obs_U(b′)
137.         ∧ let (iuis,ouis)=mereo_U(u),(iuis′,ouis′)=mereo_U(u′),
137.               ui=uid_U(u),ui′=uid_U(u′) in
137.            ui ∈ iuis ∧ ui′ ∈ ouis′ ⇒
137(a).               attr_cur_oF(us′)(ui′) ⊖ attr_leak_oF(us′)(ui′)
137(b).             = attr_cur_iF(us)(ui) ⊕ attr_leak_iF(us)(ui)
137.            end
137.      comment: b′ precedes b
```

- From the above two laws one can prove the **theorem:**
  - what is pumped from the wells equals
  - what is leaked from the systems plus what is output to the sinks.
- We need formalising the flow and leak summation functions. ■

6.2. Continuous Behaviours

- This section is still under research and development.
- The aim of this section is to relate
  - discrete behaviour domain models of some fragments of a domain
  - to continuous behaviour domain models of other fragments of that domain.
- By a continuous behaviour model$_\delta$ we mean
  - *a domain description that emphasises*
  - *the behaviour of materials, that is,*
  - *how they flow through parts, and related matters.*

6.2.1. Fluid Dynamics

- Continuous behaviour domain models classically express
  - the fluid dynamics$_\delta$
    - of flows of fluids,
    - that is, the natural science of
    - liquids and gasses.

- The natural science of fluids
  - (from `Wikipedia:`)
    - *“are based on foundational axioms of fluid dynamics*
    - *which are the conservation laws,*
    - *specifically, conservation of mass,*
    - *conservation of linear momentum*
    - *(also known as Newton’s Second Law of Motion),*
    - *and conservation of energy*
    - *(also known as First Law of Thermodynamics).*
    - *These are based on classical mechanics.*
    - *They are expressed using the Reynolds Transport Theorem.”*

6.2.1.1 Descriptions of Continuous Domain Behaviours

- We are not going to exemplify such descriptive natural science models.
- Their mathematics, besides being elegant and beautiful,
  - includes familiarity with
  - Bernoulli Equations,
  - Navier Stokes Equations, etc.
- For continuous behaviour domain models
  - we shall refer to such mathematical models
  - of the natural science of fluids.

6.2.1.2 Prescriptions of Required Continuous Domain Behaviours

- By a prescriptive domain model$_\delta$ we mean
  - *a desirable behaviour specification*
  - *as in, for example, a requirements prescription*
  - of a continuous time dynamic system.
- We are also not going to illustrate prescriptive domain models.
  - Their mathematics, besides also being elegant and beautiful,
    - is based on the descriptive natural science models;
    - but are now part of the engineering realm of *Control Theory*.
    - It includes such disciplines as
      - fuzzy control [Michel-etal-2010],
      - stochastic control [Karlin+Taylor1998] and
      - adaptive control [aastroem89], etc.

Example: 48 Pipelines: Fluid Dynamics and Automatic Control.

- We refer to Example 49 on Slide 308.
- In that example, next, we expect domain models
  - for the fluid dynamics of individual pipeline units: wells, pumps, pipes, valves, forks, joins and sinks,
  - as well as models (one or more) for sequences of such units,
  - extending, preferably to entire nets: from wells to sinks.
- And we expect **requirements description models**
  - again for each of some of the individual units:
    - pumps and valves in particular:
    - when they need and how they are **controll**ed:
    - regulating pumps and valves and
    - which unit **attribute**s need be **monitor**ed. ■

6.2.2. A Pipeline System Behaviour

- We shall model the behaviours of a composite pipeline system.
  - We shall be using basically the same form of the description as first illustrated in Sects. 2.8.2–2.8.7 (Slides 94–105) of Example 4.
  - That system, Sects. 2.8.2–2.8.7, can be interpreted as illustrating the central monitoring of vehicles spread over a wide geographical area.
  - The system to be illustrated in Example 49 can likewise be interpreted as illustrating the central monitoring of pipeline units (and their oil) spread over a wide geographical area.

Example: 49 A Pipeline System Behaviour.

- We consider (cf. Examples 30 on Slide 210 and 31 on Slide 212) the pipeline system units to represent also the following behaviours:
  - pls:PLS, Item 126(a) on Slide 288, to also represent the system process, pipeline_system, and for each kind of unit, cf. Example 30, there are the unit processes:
    - unit,
    - well (Item 98(c) on Slide 210),
    - pipe (Item 98(a)),
    - pump (Item 98(a)),
    - valve (Item 98(a)),
    - fork (Item 98(b)),
    - join (Item 98(b)) and
    - sink (Item 98(d) on Slide 210).

channel
  { pls_u_ch[ui]:ui:UI•i ∈ UIs(pls) } MUPLS
  { u_u_ch[ui,uj]:ui,uj:UI•{ui,uj}⊆UIs(pls) } MUU
type
  MUPLS, MUU
value
  pipeline_system: PLS → **in**,**out** { pls_u_ch[ui]:ui:UI•i ∈ UIs(pls) } **Unit**
  pipeline_system(pls) ≡ ‖ { unit(u)|u:U•u ∈ obs_Us(pls) }
  unit: U → **Unit**
  unit(u) ≡
98(c).   is_We(u) → well(uid_U(u))(u),
98(a).   is_Pu(u) → pump(uid_U(u))(u),
98(a).   is_Pi(u) → pipe(uid_U(u))(u),
98(a).   is_Va(u) → valve(uid_U(u))(u),
98(b).   is_Fo(u) → fork(uid_U(u))(u),
98(b).   is_Jo(u) → join(uid_U(u))(u),
98(d).   is_Si(u) → sink(uid_U(u))(u)

- We illustrate essentials of just one of these behaviours.

```
98(b). fork: ui:UI → u:U → out,in pls_u_ch[ ui ],
                 in  { u_u_ch[ iui,ui ] | iui:UI • iui ∈ sel_UIs_in(u) }
                 out { u_u_ch[ ui,oui ] | iui:UI • oui ∈ sel_UIs_out(u) }  Unit
98(b). fork(ui)(u) ≡
98(b).     let u′ = core_fork_behaviour(ui)(u) in
98(b).     fork(ui)(u′) end
```

- The core_fork_behaviour(ui)(u) distributes
  - what oil (or gas) in receives,
    - on the one input sel_UIs_in(u) = {iui},
    - along channel u_u_ch[iui]
  - to its two outlets
    - sel_UIs_out(u) = {oui$_1$,oui$_2$},
    - along channels u_u_ch[oui$_1$], u_u_ch[oui$_2$].

- The core_· · · _behaviour[s](ui)(u) also communicate with the pipeline_system behaviour.
  - What we have in mind here is to model a traditional supervisory control and data acquisition, SCADA system.

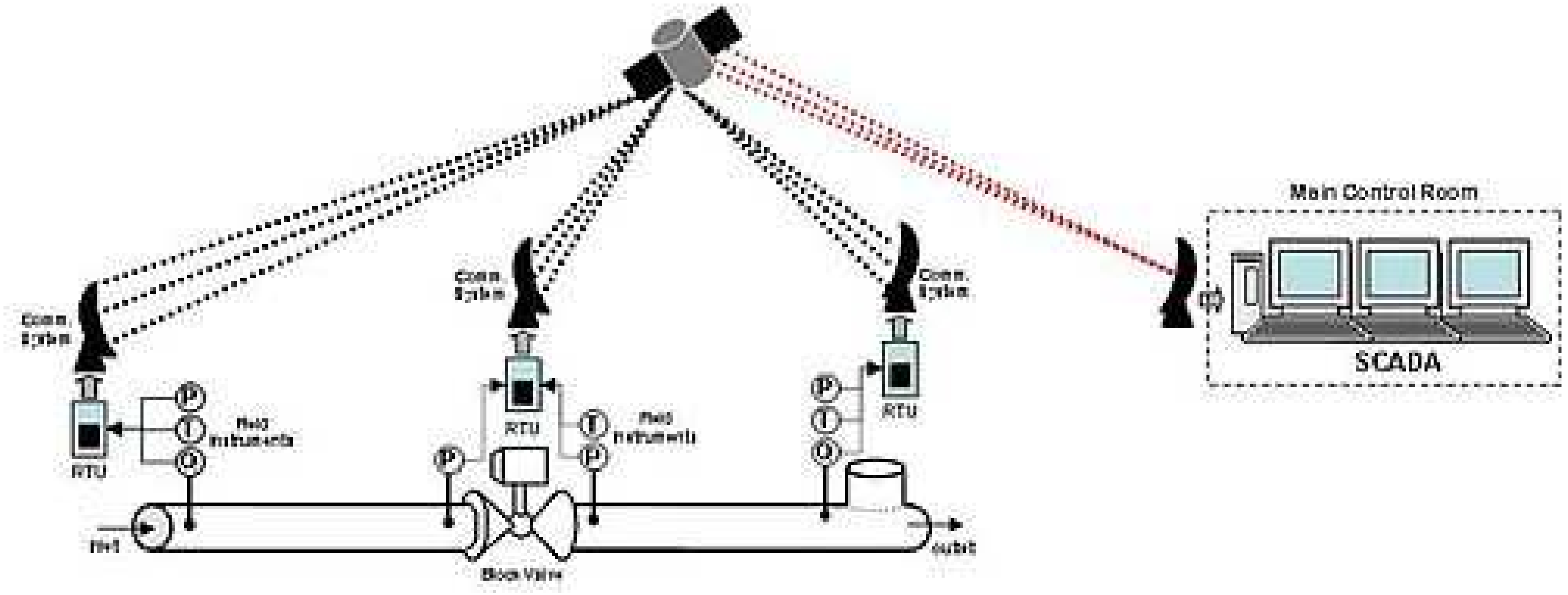


Figure 2: A supervisory control and data acquisition system

138. `SCADA` is then part of the scada_pipeline_system behaviour.

```
138.   scada_pipeline_system: PLS →
138.       in,out { pls_u_ch[ ui ]:ui:UI·i ∈ UIs(pls) }  Unit
138.   scada_pipeline_system(pls) ≡
138.       scada(props(pls)) ∥ pipeline_system(pls)
```

- props was defined on Slide 205.

• We refer to Example 48 on Slide 306:

- for all the core_· · · _behaviours
  - we expect the scada monitor
  - to be expressed in terms of a prescriptive domain model
  - which prescribes some optimal form of control of the pipeline net.

139. scada non-deterministically (internal choice, $\sqcap$), alternates between continually

   a doing own work,

   b acquiring data from pipeline units, and

   c controlling selected such units.

```
type
139.    Props
value
139.    scada: Props → in,out { pls_ui_ch[ ui ] | ui:UI•ui ∈ ∈ uis }  Unit
139.    scada(props) ≡
139(a).       scada(scada_own_work(props))
139(b).     ⊓ scada(scada_data_acqui_work(props))
139(c).     ⊓ scada(scada_control_work(props))
```

- We leave it to the listeners imagination to describe scada_own_work.

140. The scada_data_acqui_work

    a non-deterministically, external choice, ⌷, offers to accept data,

    b and scada_input_updates the scada state —

    c from any of the pipeline units.

```
value
140.    scada_data_acqui_work: Props → in,out { pls_ui_ch[ ui ] | ui:UI•ui ∈ ∈
140.    scada_data_acqui_work(props) ≡
140(a).     ⌷ { let (ui,data) = pls_ui_ch[ ui ] ? in
140(b).            scada_input_update(ui,data)(props) end
140(c).            | ui:UI • ui ∈ uis }

140(b).  scada_input_update: UI × Data → Props → Props
type
140(a).  Data
```

141. The scada_control_work

a **analyses** the scada state (**props**) thereby selecting a pipeline unit, **ui**, and the controls, **ctrl**, that it should be subjected to;

b informs the units of this control, and

c **scada_output_update**s the scada state.

```
141.    scada_control_work: Props → in,out { pls_ui_ch[ ui ] | ui:UI•ui ∈ ∈ uis
141.    scada_control_work(props) ≡
141(a).      let (ui,ctrl) = analyse_scada(ui,props) in
141(b).      pls_ui_ch[ ui ] ! ctrl ;
141(c).      scada_output_update(ui,ctrl)(props) end

141(c).  scada_output_update UI × Ctrl → Props → Props
type
141(a).  Ctrl
```

SOFTWARE ENGINEERING 2
SPECIFICATION OF SYSTEMS AND LANGUAGES
软件工程 系统与语言规约 卷2

SOFTWARE ENGINEERING 3
DOMAINS, REQUIREMENTS AND SOFTWARE DESIGN
软件工程 领域、需求与软件设计 卷3

See You in 30 Minutes — Thanks !

Welcome Back — Thanks !

Lecture 4: 16:00–16:40 + 16:50–17:30
Discrete Perdurant and Contiuous Entities

7. A Domain Discovery Calculus

$\boxed{\text{TO BE WRITTEN}}$

7.1. An Overview

$\boxed{\text{TO BE WRITTEN}}$

7.1.1. Domain Analysers

$\boxed{\text{MORE TO COME}}$

- IS_ENTITY,
  IS_ENDURANT,
  IS_PERDURANT,
  IS_DISCRETE,
  IS_CONTINUOUS,
  IS_MATERIALS_BASED,
  IS_ATOMIC,
  IS_COMPOSITE and
  HAS_CONCRETE_TYPES.

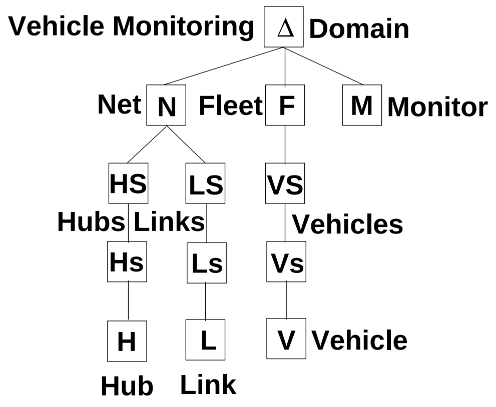
7.1.2. Domain Discoverers

$\boxed{\text{MORE TO COME}}$

- PART_SORTS,
  MATERIAL_SORTS,
  PART_TYPES,
  UNIQUE_ID,
  MEREOLOGY,
  ATTRIBUTES,
  ACTION_SIGNATURES,
  EVENT_SIGNATURES and
  BEHAVIOUR_SIGNATURES.

7.1.3. Domain Indexes

- We first made a reference to the concept of a “domain lattice” in Sect. (Slide 45).
- In Fig. 3 we show a similar “lattice” for the domain of road transport systems as illustrated in this seminar.



<Δ>,
<Δ,N>, <ΔF>, <Δ,M>
<Δ,N,HS>, <Δ,N,LS>, <ΔF,VS>
<Δ,N,HS,Hs>, <Δ,N,LS,Ls>, <ΔF,VS,Vs>,
<Δ,N,HS,Hs,H>, <Δ,N,LS,Ls,L>, <Δ,F,VS,Vs,V>

Figure 3: A domain lattice for the road transport system and the full set of domain indexes

$\boxed{\text{MORE TO COME}}$

7.2. Domain Analysers

$\boxed{\text{TO BE WRITTEN}}$

7.2.1. Some Meta-meta Discoverers

- IS_ENTITY $\boxed{\text{MORE TO COME}}$
- IS_ENDURANT $\boxed{\text{MORE TO COME}}$
- IS_PERDURANT $\boxed{\text{MORE TO COME}}$
- IS_DISCRETE $\boxed{\text{MORE TO COME}}$
- IS_CONTINUOUS $\boxed{\text{MORE TO COME}}$

7.2.2. IS_MATERIALS_BASED

IS_MATERIALS_BASED

- An early decision has to be made as to whether a domain is significantly based on materials or not:

142. IS_MATERIALS_BASED($\langle\Delta_{\text{Name}}\rangle$).

- If Item 142 holds of a domain Δ_{Name}
  - then the **domain describer** can apply
  - MATERIAL_SORTS (Item 145 on Slide 334).

Example: 50 Is Materials-based Domain.

- Example 44 on Slide 288 Item 126 on Slide 289.

7.2.3. IS_ATOMIC

IS_ATOMIC

- The IS_ATOMIC analyser serves that purpose:

value

  IS_ATOMIC: Index $\xrightarrow{\sim}$ **Bool**

  IS_ATOMIC(ℓ^$\langle$t$\rangle$) $\equiv$ **true** | **false** | **chaos**

Example: 51 Is Atomic Type. The IS_ATOMIC analyser has been applied in the following cases in Example 4:

- Sect. 2.1.1 Item 1(c) (M) on Slide 38,
- Sect. 2.1.2 Item 4(b) (V) on Slide 41,
- Sect. 2.1.3 Item 5(b) (H) on Slide 44 and
- Sect. 2.1.3 Item 6(b) (L) on Slide 44.

7.2.4. IS_COMPOSITE

IS_COMPOSITE

- The IS_COMPOSITE analyser is
  - similarly applied by the **domain describer**
  - to a **part type t**
  - to help decide whether **t** is a **composite type**.

value

$\mathbb{IS\_COMPOSITE}$: Index $\overset{\sim}{\rightarrow}$ **Bool**

$\mathbb{IS\_COMPOSITE}(\ell^\frown\langle \mathrm{t}\rangle) \equiv$ **true** | **false** | **chaos**

Example: 52 **Is Composite Type.** The IS_COMPOSITE analyser has been applied in the following cases in Example 4:

- N: Sect. 2.1.2 Items 2(a) and 2(b) on Slide 39,
- HS: Sect. 2.1.2 Item 2(a) on Slide 39,
- Hs: Sect. 2.1.3 Item 5(a) on Slide 44,
- LS: Sect. 2.1.2 Item 2(b) on Slide 39,
- Ls: Sect. 2.1.3 Item 6(a) on Slide 44,
- F: Sect. 2.1.2 Item 3 on Slide 40,
- VS: Sect. 2.1.2 Item 4(b) on Slide 41 and
- Vs: Sect. 2.1.2 Item 4(a) on Slide 41. ■

7.2.5. HAS_A_CONCRETE_TYPE

HAS_A_CONCRETE_TYPE

143. Thus we introduce the analyser:

143 $\mathbb{HAS\_A\_CONCRETE\_TYPE}$: Index $\xrightarrow{\sim}$ **Bool**
143 $\mathbb{HAS\_A\_CONCRETE\_TYPE}(\ell\frown\langle t\rangle)$: **true** | **false** | **chaos**

Example: 53 Has Concrete Types.

- The HAS_CONCRETE_TYPE analyser has been applied in the following cases in Example 4:
  - VS, Vs: Sect. 2.1.2 Item 4(a) on Slide 41,
  - HS, Hs: Sect. 2.1.3 Item 5(a) on Slide 44,
  - LS, Ls: Sect. 2.1.3 Item 6(a) on Slide 44 ■

7.3. Domain Discoverers

7.3.1. PART_SORTS

PART_SORTS I/II

144. The part type discoverer PART_SORTS
   a applies to a simply indexed domain, $\ell^\frown\langle\mathsf{t}\rangle$,
   b where **t** denotes a composite type, and yields a pair
      i. of narrative text and
      ii. formal text which itself consists of a pair:
         A. a set of type names
         B. each paired with a part (sort) observer.

PART_SORTS II/II

value

144. PART_SORTS: Index $\overset{\sim}{\rightarrow}$ (**Text**×RSL)

144(a). PART_SORTS(ℓ⌢⟨t⟩):

144((b))i. [narrative, possibly enumerated texts ;

144((b))iiA. **type** t_1,t_2,...,t_m,

144((b))iiB. **value** obs_t_1:t→t_1,obs_t_2:t→t_2,...,obs_t_m:t→t_m

144(b). **pre**: IS_COMPOSITE(ℓ⌢⟨t⟩)]

Example: 54 Discover Part Sorts.

- We refer to Example 4. The PART_SORTS discoverer has been applied in the followig cases:
  - Δ: Sect. 2.1.1 Items 1(a)–1(c) on Slide 38,
  - N, HS, LS: Sect. 2.1.2 Items 2(a)–2(b) on Slide 39,
  - HS: Sect. 2.1.2 Item 5 on Slide 44,
  - LS: Sect. 2.1.2 Item 6 on Slide 44,
  - Hs: Sect. 2.1.2 Item 5(a) on Slide 44,
  - Ls: Sect. 2.1.2 Item 6(a) on Slide 44,
  - F, VS: Sect. 2.1.2 Item 3 on Slide 40, and
  - VS, Vs: Sect. 2.1.2 Item 4(a) on Slide 41. ■

7.3.2. MATERIAL_SORTS

MATERIAL_SORTS – I/II

145. The MATERIAL_SORTS discovery function applies to a domain, usually designated by $\langle \Delta_{\mathsf{Name}} \rangle$ where Name is a pragmatic hinting at the domain by name.
146. The result of the **domain discoverer** applying this meta-function is some narrative text
147. and the **type**s of the discovered **material**s
148. usually affixed a comment
    a which lists the "somehow related" part types
    b and their related materials observers.

MATERIAL_SORTS II/II

```
145. MATERIAL_SORTS: ⟨Δ⟩ → (Text × RSL)
145. MATERIAL_SORTS(⟨Δ_Name⟩):
146.     [ narrative text ;
147.       type $M_a$, $M_b$, ..., $M_c$  materials
148.       comment: related part types: $P_i$, $P_j$, ..., $P_k$
148.                       obs_$M_n$ : $P_m$ → $M_n$, ... ]
142.     pre: IS_MATERIALS_BASED(⟨Δ_Name⟩)
```

Example: 55 Material Sort.

- The MATERIAL_SORTS discoverer has been applied:
  - O: Example , Items 126 and 126(c) on Slide 289. ■

7.3.3. PART_TYPES

PART_TYPES I/II

149. The PART_TYPES discoverer applies to a composite sort, t, and yields a pair
   a of narrative, possibly enumerated texts [omitted], and
   b some formal text:
      i. a type definition, $\mathsf{t}_c = \mathsf{te}$,
      ii. together with the sort definitions of so far undefined type names of te.
      iii. An observer function observes t_c from t.
      iv. The PART_TYPES discoverer is not defined if the designated sort is judged to not warrant a concrete type definition.

PART_TYPES II/II

149. $\mathbb{PART\_TYPES}$: Index $\overset{\sim}{\to}$ (**Text** $\times$ RSL)
149. $\mathbb{PART\_TYPES}(\ell^\frown\langle t\rangle)$:
149(a). [narrative, possibly enumerated texts ;
149((b))i. **type** t_c = te,
149((b))ii. t_α, t_β, ..., t_γ,
149((b))iii. **value** obs_t_c: t $\to$ t_c
149((b))iv. **pre**: $\mathbb{HAS\_CONCRETE\_TYPE}(\ell^\frown\langle t\rangle)$]
149((b))ii. **where:** type expression te contains
149((b))ii. type names t_α, t_β, ..., t_γ

Example: 56 Part Types.

- The PART_TYPES discoverer has been applied in Example 4:
  - VS, Vs: Sect. 2.1.2 Item 4(a) on Slide 41,
  - HS, Hs: Sect. 2.1.3 Item 5 on Slide 44, and
  - LS, Ls: Sect. 2.1.3 Item 6 on Slide 44. ■

7.3.4. UNIQUE_ID

UNIQUE_ID

150. For every part type **t** we postulate a unique identity analyser function **uid_t**.

```
value
150. UNIQUE_ID: Index → (Text×RSL)
150. UNIQUE_ID(ℓ^⟨t⟩):
150. [ narrative, possibly enumerated text ;
150.      type  ti
150.      value uid_t: t → ti ]
```

Example: 57 Unique ID.

- We refer to Example 4, Sect. 2.2.1 Slide 47:
  - LI, Item 7(a),
  - HI, Item 7(b) and
  - VI, Item 7(c). ■

7.3.5. MEREOLOGY

MEREOLOGY I/II

151. Let type names t_1, t_2, ..., t_n
denote the types of all parts of a domain.

152. Let type names ti_1, ti_2, ..., ti_n[19], be the corresponding
type names of the unique identifiers of all parts of that domain.

153. The mereology analyser MEREOLOGY is a generic function
which applies to a pair of an index and an index set
and yields some structure of unique identifiers.
We suggest two possibilities,
but otherwise leave it to the domain analyser
to formulate the mereology function.

154. Together with the "discovery" of the mereology function
there usually follows some axioms.

MEREOLOGY II/II

```
type
151.  $t_1$, $t_2$, ..., $t_n$
152.  $t_{idx}$ = $ti_1$ | $ti_2$ | ... | $ti_n$

153.  MEREOLOGY: Index $\overset{\sim}{\rightarrow}$ Index-set $\overset{\sim}{\rightarrow}$ (Text×RSL)
153.  MEREOLOGY($\ell$^⟨t⟩)({$\ell_i$^⟨$t_j$⟩,...,$\ell_k$^⟨$t_l$⟩}):
153.      [ narrative, possibly enumerated texts ;
153.         either:  {}
153.         or:      value mereo_t: t → $ti_x$
153.         or:      value mereo_t: t → $ti_x$-set × $ti_y$-set × ... × $ti_x$-set
154.                  axiom $\mathcal{P}$redicate over values of t′ and $t_{idx}$ ]
```

where none of the ti_x, ti_y, . . . , ti_z are equal to ti.

[19]We here assume that all parts have unique identifications.

Example: 58 Mereologies.

- The MEREOLOGY discoverer was applied in
  - ◈ Example 4, Sect. 2.2.2.2, Items 8(a)–9(b) on Slide 49,
  - ◈ Example 18, Items 74–77 on Slide 179,
  - ◈ Example , Items 79–80(e) on Slide 183 and
  - ◈ Example , Items 96–98(d) on Slide 211.

7.3.6. ATTRIBUTES

ATTRIBUTES I/II

155. Attributes have types.
    We assume **attribute type names** to be distict from **part type names**.
156. ATTRIBUTES applies to parts of type **t** and yields a pair of
    a a narrative text and
    b formal text, here in the form of a pair
        i. a set of one or more attribute types, and
        ii. a set of corresponding attribute observer functions **attr_at**, one for each attribute sort **at** of **t**.

ATTRIBUTES II/II

type
155. at = at_1 | at_2 | ... | at_n
value
156. ATTRIBUTES: Index → (**Text**×RSL)
156. ATTRIBUTES(ℓ⌢⟨t⟩):
156(a). [narrative, possibly enumerated texts ;
156((b))i. **type** at_1, at_2, ..., at_m
156((b))ii. **value** attr_at_1:t→at_1,attr_at_2:t→at_2,...,attr_at_m:t→at_m]

- where m≤n

Example: 59 Attributes.

- The ATTRIBUTES discoverer was applied in
  - Example 4, Sect. 2.2.3 for attributes of
    - Links, Items 10–10(c) Slides 52–53,
    - Hubs, Items 11–11(c) Slides 54–55, and
    - Vehicles, Items 12–12 Slides 56–57;
  - as well as in many other examples.

7.3.7. ACTION_SIGNATURES

ACTION_SIGNATURES I/II

157. The ACTION_SIGNATURES meta-function,
besides narrative texts, yields

  a a set of auxiliary sort or concrete type definitions and

  b a set of action signatures each consisting of
  an action name and
  a pair of definition set and range type expressions where

  c the type names that occur in these type expressions
  are defined by in the domains indexed by the index set.

ACTION_SIGNATURES II/II

157 ACTION_SIGNATURES: Index $\rightarrow$ Index-**set** $\overset{\sim}{\rightarrow}$ (**Text**$\times$RSL)
157 ACTION_SIGNATURES($\ell\widehat{\ }\langle t\rangle$)($\{\ell_1\widehat{\ }\langle t_1\rangle,\ell_2\widehat{\ }\langle t_2\rangle,\ldots,\ell_n\widehat{\ }\langle t_n\rangle\}$):
157 [narrative, possibly enumerated texts ;
157 **type** $t_a,t_b,\ldots\ t_c,$
157(b) **value**
157(b) $act_i{:}te_{i_d}\overset{\sim}{\rightarrow}te_{i_r},act_j{:}te_{j_d}\overset{\sim}{\rightarrow}te_{j_r},\ldots,act_k{:}te_{k_d}\overset{\sim}{\rightarrow}te_{k_r}$
157(c) **where:**
157(c) type names in $te_{(i|j|\ldots|k)_d}$ and in $te_{(i|j|\ldots|k)_r}$ are either
157(c) type names t_a, t_b, ... t_c or are type names defined by the
157(c) indices which are prefixes of $\ell_m\widehat{\ }\langle T_m\rangle$ and where T_m is
157(c) in some signature $\mathsf{act}_{i|j|\ldots|k}$]

Example: 60 Action Signatures.

- The ACTION_SIGNATURES discoverer was applied in
  - Example 4: ins_H, Item 37 on Slide 82,
  - Sect. 5.2.3, see Example 33 on Slide 229,
  - etcetera.

7.3.8. EVENT_SIGNATURES

EVENT_SIGNATURES I/II

158. The EVENT_SIGNATURES meta-function, besides narrative texts, yields

a a set of auxiliary event sorts or concrete type definitions and

b a set of event signatures each consisting of

- an event name and
- a pair of definition set and range type expressions

where

c the type names that occur in these type expressions
are defined either in the domains indexed by the indices
or by the auxiliary event sorts or types.

EVENT_SIGNATURES II/II

```
158     EVENT_SIGNATURES: Index → Index-set $\xrightarrow{\sim}$ (Text×RSL)
158     EVENT_SIGNATURES($\ell$⌢⟨t⟩)({$\ell_1$⌢⟨$t_1$⟩,$\ell_2$⌢⟨$t_2$⟩,...,$\ell_n$⌢⟨$t_n$⟩}):
158(a)      [ narrative, possibly enumerated texts omitted ;
158(a)       type  $t_a$,$t_b$,... $t_c$,
158(b)       value
158(b)         evt_pred$_i$: te$_{d_i}$ × te$_{r_i}$ → Bool
158(b)         evt_pred$_j$: te$_{d_j}$ × te$_{r_j}$ → Bool
158(b)         ...
158(b)         evt_pred$_k$: te$_{d_k}$ × te$_{r_k}$ → Bool ]
```

158(c) **where:** t is any of t_a,t_b,...,t_c or type names listed in in indices; type names of the '*d*'efinition set and '*r*'ange set type expressions te$_d$ and te$_r$ are type names listed in domain indices or are in t_a,t_b,...,t_c, the auxiliary discovered event types. ■

Example: 61 Event Signatures.

- Example 4, Sect. 2.7 Item 38 on Slide 85.

7.3.9. DISCRETE_BEHAVIOUR_SIGNATURES

BEHAVIOUR_SIGNATURES I/II

159. The BEHAVIOUR_SIGNATURES meta-function, besides narrative texts, yields
160. It applies to a set of indices and results in a pair,
    - a a narrative text and
    - b a formal text:
        - i. a set of one or more message types,
        - ii. a set of zero, one or more channel index types,
        - iii. a set of one or more channel declarations,
        - iv. a set of one or more process signatures with each signature containing a behaviour name, an argument type expression, a result type expression, usually just **Unit**, and
        - v. an input/output clause which refers to channels over which the signatured behaviour may interact with its environment.

BEHAVIOUR_SIGNATURES II/II

```
159.        BEHAVIOUR_SIGNATURES: Index→ Index-set $\xrightarrow{\sim}$ (Text×RSL)
159.        BEHAVIOUR_SIGNATURES($\ell$⌢⟨t⟩)({$\ell_1$⌢⟨$t_1$⟩,$\ell_2$⌢⟨$t_2$⟩,...,$\ell_n$⌢⟨$t_n$⟩}):
160(a).              [ narrative, possibly enumerated texts ;
160((b))i.            type    m = $m_1$ | m $_2$ | ... | $m_\mu$, $\mu\geq1$
160((b))ii.                   i = $i_1$ | $i_2$ | ... | $i_n$, $n\geq0$
160((b))iii.          channel c:m, {vc[x]|x:$i_a$}:m, {mc[x,y]|x:$i_b$,y:$i_c$}:m,...
160((b))iv.           value
160((b))iv.               $bhv_1$: $ate_1$ → $inout_1$ $rte_1$,
160((b))iv.               ... ,
160((b))iv.               $bhv_m$: $ate_m$ → $inout_m$ $rte_m$. ]
160((b))iv.   where   type expressions $atei_i$ and $rte_i$ for all i involve at least
160((b))iv.             two types $t'_i$, $t''_j$ of respective indexes $\ell_i$⌢⟨$t_i$⟩, $\ell_j$⌢⟨$t_j$⟩,
160((b))v.    where   Unit may appear in either $ate_i$ or $rte_j$ or both.
160((b))v.    where   $inout_i$: in k | out k | in,out k
160((b))v.    where   k: c or vc[x] or {vc[x]|x:$i_a$•x ∈ xs} or
160((b))v.               {mc[x,y]|x:$i_b$,y:$i_c$ • x ∈ xs ∧ y ∈ ys} or ...
```

Example: 62 Behaviour Signatures.

- The BEHAVIOUR_SIGNATURES discoverer was applied in several examples:
  - Example 4, Sect. 2.8.5 Items 61–63 on Slide 98;
  - Sects. 5,4,3 to 5.4.4 inclusive, Slides 250–254;
  - Example 49 on Slide 308;
  - etcetera.

7.4. Some Technicalities

7.4.1. Order of Analysis and "Discovery"

- Analysis and "discovery", that is, the "application" of
  - the analysis meta-functions and
  - the "discovery" meta-functions
- has to follow some order:
  - starts at the "root", that is with index $\langle\Delta\rangle$,
  - and proceeds with indices appending part domain type names already discovered.

7.4.2. Analysis and “Discovery” of “Leftovers”

- The analysis and discovery meta-functions focus on types, that is, the types
  - of abstract parts, i.e., sorts,
  - of concrete parts, i.e., concrete types,
  - of unique identifiers,
  - of mereologies, and of
  - attributes – where the latter has been largely left as sorts.

- In this seminar we do not suggest any meta-functions for such analyses that may lead to
  - concrete types from non-part sorts, or to
  - action, event and behaviour definitions
    - say in terms of pre/post-conditions,
    - etcetera.
  - So, for the time, we suggest, as a remedy for the absence of such "helpers", good "old-fashioned" domain engineer ingenuity.

7.5. Laws of Domain Descriptions

- By a domain description law we shall understand
  - some desirable property
  - that we expect (the 'human') results of
  - the (the 'human') use of the domain description calculus
  - to satisfy.
- We may think of these laws as axioms
  - which an ideal domain description ought satisfy,
  - something that domain describers should strive for.

Notational Shorthands:

- $(f;g;h)(\Re) = h(g(f(\Re)))$
- $(f_1;f_2;\ldots;f_m)(\Re) \simeq (g_1;g_2;\ldots;g_n)(\Re)$
  means that the two "end" states are equivalent modulo appropriate renamings of types, functions, predicates, channels and behaviours.
- $[f;g;\ldots;h;\alpha]$
  stands for the Boolean value yielded by α (in state $\Re$).

7.5.1. 1st Law of Commutativity

- We make a number of assumptions:
  - ⟐ the following two are well-formed indices of a domain:
    - ⦾ ι': $\langle\Delta\rangle\hat{\ }\ell'\hat{\ }\langle\mathsf{A}\rangle$,
    - ⦾ ι'': $\langle\Delta\rangle\hat{\ }\ell''\hat{\ }\langle\mathsf{B}\rangle$,

    where ℓ' and ℓ'' may be different or empty ($\langle\rangle$) and A and B are distinct;
  - ⟐ that $\mathcal{F}$ and $\mathcal{G}$ are two, not necessarily distinct discovery functions; and
  - ⟐ that the domain at ι' and at ι'' have not yet been explored.

- We wish to express,
  - as a desirable property of **domain description development**
  - that exploring domain Δ at
    - either ι' first and then ι''
    - or at ι'' first and then ι',
  - the one right after the other (hence the ";"),
  - ought yield the same partial description fragment:

161. $(\mathcal{G}(\iota'') \ ; \ (\mathcal{F}(\iota')))(\Re) \simeq (\mathcal{F}(\iota') \ ; \ (\mathcal{G}(\iota'')))(\Re)$

When a **domain description development** satisfies Law 161.,
under the above assumptions,

- then we say that the development,
- modulo type, action, event and behaviour name "assignments",
- satisfies a mild form of commutativity.

7.5.2. 2nd Law of Commutativity

- Let us assume
  - that we are exploring the sub-domain at index
  - ι: $\langle\Delta\rangle$^ℓ^$\langle\mathsf{A}\rangle$.
- Whether we
  - first "discover" $\mathcal{A}$ttributes
  - and then $\mathcal{M}$ereology (including $\mathcal{U}$nique identifiers)

  or

  - first "discover" $\mathcal{M}$ereology (including $\mathcal{U}$nique identifiers)
  - and then $\mathcal{A}$ttributes

  should not matter.

- We make some abbreviations:
  - $\mathcal{A}$ stand for the $\mathbb{ATTRIBUTES}$,
  - $\mathcal{U}$ stand for the $\mathbb{UNIQUE\_IDENTIFIER}$,
  - $\mathcal{M}$ stand for the $\mathbb{MEREOLOGY}$,
  - ι for index $\langle\Delta\rangle^\frown\ell^\frown\langle\mathsf{A}\rangle$, and
  - $\iota\mathsf{s}$ for a suitable set of indices.
- Thus we wish the following law to hold:

162. $(\mathcal{A}(\iota);\mathcal{U}(\iota);\mathcal{M}(\iota)(\iota s))(\Re) \simeq$
$(\mathcal{U}(\iota);\mathcal{M}(\iota)(\iota s);\mathcal{A}(\iota))(\Re) \simeq$
$(\mathcal{U}(\iota);\mathcal{A}(\iota);\mathcal{M}(\iota)(\iota s))(\Re)$.

  - here modulo attribute and unique identifier type name renaming.

7.5.3. 3rd Law of Commutativity

- Let us again assume
  - ⟐ that we are exploring the sub-domain at index
  - ⟐ ι: $\langle\Delta\rangle\widehat{\ }\ell\widehat{\ }\langle\mathsf{A}\rangle$
  - ⟐ where $\iota\mathsf{s}$ is a suitable set of indices.
- Whether we are
  - ⟐ exploring actions, events or behaviours at that domain index
  - ⟐ in that order,
  - ⟐ or some other order

ought be immaterial.

- Hence with
  - $\mathcal{A}$ now standing for the $\mathbb{ACTION\_SIGNATURES}$,
  - $\mathcal{E}$ standing for the $\mathbb{EVENT\_SIGNATURES}$,
  - $\mathcal{B}$ standing for the $\mathbb{BEHAVIOUR\_SIGNATURES}$,
- discoverers, we wish the following law to hold:

163.
$$\begin{aligned}
&(\mathcal{A}(\iota)(\iota s);\mathcal{E}(\iota)(\iota s);\mathcal{B}(\iota)(\iota s))(\Re) \simeq\\
&(\mathcal{A}(\iota)(\iota s);\mathcal{B}(\iota)(\iota s);\mathcal{E}(\iota)(\iota s))(\Re) \simeq\\
&(\mathcal{E}(\iota)(\iota s);\mathcal{A}(\iota)(\iota s);\mathcal{B}(\iota)(\iota s))(\Re) \simeq\\
&(\mathcal{E}(\iota)(\iota s);\mathcal{B}(\iota)(\iota s);\mathcal{A}(\iota)(\iota s))(\Re) \simeq\\
&(\mathcal{B}(\iota)(\iota s);\mathcal{A}(\iota)(\iota s);\mathcal{E}(\iota)(\iota s))(\Re) \simeq\\
&(\mathcal{B}(\iota)(\iota s);\mathcal{E}(\iota)(\iota s);\mathcal{A}(\iota)(\iota s))(\Re).
\end{aligned}$$

  - here modulo action function, event predicate, channel, message type and behaviour (and all associated, auxiliary type) renamings.

7.5.4. 1st Law of Stability

- Re-performing
  - the same discovery function
  - over the same sub-domain,
  - that is with identical indices,
  - one or more times,

  ought not produce any new description texts.
- That is:

164. $(\mathcal{D}(\iota)(\iota\mathsf{s});\mathcal{A}\mathsf{\_and\_}\mathcal{D}\mathsf{\_seq})(\Re) \simeq$
$(\mathcal{D}(\iota)(\iota\mathsf{s});\mathcal{A}\mathsf{\_and\_}\mathcal{D}\mathsf{\_seq};\mathcal{D}(\iota)(\iota\mathsf{s}))(\Re)$

- where
  - $\mathcal{D}$ is any discovery function,
  - $\mathcal{A}\mathsf{\_and\_}\mathcal{D}\mathsf{\_seq}$ is any specific sequence of intermediate analyses **and** discoveries, and where
  - ι and $\iota\mathsf{s}$ are suitable indices, respectively sets of indices.

7.5.5. 2nd Law of Stability

- Re-performing
  - ◈ the same analysis functions
  - ◈ over the same sub-domain,
  - ◈ that is with identical indices,
  - ◈ one or more times,

  ought not produce any new analysis results.

- That is:

165. $[\mathcal{A}(\iota)] \;=\; [\mathcal{A}(\iota);\ldots;\mathcal{A}(\iota)]$

- where
  - ◈ $\mathcal{A}$ is any analysis function,
  - ◈ "$\ldots$" is any sequence of intermediate analyses and discoveries, and where
  - ◈ ι is any suitable index.

7.5.6. Law of Non-interference

- When performing a discovery meta-operation, $\mathcal{D}$
  - on any index, ι, and possibly index set, $\iota\mathsf{s}$, and
  - on a repository state, $\Re$,
  - then using the $[\mathcal{D}(\iota)(\iota\mathsf{s})]$ notation
  - expresses a pair of a narrative text and some formulas, $[\mathsf{txt},\mathsf{rsl}]$,
  - whereas using the $(\mathcal{D}(\iota)(\iota\mathsf{s}))(\Re)$ notation
  - expresses a next repository state, $\Re'$.
- What is the "difference" ?
- Informally and simplifying we can say that the relation between the two expressions is:

166. $[\mathcal{D}(\iota)(\iota\mathsf{s})]$: [txt,rsl]
$(\mathcal{D}(\iota)(\iota\mathsf{s}))(\Re) = \Re'$
where $\Re' = \Re \cup \{[\text{txt,rsl}]\}$

- We say that when 166. is satisfied
  - for any discovery meta-function $\mathcal{D}$,
  - for any indices ι and ιs
  - and for any repository state $\Re$,

  then the repository is not interfered with,
  - that is, “*what you see is what you get:*”

  and therefore that
  - the discovery process satisfies the law on non-interference.

7.6. Discussion

- The above is just a hint at domain development laws that we might wish orderly developments to satisfy.
- We invite the audience to suggest other laws.
- The laws of the analysis and discovery calculus
  - forms an ideal set of expectations
  - that we have of not only one domain describer
  - but from a domain describer team
  - of two or more domain describers
  - whom we expect to work, i.e., loosely collaborate,
  - based on “near”-identical domain development principles.

- These are quite some expectations.
  - But the whole point of
    - a highest-level
    - academic scientific education and
    - engineering training
  - is that one should expect commensurate development results.

- Now, since the ingenuity and creativity in the analysis and discovery process does differ between **domain developers**
  - we expect that a daily process of *“buddy checking”*,
  - where individual team members present their findings
  - and where these are discussed by the team
  - will result in adherence to the laws of the calculus.
- The laws of the analysis and discovery calculus
  - expressed some properties that we wish the repository to exhibit.

- We have deliberately abstained from "over-defining"
  - the structure of repositories and
  - the "hidden" operations (i.e., 'update', etc.)

  repositories.
- We expect further
  - research into,
  - development of,
  - possible changes to
  - and use

  of the calculus to yield such insight as to lead to
  - a firmer understanding of
  - the nature of repositories.

- In the analysis and discovery calculus
  - such as we have presented it
- we have emphasised
  - the types of parts, sorts and immediate part concrete types, and
  - the signatures of actions, events and behaviours —
  - as these predominantly featured type expressions.

- We have therefore, in this seminar, not investigated, for example,
  - pre/post conditions of action function,
  - form of event predicates, or
  - behaviour process expressions.
- We leave that, substantially more demanding issue, for future explorative and experimental research.

8. Requirements Engineering

- We shall give a terse overview of some facets of **requirements engineering**.
  - Namely those which "relate" **domain engineering** to **requirements engineering**.
  - The relation is the following:
    - one can "derive",
      - not automatically,
      - but systematically,
    - **domain requirements** and significant aspects of
    - **interface requirements**
  - from **domain descriptions**.

8.1. A Requirements “Derivation”

8.1.1. Definition of Requirements

IEEE Definition of ‘Requirements’

- By a requirements we understand (cf. IEEE Standard 610.12 [ieee-610.12]):
  - ◈ *“A condition or capability needed by a user to solve a problem or achieve an objective”.*

8.1.2. The Machine = Hardware + Software

- By 'the **machine**' we shall understand the
  - ◈ **software** to be developed and
  - ◈ **hardware** (equipment + base software) to be configured

  for the domain application.

8.1.3. Requirements Prescription

- The core part of the requirements engineering of a computing application is the **requirements prescription**.
  - A requirements prescription tells us which parts of the domain are to be supported by ‘the machine’.
  - A requirements is to satisfy some **goal**s.
  - Usually the **goal**s cannot be prescribed in such a manner that they can serve directly as a basis for software design.
  - Instead we derive the requirements from the domain descriptions and then argue
    (incl. prove) that the **goal**s satisfy the requirements.
  - In this colloquium we shall not show the latter
    but shall show the former.

8.1.4. Some Requirements Principles

The “Golden Rule” of Requirements Engineering

- Prescribe only such requirements
  - that can be objectively shown to hold
  - for the designed software.

An “Ideal Rule” of Requirements Engineering

- When prescribing (including formalising) requirements,
  - formulate tests (theorems, properties for model checking)
  - whose actualisation show adherence to the requirements.

- We shall not show adherence to the above rules.

8.1.5. A Decomposition of Requirements Prescription

- We consider three forms of requirements prescription:
  - the **domain requirements**,
  - the **interface requirements** and
  - the **machine requirements**.
- Recall that the machine is the hardware and software (to be required).
  - **Domain requirements** are those whose technical terms are from the domain only.
  - **Machine requirements** are those whose technical terms are from the machine only.
  - **Interface requirements** are those whose technical terms are from both.

8.1.6. An Aside on Our Example

- We shall continue our "ongoing" example.
- Our requirements is for a tollway system.
- By a **requirements goal** we mean
  - *an objective*
  - *the system under consideration*
  - *should achieve* [LamsweerdeIEEE2001].
- The **goal**s of having a tollway system are:
  - to decrease transport times
    between selected hubs of a general net; and
  - to decrease traffic accidents and fatalities
    while moving on the tollway net
    as compared to comparable movements on the general net.

- The tollway net, however, must be paid for by its users.
  - Therefore tollway net entries and exits occur at tollway plazas
  - with these plazas containing entry and exit toll collectors
  - where tickets can be issued,
    respectively collected and
    travel paid for.
- We shall very briefly touch upon these toll collectors,
  in the Extension part (as from Slide 400) below.
- So all the other parts of the next section
  serve to build up to the Extension section.

8.2. Domain Requirements

- Domain requirements cover all those aspects of the domain —
  - ◈ parts and materials,
  - ◈ actions,
  - ◈ events and
  - ◈ behaviours —
- which are to be supported by 'the machine'.

- Thus domain requirements are developed by systematically "revising" cum "editing" the domain description:
  - which parts are to be **projected:** left in or out;
  - which general descriptions are to be **instantiated** into more specific ones;
  - which non-deterministic properties are to be made more **determinate**; and
  - which parts are to be **extended** with such computable domain description parts which are not feasible without IT.

- Thus
  - projection,
  - instantiation,
  - determination and
  - extension

  are the basic engineering tasks of domain requirements engineering.

- An example may best illustrate what is at stake.
- The example is that of a tollway system —
  - in contrast to the general nets.
  - See Fig. 4 on the facing slide.

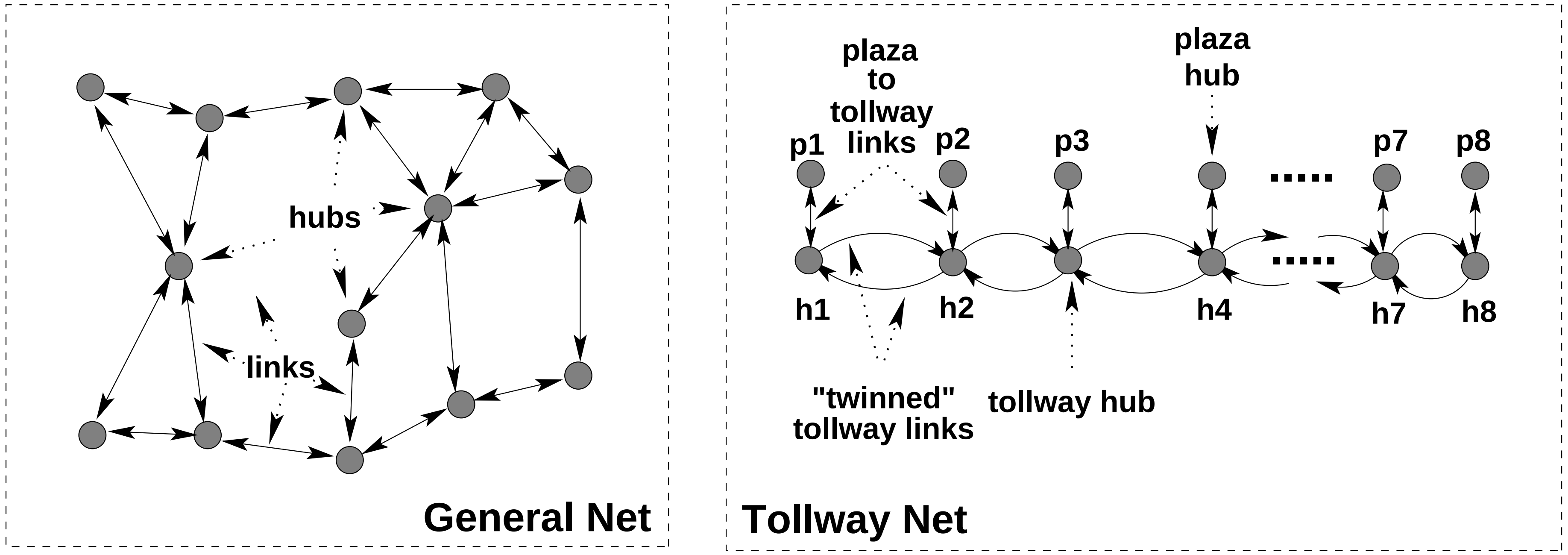


Figure 4: General and Tollway Nets

8.2.1. Projection

- By **domain projection**$_\delta$ we mean that a subset of the **domain description** is kept.
- In the tollway example we actually keep all the parts, their properties and therefore the types and functions derived from these,
- Thus we keep:
  - 1(a)–1(c) (N, F, M)
  - 2–2(b) (HS, LS),
  - 5(a)–6(b) (Hs, Ls, H, L),
  - 7(a)–7(b) (HI, LI),
  - 10(a)–10(c) (LΣ, LΩ, LEN, LOC) and
  - 11(a)–11(c) (HΣ, HΩ, LOC) ,
  - 3–4(b), 7(c) (VS, Vs, V),
  - 8(a)–9(b) (**mereo**_L),
  - 12(a)–12((a))iii, 13 (VP, atH, onL, FRAC, **attr**_VP),
- We do not keep any actions or events (!),
- But we keep the behaviours:
  - 59–59(b) (trs),
  - 61–63 (trs, veh, mon),
  - 65–65(d), 64–68 (veh),
  - 69–71(d) (mon).

8.2.2. Instantiation

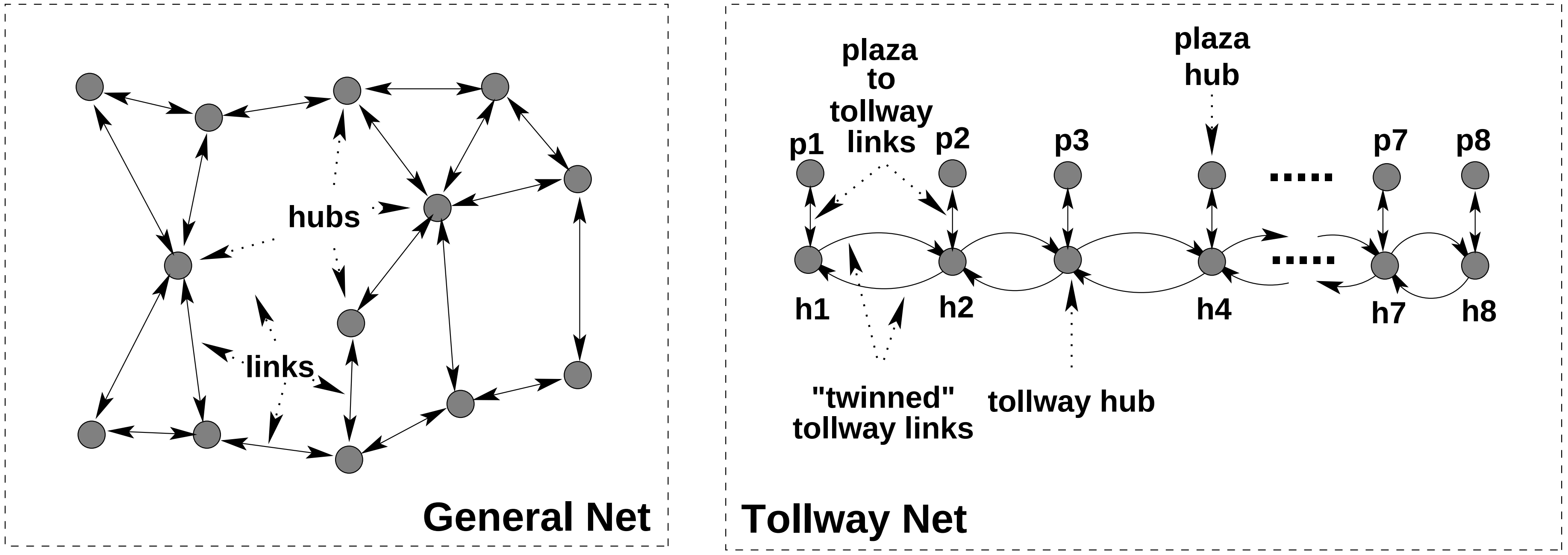


Figure 5: General and Tollway Nets

- From the general net model of earlier formalisations
  we instantiate, that is, make more concrete,
  the tollway net model now described.

167. The net is now concretely modelled as a pair of sequences.
168. One sequence models the plaza hubs, their plaza-to-tollway link and the connected tollway hub.
169. The other sequence models the pairs of “twinned” tollway links.
170. From plaza hubs one can observe their hubs and the identifiers of these hubs.
171. The former sequence is of m such plaza “complexes” where $m \geq 2$; the latter sequence is of $m - 1$ “twinned” links.
172. From a tollway net one can abstract a proper net.

type
167. $\mathsf{TWN} = \mathsf{PC}^* \times \mathsf{TL}^*$
168. $\mathsf{PC} = \mathsf{PH} \times \mathsf{L} \times \mathsf{H}$
169. $\mathsf{TL} = \mathsf{L} \times \mathsf{L}$
value
168. obs_H: PH $\rightarrow$ H, obs_HI: PH $\rightarrow$ HI
axiom
171. $\forall$ (pcl,tll):TWN •
171. $2 \leq$ **len** pcl $\wedge$ **len** pcl=**len** tll+1

value
172. abs_HsLs: TWN $\rightarrow$ (Hs$\times$Ls)
172. abs_HsLs(pcl,tll) **as** (hs,ls)
172. **pre**: wf_TWN(pcl,tll)
172. **post**:
172. hs = {h,h′|(h,_,h′):PC • (h,_,h′)$\in$ **elems** pcl}
172. $\wedge$ ls = {l|(_,l,_):PC • (_,l,_)$\in$ **elems** pcl} $\cup$
172. {l,l′|(l,l′):TL•(l,l′)$\in$ **elems** tll}

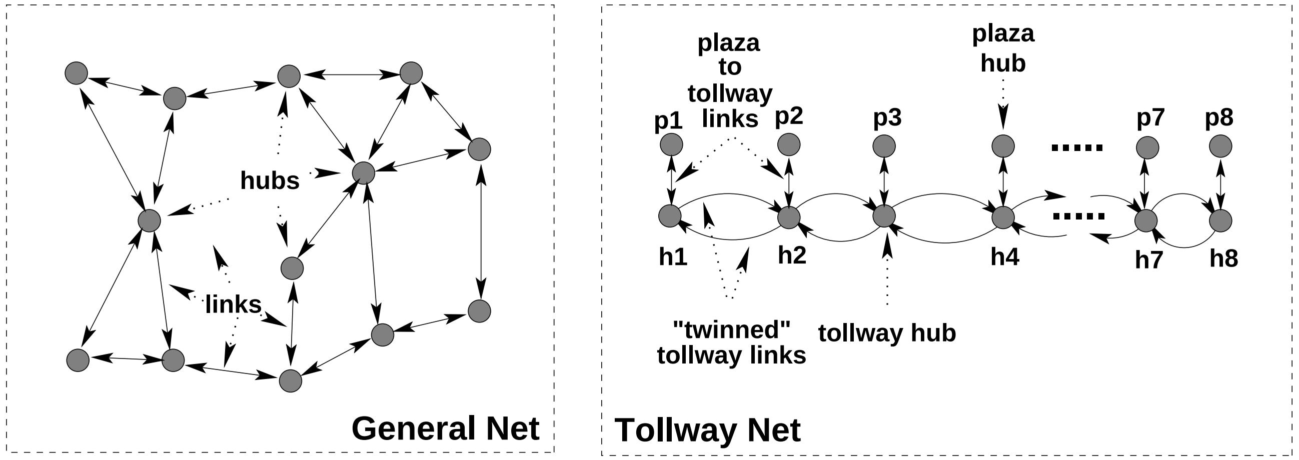


Figure 6: General and tollway Nets

8.2.2.1 Model Well-formedness wrt. Instantiation

- Instantiation restricts general nets to tollway nets.
- Well-formedness deals with proper mereology:
  that observed identifier references are proper.
- The well-formedness of instantiation of the tollway system model can be defined as follows:

173. The i'plaza complex, (p_i, l_i, h_i), is instantiation-well-formed if
  a link l_i identifies hubs p_i and h_i, and
  b hub p_i and hub h_i both identifies link l_i; and if
174. the i'th pair of twinned links, tl_i, tl'_i,
  a has these links identify the tollway hubs of the i'th and $i{+}1$'st plaza complexes ((p_i, l_i, h_i) respectively $(p_{i+1}, l_{i+1}, h_{i_1})$).

```
value
  Instantiation_wf_TWN: TWN → Bool
  Instantiation_wf_TWN(pcl,tll) ≡
173.     ∀ i:Nat · i ∈ inds pcl⇒
173.        let (pi,li,hi)=pcl(i) in
173(a).        obs_LIs(li)={obs_HI(pi),obs_HI(hi)}
173(b).      ∧ obs_LI(li)∈ obs_LIs(pi)∩ obs_LIs(hi)
174.      ∧ let (li′,li″) = tll(i) in
174.         i < len pcl ⇒
174.           let (pi′,li‴,hi′) = pcl(i+1) in
174(a).           obs_HIs(li) = obs_HIs(li′)
174(a).           = {obs_HI(hi),obs_HI(hi′)}
    end end end
```

8.2.3. Determination

- By **domain determination**$_\delta$ we mean, as illustrated in this example,
  - making **part property values**
  - less **in-determinate**, i.e.,
  - more **determinate**.
- The state sets contain only one set.
  - Twinned tollway links allow traffic only in opposite directions.
  - Plaza to tollway hubs allow traffic in both directions.
  - tollway hubs allow traffic to flow freely from
    - plaza to tollway links
    - and from incoming tollway links
    - to outgoing tollway links
    - and tollway to plaza links.

8.2.3.1 Model Well-formedness wrt. Determination

- We need define well-formedness wrt. determination.
- Please study Fig. 7.

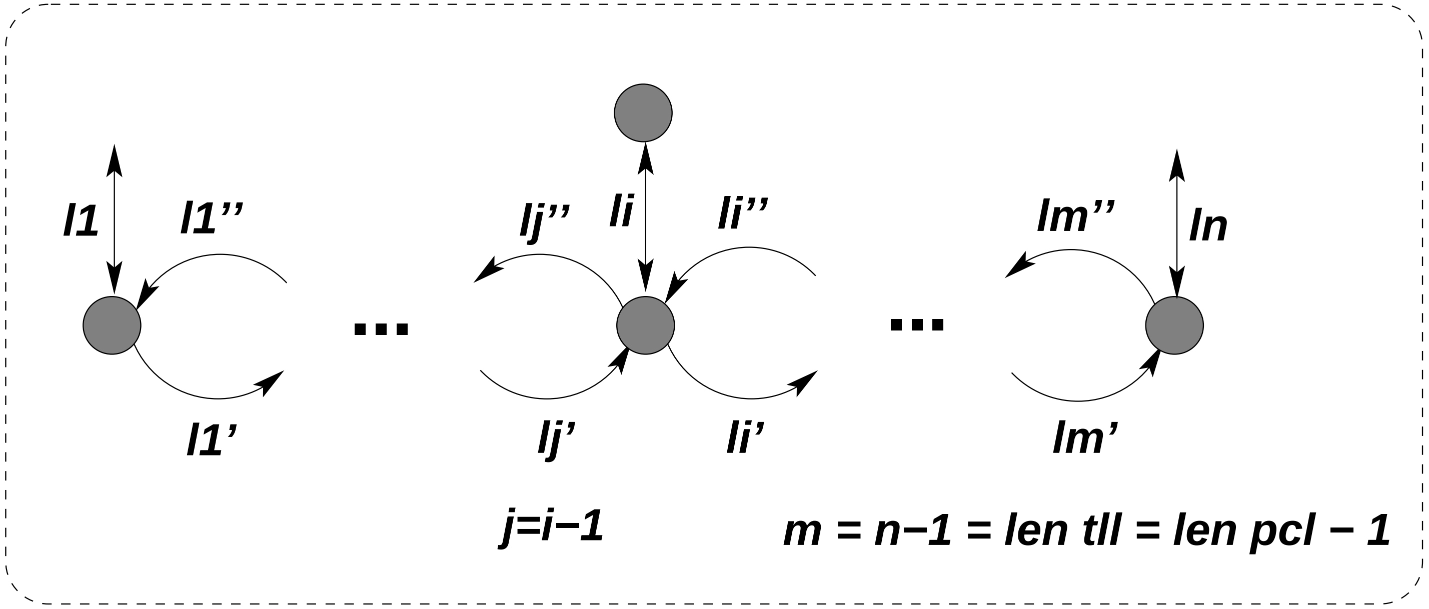


Figure 7: Hubs and Links

175. All hub and link state spaces contain just one hub, respectively link state.

176. The i'th plaza complex, pcl(i):(p_i, l_i, h_i) is determination-well-formed if

    a l_i is open for traffic in both directions and

    b p_i allows traffic from h_i to "revert"; and if

177. the i'th pair of twinned links (li', li'') (in the context of the i+1st plaza complex, pcl(i+1):$(p_{i+1}, l_{i+1}, h_{i+1})$) are determination-well-formed if

    a link l_i' is open only from h_i to h_{i+1} and

    b link l_i'' is open only from h_{i+1} to h_i; and if

178. the jth tollway hub, h_j (for $1 \leq j \leq \mathbf{len}\,$pcl) is determination-well-formed if, depending on whether j is the first, or the last, or any "in-between" plaza complex positions,

    a [the first:] hub $i = 1$ allows traffic in from l_1 and l_1'', and onto l_1 and l_1'.

    b [the last:] hub $j = i + 1 = \mathbf{len}\,$pcl allows traffic in from $l_{\mathbf{len}\,\text{tll}}$ and $l''_{\mathbf{len}\,\text{tll}-1}$, and onto $l_{\mathbf{len}\,\text{tll}}$ and $l'_{\mathbf{len}\,\text{tll}-1}$.

    c [in-between:] hub $j = i$ allows traffic in from l_i, l_i'' and l_i' and onto l_i, l_{i-1}' and l_i''.

```
value
176.     Determination_wf_TWN: TWN → Bool
176.     Determination_wf_TWN(pcl,tll) ≡
176.       ∀ i:Nat• i ∈ inds tll ⇒
176.         let (pi,li,hi) = pcl(i),
176.             (npi,nli,nhi) = pcl(i+1), in
176.             (li′,li″) = tll(i) in
175.            obs_HΩ(pi)={obs_HΣ(pi)}∧obs_HΩ(hi)={obs_HΣ(hi)}
175.        ∧ obs_LΩ(li)={obs_LΣ(li)}∧obs_LΩ(li′)={obs_LΣ(li′)}
175.        ∧ obs_LΩ(li″)={obs_LΣ(li″)}
176(a).       ∧ obs_LΣ(li)
176(a).            = {(obs_HI(pi),obs_HI(hi)),(obs_HI(hi),obs_HI(pi))}
176(a).       ∧ obs_LΣ(nli)
176(a).            = {(obs_HI(npi),obs_HI(nhi)),(obs_HI(nhi),obs_HI(npi))}
176(b).       ∧ {(obs_LI(li),obs_LI(li))}⊆obs_HΣ(pi)
176(b).       ∧ {(obs_LI(nli),obs_LI(nli))}⊆obs_HΣ(npi)
177(a).       ∧ obs_LΣ(li′)={(obs_HI(hi),obs_HI(nhi))}
177(b).       ∧ obs_LΣ(li″)={(obs_HI(nhi),obs_HI(hi))}
178.        ∧ case i+1 of
178(a).           2 → obs_HΣ(h_1)=
178(a).                  {(obs_LΣ(l_1),obs_LΣ(l_1)), (obs_LΣ(l_1),obs_LΣ(l_1″)),
178(a).                  (obs_LΣ(l″_1),obs_LΣ(l_1)), (obs_LΣ(l″_1),obs_LΣ(l′_1))},
178(b).           len pcl → obs_HΣ(h_i+1)=
178(b).                 {(obs_LΣ(l_len pcl),obs_LΣ(l_len pcl)),
178(b).                  (obs_LΣ(l_len pcl),obs_LΣ(l′_len tll)),
178(b).                  (obs_LΣ(l″_len tll),obs_LΣ(l_len pcl)),
178(b).                  (obs_LΣ(l″_len tll),obs_LΣ(l′_len tll))},
178(c).           _ → obs_HΣ(h_i)=
178(c).                 {(obs_LΣ(l_i),obs_LΣ(l_i)), (obs_LΣ(l_i),obs_LΣ(l′_i)),
178(c).                  (obs_LΣ(l_i),obs_LΣ(l″_i−1)), (obs_LΣ(l″_i),obs_LΣ(l′_i)),
178(c).                  (obs_LΣ(l″_i),obs_LΣ(l′_i−1)), (obs_LΣ(l″_i),obs_LΣ(l′_i))}
176.         end end
```

8.2.4. Extension

- By domain extension$_\delta$ we understand the
  - ◈ *introduction of domain entities, actions, events and behaviours that were not feasible in the original domain,*
  - ◈ *but for which, with computing and communication,*
  - ◈ *there is the possibility of feasible implementations,*
  - ◈ *and such that what is introduced become part of the emerging domain requirements prescription.*

8.2.4.1 Backgorund

- The road traffic monitoring domain of Example 4,
  - notably the sections on vehicle and monitor behaviours, (Items 65–71(d) Slides 100–104),
  - illustrated the **intangible abstraction** of road traffic
  - in the form of the recording of a discrete version of that traffic:[20]

46. $\mathrm{d}\mathbb{T}$

45. $\mathrm{dRTF} = \mathrm{d}\mathbb{T} \underset{m}{\rightarrow} (\mathrm{VI} \underset{m}{\rightarrow} \mathrm{VP})$

- by the **road traffic system**:

value

59. $\mathrm{trs}() =$

59(a). $\parallel \{\mathrm{veh}(\underline{\textbf{uid}}\_\mathrm{V}(\mathrm{v}))(\mathrm{v})(\mathrm{vpm}(\underline{\textbf{uid}}\_\mathrm{V}(\mathrm{v})))|\mathrm{v}:\mathrm{V}\cdot\mathrm{v} \in \mathrm{vs}\}$

59(b). $\parallel \mathrm{mon}(\mathrm{mi})(\mathrm{m})([\,\mathrm{t}_0 \mapsto \mathrm{vpm}\,])$

[20] In dRTF we change V into a reference to vehicles VI.

- We say that the **road traffic, dRTF** is **intangible**
  - ◈ since the **dRTF** function,
  - ◈ being a function, is an intangible.
- The **domain extension** is now making that "function" a **tangible** notion.
- There is no presumption,
  - ◈ in defining the **mon**itor behaviour,
  - ◈ that there is indeed a mechanised behaviour,
  - ◈ i.e., a computerised process
  - ◈ that "implements" that **mon**itor.
- Since
  - ◈ one can speak of the **mon**itor behaviour,
  - ◈ one can, as well define it.

8.2.4.2 The Extension

- We now “implement” a version of the above **monitor** behaviour.
- The proposed **domain extension** builds upon
  - the **monitor**
  - and the ability of **vehicles**
    - to communicate their **vehicle positions**
    - to the **monitor**, cf.
      - Items 65(a) and 65(a) Slide 101,
      - Items 66(a), 66((c))i and 66((c))iiA Slide 102 and
      - Item 71(a) Slide 105.

- Instead of this “directness”
  - we interpret links and hubs of the tollway system
  - as behaviours
  - endowed with sensors.
- Vehicle behaviours now interact with link and hub behaviours
  - communicating their positions
  - which the link and hub behaviours
  - communicate to a tollway system monitor.
- The **domain extension** then consists of
  - the extension of links and hubs with sensors and
  - the modelling of their vehicle interactions and
  - their interaction with the tollway system monitor.

8.2.4.3 The Formalisation

- We introduce

179. rather simple **link** and **hub** behaviours, and
180. an array of channels for the interaction of **veh**icle behaviours with **link** and **hub** behaviours.

- And we modify

181. the **veh**icle and **mon**itor behaviours and
182. the **veh**icle/**mon**itor **channel**

- the latter to now serve at the **channel**
- for **link** and **hub** interactions
- with the refined **mon**itor behaviour.

value

172. $(\mathrm{hs,ls}):(\mathrm{Hs,Ls}) = \mathrm{abs\_HsLs(twn)}$

22. $\mathrm{his:HI}\textbf{-set} = \{\underline{\textbf{uid\_}}\mathrm{H(h)}|\mathrm{h:H{\cdot}h} \in \mathrm{hs}\}$

21. $\mathrm{lis:LI}\textbf{-set} = \{\underline{\textbf{uid\_}}\mathrm{L(l)}|\mathrm{l:L{\cdot}l} \in \mathrm{ls}\}$

channel

180. $\{\mathrm{vlh\_ch[\,vi,si\,]}|\mathrm{vi:VI,si:(LI|HI){\cdot}vi} \in \mathrm{vis} \wedge \mathrm{si} \in \mathrm{lis} \cup \mathrm{his}\}\mathrm{:VP}$

182. $\{\mathrm{lhm\_ch[\,si,mi\,]}|\mathrm{si:(LI|HI){\cdot}si} \in \mathrm{lis} \cup \mathrm{his}\}\mathrm{:(VI{\times}VP)}$

value

180. $\mathrm{link:\ li:LI} \to \mathrm{L} \to \textbf{in}\ \{\ \mathrm{vlh\_ch[\,vi,si\,]}|\mathrm{si:LI{\cdot}si} \in \mathrm{lis}\ \}\ \ \textbf{Unit}$

180. $\mathrm{hub:\ hi:HI} \to \mathrm{H} \to \textbf{in}\ \{\ \mathrm{vlh\_ch[\,vi,si\,]}|\mathrm{si:HI{\cdot}si} \in \mathrm{his}\ \}\ \ \textbf{Unit}$

179. $\mathrm{link(li)(l)} \equiv$

179. $\quad(...\sqcap\ \square\ \{\textbf{let}\ \mathrm{(vi,vp)} = \mathrm{vlh\_ch[\,vi,li\,]?}\ \textbf{in}\ \mathrm{lhm\_ch[\,li,mi\,]!(vi,vp)}|\mathrm{vi:VI{\cdot}vi} \in \mathrm{vis}\ \textbf{end}\});\mathrm{li}$

179. $\mathrm{hub(hi)(h)} \equiv$

179. $\quad(...\sqcap\ \square\ \{\textbf{let}\ \mathrm{(vi,vp)} = \mathrm{vlh\_ch[\,vi,hi\,]?}\ \textbf{in}\ \mathrm{lhm\_ch[\,hi,mi\,]!(vi,vp)}|\mathrm{vi:VI{\cdot}vi} \in \mathrm{vis}\ \textbf{end}\});\mathrm{l}$

59. $\mathrm{trs()} =$

59(a). $\quad\|\ \{\mathrm{veh}(\underline{\textbf{uid\_}}\mathrm{V(v))(v)(vpm(}\underline{\textbf{uid\_}}\mathrm{V(v)))}|\mathrm{v:V{\cdot}v} \in \mathrm{vs}\}$

59(b). $\quad\|\ \mathrm{mon(mi)(m)([\,t_0 \mapsto vpm\,])}$

179. $\quad\|\ \{\textbf{link(}\underline{\textbf{uid\_}}\textbf{L(l))(l)}|\textbf{l:L{\cdot}l} \in \textbf{ls}\}$

179. $\quad\|\ \{\textbf{hub(}\underline{\textbf{uid\_}}\textbf{H(h))(h)}|\textbf{h:H{\cdot}h} \in \textbf{hs}\}$

- The modifications to the **vehicle** behaviour is shown in
  - Items 65(a)′, 65((b))ii′, 66(a)′, 66((c))i′, 66((c))iiA′ and 71(a)′
  - (Slides 407–408).

```
65.       veh(vi)(v)(vp:atH(fli,hi,tli)) ≡
65(a)′.          vlh_ch[vi,hi]!(vi,vp) ; veh(vi)(v)(vp)
65(b).        ⊓
65((b))i.        let {hi′,thi}=mereo_L(get_L(tli)(n)) in assert: hi′=hi
65((b))ii′.      vlh_ch[vi,tli]!(vi,onL(hi,tli,0,thi)) ;
65((b))iii.      veh(vi)(v)(onL(hi,tli,0,thi)) end
65(c).        ⊓
65(d).           stop
```

```
64.         veh(vi)(v)(vp:onL(fhi,li,f,thi)) ≡
66(a)′.          vlh_ch[vi,li]!(vi,vp) ; veh(vi)(v)(vp)
66(b).        ⌈⌉
66(c).           if f + $\delta$<1
66((c))i′.          then vlh_ch[vi,li]!(vi,onL(fhi,li,f+$\delta$,thi)) ;
66((c))i.                veh(vi)(v)(onL(fhi,li,f+$\delta$,thi))
66((c))ii.          else let li′:LI•li′ ∈ mereo_H(get_H(thi)(n)) in
66((c))iiA′.             vlh_ch[vi,thi]!(vi,atH(li,thi,li′)) ;
66((c))iiB.              veh(vi)(v)(atH(li,thi,li′)) end end
67.           ⌈⌉
68.              stop

69.         mon(mi)(m)(rtf) ≡
70.             mon(mi)(own_mon_work(m))(rtf)
71.         ⌈⌉
71(a)′.       ⌈⌉ { let ((vi,vp),t) = (lhm_ch[si,mi]?,clk_ch?) in
71(b).             let rtf′ = rtf † [t ↦ rtf(max dom rtf) † [vi ↦ vp]] in
71(c).             mon(mi)(m)(rtf′) end
71(d).             end | si:(LI|HI) • si ∈ lis ∪ his}
```

- The extension, in this example, does not really amount to much.
  - We say that we have extended links and hubs with sensors.
  - But we have not really modelled these sensors.
  - We have modelled their intent, but not their extent.
  - A more complete extension,
    - which has to be done, but which is not shown in this seminar,
    - would now model these sensors
    - as they rely on the **unique vehicle identifier** to be sensed.

- We shall, regrettably, omit this aspect of our presentation of the extension.
  - Whichever sensor technology is chosen, it must be described.
  - A description includes both it proper and its erroneous functioning.
  - Such (IT equipment &c.) descriptions may be expressed in a number of steps:
    - First, as here, a `RSL/CSP` `[CARH:Electronic,TheSEBook1wo]`. model.
    - Then a "derived" description models temporal properties using `Duration Calculus, DC [zcc+mrh2002]`, or `Temporal Logic of Actions, TLA+ [Lamport-TLA+02]`.
    - Finally a timed-automata `[AluDil:94,olderogdirks2008]` model which "implements" the `DC` model.

8.3. Interface Requirements Prescription

- A systematic reading of the domain requirements shall
  - result in an identification of all shared
    - parts and materials,
    - actions,
    - events and
    - behaviours.
- An entity is said to be a **shared entity**$_\delta$
  - if it is present
  - in some related forms,
  - in both
    - the **domain** and
    - the **machine**.

- Each such **shared phenomenon** shall then be individually dealt with:
  - **part** and **materials sharing** shall lead to interface requirements for **data initialisation and refreshment;**
  - **action sharing** shall lead to interface requirements for **interactive dialogues between the machine and its environment;**
  - **event sharing** shall lead to interface requirements for **how events are communicated between the environment of the machine and the machine.**
  - **behaviour sharing** shall lead to interface requirements for **action and event dialogues between the machine and its environment.**

8.3.1. Shared Parts

- The main **shared part**s of the main example of this section are
  - the net, hence the hubs and the links.
- As domain parts they repeatedly undergo changes with respect to the values of a great number of attributes and otherwise possess attributes — most of which have not been mentioned so far:
  - length, cadestral information, namings,
  - wear and tear (where-ever applicable),
  - last/next scheduled maintenance (where-ever applicable),
  - state and state space, and
  - many others.

- We "split" our interface requirements development into two separate steps:
  - the development of $d_{r.net}$
    - (the common domain requirements for the shared hubs and links),
  - and the co-development of $d_{r.db:i/f}$
    - (the common domain requirements for the interface between $d_{r.net}$ and DB_{rel} —
- under the assumption of an available relational database system DB_{rel}

- When planning the common domain requirements for the net, i.e., the hubs and links,
  - we enlarge our scope of requirements concerns beyond the two so far treated ($d_{r.toll}$, $d_{r.maint.}$)
  - in order to make sure that the shared relational database of nets, their hubs and links, may be useful beyond those requirements.

- We then come up with something like
  - hubs and links are to be represented as tuples of relations;
  - each net will be represented by a pair of relations
    - a hubs relation and a links relation;
    - each hub and each link may or will be represented by several tuples;
  - etcetera.
- In this database modelling effort it must be secured that "standard" actions on nets, hubs and links can be supported by the chosen relational database system DB_{rel}.

8.3.1.1 Data Initialisation

- As part of $d_{r.net}$ one must prescribe **data initialisation**, that is provision for
  - an interactive user interface dialogue with a set of proper display screens,
    - one for establishing net, hub or link attributes (names) and their types and,
    - for example, two for the input of hub and link attribute values.
  - Interaction prompts may be prescribed:
    - next input,
    - on-line vetting and
    - display of evolving net, etc.
  - These and many other aspects may therefore need prescriptions.
- Essentially these prescriptions concretise the insert link action.

8.3.1.2 Data Refreshment

- As part of $d_{r.net}$ one must also prescribe **data refreshment**:
  - an interactive user interface dialogue
    with a set of proper display screens
    - one for updating net, hub or link attributes (names)
      and their types and,
    - for example, two for the update of hub and link
      attribute values.
  - Interaction prompts may be prescribed:
    - next update,
    - on-line vetting and
    - display of revised net, etc.
  - These and many other aspects may therefore need prescriptions.
- These prescriptions concretise remove and insert link actions.

8.3.2. Shared Actions

- The main **shared action**s are related to
  - the entry of a vehicle into the tollway system and
  - the exit of a vehicle from the tollway system.

8.3.2.1 Interactive Action Execution

- As part of $d_{r.toll}$ we must therefore prescribe
  - the varieties of successful and less successful sequences
  - of interactions between vehicles (or their drivers) and the toll gate machines.
- The prescription of the above necessitates determination of a number of external events, see below.
- (Again, this is an area of embedded, real-time safety-critical system prescription.)

8.3.3. Shared Events

- The main **shared external events** are related to
  - the entry of a vehicle into the tollway system,
  - the crossing of a vehicle through a tollway hub and
  - the exit of a vehicle from the tollway system.
- As part of $d_{r.toll}$ we must therefore prescribe
  - the varieties of these events,
  - the failure of all appropriate sensors and
  - the failure of related controllers:
    - gate opener and closer (with sensors and actuators),
    - ticket “emitter” and “reader” (with sensors and actuators),
    - etcetera.
- The prescription of the above necessitates extensive fault analysis.

8.3.4. Shared Behaviours

- The main **shared behaviours** are therefore related to
  - the journey of a vehicle through the tollway system and
  - the functioning of a toll gate machine during "its lifetime".
- Others can be thought of, but are omitted here.
- In consequence of considering, for example, the journey of a vehicle behaviour, we may "add" some further, extended requirements:
  - requirements for a vehicle statistics "package";
  - requirements for tracing supposedly "lost" vehicles;
  - requirements limiting tollway system access in case of traffic congestion; etcetera.

8.4. Machine Requirements

- The machine requirements
  - make hardly any concrete reference to the domain description;
  - so we omit its treatment altogether.

8.5. Discussion of Requirements “Derivation”

- We have indicated
  - how the **domain engineer**
  - and the **requirements engineer**
  - can work together
  - to “derive” significant fragments
  - of a **requirements prescription**.

- This puts **requirements engineering** in a new light.
  - Without a previously existing **domain descriptions**
  - the **requirements engineer** has to do double work:
    - both **domain engineering**
    - and **requirements engineering**
  - but without the **principles** of **domain description**,
    - as laid down in this seminar
  - that job would not be so straightforward as we now suggest.

9. Conclusion

- This seminar,
  - meant as the basis for my tutorial
  - at FM 2012 (CNAM, Paris, August 28),
  - "grew" from a paper being written for possible journal publication.
    - Sections 3–7 possibly represent two publishable journal papers.
    - Section 8 has been "added" to the 'tutorial' notes.

- The style of the two tutorial “parts”,
  - Sects. 3–7 and
  - Sect. 8
  - are, necessarily, different:
    - Sects. 3–7
      are in the form of research notes,
    - whereas Sect. 8
      is in the form of “lecture notes” on methodology.
  - Be that as it may. Just so that you are properly notified !

9.1. Comparison to Other Work

- In this section we shall only compare
  - our contribution to domain science & engineering as presented in this seminar
  - to that found in the broader literature with respect to the computer science and software engineering term ‘domain’.
- Finally we shall also not compare
  - our work on a description calculus
  - as we find no comparable literature!

- Our comparison hinges on basically the following two facets:
  - domain analysis and
  - domain description.
- We shall see that the former term, seen across the surveyed literature,
  - covers techniques that are claimed used in many steps of software engineering,
  - but that they seldom, if ever, involve formal concept analysis as we understand it
  - (cf. Sects. 3.2 (Slide 124), 4.1.4 (Slide 142) and 5.1 (Slide 222)).

9.1.1. Ontological Engineering:

- Ontological engineering is described mostly on the Internet, see however [Benjamins+Fensel98].
- Ontology engineers build ontologies.
- And ontologies are, in the tradition of ontological engineering, *"formal representations of a set of concepts within a domain and the relationships between those concepts"* — expressed usually in some logic.
- Published ontologies usually consists of thousands of logical expressions.
- These are represented in some, for example, low-level mechanisable form so that they can be interchanged between ontology groups building upon one-anothers work and processed by various tools.

- There does not seem to be a concern for “deriving” such ontologies into requirements for software.
- Usually ontology presentations
  - either start with the presentation
  - or makes reference to its reliance

  of an upper ontology.
- Instead the ontology databases
  - appear to be used for the computerised
  - discovery and analysis
  - of relations between ontologies.

- The TripTych form of domain science & engineering differs from conventional **ontological engineering** in the following, essential ways:
  - The TripTych domain descriptions rely essentially on a "built-in" **upper ontology**:
    - types, abstract as well as model-oriented (i.e., concrete) and
    - actions, events and behaviours.
  - Domain science & engineering is not, to a first degree, concerned with modalities, and hence do not focus on the modelling of
    - knowledge and belief,
    - necessity and possibility, i.e., alethic modalities,
    - epistemic modality (certainty),
    - promise and obligation (deontic modalities),
    - etcetera.

9.1.2. Knowledge and Knowledge Engineering:

- The concept of knowledge has occupied philosophers since Plato.
  - No common agreement on what ‘knowledge’ is has been reached.
  - From Wikipedia we may learn that
    - *knowledge is a familiarity with someone or something;*
      - *it can include facts, information, descriptions, or skills acquired through experience or education;*
      - *it can refer to the theoretical or practical understanding of a subject;*
    - *knowledge is produced by socio-cognitive aggregates*
      - *(mainly humans)*
      - *and is structured according to our understanding of how human reasoning and logic works.*

- The aim of **knowledge engineering** was formulated, in 1983, by an originator of the concept, Edward A. Feigenbaum `[Feigenbaum83]`:
  - **knowledge engineering** is an engineering discipline
  - that involves integrating knowledge into computer systems
  - in order to solve complex problems
  - normally requiring a high level of human expertise.

- **Knowledge engineering** focuses on
  - continually building up (acquire) large, shared data bases (i.e., **knowledge bases**),
  - their continued maintenance,
  - testing the validity of the stored 'knowledge',
  - continued experiments with respect to **knowledge representation**,
  - etcetera.

- Knowledge engineering can, perhaps, best be understood in contrast to algorithmic engineering:
  - ⟐ In the latter we seek more-or-less conventional, usually imperative programming language expressions of algorithms
    - ⚭ whose algorithmic structure embodies the knowledge
    - ⚭ required to solve the problem being solved by the algorithm.
  - ⟐ The former seeks to solve problems based on an interpreter inferring possible solutions from logical data. This logical data has three parts:
    - ⚭ a collection that "mimics" the semantics of, say, the imperative programming language,
    - ⚭ a collection that formulates the problem, and
    - ⚭ a collection that constitutes the knowledge particular to the problem.
- We refer to `[BjornerNilsson1992]`.

- The concerns of `TripTych` domain science & engineering is based on that of algorithmic engineering.
  - Domain science & engineering is not aimed at
    - letting the computer solve problems based on
    - the knowledge it may have stored.
  - Instead it builds models based on knowledge of the domain.
- Further references to seminal exposés of **knowledge engineering** are `[Studer1998,Kendal2007]`.

9.1.3. Prieto-Dĩaz: Domain Analysis:

- There are different “schools of domain analysis”.
  - **Domain analysis**, or **product line analysis** (see below), as it was first conceived in the early 1980s by James Neighbors
    - is the analysis of related software systems in a domain
    - to find their common and variable parts.
    - It is a model of wider business context for the system.
  - This form of domain analysis turns matters “upside-down”:
    - it is the set of software “systems” (or packages)
    - that is subject to some form of inquiry,
    - albeit having some domain in mind,
    - in order to find common features of the software
    - that can be said to represent a named domain.

- In this section we shall mainly be comparing the `TripTych` approach to domain analysis to that of Reubén Prieto-Dĩaz's approach `[Prieto-Diaz:1987,Prieto-Diaz:1990,Prieto-Diaz:1991]`.
- Firstly, the two meanings of **domain analysis** basically coincide.
- Secondly, in, for example, `[Prieto-Diaz:1987]`, Prieto-Dĩaz's domain analysis is focused on the very important stages that precede the kind of **domain modelling** that we have described.

- Major concerns of Prieto-Dĩaz's approach are
  - **selection** of what appears to be similar, but specific entities,
  - **identification** of common features,
  - **abstraction** of entities and
  - **classification**.
- In comparison
  - **selection** and **identification** is assumed in our approach, but using Ganter & Wille's *Formal Concept Analysis* [Wille:ConceptualAnalysis1999] where Prieto-Dĩaz really does not report on a systematic, let alone formal approach to identification.
  - **Abstraction**
    - (from values to types and signatures) and
  - **classification**
    - into parts, materials, actions, events and behaviours
  - is what we have focused on;
  - as we have also focused on their formalisation.

- All-in-all we find Prieto-Dĩaz's work relevant to our work:
  - ◈ relating to it by providing guidance to pre-modelling steps,
  - ◈ thereby emphasising issues that are necessarily informal,
  - ◈ yet difficult to get started on by most software engineers.
- Where we might differ is on the following:
  - ◈ although Prieto-Dĩaz does mention a need for **domain specific languages**,
  - ◈ he does not show examples of **domain descriptions** in such `DSL`s.
  - ◈ We, of course, basically use mathematics as the `DSL`.
- In the `TripTych` approach to **domain analysis**
  - ◈ we provide a full ontology and
  - ◈ suggest a **domain description calculus**.
- In our approach
  - ◈ we do not consider requirements, let alone software components,
  - ◈ as does Prieto-Dĩaz,

  but we find that that is not an important issue.

9.1.4. Software Product Line Engineering:

- Software product line engineering, earlier known as domain engineering,
  - is the entire process of **reusing domain knowledge** in the production of new software systems.
- Key concerns of **software product line engineering** are
  - **reuse**,
  - the building of repositories of **reusable software components**, and
  - **domain specific language**s with which to, more-or-less automatically build software based on **reusable software component**s.

- These are not the primary concerns of `TripTych` domain science & engineering.
  - But they do become concerns as we move from domain descriptions to requirements prescriptions.
  - But it strongly seems that software product line engineering is not really focused on the concerns of domain description — such as is `TripTych` domain engineering.
  - It seems that software product line engineering is primarily based, as is, for example, `FODA: Feature-oriented Domain Analysis`, on analysing features of software systems.
  - Our `[dines-maurer]` puts the ideas of software product lines and model-oriented software development in the context of the `TripTych` approach.
- Notable sources on software product line engineering are `[dom:Bayer:1999,dom:Weiss:1999,dom:Ardis:2000,dom:Thiel:20`

9.1.5. M.A. Jackson: Problem Frames:

- The concept of **problem frames** is covered in `[mja2001a]`.
- Jackson's prescription for software development focuses on the "triple development" of descriptions of
  - the **problem world**,
  - the **requirements** and
  - the **machine** (i.e., the **hardware** and **software**) to be built.
- Here **domain analysis** means, the same as for us, the **problem world analysis**.

- In the **problem frame** approach the software developer plays three, that is, all the **TripTych** rôles:
  - domain engineer,
  - requirements engineer and
  - software engineer

  "all at the same time",
- well, iterating between these rôles repeatedly.
- So, perhaps belabouring the point,
  - **domain engineering** is done only to the extent needed by the prescription of **requirements** and the **design** of **software**.
- These, really are minor points.

- But in “restricting” oneself to consider
  - only those aspects of the domain which are mandated by the **requirements prescription**
  - and **software design**

  one is considering a potentially smaller fragment [Jackson2010Facs] of the domain than is suggested by the `TripTych` approach.
- At the same time one is, however, sure to
  - consider aspects of the domain
  - that might have been overlooked when pursuing **domain description development**
  - the `TripTych`, “more general”, approach.

.

9.1.6. Domain Specific Software Architectures (DSSA):

- It seems that the concept of **DSSA**
  - was formulated by a group of **ARPA**[21] project "seekers"
  - who also performed a year long study (from around early-mid 1990s);
  - key members of the **DSSA** project were Will Tracz, Bob Balzer, Rick Hayes-Roth and Richard Platek [dom:Trasz:1994].
- The [dom:Trasz:1994] definition of domain engineering is *"the process of creating a DSSA:*
  - *domain analysis and domain modelling*
  - *followed by creating a software architecture*
  - *and populating it with software components."*

[21] ARPA: The US DoD Advanced Research Projects Agency

- This definition is basically followed also by [Mettala+Graham:1992,Shaw+Garlan:1996,Medvidovic+Colbert:2
- Defined and pursued this way, DSSA appears,
  - notably in these latter references, to start with the
  - with the analysis of software components, “per domain”,
  - to identify commonalities within application software,
  - and to then base the idea of software architecture
  - on these findings.

- Thus DSSA turns matter “upside-down” with respect to TripTych **requirements development**
  - by starting with **software components**,
  - assuming that these satisfy some **requirements**,
  - and then suggesting **domain specific software**
  - built using these components.
- This is not what we are doing:
  - We suggest that **requirements**
    - can be “derived” systematically from,
    - and related back, formally to **domain descriptions**s
    - without, in principle, considering **software components**,
    - whether already existing, or being subsequently developed.

- Of course, given a **domain descriptions**
  - it is obvious that one can develop, from it, any number of **requirements prescriptions**
  - and that these may strongly hint at shared, (to be) implemented **software components**;
- but it may also, as well, be the case
  - two or more **requirements prescriptions**
  - "derived" from the same **domain description**
  - may share no **software components** whatsoever !
- So that puts a "damper" of my "enthusiasm" for `DSSA`.

- It seems to this author that had the `DSSA` promoters
  - based their studies and practice on also using formal specifications,
  - at all levels of their study and practice,
  - then some very interesting insights might have arisen.

9.1.7. Domain Driven Design (DDD)

- Domain-driven design (DDD)[22]
  - *"is an approach to developing software for complex needs*
  - *by deeply connecting the implementation to an evolving model of the core business concepts;*
  - *the premise of domain-driven design is the following:*
    - *placing the project's primary focus on the core domain and domain logic;*
    - *basing complex designs on a model;*
    - *initiating a creative collaboration between technical and domain experts to iteratively cut ever closer to the conceptual heart of the problem."*[23]

[22] Eric Evans: http://www.domaindrivendesign.org/

[23] http://en.wikipedia.org/wiki/Domain-driven_design

- We have studied some of the `DDD` literature,
  - ◈ mostly only accessible on `The Internet`, but see also `[Haywood2009]`,
  - ◈ and find that it really does not contribute to new insight into **domains** such as wee see them:
  - ◈ it is just "plain, good old software engineering cooked up with a new jargon.

9.1.8. Feature-oriented Domain Analysis (FODA):

- Feature oriented domain analysis (FODA)
  - is a domain analysis method
  - which introduced feature modelling to domain engineering
  - FODA was developed in 1990 following several U.S. Government research projects.
  - Its concepts have been regarded as critically advancing software engineering and software reuse.
- The US Government supported report [KyoKang+et.al.:1990] states: "*FODA is a necessary first step*" for software reuse.

- To the extent that
  - TripTych domain engineering
  - with its subsequent **requirements engineering**

  indeed encourages reuse at all levels:
  - **domain descriptions** and
  - **requirements prescription**,

  we can only agree.
- Another source on FODA is [Czarnecki2000].
- Since FODA “leans” quite heavily on ‘Software Product Line Engineering’ our remarks in that section, above, apply equally well here.

9.1.9. Unified Modelling Language (UML)

- Three books representative of UML are [Booch98,Rumbaugh98,Jacobson99].
- The term **domain analysis** appears numerous times in these books,
  - yet there is no clear, definitive understanding
  - of whether it, the **domain**, stands for entities in the domain such as we understand it,
  - or whether it is wrought up, as in several of the 'approaches' treated in this section, to wit, Items [3,4,6,7,8], with
    - either **software design** (as it most often is),
    - or **requirements prescription**.

- Certainly, in UML,
  - in [Booch98,Rumbaugh98,Jacobson99] as well as
  - in most published papers claiming "adherence" to UML,
  - that domain analysis usually
    - is manifested in some UML text
    - which "models" some **requirements** facet.
  - Nothing is necessarily wrong with that;
  - but it is therefore not really the TripTych form of **domain analysis**
    - with its concepts of abstract representations of endurant and perdurants, and
    - with its distinctions between **domain** and **requirements**, and
    - with its possibility of "deriving"
      - **requirements prescriptions** from
      - **domain descriptions**.

- There is, however, some important notions of UML
  - and that is the notions of
    - class diagrams,
    - objects, etc.
  - How these notions relate to the **discovery**
    - of part types, unique part identifiers, mereology and attributes, as well as
    - action, event and behaviour signatures and channels,
  - as discovered at a particular **domain index**,
  - is not yet clear to me.
  - That there must be some relation seems obvious.
- We leave that as an interesting, but not too difficult, research topic.

9.1.10. Requirements Engineering:

- There are in-numerous books and published papers on requirements engineering.
  - A seminal one is `[AvanLamsweerde2009]`.
  - I, myself, find `[SorenLauesen2002]` full of very useful, non-trivial insight.
  - `[Dorfman+Thayer:1997:IEEEComp.Soc.Press]` is seminal in that it brings a number or early contributions and views on requirements engineering.

- Conventional text books, notably `[Pfleeger2001,Pressman2001,Sommerville2006]` all have their "mandatory", yet conventional coverage of **requirements engineering**.
  - None of them "derive" requirements from domain descriptions,
    - yes, OK, from domains,
    - but since their description is not mandated
    - it is unclear what "the domain" is.
  - Most of them repeatedly refer to **domain analysis**
    - but since a written record of that **domain analysis** is not mandated
    - it is unclear what "domain analysis" really amounts to.

- Axel van Laamsweerde's book [AvanLamsweerde2009] is remarkable.
  - Although also it does not mandate descriptions of domains
  - it is quite precise as to the relationships between domains and requirements.
  - Besides, it has a fine treatment of the distinction between **goals** and **requirements**,
  - also formally.
- Most of the advices given in [SorenLauesen2002]
  - can beneficially be followed also in
  - TripTych requirements development.
- Neither [AvanLamsweerde2009] nor [SorenLauesen2002] preempts TripTych requirements development.

9.1.11. Summary of Comparisons

- It should now be clear from the above that there are basically two notions from above that relate to our notion of **domain analysis**.
  - (i) Prieto-Dĩaz's notion of *'Domain Analysis'*, and
  - (ii) Jackson's notion of *Problem Frames*.
- But it should also be clear that none of the surveyed literature,
  - except, of course, Ganter & Wille's `[Wille:ConceptualAnalysis1999]` *Formal Concept Analysis, Mathematical Foundations*,
  - covers our notion of **domain analysis**
  - as it hinges crucially on Ganter & Wille's **formal concept analysis**.

9.2. What Have We Omitted: Domain Facets

- One can further structure **domain descriptions** along the lines of the following **domain facets**:
  - ◈ intrinsics,
  - ◈ support technologies,
  - ◈ rules & regulations,
  - ◈ incl. scripts,
  - ◈ organisation & management and
  - ◈ human behaviour

  of domains.
- We refer to `[dines:facs:2008]` for an early treatment of domain facets.

9.2.1. Intrinsics

- By intrinsics$_\delta$ we shall mean
  - *the entities*
  - *in terms of which all other domain facets*
  - *are expressed.*

Example: 63 Road Transport System Intrinsics.

- We refer to Example 4.
  - The following parts are typical of intrinsic parts:
  - N, HS, Hs, LS, Ls, H, L; F, VS, Vs, V. ■

9.2.2. Support Technologies

- By a **support technology**$_\delta$ we shall mean
  - *a human (soft technological) or a hard technological*
  - *means of supporting,*
  - *that is, presenting entities*
  - *and carrying out functions: actions and behaviours.*

Example: 64 Tollroad System Support Technologies.

- We refer to Example (Slides 400–410).
  - The **link sensors**,
  - the **hub sensors**, and
  - the **monitor**

  are examples of **support technologies**. ■

9.2.3. Rules & Regulations

9.2.3.1 Rules

- By a **rule**$_\delta$ we shall mean
  - *some, usually syntactically expressed predicate*
  - *which expresses whether an action (say of a behaviour)*
  - *violates some state property.*

Example: 65 Road Transport System Rules.

- We refer to Sect. 8.2.4 (Slides 400–410).
  - If a vehicle somehow disables its ability to be sensed
  - then a rule has been violated. ■

9.2.3.2 Regulation

- By a **regulation**$_\delta$ we shall mean
  - *some, usually syntactically expressed state-to-state transformer*
  - *which expresses how an erroneous state*
  - *resulting from a rule violation*
  - *can be restored to a state*
  - *in which rule adherence is "restored".* ■

Example: 66 Road Transport System Regulations.

- We refer to Sect. 8.2.4 (Slides 400–410).
  - A pseudo vehicle identification and position
  - replaces a failed sensing of a vehicle at a hub or link.
  - Additional precautionary measures may be taken. ■

9.2.4. Scripts

- By a script$_\delta$ we shall mean
  - *a usually syntactic text which*
  - *describes as set of actions*
  - *expected to be taken by human actors of a system,*
  - *including the assumptions under which*
  - *these actions, or alternatives are to be taken.*

Example: 67 Pipeline System Scripts.

- We refer to Example 49.
  - When closing a valve somewhere along a route
    - all pumps upstream from the valve must first be shut down.
  - Similarly when starting a pump somewhere along a route
    - all valves downstream from the pump must first be opened.
  - For a specific pipeline net this gives rise to a number of scripts, basically one for each pump and valve action. ■

9.2.5. Organisation & Management

9.2.5.1 Organisation

- By organisation$_\delta$ we shall mean
  - *a partitioning of*
    - *parts,*
    - *actions and*
    - *behaviours.*

Example: 68 Tollroad System Organisation.

- We refer to Sect. 8.2.4 (Slides 400–410).
  - A simplest reasonable organisation is
    - the set of links and hubs, including their sensors,
    - and the monitor. ■

9.2.5.2 Management

- By management$_\delta$ we shall mean
  - *a partitioning of human staff*
  - *into possibly a hierarchy*
  - *strategy, tactics and operational managers,*
  - *each taking care of the monitoring and control*
  - *of the rules & regulations for*
  - *decreasing size sets of organisation partitions.*

Example: 69 Tollroad System Management.

- We refer to Sect. 8.2.4 (Slides 400–410).
  - There is one **strategic management structure** for up to several tollroad systems.
    - It is to be commonly described wrt., for example,
      - policies of fixed or varying fee structures; etcetera.
  - In the case of tollroad systems it seems reasonable to also have just one **tactical management structure**.
    - It is to be commonly described wrt., for example,
      - when to invoke one from a set of fee structures; etcetera.
  - Etcetera.

9.2.6. Human Behaviour

- By human behaviour$_\delta$ we shall mean
  - *the sometimes diligent,*
  - *sometimes sloppy,*
  - *sometimes delinquent, or*
  - *sometimes outright criminal*

  carrying out of actions and behaviours of the domain.
- We omit giving examples.

9.3. What Needs More Research

- $\boxed{\text{MORE TO COME}}$

9.3.1. Modelling Discrete & Continuous Domains

- $\boxed{\text{MORE TO COME}}$

9.3.2. Domain Types and Signatures Form Galois Connections

- We plan, in the Fall of 2012, to study
  - whether an altogether different treatment of
  - endurant domain entity types and
  - perdurant domain entity signatures
  - can illuminate the veracity of the title of this section.

9.3.3. A Theory of Domain Facets ?

- We refer to Sect. 9.2. $\boxed{\text{MORE TO COME}}$

9.3.4. Other Issues

- $\boxed{\text{MORE TO COME}}$

9.4. What Have We Achieved

- We claim that there are four major contributions having been lectured upon:
  - (i) strongly hinting that *domain types and signatures form Galois connections*,
  - (ii) the separation of **domain engineering** from **requirements engineering**,
  - (iii) the separate treatment of **domain science & engineering**:
    - as "free-standing" with respect, ultimately, to computer science,
    - and endowed with quite a number of **domain analysis principle**s and **domain description principle**s; and

- (iv) the identification of a number of techniques
  - for “deriving” significant fragments of **requirements prescriptions** from **domain descriptions** —
  - where we consider this whole relation between **domain engineering** and **requirements engineering** to be novel.

- Yes, we really do consider the possibility of a systematic
  - ‘derivation’ of significant fragments of **requirements prescriptions** from **domain descriptions**
  - to cast a different light on **requirements engineering**.

- What we have not shown in this seminar is
  - the concept of **domain facets**;
  - this concept is dealt with in `[dines:facs:2008]` —
  - but more work has to be done to give a firm theoretical understanding of **domain facets** of
    - domain intrinsics,
    - domain support technology,
    - domain scripts,
    - domain rules and regulations,
    - domain management and organisation, and
    - human domainbehaviour.

9.5. General Remarks

- Perhaps belaboring the point:
  - ◈ one can pursue creating and studying **domain descriptions**
  - ◈ without subsequently aiming at **requirements development**,
  - ◈ let alone **software design**.
- That is, **domain descriptions**
  - ◈ can be seen as
    - ⦾ "free-standing",
    - ⦾ of their "own right",
    - ⦾ useful in simply just understanding
    - ⦾ domains in which humans act.

- Just like it is deemed useful
  - that we study “Mother Nature”,
  - the physical world around us,
  - given before humans “arrived”;
- so we think that
  - there should be concerted efforts to study and create **domain models**,
  - for use in
    - studying “our man-made domains of discourses”;
    - possibly proving laws about these domains;
    - teaching, from early on, in middle-school, the domains in which the middle-school students are to be surrounded by;
    - etcetera

- How far must one formalise such **domain descriptions** ?
  - ⟐ Well, enough, so that possible laws can be mathematically proved.
  - ⟐ Recall that **domain descriptions** usually will or must be developed by **domain researchers** — not necessarily **domain engineers** —
    - ⚭ in research centres, say universities,
    - ⚭ where one also studies physics.

- And, when we base **requirements development** on **domain descriptions**,
  - as we indeed advocate,
  - then the **requirements engineers**
  - must understand the formal **domain descriptions**,
  - that is, be able to perform formal
    - domain projection,
    - domain instantiation,
    - domain determination,
    - domain extension,

    etcetera.

- This is similar to the situation in classical engineering
  - which rely on the sciences of physics,
  - and where, for example,
    - *Bernoulli's equations*,
    - *Navier-Stokes equations*,
    - *Maxwell's equations*,
    - etcetera
  - were developed by physicists and mathematicians,
  - but are used, daily, by engineers:
    - read and understood,
    - massaged into further differential equations, etcetera,
    - in order to calculate (predict, determine values), etc.

- Nobody would hire non-skilled labour
  - ⟐ for the engineering development of airplane designs
    - ⚭ unless that “labourer” was skilled in *Navier-Stokes equations*,

  or
  - ⟐ for the design of mobile telephony transmission towers
    - ⚭ unless that person was skilled in *Maxwell’s equations*.

- So we must expect a future, we predict,
  - where a subset of the software engineering candidates from universities
    - are highly skilled in the development of
      - formal **domain descriptions**
      - formal **requirements prescriptions**
  - in at least one domain, such as
    - *transportation*, for example,
      - air traffic,
      - railway systems,
      - road traffic and
      - shipping;

      or
    - *manufacturing*,
    - *services* (health care, public administration, etc.),
    - *financial industries*, or the like.

9.6. Acknowledgements

- I thank the tutorial organisers of the `FM 2012` event
  for accepting my Dec. 31. 2011 tutorial proposal.
- I thank that part of participants
  - ⟐ who first met up for this tutorial this morning (Tuesday 28 August, 2012)
  - ⟐ to have remained in this room for most, if not all of the time.
- I thank colleagues and PhD students around Europe
  - ⟐ for having listened to previous,
  - ⟐ somewhat less polished versions of this seminar.
  - ⟐ I in particular thank Dr. Magne Haveraaen
    of the University of Bergen for providing an important step
    in the development of the present material.
- And I thank my wife
  - ⟐ for her patience during the spring and summer of 2012
  - ⟐ where I ought to have been tending to the garden, etc. !

Thanks for Today — Many Thanks !