Lecture 0: Seminar Overview

Towards a Theory of Domain Descriptions

— a gentle introduction —

Dines Bjørner

Fredsvej 11, DK 2840 Holte, Denmark

February 21, 2012: 12:38

Summary

- We seek foundations for a possible theory of domain descriptions.
  - Part 2 informally outlines what we mean by a domain.
  - Part 3 informally outlines the entities whose description form a description of a domain.
  - Part 4 then suggests one way of formalising such description parts[1]. There are other ways of formally describing domains[2], but the one exemplified can be taken as generic for other description approaches.
  - Part 5 suggests some 'domain discoverers'.

[1] The exemplified description approach is model-oriented, specifically the `RAISE` cum `RSL` approach.

[2] Other model-oriented approaches are those of `Alloy`, `Event B`, `VDM` and `Z`. Property-oriented description approaches include `CafeOBJ`, `Casl` and `Maude`

- These lectures reflect our current thinking.
- Through
  - seminar presentations,
  - their preparation and
  - post-seminar revisions

  it is expected that they will be altered and honed.

Lecture Overview

End Lecture 0: Seminar Overview

Lecture 1: Introduction

1. Introduction

- In this section we shall cover a number of concepts that lie at the foundation of
  - the theory and practice of
  - domain science and engineering.
- These are general issues such as
  - types and values, and
  - algebras.

  and more special, ontological issues such as
  - things, particulars, individuals, entities and parts,
  - endurants and perdurants, and
  - properties, attributes, qualities and tropes.
- But first we shall put the concept of domain engineering in a proper perspective.

1.1. Rôles of Domain Engineering

- By domain engineering we shall understand
  - the engineering[3] of domain descriptions,
  - their study, use and maintenance.
- In this section
  - we shall focus on the use of domain descriptions
    - (i) in the construction of requirements and in the design of software, and
    - (ii) more generally
      - in the study of man-made domains
      - in a search for possible laws.

[3] Engineering is the discipline, art, skill and profession of acquiring and applying scientific, mathematical, economic, social, and practical knowledge, in order to design and build structures, machines, devices, systems, materials and processes . . . [http://en.wikipedia.org/wiki/Engineering]

1.1.1. Software Development

- We see domain engineering as a first in a triptych phased software engineering:
  - (I) domain engineering,
  - (II) requirements engineering and
  - (III) software design.
- Parts 3–4 of these lectures cover some engineering aspects of domain engineering.

1.1.1.1. Requirements Construction

- As shown elsewhere[4] domain descriptions, $\mathcal{D}$, can serve as a firm foundation for requirements engineering.
  - This done is by systematically "deriving" major part of the requirements from the domain description.
  - The 'derivation' is done in steps of refinements and extensions.
  - Typical steps reflect such 'algebraic operations' as
    - projection,
    - determination,
    - fitting,
    - instantiation,
    - extension,
    - etcetera

[4] From Domains to Requirements. LNCS 5065, Springer

- In "injecting" a domain description, $\mathcal{D}$, in a requirements prescription, $\mathcal{R}$,
  - the requirements engineer endeavors to satisfy *goals*, $\mathcal{G}$,
  - where goals are meta-requirements, that is,
  - are a kind of higher-order requirements
  - which can be uttered, that is, postulated,
  - but cannot be formalised in a way from which we can "derive" a software design.
- So, to us, domain engineering becomes an indispensable part of software engineering.

1.1.1.2. Software Design

- Finally, from the requirements prescription, $\mathcal{R}$,
  - software, $\mathcal{S}$, can be designed
  - through a series of refinements and transformations
  - such that one can prove $\mathcal{D}, \mathcal{S} \models \mathcal{R}$,
  - that is, the software design, $\mathcal{S}$, models, i.e.,
  - is a correct implementation of the requirements, $\mathcal{R}$,
  - where the proof makes assumptions about the domain, $\mathcal{D}$.

1.1.2. Domain Studies

- But one can pursue developments of domain descriptions whether or not one subsequently wishes to pursue requirements and software design.
  - Just as physicists study "mother nature" in order just to understand,
  - so domain scientists cum engineers can study, for example, man-made domains — just to understand them.

- Such studies of man-made domains seem worthwhile.
  - Health care systems appear to be quite complex, embodying hundreds or even thousands of phenomena and concepts: parts, actions, events and behaviours.
  - So do
    - container lines,
    - manufacturing,
    - financial services,
    - liquid and gaseous material distribution (pipelines),
    - etcetera.
  - Proper studies of each of these entails many, many years of work.

1.2. Some Preliminary Notions

- We first dwell on the "twinned" notions 'type' and 'value'.
- And then we summarise the notions of
  - (universal, or abstract) algebras,
  - heterogeneous algebras and
  - 'behavioural' algebras.
- The latter notion, behavioural algebra, is a "home-cooked" term.
- The algebra section is
  - short on definitions and
  - long on examples.

1.2.1. Types and Values

- Values $(0, 1, 2, \ldots)$ have types (**integer**).
  - We observe values (**false, true**)),
  - but we speak of them by their types (**Bool**ean);
  - that is:
    - types are abstract concepts
    - whereas (actual) values are (usually) concrete phenomena.
- By a type we shall here, simplifying, mean a way of characterising a set of entities (of similar "kind").
- Entity values and types are related:
  - when we observe an entity we observe its value;
  - and when we say that an entity is of a given type, then we (usually) mean that the observed entity is but one of several entities *of that type*.

Example 1 (Types and Values of Parts) Three naïve examples

When we say, or write, *the* [or *that*] *net*, we mean

1. an entity, a specific value, n,
2. of type *net*, N.

type
2. N
value
1. n:N

When we say, or write, *the* [or *that*] *account*, we mean

3. an entity, a specific value, a,
4. of type *account*, A.

type
4. A
value
3. a:A

When we say, or write, *the* [or *that*] *container*, we mean

5. an entity, a specific value, c,
6. of type *container*, C.

type
6. C
value
5. c:C

●

Example 2 (Types and Values of Actions, Events and Behaviours) We continue the example above:

- A set of actions that all insert hubs in a net have the common signature:

value

insert: H → N $\xrightarrow{\sim}$ N

- The type expression H→N$\xrightarrow{\sim}$N demotes an infinite set of
  - functions from Hubs
  - to partial functions from Nets
  - to Nets.
- The **value** clause insert: H→N$\xrightarrow{\sim}$N
  - names a function value in that infinite set insert
  - and non-deterministically selects an arbitrary value in that infinite set.

- The functions are partial ($\xrightarrow{\sim}$)
  - since an argument Hub
  - may already "be" in the N
  - in which case the insert function is not defined.
- A set of events that all result in a link of a net being broken can be characterised by the same predicate signature:

value

link_disappearance: N × N → **Bool**

- The set of behaviours that focus only on the insertion and removal of hubs and links in a net have the common signature:

type

Maintain = Insert_H | Remove_H | Insert_L | remove_L

value

maintain_N: N → Maintain* → N

maintain_N: N → Maintain$^{\omega}$ → **Unit**

•

1.2.2. Algebras

1.2.2.1. Abstract Algebras

- By an *abstract algebra* we shall understand
  - a set of parts $(e_1, e_2, \ldots)$ called the *carrier*, A (a type), of the algebra, and
  - a set of functions, $f_1, f_2, \ldots, f_n$, [each] in Ω, over these.
- Writing $f_i(e_{j_1}, e_{j_2}, \ldots, e_{j_m})$,
  - where f_i is in Ω of signature:

$$\textbf{signature}\,\omega : A^n \rightarrow A$$

  - and each e_{j_ℓ} $(\ell : \{1..m\})$ is in A.
- The operation $f_i(e_{j_1}, e_{j_2}, \ldots, e_{j_m})$ is then meant to designate
  - either **chaos** (a totally undefined quantity)
  - or some e_k in A.

1.2.2.2. Heterogeneous Algebras

- A *heterogeneous algebra*
  - has its carrier set, A, consist of a number of usually disjoint sets,
  - also referred to as sub-types of A: A_1, A_2, ..., A_n, and
  - a set of operations, ω:Ω, such that each operation, ω, has a *signature*:

$$\textbf{signature}\ \omega : A_i \times A_j \times \cdots \times A_k \ \rightarrow \ A_r$$

  - where $A_i, A_j, \ldots, A_k$ and A_r are in $\{A_1, A_2, \ldots, A_n\}$.

Example 3 (Heterogeneous Algebras: Platoons) We leave it to the reader to fill in missing narrative and to decipher the following formalisation.

7. There are vehicles.
8. A platoon is a set of one or more vehicles.

type
7. V
8. P = {| p · p:V-**set** ∧ p≠{} |}

9. A vehicle can join a platoon.
10. A vehicle can leave a platoon.
11. Two platoons can be merged into one platoon.
12. A platoon can be split into two platoons.

9. join_0: V × P → P
9. join_0(v,p) ≡ p ∪ {v} **pre**: v ∉ p

10. leave_0: V × P → P
10. leave_0(v,p) ≡ p\{v} **pre**: v ∈ p

11. merge_0: P × P → P
11. merge_0(p,p′) ≡ p ∪ p′ **pre**: p ≠ {} ≠ p′ ∧ p ∩ p′ = {}

12. split_0: P → P**-set**
12. split_0(p) ≡ **let** p′,p″:P · p′ ∪ p″ = p **in** {p′,p″} **end pre**: **card** p ≥ 2

- The above formulas define a heterogeneous algebra with
  - types V and P and
  - operations (or actions) *join_0*, *leave_0*, *merge_0*, and *split_0*.

●

1.2.2.3. Behavioral Algebras

- An abstract algebra is characterised
  - by the one type, A, of its parts and
  - by its operations all of whose signatures are of the form $A \times A \times \cdots \times A \rightarrow A$.
- A heterogeneous algebra is an abstract algebra and is further characterised
  - by two or more types, A_1, A_2, ..., A_m, and
  - by a set of operations of usually distinctly typed signatures.
- A behavioral algebra is a heterogeneous algebra and is further characterised
  - by a set of events and
  - by a set of behaviours where
    - events are like actions and
    - behaviours are sets of sequences of actions, events and behaviours.

Example 4 (A Behavioural Algebra: A System of Platoons and Vehi

Our example may be a bit contrived.

- We have yet to unfold, as we do in this paper, enough material to give more realistic examples.

13. A well-formed platoon/vehicle system consists of a pair:
    (a) *convoys* which is a varying set of [non-empty] platoons and
    (b) *reservoir* which is a varying set of vehicles —
    (c) such that the *convoys* platoons are disjoint, no vehicles in common, and
    (d) such that *reservoir* have no vehicle in common with any platoon in *convoys*.

14. Platoons are characterised by unique platoon identifiers.
15. These identifiers can be observed from platoons.
16. Vehicles from the *reservoir* behaviour may join [leave] a platoon whereby they leave [respectively join] the pool.
17. Two platoons may merge into one, and a platoon may split into two.
18. Finally, vehicles may enter [exit] the system by entering [exiting] *reservoir*.

type
13. S = {| (c,r):C×R•r ∩ ∪ c = {} |}
13(a). C = {| c:P**-set** • wf_C(c) |}
value
13(c). wf_C: C → **Bool**
13(c). wf_C(c) ≡ ∀ p,p′:P•{p,p′}⊆c ⇒ p≠{}≠p′ ∧ p ∩ p′ = {}
type
13(b). R = V**-set**
value
16. join_1: S ~→ S
16. leave_1: S ~→ S
17. merge_1: S ~→ S
17. split_1: S ~→ S
18. enter_1: S ~→ S
18. exit_1: S ~→ S

19. join_1 selects an arbitrary vehicle in $r{:}R$ and an arbitrary platoon p in $c{:}C$, joins v to p in c and removes v from r.
20. leave_1 selects a platoon p in c and a vehicle v in p, removes v from p in c and joins v to r.
21. merge_1 selects two distinct platoons p,p' in c, removes them from c, takes their union and adds to c.
22. split_1 selects a platoon p in c, one which has at least to vehicles,
23. and partitions p into p' and p'', removes p from c and joins p' and p'' to c.
24. enter_1 joins a fresh vehicle v to r.
25. exit_1 removes a vehicle v from a non-empty r.

19. join_1(c,r) ≡
19. **let** v:V•v ∈ r,p:P•p ∈ c **in**
19. (c\{p} ∪ {**join_0(v,p)**},r\{v}) **end**

20. leave_1(c,r) ≡
20. **let** v:V,p:P•p ∈ c ∧ v ∈ p **in**
20. (c\{p} ∪ {**leave_0(v,p)**},r ∪ {v}) **end**

21. merge_1(c,r) ≡
21. **let** p,p′:P•p≠p′∧{p,p′}⊆c **in**
21. (c\{p,p′} ∪ {**merge_0(p,p')**},r) **end**

22. split_1(c,r) ≡
23. **let** p:P•p ∈ c∧**card** p≥2 **in**
23. **let** p′,p″:P•p ∪ p′ = p **in**
23. (c\{p} ∪ **split_0(p)**,r) **end end**

24. enter_1(c,r) ≡ (c,**let** v:V•v∉ r ∪ ∪ c **in** r ∪ {v} **end**)
25. exit_1()(c,r) ≡ (c,**let** v:V•v ∈ r **in** r\{v} **end**) **pre**: r≠{}

- The above model abstracts an essence of the non-deterministic behaviour of a platooning system.
- We make no assumptions about
  - which vehicles are joined to or leave which platoons,
  - which platoons are merged,
  - which platoon is split nor into which sub-platoons, and
  - which vehicle enters and exits the reservoir state.

26. We model the above *system* as a behaviour which is composed from a pair of concurrent behaviours:
    (a) a *convoys* behaviour and
    (b) a *reservoir* behaviour
    (c) where these behaviours interact via a **channel** *cr_ch* and
    (d) where the entering of "new" and exiting of "old" vehicles occur on a **channel** *io_ch*
27. Hence the communications between the *reservoir* behaviour and the *convoys* behaviour are of three kinds: *J*oining (moving) a vehicle to a ("magically"[5]) named platoon from the *reservoir* behaviour, *R*emoving [moving] a vehicle from a named platoon to (*mkV(v)*) the *reservoir* behaviour

[5] In this example we skip the somewhat 'technical' details as to how the *reservoir* behaviour obtains knowledge of platoon names.

type
27. M == mkJ(v:V) | mkR | mkV(v:V)
channel
26(c). cr_ch:M
26(d). io_ch:V
value
26. system: S → **Unit**
26. system(c,r) ≡ convoys(c) ‖ reservoir(r)

28. The *convoys* behaviour non-deterministically ($\sqcap$) chooses either to
    (a) merge platoons, or to
    (b) split platoons, or to
    (c) interact with the *reservoir* behaviour via channel *ct_ch*
    (d) and based on that interactions
        i. to either join a[n arbitrary] vehicle *v* to a platoon, or
        ii. to remove a named vehicle, *v*, from a platoon
        iii. while "moving' that vehicle to *reservoir*.

28. convoys: C → **in**,**out** cr_ch **Unit**
28. convoys(c) ≡ convoys(merge(c)) ∏ convoys(split(c)) ∏ convoys(interact(c))

28(c). interact: C → **in**,**out** cr_ch C
28(c). interact(c) ≡
28(c). **let** m = cr_ch ? **in**
28(d). **case** m **of**
28((d))i. mkJ(v) → join_vehicle(v,c),
28((d))ii. mkR → **let** (c′,v)=remove_vehicle(c) **in**
28((d))iii. ct_ch!mkV(v) ; c′
28(c). **end end end**

29. The *merge_platoons* behaviour

(a) non-deterministically chooses two platoons of *convoys* (p,p'),

(b) removes the two platoons from *convoys* and adds the *merge* of these two platoons to *convoys*.

(c) If *convoys* contain less than two platoons then *merge_platoons* is undefined.

29. merge_platoons: C → C
29. merge_platoons(c) ≡
29(a). **let** p,p′,p″:P · p≠p′∧{p,p′}⊆ c **in**
29(b). c\{p,p′} ∪ {**merge_0(p,p')**} **end**
29(b). **pre**: **card** c ≥ 2

30. The *split_platoons* function

(a) non-deterministically chooses a platoon, p, of two or more vehicles in *convoys*,

(b) removes the chosen platoon from *convoys* and inserts the split platoons into *convoys*.

(c) If there are no platoons in c with two or more vehicles then *split_platoons* is undefined.

30. split_platoons: C $\xrightarrow{\sim}$ C
30. split_platoons(c) ≡
30(a). **let** p:P · p ∈ c ∧ **card** p ≥ 2 **in**
30(b). c\{p} ∪ {**split_0(p)**} **end**
30(c). **pre**: ∃ p:P · p ∈ c ∧ **card** p ≥ 2

31. The *reservoir* behaviour interacts with the *convoys* behaviour and with "an external", that is, undefined behaviour through channels *ct_ch* and *io_ch*.
    The *reservoir* behaviour [external] non-deterministically chooses between
    (a) importing a vehicle from "the outside",
    (b) exporting a vehicle to "the outside",
    (c) moving a vehicle to the *convoys* behaviour, and
    (d) moving a vehicle from the *convoys* behaviour.

31. reservoir: R → **in**,**out** cr_ch, io_ch **Unit**
31. reservoir(r) ≡
31(a). (r ∪ {io_ch?}),
31(b). [] **let** v:V · v ∈ t **in** io_ch!mkV(v) ; reservoir(r\{v}) **end**
31(c). [] **let** v:V · v ∈ t **in** ct_ch!mkJ(v) ; reservoir(r\{v}) **end**
31(d). [] **let** mkV(v) = ct_ch? **in** reservoir(r ∪ {v}) **end**

- We may consider Items 31(a)–31(b) as designating events.
- This example designates a behavioural algebra.

●

1.2.3. On 'Method' and 'Methodology'

- By a *method* we shall understand
  - a set of *principles*, *techniques* and *tools*
  - where the principles help
    - ∗ *select* and
    - ∗ *apply*

    these techniques and tools
  - such that an *artifact*, here a domain description, can be constructed.
- By *methodology* we shall understand
  - the *knowledge* and *study* of one or more methods.
- Languages,
  - whether informal, as English,
  - or formal, as RSL,

  are *tools*.

1.3. An Ontology of Descriptions

- *"By ontology we mean*
  - *the philosophical study*
    - *of the nature of being, existence, or reality as such,*
    - *as well as the basic categories of being and their relations.*
- *Traditionally listed as a part of the major branch of philosophy known as metaphysics,*
  - *ontology deals with questions concerning*
  - *what entities exist or can be said to exist,*
  - *and how such entities can be grouped,*
  - *related within a hierarchy,*
  - *and subdivided according to similarities and differences."*[6]

[6] http://en.wikipedia.org/wiki/Ontology

1.3.1. Entities and Properties

- A main stream of philosophers [MellorOliver1997,ChierchiaEtAl1998,ChrisFox2000] appear to agree that there are two categories of discourse:
  - entities[7] and
  - properties.
- Once we say that, a number of questions arise:
  - (Q_1) What counts as an entity ?
  - (Q_2) What counts as a property ?
  - (Q_3) Are properties entities ?
  - (Q_4) Can properties predicate properties ?
- We shall take no and yes to be answers to Q_3 and Q_4.
- These lectures shall answer Q_1 and Q_2

[7] The literature [LeonardGoodman1940,BowmanLClarke81,BowmanLClarke85, MellorOliver1997,ChierchiaEtAl1998,ChrisFox2000,MarcusRossberg2008] alternatively refer to entities by the term individuals.

1.3.2. Things, Entities, Particulars and Individuals

- We shall consider the terms
  - 'thing', – 'entity', – 'particular' – 'individual'

  to be synonymous
- although some philosophers make minute distinctions as we shall see from the below selections.

- *An **entity**[8] is something that has a distinct, separate existence, although it need not be a material existence.*
- *In particular, abstractions and legal fictions are usually regarded as entities.*
- *In general, there is also no presumption that an entity is animate.*

[8] From http://en.wikipedia.org/wiki/Entity

- In algebras **entities** are used as the term for "the things" to which operations are applied and where these applications yield further entities.
- From now on, that is, in the context of domains, entities are "the things" we observe as phenomena in the domains or abstract from these as concepts.
- Domain entities are seen as either
  - parts or
  - actions or
  - events or
  - behaviours.
- We shall later discuss this categorisation of entities.
- It is certainly a matter for reflection.

- *In philosophy,* ***particulars***[9] *are concrete entities existing in space and time as opposed to abstractions.*
- *There are, however, theories of abstract particulars or tropes (properties, qualities, attributes).*
  - *For example, Socrates is a particular (there's only one Socrates-the-teacher-of-Plato and one cannot make copies of him, e.g., by cloning him, without introducing new, distinct particulars).*
  - *Redness, by contrast, is not a particular, because it is abstract and multiply instantiated (my bicycle, this apple, and that woman's hair are all red).*

[9] http://en.wikipedia.org/wiki/Particular

- *Sybil Wolfram writes*[10]*:*
  - *Particulars include only individuals of a certain kind:*
    - *as a first approximation individuals*
    - *with a definite place in space and time,*
      - *such as persons and material objects or events,*
    - *or which must be identified through such individuals,*
      - *like smiles or thoughts.*

[10] Sybil Wolfram, *Philosophical Logic*, Routledge, London and New Youk, 1989, ISBN 0 415 02317 3, page 55

- *Some terms are used by philosophers with a rough-and-ready idea of their meaning.*
  - *This can occur if there is lack of agreement about the best definition of the term.*
  - *In formulating a solution to the problem of universals, the term 'particular' can be used to describe the particular instance of redness of a certain apple as opposed to the 'universal' 'redness' (being abstract).*

- Some philosophers use the term **individual** where others (and we) use the term entity or particular.
  - *The term 'individual' was, in this context, first introduced by [LeonardGoodman1940].*

1.3.3. Categories of Entities

- We shall promulgate the following classes of entities:
  - parts, and
  - operations.

  where we further "sub-divide" operations into
  - actions,
  - events and
  - behaviours
- That is, we can predicate entities, e, as follows:
  - $\mathbb{IS\_PART}(e)$,
  - $\mathbb{IS\_ACTION}(e)$,
  - $\mathbb{IS\_EVENT}(e)$ and
  - $\mathbb{IS\_BEHVAIOUR}(e)$.
- We shall justify the above categorisation through these lectures.
- So parts, actions, events and behaviours form an ontology of descriptions.

1.3.4. Endurants and Perdurants

- ***Endurant***[11], *also known as continuant, or in some cases 'substance'.*
  - *Endurants are those entities that can be observed,*
    - *that is, perceived as a complete concept,*
    - *at no matter which given snapshot of time.*
  - *Were we to freeze time we would still be able to perceive/conceive the entire endurant.*
  - *Examples are material objects, such as an apple or a human, and abstract 'fiat' objects, such as an organisation or the border of a country.*
- In our description ontology we shall call the endurants **parts**.

[11] From http://en.wikipedia.org/wiki/Formal_ontology#Endurant

- ***Perdurant***[12], *also known as occurrent, accident or happening.*
  - *Perdurants are those entities for which only a part exists if we look at them at any given snapshot in time.*
  - *When we freeze time we can only see a part of the perdurant.*
  - *Perdurants are often what we know as processes, for example 'running'.*
  - *If we freeze time*
    - *then we only see a part 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.*
- In our description ontology we shall call the perdurants
  - actions or
  - events or
  - behaviours.

[12] From http://en.wikipedia.org/wiki/Formal_ontology#Perdurant

1.3.5. Universals

- *A **universal***[13]
  - *is what particulars have in common, namely characteristics or qualities.*
  - *In other words, universals are repeatable or recurrent 'quantities' that can be instantiated or exemplified by many particulars.*
  - *For example,*
    - *suppose there are two chairs in a room, each of which is green.*
    - *These two chairs both share the quality of "chairness," as well as greenness or the quality of being green.*

[13] http://en.wikipedia.org/wiki/Universal_(metaphysics)

1.3.6. Properties, Qualities, Tropes, Attributes
1.3.6.1. Properties

- *In modern philosophy, logic, and mathematics a property*[14] *is an attribute of an object; a red object is said to have the property of redness.*
- *The property may be considered a form of object in its own right, able to possess other properties.*
- *A property however differs from individual objects in that it may be instantiated, and often in more than one thing.*
- *It differs from the logical concept of class by not having any concept of extensionality, and from the philosophical concept of class in that a property is considered to be distinct from the objects which possess it.*
- *Understanding how different individual entities (or particulars) can in some sense have some of the same properties is the basis of the problem of universals.*
- *The terms attribute and quality have similar meanings.*

[14] From http://en.wikipedia.org/wiki/Property_(philosophy)

1.3.6.2. Qualities

- *Qualities*[15] *do not exist on their own, but they need an entity (in many formal ontologies,* but not in ours, *this entity is restricted to be an endurant) which they occupy.*
- *Examples of qualities and the values they assume are colors (red color), or temperatures (warm).*
- *Most formal* `upper level`[16] *ontologies recognize qualities, attributes, tropes, or something related, although the exact classification may differ.*

[15] From http://en.wikipedia.org/wiki/Formal_ontology#Qualities

[16] An `upper level ontology` is an ontology which describes very general concepts that are the same across a number of domains.

- *Some see qualities and the values they can assume (sometimes called quale) as a separate hierarchy besides endurant and perdurant.*
- *Others classify qualities as a subsection of endurants, e.g. the dependent endurants.*
- *Others consider property-instances or tropes that are single characteristics of individuals to be the atoms of the ontology, the simpler entities of which all other entities are composed, so that all the entities are sums or bundles of tropes.*

1.3.6.3. Tropes

- *Trope theory*[17] *in metaphysics is a version of nominalism.*
- *Here, a trope is a particular instance of a property, like the specific redness of a rose, or the specific nuance of green of a leaf.*
- *Trope theories assume that universals are unnecessary.*
- *This use of the term goes back to D. C. Williams (1953).*
- *The basic problem has been discussed previously in philosophy without using the term "trope".*

[17] From http://en.wikipedia.org/wiki/Trope_(philosophy)#Trope_theory_in_philosophy_.28metaphysics.29

1.3.6.4. Attributes

- *We shall use the term 'attribute'*
- *as a collective term for 'property', 'quality' and 'trope'.*

1.3.7. Nominalism

- *Nominalism*[18] *is a metaphysical view in philosophy according to which*
  - *general or abstract terms and predicates exist,*
  - *while universals or abstract objects,*
    - *which are sometimes thought to correspond to these terms, do not exist.*

[18] From http://en.wikipedia.org/wiki/Nominalism

- *Thus, there are at least two main versions of nominalism.*
  - *One version denies the existence of universals —*
    - *things that can be instantiated*
    - *or exemplified by many particular things (e.g. strength, humanity).*
  - *The other version specifically denies the existence of abstract objects —*
    - *objects that do not exist in space and time.*

1.4. Structure of These Lectures

End Lecture 1: Introduction

Lecture 2: Domains

2. Domains

- By an observable phenomenon we shall here understand something that can be sensed by one or more of our five sense organs.
- By a domain we shall here informally understand
  - an area of human activity
  - characterised by observable phenomena:
    - entities and their
    - properties,
  - and abstractions, i.e., concepts, thereof.
- In Part 2.2 we suggest a more formal way of characterising a domain.
- But first we give some rough sketch hints as to what domains are.

2.1. Informal Characterisation

- There are several forms of observable phenomena.
- There are the entities:
  - parts,
  - actions,
  - events, and
  - behaviours

  of the domain.
- Then there are the properties of these entities:
  - (i) their **unique identifications**,
  - (ii) the **mereology** of parts, and
  - (iii) the **attributes** of
    - parts: types and values, whether atomic or composite, and of
    - actions, events and behaviours: signatures and values.

- We will just examine one of the part properties.

2.2. Mereology

- Mereology, to us, is the study and knowledge
  - about how physical and conceptual parts relate and
  - what it means for a part to be related to another part:
    - *being adjacent to*,
    - *being contained properly within*,
    - *being properly overlapped with*,
    - etcetera.

- By physical parts we mean
  - such spatial individuals
  - which can be pointed to.
- **Examples:**
  - *a road net*
    (consisting of street segments and street intersections);
  - *a street segment*
    (between two intersections);
  - *a street intersection;*
  - *a vehicle; and*
  - *a platoon*
    (of sequentially adjacent vehicles).

- By a conceptual part we mean
  - an abstraction with no physical extent,
  - which is either present or not.
- **Examples:**
  - *a bus timetable*
    - *(not as a piece or booklet of paper,*
    - *or as an electronic device, but)*

    as an image in the minds of potential bus passengers; and
  - *routes of a pipeline, that is, adjacent sequences of pipes, valves, pumps, forks and joins, for example referred to in discourse: take "such-and-such" a route".*
  - The tricky thing here is that a route may be thought of as being both a concept or being a physical part — in which case one ought give them different names: a planned route and an actual route, for example.

- The mereological notion of subpart, that is: *contained within* can be illustrated by **examples:**
  - *the intersections and street segments are subparts of the road net;*
  - *vehicles are subparts of a platoon; and*
  - *pipes, valves, pumps, forks and joins are subparts of pipelines.*
- The mereological notion of adjacency can be illustrated by **examples:**
  - *the pipes of a pipeline are adjacent (that is, connected) to other pipes or valves or pumps or forks or joins, etcetera;*
  - *two immediately neighbouring vehicles of a platoon are adjacent.*
  - *We shall mereologically model adjacency by the mereology notion of overlap.*

- The mereological notion of proper overlap can be illustrated by **examples:**
  - *two routes of a pipelines may overlap; and*
  - *two conceptual bus timetables may overlap with some, but not all bus line entries being the same.*

2.3. Rough Sketch Hints of Domains

Example 5 (Domains) We present a number of examples:

- *Container Line:*
  - A container line consists of a number of *container vessels* capable of holding (usually thousands of) *containers* being transported, by the vessels, between *container terminal ports* across the seven seas.
  - A container vessel has its containers ordered in *bays*, *rows*, and *stacks* with *container terminal port cranes* depositing or removing ("lifting") *containers* onto or from port side *stack tops*.
  - Container vessels sail specific *routes* with a route being designated by a sequence of *container terminal port visits* where a *container terminal port visit*, amongst others, has a *container terminal port name*, *estimated and actual arrival times*, etc.
  - Etcetera.

- *Financial Service Industry:*
  - A financial service industry consists of a number of *"high street" (i.e., deposit/demand) banks*, *savings & loan institutes*, *commercial banks*, other forms of banks, *insurance companies* (of differing specialisations), *stock/commodity exchanges* with their *brokers* and *traders*, one or more forms of *finance "watchdog" institutions* (SEC, FDIC, etc.), etc.
  - A *bank* had *clients* and *clients* have one or more *accounts* having *account numbers* and *account balances* with *clients opening* and *closing accounts*, *depositing monies* into, and *withdrawing monies* from *accounts*, etc.
  - Etcetera.

- *Health Care System:*
  - A health care system consists of a number of *private physicians*, *hospitals*, *pharmacies*, *health insurance companies*, a *pharmaceutical industry*, *patients*, etc.
  - A *hospital* consists of a number or *wards* (etc.) with each *ward* consisting of a number or *bedrooms* (etc.) with each *bedroom* consisting of a number of *beds* (etc.), etcetera.
  - Etcetera.

- *Pipeline System:*
  - A pipeline system consists of sequences of units: pumps, pipes, valves, forks and joins such that a fork connects to one pipe at the input and two at the output and a join connects two pipes at the input and one at the output, such that the first unit is a pump and is connected at the input to a well and the last unit is a valve and is connected to a sink at the output.
  - A pump, when active (i.e., pumping) should be moving a certain volume of gas or liquid from the input to the put per time unit.
  - A valve when closed prevents flow of gas or liquid from the input to the put, whereas when open unhindered permits such a flow.
  - Etcetera.

- *Transportation System:*
  - Transportation involves, say, three sub-domains: a transport net, a fleet of vehicles, and a community of vehicle drivers and vehicle passengers.
  - A transport net consists of hubs and links such that a link is connected to exactly two distinct hubs and a hub is connected to zero, one or more links.
  - Vehicles are positioned along the net: at hubs or on links and may be standing still or moving — while transporting freight, the driver and zero, one or more passengers.
  - Etcetera.

●

2.4. What are Domains ?

- So what is a domain ?
- We can answer this in three ways:
  - as above, by giving examples,
  - or, as we now do,
    - by an informal characterisation, or
    - by a more formal characterisation.

2.4.1. An Informal Characterisation of Domains

- A *domain* is a set of observable entities and abstractions of these, that is, of
  - *parts*
    (some of which form states),
  - *actions*
    (operation applications causing state changes),
  - *events*
    ("spurious" state changes not [intentionally] caused by actions) and
  - *behaviours*
    (seen as set of sequences of sets of actions, events and behaviours).

- Whereas some entities are manifested
  - spatio-physically, that is,
  - we can point to them,
- others cannot,
  - they are either abstractions of parts,
  - or they are actions, events and behaviours.
- These latter can, however, be characterised
  - by function definitions, event predicates and behaviour definitions
  - which [when applied] denote actions, events and behaviours.

2.4.2. A Formal Characterisation of Domains

- A domain is a behavioral algebra described as consisting of
  - usually two or more type descriptions,
  - usually two or more function and event descriptions, and
  - usually one or more behaviour descriptions,
    - which contain channel descriptions and
    - behaviour process descriptions.

• • •

2.5. Six Examples

2.5.1. Air Traffic

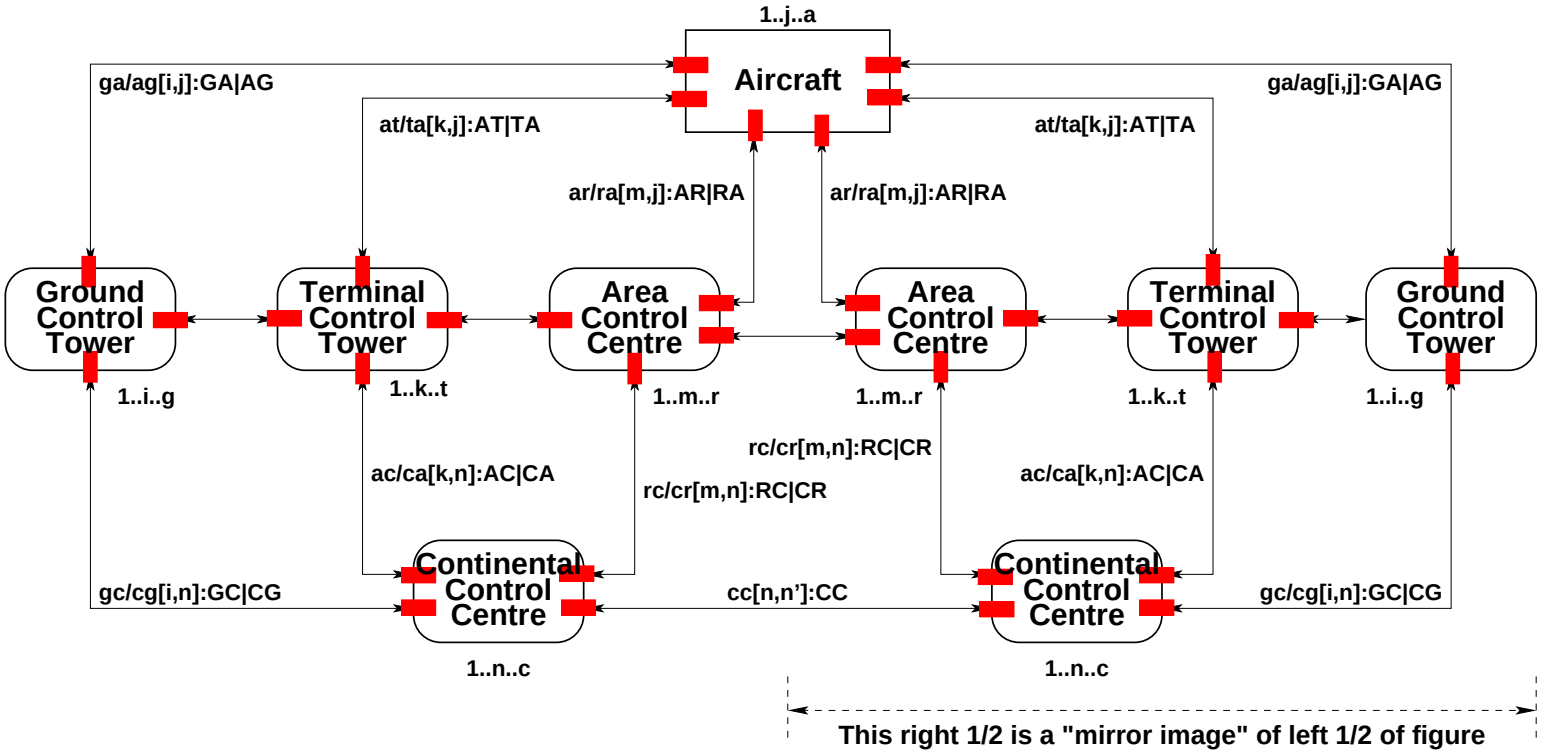

Figure 1: An air traffic system

2.5.2. Buildings

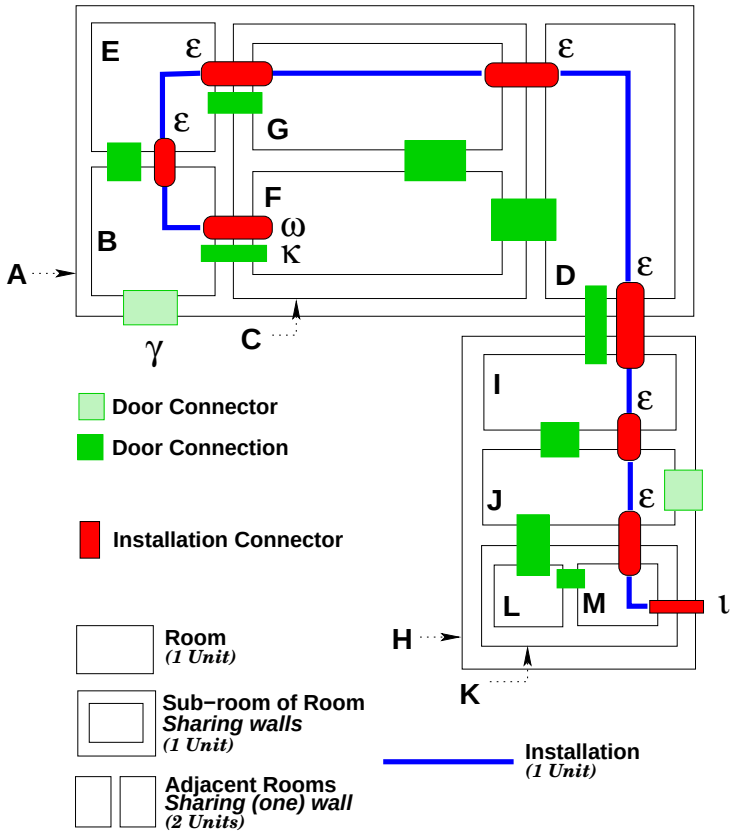

Figure 2: A building plan with installation

2.5.3. Financial Service Industry

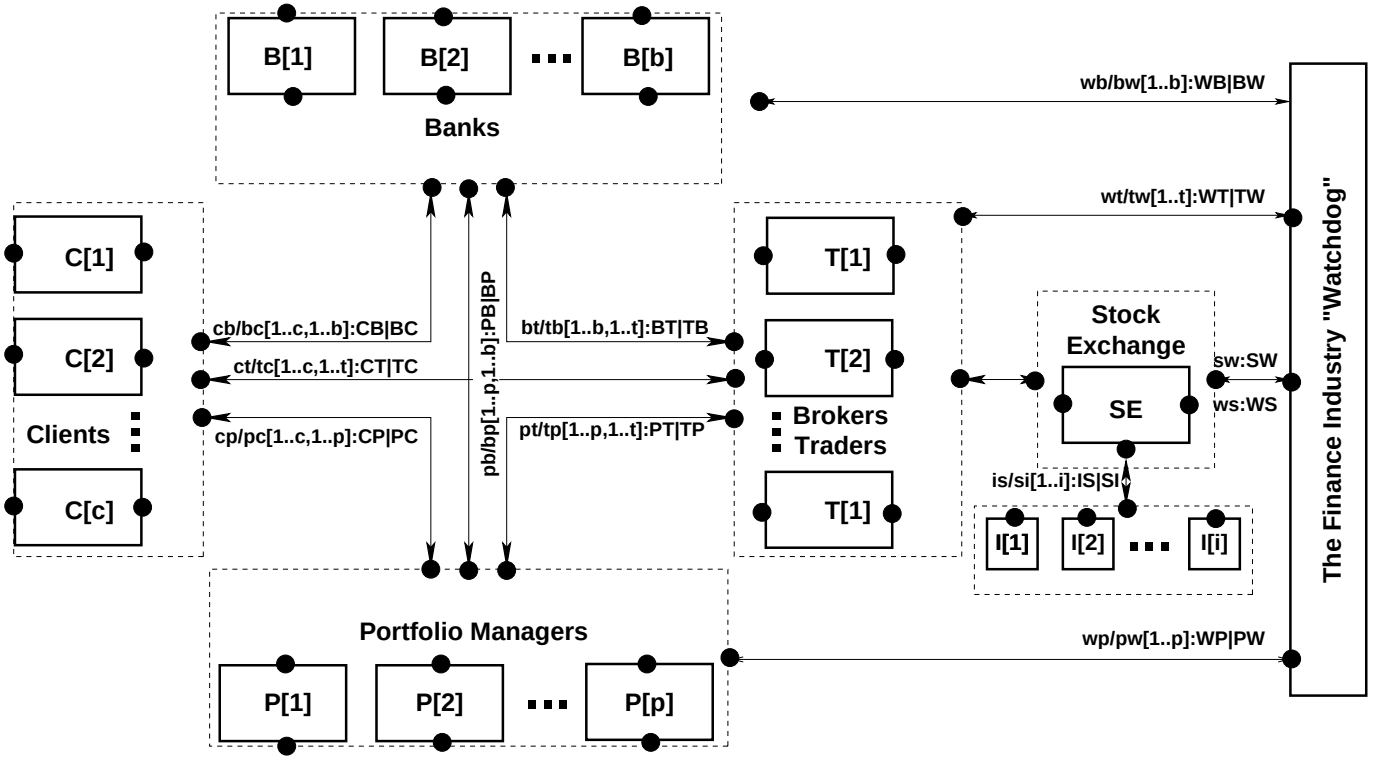

Figure 3: A financial service industry

2.5.4. Machine Assemblies

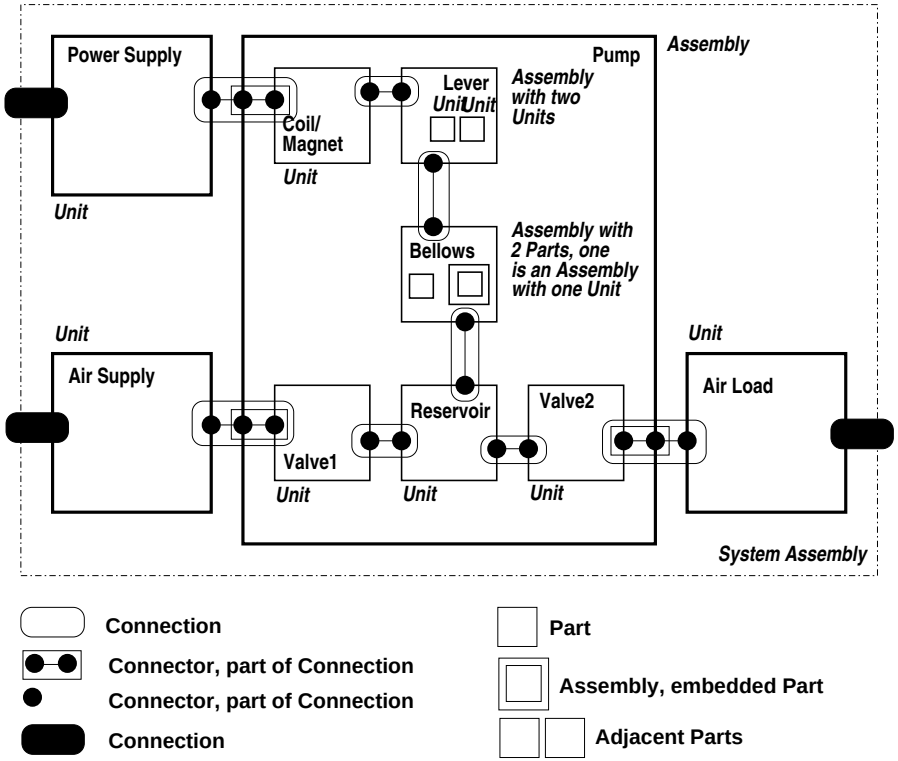

Figure 4: An air pump, i.e., a physical mechanical system

2.5.5. Oil Industry

2.5.5.0.1. "The" Overall Assembly

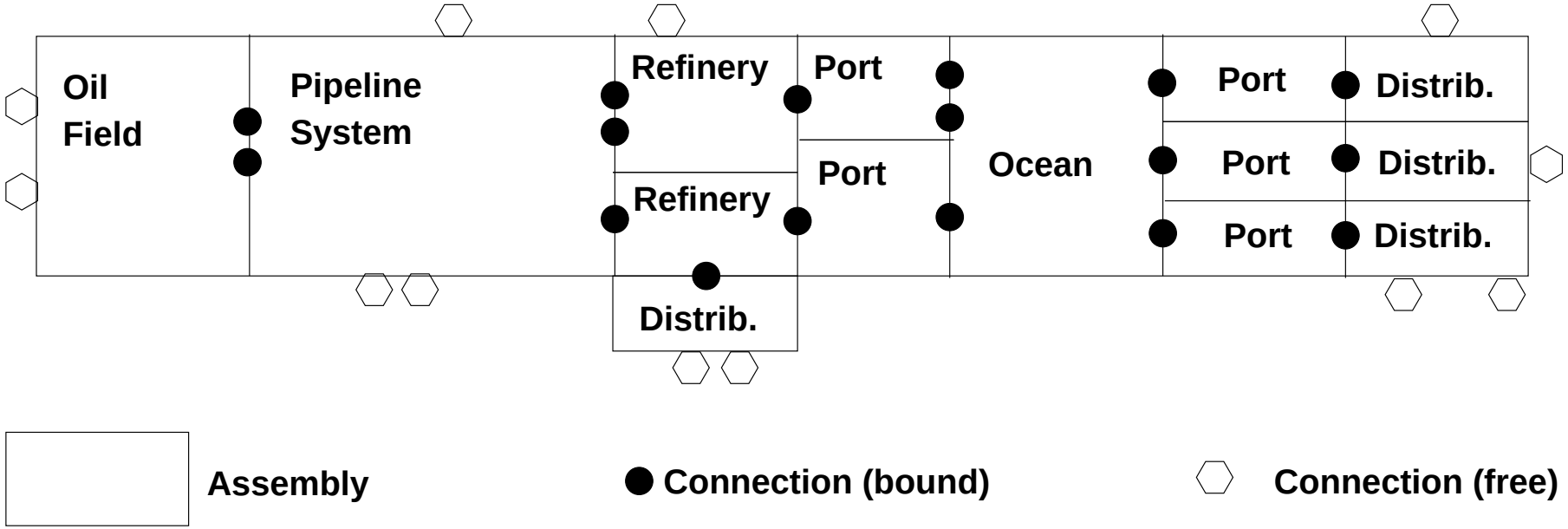

Figure 5: A Schematic of an Oil Industry

2.5.5.0.2. A Concretised Composite parts

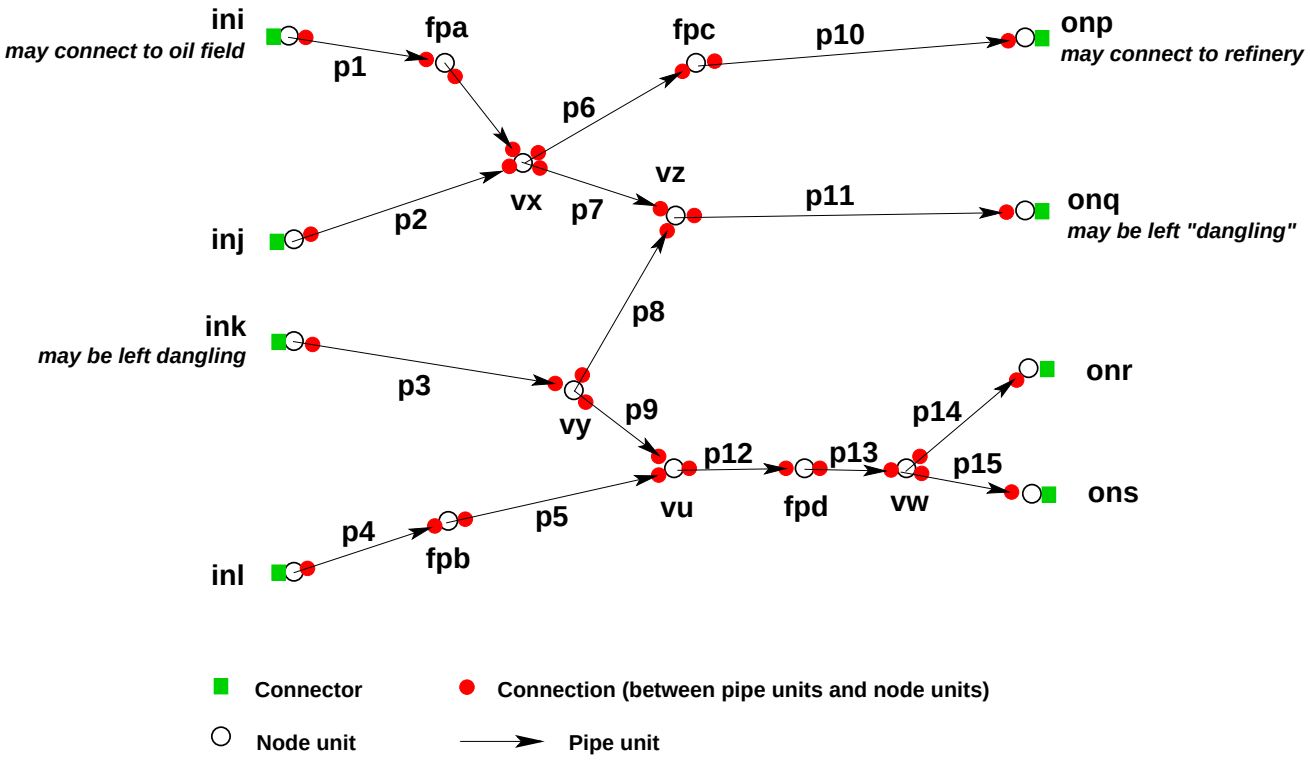

Figure 6: A Pipeline System

2.5.6. Railway Nets

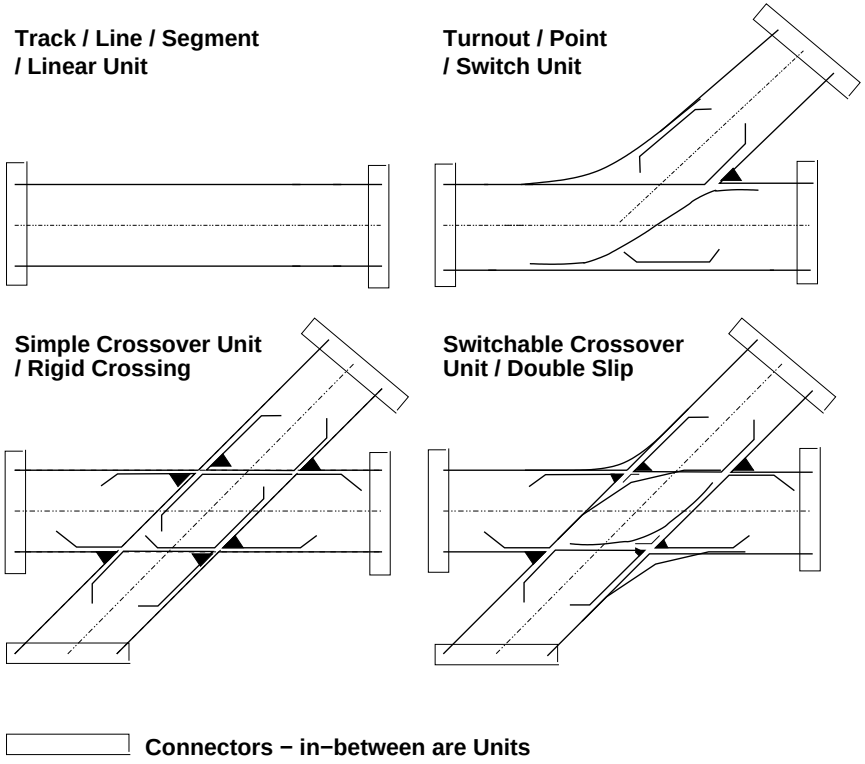

Figure 7: Four example rail units

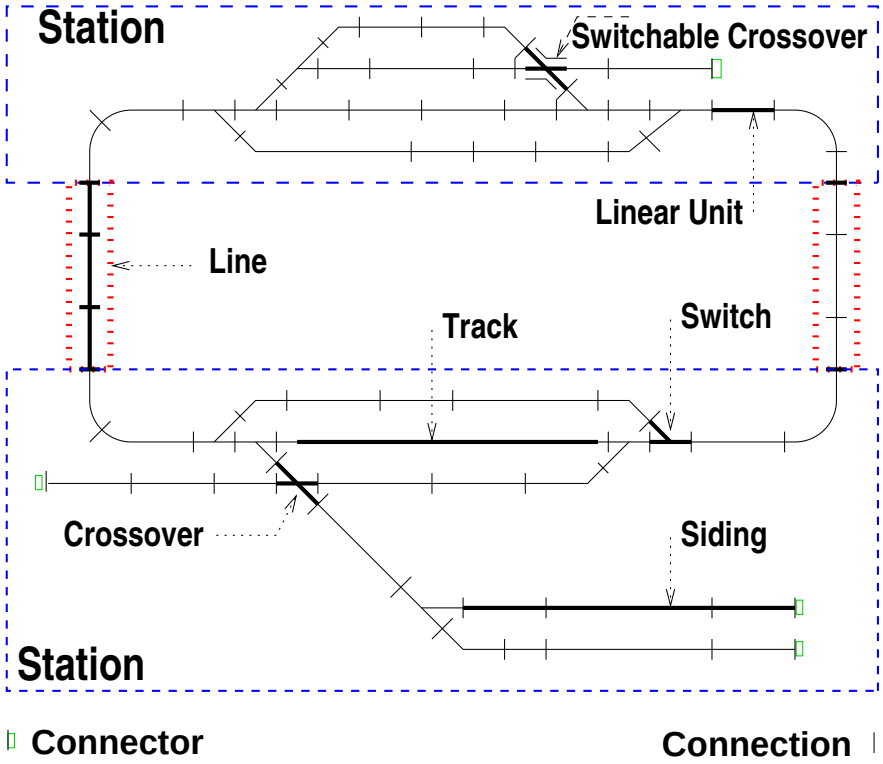

Figure 8: A "model" railway net. An Assembly of four Assemblies:
Two stations and two lines; Lines here consist of linear rail units;
stations of all the kinds of units shown in Fig. 7 on the previous page.
There are 66 connections and four "dangling" connectors

2.6. Discussion

-
-
-

End Lecture 2: Domains

Lecture 3: Entities

3. Entities

- By a domain entity we mean
  - a *fact*[19] which is
    - either a *part*
    - or an *action*
    - or an *event*
    - or a *behaviour*.
  - In contrast to facts we have *concepts*, that is, abstractions derived from facts.
  - Concepts can also be considered entities.
- Domain entities are
  - the things, the tangible, spatial facts we observe and
  - the concepts we abstract from these.

[19] We use the terms 'fact', 'entity', 'particular', 'thing' and 'individual' synonymously

3.0.0.0.1. Examples

Example 6 (Domain Entities) One example per each of four entity categories:

- *part:* transport net;
- *action:* insertion of link;
- *event:* disappearance of a link segment (that is, fraction of a link); and
- *behaviour:* movement of vehicles along net. •

3.1. Parts

- By a part we understand
  - a manifest, an endurant,
    - ∗ that is, something we can point to,
    - ∗ inert, possibly dynamic, i.e., animate,

    phenomenon
  - or a concept thereof,
  - something that we might (later on) represent as data by a computer.

3.1.0.0.1. Examples

Example 7 (Parts) Five domain examples:

- *Container line:*
  - container,
  - container vessel,
  - container terminal port,
  - bill of lading, etc.
- *Financial service industry:*
  - bank,
  - bankbook,
  - money (notes, coins),
  - insurance policy,
  - stock certificate, etc.
- *Transportation:*
  - net,
  - link,
  - hub,
  - vehicle,
  - driver, etc.

- *Health care:*
  - hospital,
  - ward,
  - bed,
  - patient,
  - medical staff,
  - medical record,
  - medicine,
  - surgery
  - instruments,
  - health insurance policy, etc.
- *Pipeline system:*
  - well,
  - pump,
  - pipe,
  - valve,
  - fork,
  - join,
  - sink,
  - pipeline, etc.

•

3.1.1. Atomic Parts

- By an atomic part we shall understand
  - a part
  - which we, as observers, have decided
  - form an indivisible whole,
  - that is, one for which it is not, in a current context, relevant to speak of meaningful subparts.

3.1.1.0.1. Examples

Example 8 (Atomic Parts) Five domain examples:

- *Container Line:* container, bill of lading, way bill.
- *Financial Service Industry:* bankbook, money; insurance policy; stock certificate.
- *Health Care System:* bed, patient, medical record, health insurance policy.
- *Pipeline System:* well, pump, pipe, valve, fork, join, sink.
- *Transportation System:* link, hub, vehicle, driver. •

3.1.2. Composite Parts

- By a composite part we shall understand
  - a part
  - which we, as observers, have decided
  - consists of one or more proper parts
  - also referred to as subparts.

3.1.2.0.1. Examples

Example 9 (Composite Parts and Subparts) Five domain examples:

- *Container Line:* container vessel and its bays; bay and its rows; row and its stacks, stack and its containers.
- *Financial Service Industry:* bank and its accounts.
- *Health Care System:* hospital and its wards; ward and its bedrooms, bedroom and its beds.
- *Pipeline System:* pipeline and its wells, pumps, pipes, valves, forks, joins and sinks.
- *Transportation System:* net and its hubs and links. •

3.1.3. Part Attributes

- By an attribute we shall mean a pair:
  - a type name and
  - a value (of that type).

3.1.3.1. Atomic Part Attributes

- By the attributes of an atomic part we mean
  - the set of properties (type names and values)
  - that we have decided
  - together characterise that atomic part
  - (and all of the atomic parts of the same type).

3.1.3.1.1. Examples

Example 10 (Atomic Part Attributes) Five domain examples:

- *Container line:* `container attributes:` length, width, height, weight, refrigerated or not refrigerated, physical location, contents, etc.
- *Financial service industry:* `account attributes:` interest rate (on loans), yield (on deposits), owner(s), maximum credit, current balance, etc.
- *Health care:* `patient attributes`: name, central personal registration identifier, gender, birth date, birth place, nationality, weight, height, insurance policies, medical record, etc.
- *Pipeline system:* `pipe attributes`: circular diameter, length, location, maximum laminar flow, current flow, guaranteed maximum leak (in volume/second), current leak, etc.
- *Transportation:* `link attributes`: length, location, link state (open in one direction or the other or open in both or closed in both directions), link type (road, rail, sea, air), etc. •

3.1.3.2. Composite Part Attributes

- By the attributes of a composite part we mean
  - the set of properties (type names and values)
  - (exclusive of all subparts of that composite part)
  - that we have decided
  - together characterise that composite part
  - (and all of the composite parts of the same type).

3.1.3.2.1. Examples

Example 11 (Composite Part Attributes) Five domain examples:

- *Container Line Attributes:*
  - name,
  - legal residence,
  - incorporated ?,
  - responsible capital,
  - organisation,
  - subsidiaries,
  - budget,
  - accounts,
  - etcetera.
- *Financial Service Industry Attributes, Bank:*
  - name of bank,
  - kind of bank,
  - legal residence,
  - responsible capital,
  - organisation,
  - subsidiaries,
  - budget,
  - accounts,
  - etcetera.

- *Health Care System Attributes, Hospital:*
  - name,
  - kind of hospital,
  - legal residence,
  - legal owner,
  - organisation,
  - financing,
  - budget,
  - accounts,
  - etcetera.
- *Pipeline System Attributes:*
  - name,
  - legal residence,
  - legal owner,
  - financing,
  - geography,
  - subcontractors,
  - budget,
  - accounts,
  - etcetera.

- *Transportation System Attributes:*
  - name,
  - kind of transport system[20],
  - legal residence ,
  - legal owner,
  - financing,
  - geography,
  - subcontractors,
  - budget,
  - accounts,
  - etcetera. •

• • •

[20] whether a road system or a rail/train system, or an airline, or a shipping company, etc.

3.1.3.3. Static Part Attributes

- A part attribute is static
- if that part
- never changes its value.

3.1.3.3.1. Examples

Example 12 (Static Part Attributes) Two examples:

- **Patients:**
  - name,
  - central personal registration identifier,
  - gender, and
  - birthplace.
- **Links:** length.

3.1.3.4. Dynamic Part Attributes

- A part attribute is dynamic
- if that part
- can change its value.

3.1.3.4.1. Examples

Example 13 (Dynamic Part Attributes) Three examples:

- The
  - height,
  - weight,
  - blood pressure,
  - blood sugar,
  - temperature, and
  - PMR

  of a patient are dynamic attributes.

- A hub typically can
  - connect a number of distinct links and
  - thus can attain either one of number of hub states
  - each hub state being a possibly empty set of pairs, $(\mathsf{li}_j,\mathsf{li}_k)$,
  - of not necessarily distinct link identifiers (li) of the links connected to that hub.

  The state of a hub is a dynamic attribute.
- Similarly for link states. •

3.1.3.5. Indivisibility of Attributes

- Given a part of some kind (i.e., having some set of attributes), whether atomic or composite,
- one cannot "remove" an attribute from that entity
- and still retain the entity as being of that kind.

3.1.3.5.1. Examples

Example 14 (Indivisibility of Attributes) Two examples:

- One cannot remove the attribute 'height' from an entity of kind person
- and one cannot remove the attribute 'kind of transport system' from an entity of kind 'transport system'. •

3.1.4. Subparts Are Parts

- By a subpart, p', of a part, p, we thus
  - mean an entity which is not the same as the part, that is $p \neq p'$.
  - We say that a part, p', is a *proper part* of another part if it is a subpart of that part.
  - So by proper part of p and subpart of p we mean the same.

3.1.4.1. Examples

Example 15 (Sets of Hubs and Hubs – Sets of Links and Links)
From a net we observe sets of hubs and sets of links:

- A set of hubs is a value of the type sets of hubs.
- A hub is a value of type hub.
- A set of links is a value of the type sets of links.
- A link is a value of type link. •

3.1.5. Subpart Types Are Not Subtypes

- Thus, by a subpart type
  - we mean a part type
  - but the type of the subpart cannot be the same as the type of the part of which it is a subpart.

3.1.5.1. Examples

Example 16 (Part and Subpart Types) We refer to Example 15.

- Let a part be a transportation net, *n:N*.
- A subpart of a transportation net, *n:N*, is, for example, the part *hs:HS*, which is the set of all hubs of the net,
- and a hub, *h:H*, which is a part of *hs:HS*, is a subpart of *hs:HS*.
- And all these subparts are of different types, to wit: *HS* and *H*,
- and, as we shall see, *LS* and *L*, are not subtypes of type *N*. •

- By a 'union', A, of *disjoint types*, say B, C, ..., D, that is: $A{=}B|C|...|D$, we mean
  - a type whose values are either of type B or of type C or ... of type D,
  - and where every type value is of exactly one of the types B, C, ..., D.
- These types, B, C, ..., D, are subtypes of A.
- Thus subpart types are not the same as subtypes of the part of which the subpart is a proper part.
- To be consistent we rule out the possibility of defining types recursively.

3.1.6. Mereology of Composite Parts

- By the mereology of a composite part we understand
  - the number of subparts
  - of respective kinds (types)
  - of that composite entity and
  - how the subparts are related to one another.

3.1.6.1. Examples

Example 17 (Mereology of Composite parts) Five domain examples:

- *Container Line System:* A container vessel contains a number of uniquely identified bays, bays consists of a sequentially indexed sequence of (usually several) rows, and rows consist of a sequentially indexed sequence of (usually several) stacks, and stacks consists of a sequence of zero or more containers — such that access to stacks are by identity of bay, number of row, number of stack and then to the top of this possibly empty stack. Etcetera.
- *Financial Service Industry:* A bank consists of (i) a set of uniquely identified demand/deposit accounts, (ii) a set of uniquely identified savings & loan accounts, (iii) a set of uniquely identified mortgage accounts. Etcetera.

- *Health Care System:* A hospital consists of (1) a set of uniquely identified wards of kind κ_1, (2) a set of uniquely identified wards of kind κ_2, ..., and (n) a set of uniquely identified wards of kind κ_n. Etcetera.
- *Pipeline System:* A pipeline system consists of a set of units — where units a either wells, pumps, pipes, valves, forks, joins or sinks — and such that (a) a well is connected to one or more pumps, (b) a pipe is input-connected to either a pipe or a pump or a valve or a fork or a join and is output-connected to either a pipe or a pump or a valve or a fork, (c) a pump is input-connected to a pipe and is output-connected to a pipe, (d) a valve is input-connected to a pipe and is output-connected a pipe or a sink, (e) a fork is input-connected to a pipe and is output-connected to two pipes, (f) a join is input-connected to two pipes and is output-connected to a pipe, and (g) a sink is input-connected to a valve. Etcetera.

- *Transportation System:* A transport net consists of a set of hubs and a set of one or more links such that links connect exactly two distinct hubs, and thus such that hubs are connected to zero or more distinct links. Etcetera.
  - The mereology of a net can be expressed in terms of unique identifiers associated with hubs, hij, hik, ..., him, and links, lia, lib, ..., lic. ●

- Mereologically two parts, e_i, e_j, may stand in the following relationships:
  - (a) either e_i is identical to e_j,
  - (b) or e_i is fully disjoint from e_j,
  - (c) or e_i is adjacent (i.e., connects) to (disjoint from, but "touches") e_j,
  - (d) or e_i is fully contained within e_j,
  - (e) or e_i partially overlaps with e_j (that is, there are "areas" of e_i which are not overlapping with "areas" of e_j).

3.1.7. Part Descriptions

- To describe an atomic part (type) it suffices to describe all the atomic part attributes:
  - its type name,
  - the attributes, and
  - its possible contribution to the mereology of "a whole":
    - own unique identification, and
    - how it 'unique identifier'-relates to other parts.
- To describe a composite part (type) it is necessary to describe these things:
  - (i) all the composite part attributes,
  - (ii) each of the subpart types (i.e., subparts), and
  - (iii) their mereology.

3.1.7.1. Examples

Example 18 (Description of An Atomic Part) We continue our example of transport nets. A link is here considered an atomic part.

- **Type Name:** link.
- **Attributes:** length, location[21], current state[22] and state space[23], etc.
- **Unique Identification:** unique Link identifier.
- **Mereology:** a pair of unique hub identifiers. •

[21] The cartographic and cadestral location of a link may, amongst other components, include, for example, a Beziér curve description of how that link "traverses" a, or the landscape.

[22] in terms of sets of pairs of distinct identifiers of connecting hubs

[23] in terms of sets of possible link states

Example 19 (Description of A Composite Part) We continue our example of transport nets. A net is here considered a composite part.

- **Type Name:** net.
- **Attributes:** name, transport kind, legal address, legal owner, sources of financing, geographical area, maintenance subcontractors, budget, accounts, etc.
- **Unique Identification:** `not applicable.`
- **Mereology:** `not applicable.`
- **Subpart Type[s]:** set of links, set of hubs. •

3.1.8. States

- By a state we understand
  - a specific set of parts
  - such that for each of these parts
  - some attributes are dynamic.

3.1.8.1. Examples

Example 20 (States) Five domain examples:

- *Container line:* container, container vessel, container terminal port.
- *Financial service industry:* bank (as a whole), account (as a subpart).
- *Health care:* hospital, ward, bed, patient.
- *Pipeline system:* well, pump, pipe, valve, pipeline.
- *Transportation:* net, link (open in one direction, open in the opposite direction, open in both directions; closed in all [two] directions), hub (open between a specific [possibly empty] set of pairs of links connected to the hub), vehicle. •

3.2. Actions

- By an action we understand
  - a state change
  - resulting directly from the expected application of a specific function (one of several possible),
  - that is, the specific function was performed deliberately, on purpose.

3.2.0.1. Examples

Example 21 (Actions) We give examples from five domains.

- The examples are not proper descriptions of actions.
- We basically just give their names.
- These names — and the familiarity of the domains —
  - are such that the reader is "tricked into" thinking
  - oh yes, I see; but, of course,
  - only a proper action description can reveal the action.

- *Container line:*
  - loading a container;
  - unloading a container;
  - moving a container from one location (say on-board a vessel) to another location (say in a container terminal port).
- *Financial service industry:*
  - open an account,
  - deposit into an account,
  - withdraw from an account,
  - obtain account statement,
  - close account.

- *Health care:*
  - admitting a person as a patient;
  - allocating a bed in a ward to a patient;
  - medicating a patient.
- *Pipeline system:*
  - opening a pump (for pumping);
  - closing a valve.
- *Transport Net:*
  - inserting a hub;
  - inserting a link;
  - removing a hub;
  - removing a link.

•

3.3. Events

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

3.3.0.1. Example

Example 22 (Events) Five domain examples:

- *Container line:* A container falls overboard.
- *Financial service industry:* A bank goes bankrupt.
- *Health care:* A patient dies.
- *Pipeline system:* A pipe breaks.
- *Transportation:* A link disappears. ●

3.4. Behaviours

- By a behaviour we understand
  - a set of sequences of
  - actions, events and behaviours.

3.4.0.1. Example

Example 23 (Behaviours) Five domain examples:

- *Container line:* The transport of a container
  - from it being fetched at the sender,
  - via a sequence of one or more triplets of
    - loadings onto a vessel,
    - unloading at another container terminal port
    - and possibly temporary storage at that port,
  - to its final delivery at a receiver.

- *Financial service industry*, account handling:
  - the opening of an account,
  - a sequence of
    - deposits,
    - withdrawals and
    - statements
  - to the closing of that account.

- *Health care*, patient hospitalisation:
  - the admission of a patient to a hospital,
  - initial
    - anamnese,
    - analysis,
    - diagnostics and
    - treatment plan,

    via an alternating sequence of
    - treatments (including surgical operations),
    - repeated analyses,
    - evaluations and possible reformulation of
    - diagnostics and
    - treatment plan,
  - to a final discharge.

- *Pipeline system*, simple, day-to-day operations.
  - The flow of gas (or a liquid) through a pipeline net:
    - pumped from wells,
    - fed through
      - pipes,
      - valves,
      - forks and
      - joins,
    - to leaving the net at sinks.

- *Transportation:* The movement of a vehicle along a transport net:
  - from positions at
    - ∗ hub or
    - ∗ link

    positions
  - via a sequence of zero, one or more
    - ∗ hub and
    - ∗ link

    movements,
  - to a final
    - ∗ hub or
    - ∗ link

    position.

  Example 4 (Slides 23–36) illustrated a transport behaviour. •

3.5. Discussion

- We have dealt, in some detail, with the concept of parts (Part 3.1, Slides 86–115).
- Our "corresponding" treatment of actions, events and behaviours (Parts 3.2–3.4, Slides 117–127) have been far less detailed.
- The reason for this is the following.
  - Types emerge (Part 3.1) as a means of describing parts.
  - And types are indispensable in the description of action, event and behaviour signatures (Parts 3.2–3.4).
  - Types thus form the very basis for the description of all entities.
  - And we have chosen to let the type concept emerge from our treatment of parts.

- There is another reason for Part 3.1 being somewhat more detailed than Parts 3.2–3.4.
  - When studying parts we could, relatively easily, introduce such notions as
    - atomic and composite parts,
    - attributes of these, and
    - mereologies of composite parts.
  - These notions, under some disguise,
    - can likewise be found for actions, events and behaviours,
    - but they are not that easily introduced.

End Lecture 3: Entities

Lecture 4: Describing Domain Entities. Part I: Parts, Actions, Events

4. Describing Domain Entities

4.1. On Describing

- The purpose of description is
  - to use for example informal text
  - to present an entity (simple, action, event or behaviour)
  - so that the reader may
    - "picture",
    - "envisage"

    that which is being described.
- The text describing the entity
  - is said to be a syntactic quantity.
  - and the entity is then said to be a semantic quantity:
  - the syntactic text *denotes* the semantic quantity.

- We also say that the syntactic quantity
  - designates,
  - denotes,
  - indicates,
  - specifies,
  - points out,
  - gives a name
  - or title to, or
  - characterises[24]

  the semantic quantity.

4.1.1. Informal Descriptions

- In the many examples[25] of Parts 2–3 we have made several references to quite a few domain entities.
- We do not claim that we have described these entities.

[24] — eight alternative terms for the same idea!
[25] Examples 5–23

4.1.1.1. A Criterion for Description

- For us, to informally describe an entity ideally means the following:
  - *Let there be given what is claimed to be a description of some domain entity, call it* e.
  - *Let a person read and claim to have understood that description.*
  - *Now that person is confronted with some phenomenon* e'.
  - *Either that phenomenon is the same or it is of the same kind (type) as* e *or it is not.*
  - *If* e' *is of the same kind as* e *then the person must identify it as such, unequivocally.*
  - *If* e' *is not of the same kind as* e *then the person must identify it as not being so, likewise unequivocally.*
- If a description does not satisfy the above then it is not a *proper description*.

4.1.1.2. Reason for 'Description' Failure

- There can be three reasons for a description to not be proper:
  1. *either all phenomena are entities as described — the description is vacuous;*
  2. *or there are entities which were meant to be of the type or not meant to be of the type described but which*
     - *– "fall outside", respectively*
     - *– "fall inside"*

     the description;
  3. *or the description does not make sense, is "gibberish", ambiguous, or otherwise.*
- That is: a proper description, when applied to entities, "divides" their world into two non-empty and disjoint sets:
  - – the set of all entities being described by the description,
  - – and the rest !

4.1.1.3. Failure of Description Language

- But we have a problem !
  - One cannot give a precise definition of exactly the *denoting language*, that is,
    - of exactly, all and only those informal texts
    - which designate entities.
  - Firstly,
    - we have not given a sufficiently precise informal text characterisation of entities,
  - Secondly,
    - natural (cum national) languages, like English, defy such characterisations.
- We must do our best with informal language descriptions.

4.1.1.4. Guidance

- But there is help to be gotten!
- The whole purpose of Part 3 was to establish the pointers, i.e., guidelines, as to what must be described, generally:
  - parts, actions, events and behaviours,
  - and specifically:
    - whether atomic or composite parts,
    - their attributes, and, optionally, their mereology,
    - and, for composite parts, their subparts;
    - and, as a starter, the signatures of actions, events and behaviours.
- This part will continue the line
  - reviewed just above
  - and provide further hints, pointers, guidelines.

4.1.2. Formal Descriptions

- We shall,
  - in addition to the *description components*[26], outlined in Part 3
  - now join the possibility of
    - improved description precision
    - through the use of formal description.
- We argue that formal description,
  - while being used in-separately with precise informal narrative.
  - improves precision
  - while enabling formal proofs of properties of that which is denoted by the description.

[26] parts, actions, events and behaviours; attributes and possibly unique identifiers of parts, and mereology of composite (atomic) parts; subparts of composite parts; etc.

- We shall here use the term 'formal' in the sense of mathematics.
- A formal description language is here defined to have
  - a formal syntax,
  - a formal semantics, and
  - a proof system.
- We shall "unravel" an example formal description language, FDL, in this section.
  - FDL has similarities to the RAISE Specification Language, RSL,
  - but, as our informal explanation of the meaning of FDL will show, it is not RSL.
  - The similarities are "purely" syntactical.

4.2. A Formal Description Language
4.2.1. Describing Parts

- We observe parts but describe types.
- In order to describe a type we use such phrases as:
  - *a patient is characterised by a name, a central personal registration identifier, its gender, birth date, birth place, nationality, weight, height, insurance policy, medical record,* etcetera;
  - *a transport net is characterised by a set of hubs and a set of links.*
- Thus we take the nouns *name, central personal registration identifier, gender, birth date, birth place, nationality, weight, height, insurance policy, medical record, . . . , set of hubs,* and *set of links* as type names.
- The name 'patient' is also the domain name.

- That is,
  - we go from instance of an entity
  - to the type of all entities "of the same kind".
- One must take great care in not confusing the two: type and value).
- Later we shall clarify the distinction between type and domain names.

4.2.1.1. Abstract Types

- By an *abstract type* we generally mean some further unexplained set of mathematical quantities.
  - Abstract types are in contrast to concrete types
  - by which we mean such mathematical quantities as sets, Cartesians, lists, maps and functions (in general).
  - Abstract types are also referred to as *sorts*.
- The FDL clause:

  type A

  - defines what we shall here, simplifying, take as a set of values said to be of type A.
  - A is said to be a *type name* (here, more specifically, a *sort name*).

4.2.1.2. Concrete Types

- Borrowing from RSL, and, in general, discrete mathematics, we introduce FDL clauses for expressing set, Cartesian, list, map and function types.
- Let A, B, ..., C be (type or sort) names which denote some (not necessarily distinct) types, then

$$\text{A-}\mathbf{set},\ (\text{A} \times \text{B} \times ... \times \text{C}),\ \text{A}^*,\ \text{A}^\omega,\ \text{A} \xrightarrow[m]{} \text{B},\ \text{A} \rightarrow \text{B},\ \text{A} \xrightarrow{\sim} \text{B}$$

are *type expressions* which denote the following (left-to-right) concrete types:

  - set of sets of type A values;
  - sets of Cartesian values, (a,b,...,c), over types A, B, ..., C;
  - set of finite lists of elements of type A values;
  - set of possibly infinite lists of elements of type A values;
  - set of maps, that is enumerable functions from type A into type B values;
  - set of total functions from type A into type B values; respectively
  - set of partial functions from type A into type B values.

- Choosing to describe a part as a sort rather than a concrete type reflects a *principle* of abstraction.
- Modelling a concrete type in terms of, for example, a map type rather than as an indexed set reflects a modelling *technique*.

4.2.1.3. Type Definitions

- Besides
  - the sort type definitions **type** A
  - there are the concrete type definitions.
- Let A be some (unused) name (i.e., a type name), then

  type D = Type_Expression
  A | B | ... | C

  - is a concrete type definition
  - where Type_Expression is of either of the forms **A-set**, (A×B×...×C), A^*, A^ω, $A \xrightarrow[m]{} B$, A→B, $A \xrightarrow{\sim} B$ and A|B|...|C,
  - where A, B, ..., C are either type names or, more generally, other such type expressions and where
  - A|B|...|C expresses the "union" type of the A. B. ..., and C types.

Example 24 (Transport Net Types) Let us exemplify the above:

32. We focus on the transport net domain.
    That domain is "dominated" by the composite parts of nets, *n:N*.

32. **type** N

33. There are two subparts of nets:
    (a) sets, *hs:HS*, of hubs (seen as one part) and
    (b) sets, *ls:LS*, of links (also seen as one part).

33(a). HS

33(b). LS

- That is:

33(a). obs_HS: N → HS

33(b). obs_LS: N → LS

34. Hubs and links are subparts of respectively *HS* ('sets of hubs'), respectively *LS* ('sets of links'), are of type
    (a) *H* and
    (b) *L*, respectively.

34(a). H

34(b). L

That is:

type	**value**
34(a). HS = H-**set**, Hs = HS	34(a). obs_Hs: HS → H-**set**
34(a). LS = L-**set**, Ls = LS	34(b). obs_Ls: LS → L-**set**

35. Hubs have unique identifier (mereology) attributes
36. Links have unique identifier (mereology) attributes

35. HI
36. LI

- That is:

35. uid_HI: H → HI
36. uid_LI: L → LI

●

- If nothing further is expressed about A, then these values range over any mathematical object imaginable.
- Thus, to "better" characterise the intended A values the describer might then "add" some axioms to the description:

  axiom $\forall$ a:A $\cdot$ $\mathcal{P}$(a)

- where $\mathcal{P}(a)$ stands for a predicate which thus ranges over A values.

4.2.1.4. Type Properties

- In the following we shall be introducing a number of functions which analyse parts with respect to respective *properties*.
- There are three kinds of properties of interest to us:
  - the subparts of composite parts,
  - the mereology of composite parts,
  - and general attributes of parts (apart from their possible subparts and mereology).

- Every entity, whether simple, or an action, or an event, or a behaviour, has a unique identification.
  - The mere existence, in time and space, endows a part with a unique identification as follows.
    * No two spatial parts can occupy overlapping space,
    * so some abstract spatial location is a form of unique identification.
  - We consider the unique identity of a part of type A, say AI, as a general attribute.
  - We use the attribute values of AI to formulate mereologies.

Thus there are three kinds of property analysis functions.

- *Subpart observer functions*, obs_B, obs_C, ..., obs_D, which apply to composite parts (say of type A), and yield their constituent subparts, say of type B, C, ..., D:

  obs_B: A → B, obs_C: A → C, ..., obs_D: A → D;

- *Mereology functions* which apply to composite parts (say of type A), and yields elements of their mereologies.
  - Let parts of type B, C, ..., D be in some mereology relation to parts of type A.
  - That is, there are mereology functions

    mereo_Ax: B → AIx, mereo_Ay: C → AIy, ..., mereo_Az: D → AIz

    * where Ax, Ay and Az are some distinct identifiers and
    * AIx, AIy and AIyz are some type expressions over type A, typically AI, AI-**set**, (AI×...×AI), etc.

- *Attribute Functions* which apply to parts (say of type A) and yield their attributes (short of the mereologies) (say of types E, F, ..., G:

  attr_E: A → E, attr_F: A → F, ..., attr_G: A → G;

  Among the general attribute functions are the unique identification functions, say attr_A.

4.2.1.5. Subpart Type Observers

- Given a composite sort, named, say, A,
- and postulating that its values contains subparts of type B,
- one can observe these type B subparts of A using
- the likewise postulated *observer function*:

  obs_B: A → B

- If A values also contain subparts of types C, ..., E, then there also exists the additional observer functions: obs_C, ..., obs_E.
- We say that the observer functions are postulated.
  - We postulate them.
  - And we endow them with properties
  - so that they "stand out" from one another.
    - First examples of properties are given by the observer function signatures: from type A values observer function obs_B yields B values.
    - Further properties may be expressed through axioms.

Example 25 (Subpart Type Observers) From nets we observe

37. sets of hubs and
38. sets of links.

value

37. obs_HS: N → HS
38. obs_LS: N → LS

●

4.2.1.6. Unique Identifier Functions

- All parts have unique identifiers.
- This is a dogma.
- We may never need some (or any) of these unique part identifiers.
- But they are there nevertheless.

Example 26 (Unique Hub and Link Identifiers) From hubs and links we observe their unique

39. hub and
40. link

identifier attributes:

```
value
39.  uid_HI: H → HI
40.  uid_LI: L → LI
```

●

4.2.1.7. Mereologies and Their Functions

Example 27 (Transport Net Mereology) To express the mereology of transport nets we build on the unique identifications of hubs and links.

41. Links connect exactly two distinct hubs, mereo_HIs.
42. Hubs are connected to zero, one or more distinct links, mereo_LIs.

type
41. HIs = HI-**set**, HIS = HI-**set**

axiom
41. ∀ his:HIs·**card** his=2

type
42. LIs = LI-**set**, LIS = LI-**set**

value
41. mereo_L: L → HIs
42. mereo_H: H → LIs

- The above (Items 41–42) form the basis for expressing the constraints on how hubs and links are connected.

43. Given a net, its link and hub observers and the derived link and hub identifier extraction functions, the mereology of all nets must satisfy the following:
    (a) All link identifiers observed from hubs must be of links of that net and
    (b) All hub identifiers observed from links must be of hubs of that net.

44. We introduce two auxiliary functions for extracting all hub and link identifiers of a net.

axiom
43. ∀ n:N,
43. **let** ls=obs_LS(n),hs=obs_HS(n),
43. lis=xtr_LIS(n),his=xtr_HIS(n) **in**
43(a). ∀ h:H·h ∈ hs ⇒ mereo_LIs(h)⊆lis ∧
43(b). ∀ l:L·l ∈ ls ⇒ mereo_HIs(l)⊆his **end**
value
44. xtr_LIS: N → LI**-set**, xtr_HIS: N → HI**-set**
44. xtr_LIS(n)≡{uid_LI(l)|l:L·l ∈ obs_LS(n)}
44. xtr_HIS(n)≡{uid_HI(h)|h:H·h ∈ obs_HS(n)}

●

4.2.1.8. General Attributes and Their Functions

Example 28 (Hub States and State Spaces)45. With hubs we can associate

(a) a hub state, $h\sigma$, consisting of a set of pairs of link identifiers (with (li_j, li_k) in $h\sigma$ expressing that traffic is open from link l_j to link l_k via hub h),

(b) a hub state space, $h\omega$, consisting of a set of hub states,

(c) the identifiers of the links connected to hubs (part of the net mereology),

(d) such that the link identifiers recorded in the

type
45(a). HΣ = (LI×LI)set
45(b). HΩ = HΣ**-set**
value
45(c). mereo_LIs: H → LI**-set**
45(a). attr_HΣ: H → HΣ
45(b). attr_HΩ: H → HΩ
axiom
45(d). ∀ h:H · **let** lis = mereo_LIs(h), hσ = attr_HΣ(h), hω = attr_HΩ(h) **in**
45(d). **let** hs = {(li,_),(_,li)|li:LI·li ∈ lis} **in** hσ⊆hs ∧ hσ ∈ hω **end end**

●

Example 29 (Further Atomic Attributes) In addition to the unique link identifier links also have, for example, lengths, widths, possibly heights, geographic (spatial) locations, etc.

type
  LEN, WID, HEI, LOC, ...
value
  attr_LEN: L → LEN
  +: LEN × LEN → LEN
  >: LEN × LEN → **Bool**
  attr_WID: L → WID
  attr_HEI: L → HEI
  attr_LOC: L → LOC
  ...

●

4.2.2. Describing Actions
4.2.2.1. Function Names

- Actions potentially change states.
- Actions are here considered deliberate phenomena
  - in that they are caused by willful applications
  - (by agents within the domain being described)
  - of functions
  - having a specific, deliberate purpose,
  - i.e., state change in mind.

Example 30 (Transport Net Action Names) Some examples are:

- *create* an *empty net* (no hubs, no links);
- *insert* a (new) hub;
- *insert* a (new) *link* (between a pair of distinct hubs of the net);
- *delete* (existing) *hub*; and
- *delete* (existing) *link*.

●

4.2.2.2. Informal Function Descriptions

- The above examples just listed some actions by their function names.
- We did not describe these functions.
- We now do so, for two of these functions.

Example 31 (Informal Transport Net Action Descriptions) We detail two informal function descriptions.

- The *create empty net* function

46. applies to nothing and yields a net, n, of no hubs and no links;

- and the *insert link* function

47. applies to a net, n, of at least two distinct hubs, as identified, hi_j, hi_k, by the function application arguments, and a new link ℓ not in n, and yields a net, n'.

(a) The inserted link l is to be connected to the two distinct hubs identified by hj_i and hk_i, and these designate hubs of the net.
(b) l is not in the original net.
(c) its mereology designate hj_i and hk_i,
(d) and the state of these identified hubs do not allow any traffic.
(e) No new hubs are added, hence the set of hub identifiers of the net is unchanged.
(f) Only one link is added to the net, hence the set of net link identifiers is changed by only the addition of the l link identifier.
(g) Let the hubs identified by hj_i and hk_i be hj and hk, respectively, before the insertion,
(h) and by hj' and hk' after the insertion.
(i) Now the mereology contributions by the two changed hubs reflect the addition of link l.

We leave the informal descriptions of

48. *delete_L*
49. *insert_H*
50. *delete_H*

to the reader.

●

4.2.2.3. Formal Function Descriptions

- We observe actions,
  - but describe the functions
  - which when applied amount to actions.
- There are two parts to describe a function:
  - (i) the function signature, $\mathsf{a}:\mathsf{A}\rightarrow\mathsf{B}$:
    - a distinct function name, say f, and
    - a function type, $\mathsf{A}\rightarrow\mathsf{B}$, that is,
      - type of arguments A and
      - type of results B, and
  - (ii) the function definition, $\mathsf{f}(\mathsf{a},\mathsf{b})\equiv\mathcal{C}(\mathsf{a})$:
    - a symbolic function invocation, $\mathsf{f}(\mathsf{a},\mathsf{b})$, and
    - a definition body, $\mathcal{C}(\mathsf{a})$.
      - $\mathcal{C}(\mathsf{a})$ is a clause,
      - i.e., an expression in FDL,
      - whose evaluation
      - yields the function value.

- There are other ways than:

value
f: A → B
f(a,b) ≡ $\mathcal{C}$(a)

- in which to define a function, for example:

value
f: A → B
f(a) **as** b
pre $\mathcal{P}$(a)
post $\mathcal{Q}$(a,b)

Example 32 (Formal Transport Net Action Descriptions) •

The *create net* function:

value

46. create_N: **Unit** → N
46. create_N() **as** n
46. **post** obs_HS(n)={} ∧ obs_LS(n)={}

- The *insert link* function:

value

47. insert_L: (HI × HI) × L → N → N
47. insert_L((hij,hik),l)(n) **as** n′
47(a). **pre** hji≠hki ∧ {hji,hki} ⊆ xtr_HIs(n) ∧
47(b). ∧ l ∉ obs_LS(n)
47(c). ∧ mereo_LIs(l) = {hji,hki}
47(d). ∧ obs_HΣ(get_H(hji))(n)={}=obs_HΣ(get_H(hki))(n)
47(e). **post** xtr_HIs(n) = xtr_HIs(n′)
47(f). ∧ xtr_LIs(n′) = xtr_LIs(n) ∪ {uid_LI(l)}
47(g). ∧ **let** hj=get_H(hij)(n), hk=get_H(hik)(n),
47(h). hj′=get_H(hij)(n′),vhk′=get_H(hik)(n′) **in**
47(i). ∧ mereo_LIs(hj′) = mereo_LIs(hj) ∪ {uid_LI(l)}
47(i). ∧ mereo_LIs(hk′) = mereo_LIs(hk) ∪ {uid_LI(l)} **end**

51. From the postulated observer and attribute functions one can define the auxiliary *get* function:

value

51. get_H: HI → N $\xrightarrow{\sim}$ H
51. get_H(hi)(n) ≡ **let** h:H•h ∈ obs_HS(n)∧uid_HI(h)=hi **in** h **end**
    pre hi ∈ xtr_HIs(n)

- We do not narrate the informal description of "remaining" net actions (cf. Items 48– 50 on page 165), just their function signatures and pre-conditions.

48. delete_L: LI → N $\xrightarrow{\sim}$ N
48. **pre** delete_L(li): li ∈ xtr_LIs(n)
49. insert_H: H → N $\xrightarrow{\sim}$ N
49. **pre** insert_H(h): h ∉ obs_HS(n)∧mereo_LIs(h)={}∧mereo_Ω(h)={{}}
50. delete_H: HI → N $\xrightarrow{\sim}$ N
50. **pre** delete_H(hi): hi ∈ xtr_HIs(n)∧mereo_LIs(get_H(hi))(n)={}

- Given appropriate post-conditions the following theorems must be provable:

theorems:

∀ h:H · **pre**−conditions are satisfied ⇒ delete_H(uid_HI(h))(insert_H(h)(n))=n
∀ l:L · **pre**−conditions are satisfied ⇒ delete_L(uid_LI(l))(insert_L(l)(n))=n

●

4.2.2.4. Agents

- An *agent* is a behaviour
  - which invokes functions,
  - hence cause actions.
- So we simply "equate" agents with behaviours.

4.2.3. Describing Events

- We observe events. But we describe logical properties characterising classes of "the same kind" of events.

4.2.3.1. Deliberate and Inadvertent (Internal and External) Events

- Events are like actions: somehow a function was applied
  - either *deliberately* by an agent outside (that is, *external* to) the domain being described,
  - or *inadvertently* by a behaviour of (that is, *internal* to) the domain, but for another purpose than captured by the event.

Example 33 (A Deliberate [External] Event) We narrate a simple external cause / "internal" effect example:

- When one or more bank customers default on their loans
- and declare themselves unable to honour these loans
- then the bank may go bankrupt.

●

Example 34 (An Inadvertent [Internal] Action Event) We narrate a simple internal cause / "internal" effect example:

- When a bank customer, the agent, withdraws monies from an account
  - the balance of that account, if the withdrawal is completed,
    - may go negative,
    - or may go below the credit limit.
  - In either case we say that,
    - the withdrawal action, as was intended, succeeded,
    - but that an "*exceeded credit limit*" event occurred.

●

4.2.3.2. Event Predicates

- Instead of describing events
  - by directly characterising the
  - deliberate external, respectively
  - inadvertent internal
  - actions
- we suggest to describe these events
  - indirectly,
  - by characterising the logical effects,
  - say, in terms of predicates over *before/after* states.

Example 35 (Formalisation of An External Event) • The event is that of a *"link segment disappearance"*.

52. Generally we can explain *"link segment disappearances"*, for example, as follows:

53. A l_i-identified link, l, between hubs hf and ht (identified in l) is *removed*.

54. Two hubs, hf'' and ht'', and two links, lf and lt, are *inserted* — where
    (a) hub values hf and ht (the hubs in the original net) become hub values hf' and ht' in the resulting net, that is, hub values hf and ht have same respective hub identifiers as hf' and ht',
    (b) hubs hf'' and ht'' are new,
    (c) links l' and l'' are new,
    (d) link lf is *inserted* between hf' and hf'', that is, link lf identifies hubs hf' and hf'',
    and link lt is inserted between ht' and ht'', that is, link lt identifies hubs ht' and ht'',

(e) hub hf' is, in the resulting net, connected to the links hub hf was connected to in the original net "minus" link l but "plus" link lf,

(f) hub ht' is, in the resulting net, connected to the links it was connected to in the original net "minus" link l but "plus" link lt,

(g) hub hf'' is connected only to link lf
and hub ht'' is connected only to link lt,

(h) the state space of hf' suitably includes all the possibilities of entering link lf[27],

(i) the state space of ht' suitably includes all the possibilities of entering link lt[28],

(j) the state spaces of hf'' and ht'' both contains just the empty set,

(k) the states of ht'' and ht'' are both the empty set: "dead ends !",

(l) the sum of the lengths of links lf and lt is less than the length of link l, and

(m) all other non-mereology attributes of lf and lt are the same as those of link l.

55. All other links and hubs are unchanged.

[27] A substitution function replaces all link l identifiers with link lf identifiers.
[28] A substitution function replaces all link l identifiers with link lt identifiers.

value

52. link_segment_disappearance: N × N → **Bool**
52. link_segment_disappearance(n,n′) ≡
52. ∃ l:L, hf″,ht″:H, lf,lt:L •
53. {l} = obs_LS(n) \ \obs_LS(n′)
54(a). ∧ **let** hfi=uid_HI(hf), hti=uid_HI(ht) **in**
54(a). **let** hf′=get_H(hfi)(n′), ht′=get_H(hti)(n′) **in**
54(b). {hf″,ht″}∩ obs_HS(n)={} ∧ {hf″,ht″}⊆obs_HS(n′)
54(c). ∧ {lf,lt}∩ obs_LS(n)={} ∧ {lf,lt}⊆obs_LS(n′)
54(d). ∧ attr_HIs(l′)={hfi,uid_HI(hf″)} ∧ attr_HIs(l″)={hti,uid_HI(ht″)}
54(e). ∧ attr_LIs(hf)=attr_LIs(hf′)\{uid_LI(l)}∪{uid_LI(lf)}
54(f). ∧ attr_LIs(ht)=attr_LIs(ht′)\{uid_LI(l)}∪{uid_LI(lt)}
54(g). ∧ attr_LIs(hf″)={uid_LI(lf)} ∧ attr_LIs(ht″)={uid_LI(lt)}
54(h). ∧ attr_HΩ(hf′)=subst(uid_LI(l),uid_LI(lf),attr_HΩ(hf))
54(i). ∧ attr_HΩ(ht′)=subst(uid_LI(l),uid_LI(lt),attr_HΩ(ht))
54(j). ∧ attr_HΩ(hf″)={{}} ∧ attr_HΩ(ht″)={{}}
54(k). ∧ attr_HΣ(hf″)={} ∧ attr_HΣ(ht″)={}
54(l). ∧ attr_LEN(lf)+attr_LEN(t)<attr_LEN(l)
54(m). ∧ ∀ X:non_mereo_attributes(l)•attr_X(lf)=attr_X(lt)=attr_X(l)[29]
55. ∧ obs_HS(n′)\{hf′,hf″,ht′,ht″} = obs_HS(n)\{hf,ht}
55. ∧ obs_LS(n′)\{lf,lt} = obs_LS(n)\{l} **end end**

[29] The predicate in Line 54(m) on the preceding page is to be explained.

- We can express a theorem relating the above to the
  - *rem*ove and
  - *ins*ert functions.

theorem: link_segment_disappearance(n,n′) ⇒
let l:L, hf″,ht″:H, lf,lt:L · [Lines 53–54, 54(a)–54(m), 55] **in**
n′ = ins_L(lt)(ins_L(lf)(ins_H(ht″)(ins_H(hf″)(rem_L(uid_LI(l))(n)))))
end

•

End Lecture 4: 1st part: Describing Domain Entities [Parts–Events]

Lecture 5: 2nd part: Describing Domain Entities [Behaviours]

4.2.4. Describing Behaviours

4.2.4.1. Behaviour Description Languages

- As for the description of parts, actions and events[30] there exists formal ways of describing behaviours as of sequences of actions, events and behaviours:
  - some are "textual"[31]:
    - CSP,
  - some are "graphical", for example:
    - MSC,
    - Petri Nets and
    - State Charts.

[30] Part, action and event description languages were first mentioned in the 'Summary' footnotes 1– 2 on page 2: Alloy, CafeOBJ, Casl Event B, Maude RAISE/RSL , VDM and Z.

[31] The languages mentioned in Footnote 30 are textual.

4.2.4.1.1. Simple Sequential Behaviours

- A simple sequential behaviour
  - is a sequence of actions and events.
- We describe a simple sequential behaviour, for example,
  - by describing the first action or event, the second action or event, etcetera.
  - Typically we do so by listing the names, act_or_pred_fct, and arguments[32] of the action functions and event predicates.
  - One such form is:

$$
\begin{array}{l}
\textbf{let } \sigma' = \text{act\_or\_pred\_fct\_1}(\text{argl\_1})(\sigma) \textbf{ in} \\
\textbf{let } \sigma'' = \text{act\_or\_pred\_fct\_2}(\text{argl\_2})(\sigma') \textbf{ in} \\
\textbf{let } \sigma''' = \text{act\_or\_pred\_fct\_3}(\text{argl\_3})(\sigma'') \textbf{ in} \\
\ldots \\
\text{act\_or\_pred\_fct\_m}(\text{argl\_m})(\sigma'\cdots') \ldots \textbf{ end end end}
\end{array}
$$

[32] Event predicates have no arguments.

- If act_or_pred_fct_i(argl_i)(σ_i) represents an event predicate,
  - then the predicate form is pred_fct_i(σ_i, σ_i').
- We can name such simple sequential behaviours, for example:

  P: f_m(a_m)(..(f_3(a_3)(f_2(a_2)(f_1(a_1)(σ))))

4.2.4.1.2. Simple Concurrent Behaviours

- A simple concurrent behaviour
  - is a set of two or more simple sequential behaviour.
- We describe a simple concurrent behaviour
  - by a list of named behaviour descriptions
  - separated by the **parallel** behaviour composition operator ||,
  - for example, P||Q||...||R.
  - A variant form is ||{P(i)|i:Index•predicate(i)} which expresses 'distribution' of the behaviour composition operator (||) over 'expanded' terms, that is, $\mathsf{P}(\mathsf{i}_j)\|\mathsf{P}(\mathsf{i}_k)\|\ldots\|\mathsf{P}(\mathsf{i}_\ell)$.

4.2.4.1.3. Communicating Behaviours

- A communicating behaviour
  - is a behaviour which expresses willingness to engage
  - in a (synchronisation and) communication
  - with another communicating behaviour
  - or with the environment.
- In order to express 'communication' (between behaviours)
  - a notion of an output/input *channel* is introduced
  - with behaviours allowed 'access' to channel (which are therefore shared).
- In CSP channels are typed with the type of the values that can be output on a channel "between" behaviours.

- In CSP
  - output of a value (say of expression e) onto channel ch is expressed by the statement ch!e whereas
  - input of a value from channel ch is expressed by the expression ch?.

type
  M
value
  P: **Unit** → **out** ch **Unit**

Q: **Unit** → **in** ch **Unit**
P() ≡ ... ch!e ...
Q() ≡ ... ch? ...

- We thus describe a communicating behaviour by allowing one or more clauses:
  - statements of the kind ch!e and
  - expressions of the kind ch?.

4.2.4.1.4. External Non-deterministic Behaviours

- A behaviour, $\mathcal{P}$, composed from behaviours $\mathcal{P}_i, \mathcal{P}_j, \ldots, \mathcal{P}_k$ is said to exhibit internal non-determinism if the behaviour is
  - either as is behaviour $\mathcal{P}_i$,
  - or as is behaviour $\mathcal{P}_j$,
  - $\ldots$,
  - or as is behaviour $\mathcal{P}_k$,
  - and is influenced in being so by the environment of behaviour $\mathcal{P}$.
- We describe such behaviours as follows:
  P: P_i [] P_j [] ... [] P_k
  where P_i (etcetera) describes behaviour $\mathcal{P}_i$ (etcetera).
- A variant description of internal non-determinism is
  []{P(i)|i:Index·predicate(i)}.

- External influence is, for example, expressed
  - if behaviour descriptions P_i (etcetera)
  - contain either an output (ch!e) or an input (ch?) clause
  - and the environment offers to accept input, respective offers output
  - "along" the name channel.

4.2.4.1.5. Internal Non-deterministic Behaviours

- A behaviour, $\mathcal{P}$, composed (somehow) from behaviours $\mathcal{P}_i, \mathcal{P}_j, \ldots, \mathcal{P}_k$ is said to exhibit internal non-determinism if the behaviour is
  - either as is behaviour $\mathcal{P}_i$,
  - or as is behaviour $\mathcal{P}_j$,
  - $\ldots$,
  - or as is behaviour $\mathcal{P}_k$,
  - and is not influenced in being so by the environment of behaviour $\mathcal{P}$.
- We describe such behaviours as follows:
  P: P_i ⫿ P_j ⫿ ... ⫿ P_k
  where P_i (etcetera) describes behaviour $\mathcal{P}_i$ (etcetera).
- A variant description of internal non-determinism is
  ⫿{P(i)|i:Index·predicate(i)}.

- For internal non-determinism to work for expressions like the above
  - we must assume that they do not contain such
  - output (ch!e) or an input (ch?) clauses
  - for which the environment may accept input, respectively offer output.

4.2.4.1.6. General Communicating Behaviours

- A general communicating behaviour
  - is a set of sequences of actions, events and
  - (simple sequential, simple concurrent, communicating and non-deterministic)
  - behaviours
  - such that at least two separately identifiable behaviours of a set
  - share at least one channel and
  - contain respective ch!e and ch? clauses.

Example 36 (A Road Pricing Transport System Behaviour)

This example is quite extensive.

56. A road pricing transport system contains
    (a) a net n — as outlined in earlier examples — of hubs and links,
    (b) a fleet f of vehicles and
    (c) a central road pricing monitor m.

    from which we can derive the
    (d) set of hubs,
    (e) set of links and
    (f) set of vehicles.

57. With the road pricing behaviour we associate separate behaviours,
    (a) one for the net which is seen as the parallel composition of
        i. a set of hub behaviours
        ii. a set of link behaviours;
    (b) one for the fleet of vehicles which is seen as the parallel composition of
        i. a set of vehicle behaviours;
    (c) and a central road pricing monitor behaviour.

type
56(a). N
56(b). F
56(c). Monitor
value
56(b). obs_VS: F → VS

56(a). n:N
56(b). f:F
56(c). m:Monitor

56(d). hs:HS=obs_HS(n)
56(e). ls:LS=obs_LS(n)
56(f). vs:VS=obs_VS(f)

57. road_pricing_system: **Unit** → **Unit**
57. road_pricing_system() ≡ net()‖fleet()‖monitor(...)

57(a). net() ≡
57((a))i. ‖ {hub(uid_HI(h))(h)(vis)|h:H·h ∈ hs} ‖
57((a))ii. ‖ {link(uid_LI(l))(l)(vis)|l:L·l ∈ ls}

57(b). fleet() ≡
57((b))i. ‖ {vehicle(obs_VI(v))(v)(vp)|v:V·v ∈ vs}

57(c). monitor(...) ≡ ...

- The *vis* arguments of the *hub* and *link* behaviours "carry" the identifiers of current *v*ehicles currently at the hub or on the link.
- The *vp* argument of the *vehicle* behaviour "carries" the current *v*ehicle *p*osition.
- The (...) argument of the *monitor* behaviour records the history status of all vehicles on the net.

58. We associate channels as follows:
    (a) one for each pair of vehicles and hubs,
    (b) one for each pair of vehicles and links and
    (c) one for monitor (connected to vehicles).

58. **channel**
58(a). {vh_ch[uid_VI(v),uid_HI(h)]|v:V,h:H•v ∈ vd∧h ∈ hs}:VH_Msg
58(b). {vl_ch[uid_VI(v),uid_LI(h)]|v:V,l:L•v ∈ vs∧h ∈ ls}:VL_Msg
58(c). mon:VM_Msg

- We omit detailing the channel message types.

59. Vehicles are positioned
    (a) either on a link, in direction from one hub to a next, some fraction down that link,
    (b) or at a hub, in direction from one link to a next where
    (c) the fraction is a real between 0 and 1.

type
59. VP = onL | atH
59(a). onL == mk_onL(li:LI,fhi:HI,f:FRA,thi:HI)
59(b). atH == mk_atH(hi:HI,fli:LI,tli:LI)
59(c). FRA = **Real axiom** ∀ fra:FRA·0≤fra≤1

60. The *vehicle* behaviour is modelled as a CSP process which communicates with hubs, links and the monitor.
61. The *vehicle* behaviour is a relation over its position.
    If on a link, at some position,
    (a) then the vehicle may "remain" at that position,
    (b) chosen so internally non-deterministically,
    (c) or, if the vehicle position is not "infinitesimally" close to the "next" hub,
    (d) then the vehicle will move further on along the link,
    (e) some small fraction δ,
    (f) else the vehicle moves into the next hub in direction of the link named li'
    (g) where li' is in the set of links connected to that hub —
    (h) while notifying the link, the hub and the monitor of its entering the link and entering the hub.

value
61(e). δ:**Real axiom** $0<\delta \ll 1$vehc6
60. vehicle: VI $\rightarrow$ V $\rightarrow$ VP $\rightarrow$
60. **out**,**in** {vl_ch[vi,li]|li:LI•li $\in$ xtr_LIs(ls)} **out** m_ch **Unit**
61. vehicle(vi)(v)(vp:mk_onL(li,fhi,f,thi)) $\equiv$
61(a). vehicle(vi)(v)(vp)
61(b). $\prod$
61(c). **if** f + δ<1
61(d). **then** vehicle(vi)(v)(mk_onL(li,fhi,f+δ,thi))
61(e). **else let** li′:LI•li′ $\in$ mereo_LIs(get_H(thi)(n)) **in**
61(g). vh_ch[vi,thi]!enterH || vl_ch[vi,li]!leaveL || m_ch!leaveL_enterH(vi,li,thi);
61(h). vehicle(vi)(v)(mk_atH(thi,li,li′)) **end end**

62. If the vehicle is at a hub,
    (a) then the vehicle may "remain" at that same position,
    (b) chosen so internally non-deterministically,
    (c) or move on to the next link,
    (d) in direction of a next hub,
    (e) while notifying the hub and monitor of leaving the hub and the link and the monitor of entering the link.

62. vehicle(vi)(v)(vp:mk_atH(hi,fli,tli)) ≡
62(a). vehicle(vi)(v)(vp)
62(b). ⌈⌉
62(d). **let** {hi′,thi}=mereo_HIs(getL(tli)(n)) **in assert:** hi′=hi
62(e). vh_ch[vi,hi]!leaveH ‖ vl_ch[vi,tli]!enterL ‖ m_ch!leaveH_enterL(vi,hi
62(c). vehicle(vi)(v)(ml_onL(tli,hi,0,thi)) **end**

63. The monitor behaviour records the (dynamic) history of all vehicles on the net: alternating sequences of hub and link identifiers.

64. The monitor contains a price table which to every link and hub records the fee for moving along that link or hub.

type

63. $\mathrm{VW}' = \mathrm{VI} \xrightarrow{m} (\mathrm{HI}|\mathrm{LI})^*$
63. $\mathrm{VW} = \{|\mathrm{vw}:\mathrm{VW}'\cdot\mathrm{wf\_VW}(\mathrm{vh})|\}$
64. $\mathrm{Fee}, \mathrm{PT} = (\mathrm{LI}|\mathrm{HI}) \xrightarrow{m} \mathrm{Fee}$

value

63. $\mathrm{wf\_VW}(\mathrm{vh}) \equiv$
63. $\quad \forall\ \mathrm{hll}:(\mathrm{HI}|\mathrm{LI})^* \cdot \mathrm{hll} \in \mathbf{rng}\ \mathrm{vh}$
63. $\quad\quad \forall\ \mathrm{i}:\mathbf{Nat}\cdot\{\mathrm{i},\mathrm{i}+1\}\subseteq\mathbf{inds}\ \mathrm{hll} \Rightarrow$
63. $\quad\quad\quad \mathrm{is\_HI}(\mathrm{hll}(\mathrm{i}))\wedge\mathrm{is\_LI}(\mathrm{hll}(\mathrm{i}+1))\vee\mathrm{is\_LI}(\mathrm{hll}(\mathrm{i}))\wedge\mathrm{is\_HI}(\mathrm{hll}(\mathrm{i}+1))$

65. The monitor behaviour non-deterministically externally alternates between
    (a) input of messages from vehicles
        i. either when entering a link
           in which case the vehicle history is updated with that link's identifier (for that vehicle),
        ii. or when entering a hub
            in which case the vehicle history is updated with that hub's identifier (for that vehicle).
    (b) and accepting inquiries and requests relating vehicle histories and fees (designated by (...) below).

value

65. monitor: PT → VH → **in** m_ch **Unit**
65. monitor(pt)(vh) ≡
65(a). (**case** m_ch? **of**
65((a))i. leaveH_enterL(vi,hi,li) → monitor(pt)(vh † [vi ↦ vh(vi)⌢⟨li⟩])
65((a))ii. leaveL_enterH(vi,li,hi) → monitor(pt)(vh † [vi ↦ vh(vi)⌢⟨hi⟩])
65(a). **end**)
65(b). [] (...)

- We omit description of other monitor actions (Line 65(b)).

66. Link behaviours maintain a state which records the set of vehicles "currently" on the link.

67. The link behaviour expresses willingness to
    (a) accept messages from vehicles
    (b) entering links in which case the "*vehicle vi on link*" state has *vi* added, or
    (c) leaving links in which case the "*vehicle vi on link*" state has *vi* removed,
    (d) where these vehicles range over all fleet vehicles.

type
66. VIS = VI-**set**
value
67. link: li:LI → L → VIS → **in** cl_Unitlink1
67. link(li)(l)(vis) ≡
67(a). [] {**let** m = cl_vl[vi,li′]? **in assert:** li′=li
67(a). **case** m **of**
67(b). enterL → link(li)(l)(vis ∪ {vi})
67(c). leaveL → link(li)(l)(vis \ {vi})
67(d). **end** | vi:V•v ∈ xtr_VIs(vs) **end**}

- We leave it to the reader to suggest a *hub* behaviour description.

●

4.3. Temporal Issues

4.3.1. Three Abstract Time Concepts

- We shall briefly examine three aspects of time:
  - *time*, t:T,
  - *absolute time intervals*, ati:(ft:T,tt:T):ATI and
  - *relative* (non-zero) *time intervals*, rti:RTI
    (where rti is some time interval from any time tf:T to some time tt:T "later").

type
  T
  ATI = T × T **axiom** ∀ (ft,tt):ATI · tt>ft
  TI, RTI
value
  +: T × RTI → T
  −: T × T ~→ RTI **pre** t−t′: t>t′
  >,=: T × T → **Bool**
  absolute_to_relative_TI: ATI → RTI
  absolute_to_relative_TI(ta,tb) ≡ tb−ta

- The concept of time has at least two variants:
  - the **time process** for which time continually increases, and
  - a **time descriptor** which is a name for time.
- When we, colloquially say
  - *the time is now*
    we mean to refer to the process notion;
- and when we say
  - *the train departs at time* `so-and-so`
    we mean to refer to the time descriptor notion.
- We model the time process notion as a behaviour;
- and the timenotion descriptor notion as an attribute.
- Absolute and relative time intervals are descriptors (i.e., attributes).

4.3.2. Concrete Time Concepts

- Usually, when speaking of time, we say such things as: *the time is 12:34 o'clock* but mean to say the more correct `12:34 pm, February 21, 2012`.
- An absolute time interval, (t, t'), starts at some time, t, and ends some time, t', thereafter.
- The time interval $t' - t$ only make sense is $t' > t$.
- We may include the zero time interval: $t - t$.
- Several different absolute time intervals may represent the same relative time interval.
- Two absolute time intervals, (t_a, t_b) and (t_c, t_d), define the same relative time interval iff $t_b - t_a = t_d - t_e$.

4.3.3. Some Interval Relations

value

=: ATI × ATI → **Bool**
(ta,tb) = (tc,td) ≡ ta=tc ∧ tb=td

prefix: ATI × ATI → **Bool**
prefix((ta,tb),(tc,td)) ≡ ta=tc ∧ tb<td

suffix: ATI × ATI → **Bool**
suffix((ta,tb),(tc,td)) ≡ ta>tc ∧ tb=td

embed: ATI × ATI → **Bool**
embed((ta,tb),(tc,td)) ≡ ta<tc ∧ tb>td

etcetera.

4.3.4. Time Phenomena

4.3.4.1. Parts and Time

- Time descriptors may occur (as components of attributes) in parts.

Example 37 (Train Time Tables) Our example shows a classical use of time descriptors.

68. Trains and stations have names.
69. A train system time table maps train names into train ride descriptions.
70. Train ride descriptions are sequences of two or more station visits
    (a) such that "later" station visits succeed, in time, those of earlier station visits, and
    (b) such that station arrival times precede those of same station, same visit departure times.
71. A station visit is a triple: an arrival time, a station name and a departure time.

type
68. TN, SN
69. TSTT = TN $\xrightarrow[m]{}$ TRD
70. TRD′ = SV*, TRD = {|trd•wf_TRD(trd)|}
71. SV′ = T × SN × T, SV = {|sv•wf_SV(sv)|}

value
70(a). wf_TRD: TRD′ → **Bool**
70(a). wf_TRD(trd) ≡
70(a). **len** trd≥2 ∧
70(a). ∀ i:**Nat**•{i,i+1}⊆**inds** trd ⇒
70(a). **let** (_,sn,dt)=trd(i), (at,sn′,_)=trd(i+1) **in**
70(a). sn≠sn′ ∧ dt<at **end**
70(b). wf_SV: SV′ → **Bool**
70(b). wf_SV(at,_,dt) ≡ at<dt

- The occurrences of time in the part time table are as part of static attributes of that table. •

4.3.4.2. Actions and Time

- Actions, "in actual life", takes time.
  - That is, over a(n absolute) time interval.
- But we abstract from this fact.
- When such an abstractions becomes unreasonable,
  - that is, when the abstraction distorts a factual representation of the domain,
  - then we shall model the action as a behaviour,
  - typically composed from a sequence of "smaller", i.e., constituent "actions".

- Still, actions do take place at specific times.
  - So we could decide to represent this temporal aspect of action by respective time stamps.
  - Here we must distinguish between
    - the action itself, i.e., its state change
    - and the description of this action.
    - The former can be represented by a pair of (before,after) states.
    - The latter by a description of the
      - function,
      - the non-state arguments when applied, and
      - the state "in" which it was applied.

- We decide, for the moment, to not be too specific about this issue.
- The reason for this is pragmatics:
  - The actions that we do describe (informally and formally) usually represent planned, deliberate phenomena.
    - The fact that they are then "performed" at a certain time is therefore of less importance than is the desired state change.
    - Should the time at which the action is performed influence the result (including state change) then we decide to let that time be an explicit argument of the function invocation that leads to the action.
  - But not all actions are deliberate.
    - Actions that are not planned and deliberate phenomena then represent erroneous actions, that is, actions which the agent has provided with erroneous arguments.
    - For such actions time stamps can be very useful.
- Time arguments are then obtained from a separate, you can call it a global, *time behaviour*, one that is always active.

4.3.4.3. Events and Time

- There are two pragmatic aspects which separate actions from events.
  - Firstly, in contrast to actions, the times at which events occur appear to be important.
    - Event occurrences are basically spontaneous: not planned for.
    - Therefore it may be important to record the time of their occurrence.
  - Secondly, we take the view that
    - actions are primarily characterised
      by their effect on the state,
      that is, we emphasise an explicit description of the state change,
    - whereas events appears to be primarily characterisable
      in terms of properties of the event,
      that is, we emphasise a predicate description of the state change.

4.3.4.4. Behaviours and Time

- It can be relevant to record time in connection with behaviours.

Example 38 (Timed Monitor Behaviour) We modify our monitor behaviour.

72. Vehicle histories now record the time at which vehicles enter and leave hubs and links.

type
72. Time
72. VH′ = VI $\overset{m}{\rightarrow}$ (Time×(HI|LI))*
72. VH = {|vh:VH′•wf_WH(wh)|}

value
72. wf_VH: VH′ → **Bool**
72. ∀ hll:(T×(HI|LI))* • hll ∈ **rng** vh
72. ∀ i:**Nat**•{i,i+1,i+2}⊆**inds** hll ⇒
72. **let** (t,hl)=hll(i), (t′,hl′)=hll(i+1), (t″,hl″)=hll(i+2) **in**
72. t<t′<t″
72. ∧ (is_HI(hll(i))∧is_HI(hll(i+1))∧is_LI(hll(i+2))
72. ∨ is_LI(hll(i))∧is_HI(hll(i+1))∧is_HI(hll(i+2))
72. ∨ is_LI(hll(i))∧is_LI(hll(i+1))∧is_HI(hll(i+2)))
72. **end**

73. The *monitor* behaviour now, additionally,
    (a) interacts with a (global) *clock* behaviour
    (b) over a channel *clock_ch*.
    (c) The clock internally non-deterministically issues the current time or skips that point while similarly internally non-deterministically either not stepping the clock, or stepping it
    (d) with an "infinitisimal" time interval.
74. With the introduction of a clock we need redefine the system context.

channel

73(b). **channel** clock_ch: T

value

73. monitor: PT → VH → **in** m_ch; **in**,**out** clock_ch **Unit**

73(a). clock: **Unit** → **out** clock_ch **Unit**

73(c). clock(t) ≡ (**skip** ⌈⌉ clock_ch!t) ; (clock(t ⌈⌉ (t+ti)))

73(d). ti:TI **axiom:** ti is a tiny time interval

74. timed_system() = road_pricing_system() || clock(t)

75. When accepting messages from a link or a hub the *monitor* inquires with the *clock* as to *"what time is it?"* by expressing willingness to input that time.
76. That time is recorded both as the time the vehicle leaves the hub and as the time it enters the link.
77. Similar for leaving links and entering hubs.

```
value
73.   monitor(pt)(vh) ≡
65(a).      (case m_ch? of
75.           let t = clock_ch? in
76.           leaveH_enterL(vi,hi,li) → monitor(pt)(vh † [ vi ↦ vh(vi)⌢⟨(t,hi),(t,li)⟩ ])
77.           leaveL_enterH(vi,li,hi) → monitor(pt)(vh † [ vi ↦ vh(vi)⌢⟨(t,li),(t,hi)⟩ ])
65(a).      end end)
65(b).      [] (...)
```

- The timed behaviour implied in the Road Pricing Transport System Behaviour with the Timed Monitor Behaviour can be made more explicit. •

Example 39 (Traffic Behaviours) We abstract further.

78. Traffic can be abstracted as a
    (a) continuous or
    (b) discrete)

    function from time to pairs of nets[33] and vehicle positions.

    (c) Vehicles are positioned either at hubs or on links.
    (d) Vehicle hub positions, in principle, need only be represented by the hub identifier.
    (e) Vehicle link positions can be represented by the link identifier and a pair of the fraction "down" the link from a hub, here identified by that hub's hub identifier.

[33] As nets may change due, for example, to insertion and removal of hubs and links.

type

78(a). $\mathrm{cTF} = \mathrm{T} \rightarrow (\mathrm{N} \times (\mathrm{V} \xrightarrow{m} \mathrm{VP}))$

78(b). $\mathrm{dTF} = \mathrm{T} \xrightarrow{m} (\mathrm{N} \times (\mathrm{V} \xrightarrow{m} \mathrm{VP}))$

78(c). $\mathrm{VP} = \mathrm{atH} \mid \mathrm{onL}$

78(d). $\mathrm{atH} :: \mathrm{HI}$

78(e). $\mathrm{onL} :: \mathrm{LI} \times (\mathrm{FRA} \times \mathrm{HI})$

- Instead of modelling vehicle positions
  - in terms of actual vehicles, v:V,
  - we can model vehicle positions in terms of vehicle identifiers, vi:VI.
- (It all depends on the use of the traffic abstracts, cTF and dTF.)
- Similarly for choosing continuous or discrete time models.

Example 40 (Traffic Abstraction) We can introduce a behaviour, *traffic*, which,

- based on communications with a modified *vehicle* behaviour,
- "builds" a discrete traffic.
- We start by redefining the *vehicle* behaviour shown on Slide 200.

79. The *traffic* behaviour accepts inputs from the *vehicle* behaviour over a shared channel *trf_ch*.
80. Whenever the *vehicle* is invoked a message is communicated to the *traffic* behaviour with the current vehicle position.

channel
79. trf_ch:(VI×VP)
value
60. vehicle: VI → V → VP →
79. **out**,**in** {vl_ch[vi,li]|li:LI·li ∈ xtr_LIs(ls)} **out** m_ch,**trf_ch Unit**
61. vehicle(vi)(v)(vp:mk_onL(li,fhi,f,thi)) ≡
80. (**trf_ch!(vi,vp)**;
61(a). vehicle(vi)(v)(vp))
61(b). ∏
61(c). **if** f + δ<1
80. **then** (**trf_ch!(vi,mk_onL(li,fhi,f+δ,thi))** ;
61(d). vehicle(vi)(v)(mk_onL(li,fhi,f+δ,thi)))
61(e). **else let** li′:LI·li′ ∈ mereo_LIs(get_H(thi)(n)) **in**
61(g). vh_ch[vi,thi]!enterH ‖ vl_ch[vi,li]!leaveL ‖ m_ch!leaveL_enterH(vi,li,thi)
80. ‖ **trf_ch!(vi,mk_atH(thi,li,li'));**
61(h). vehicle(vi)(v)(mk_atH(thi,li,li′)) **end end**

```
value
60.   vehicle: VI → V → VP →
79.     out,in {vh_ch[ vi,hi ]|hi:HI•hi ∈ xtr_HIs(hs)} out m_ch,trf_ch Unit
62.  vehicle(vi)(v)(vp:mk_atH(hi,fli,tli)) ≡
80.    (trf_ch!(vi,vp);
62(a).   vehicle(vi)(v)(vp))
62(b).   ⌈⌉
62(d).   let {hi′,thi}=mereo_HIs(getL(tli)(n)) in assert: hi′=hi
62(e).   vh_ch[ vi,hi ]!leaveH ‖ vl_ch[ vi,tli ]!enterL ‖ m_ch!leaveH_enterL(vi,hi,tli)
80.    ‖ trf_ch!(vi,ml_onL(tli,hi,0,thi));
62(c).   vehicle(vi)(v)(ml_onL(tli,hi,0,thi)) end
```

81. Traffic is simplied in not considering the otherwise possibly dynamically changing net.
82. The *traffic* behaviour, when accepting input from some vehcile, *vi*, as to its position, *vp*, obtains, the current time, *t*, from the *clock* behaviour.
83. A technicality: if there are recordings for some vehicles in the traffic, *trf*, at that tine,
84. then *trf* is modified in one way,
85. otherwise it is modified in another way.

```
type
81. TRF = T m→ (VI m→ VP)
value
79. traffic: TRF → in trf_ch, clock_ch Unit
79. traffic(trf) ≡
82.     let (vi,vp) = trf_ch?, t = clock_ch? in
83.     if t ∈ dom trf
84.         then trf † [ t ↦ trf(t) † [ vi ↦ vp ] ]
85.         else trf ∪ [ t ↦ [ vi ↦ vp ] ] end end
```

4.3.5. Temporal Descriptions

MORE TO COME

- Examples of temporal description languages are:
  - the Interval Temporal Logic (ITL),
  - the Duration Calculus (DC), and
  - the Temporal Logic of Actions$^+$ (TLA$^+$).

MORE TO COME

End Lecture 5: Describing Domain Entities. Part II: Behaviours

Lecture 6: Discovering Domain Entities

5. Discovering Domain Entities

- Section 2 briefly characterised, informally and also a bit more formally, what we mean by a domain.
- Section 3 informally and systematically characterised the four categories of entities: parts, actions, events and behaviours.
- Section 4 more-or-less "repeated" Sect. 3's material but by now giving more terse narratives (that is, informal descriptions) and, for the fist time, also formalisations.
- Section 4 did not hint at how one discovers domain parts (i.e., their types), actions, events and behaviours.
- In this section we try unravel a set of techniques and tools — so-called 'discoverers' and 'analysers' — using which the domain describer (scientist and/or engineer) can more-or-less systematically discover, analyse and describe a domain, informally and formally.

5.1. Preliminaries

- Before we present the discoverers and analysers we need establish some concepts. These are:
  - Part Signatures — Slides 231–232
  - Domain Indices — Slides 233–234
  - Inherited Part Signatures — Slides 235–236
  - Domain Signatures — Slides 237–240
  - Simple and Compound Domain Signatures — Slides 242–243

5.1.1. Part Signatures

- Let us consider a part $p : P$.
- P is, by definition, the principal entity of a domain.
  - Now we need to identify
    - the type (P) of that part,
    - the possible type of its unique identifier,
    - the possible types of its mereology,
    - the types of its attributes, and
    - the types, $Q_1, Q_2, \ldots, Q_m$, of its proper sub-parts (if p is composite).
- We shall name that cluster of type identifications
  - the *part signature*.
- We refer to P as identifying the part signature.
- Each of the Q_i (for $i : \{1..m\}$) identifies a sub-part and hence a sub-part, i.e., a part signature.

Example 41 (Net Domain and Sub-domain Part Signatures)

The part signature of net N is here chosen to be those of

- N,
- Net_name, etc.
- VS and
- LS.

The part signatures HS and LS are

- HS = H-**set**, H
- HI
- Hub_Nm, Location, HΣ, HΩ, ...
- LS = L-**set**, L
- LI
- Link_Nm, LΣ, LΩ, LEN, ...

•

5.1.2. Domain Indices

- By a domain index we mean a list of part type names that identifies as sequence of part signatures.
- More specifically
  - The domain Δ has index $\langle\Delta,\rangle$.
  - The sub-domains, with part types A, B, ..., C of Δ has indices $\langle\Delta,A\rangle$, $\langle\Delta,B\rangle$, ..., $\langle\Delta,C\rangle$.
  - The sub-domains of sub-domain with index ℓ and with part types A, B, ..., C has indices $\ell^\frown\langle A\rangle$, $\ell^\frown\langle B\rangle$, ..., $\ell^\frown\langle C\rangle$.

Example 42 (Some Indices of The Road-pricing Transport Domain)
The sub-domain indices of the road-pricing transport domain, Δ, are:

- $\langle\Delta\rangle$,
- $\langle\Delta,N\rangle$,
- $\langle\Delta,F\rangle$,
- $\langle\Delta,Monitor\rangle$,
- $\langle\Delta,N,VS\rangle$,
- $\langle\Delta,N,LS\rangle$,
- $\langle\Delta,N,H\rangle$,
- $\langle\Delta,N,L\rangle$ and
- $\langle\Delta,F,V\rangle$. •

5.1.3. Inherited Domain Signatures

-
-
-
-

Example 43 (Inherited Domain Signatures) Some example iherits are:

-
-
-
-

5.1.4. Domain and Sub-domain Categories

- By the domain category we shall mean
  - the domain signature

  and the
  - action,
  - event and
  - behaviour

  definitions whose signatures involves
  - just the types given in the domain signature or
  - in inherited domain signatures.

Example 44 (The Road-pricing Domain Category) The road-pricing domain category consist of

- the types N, F and *Monitor*,
- the *create_N*et *create_F*leet and *create_Monitor* actions, and
- corresponding Net, Fleet and *Monitor* behaviours

●

- By a sub-domain category, of index ℓ, we shall mean
  - the sub-domain types of the sub-domain designated by index ℓ,

  and the
  - actions,
  - events and
  - behaviours

  whose signatures involves just the types
  - of the ℓ indexed sub-domain
  - or of any prefix of ℓ indexed sub-domain
  - or of the root domain.

Example 45 (A Hub Category of a Road-pricing Transport Domain)
The ancestor sub-domain types of the hub sub-domain are: HS, N and Δ.

- The hub category thus includes
  - the part (etc.) types H, HI, ...,
  - the *insert_H*ub and the *delete_H*ub actions,
  - perhaps some *saturated_hub* (and/or other) event(s),
  - but probably no hub behaviour as it would involve at least the type LI which is not in an ancestor sub-domain of the Hub sub-domain.

●

5.1.5. Simple and Compound Indexes

- By a simple index we mean a domain or a sub-domain index.
- By a compound index we mean a set of two or more distinct indices of a domain Δ.
- Compound indices, $c_{idx} : \{\ell_i, \ell_j, \ldots, \ell_k\}$, designate parts, actions, events and behaviours each of whose types and signatures involve types defined by all of the simple indexes of c_{idx}.

Example 46 (Compound Indices of the Road-pricing System)
We show just one compound index:

- $\{\langle \Delta, N, HS, H \rangle, \langle \Delta, N, LS, L \rangle\}$.

●

5.1.6. Simple and Compound Domain Categories

- By a simple domain category we shall mean any ℓ-indexed [sub-]domain category.
- By the compound domain category of compound index $c_{idx} : \{\ell_i, \ell_j, \ldots, \ell_k\}$, we shall mean
  - the set of types, actions, events and behaviours
  - as induced by compound index c_{idx}, that is,
  - parts, actions, events and behaviours
  - each of whose types and signatures involve types defined by all of the simple indexes of c_{idx}.

Example 47 (The Compound Domain Category of Hubs and Links)

The compound domain category designated by $\{\langle\Delta,N,HS,H\rangle,\langle\Delta,N,LS,L\rangle\}$ includes:

```
type
  HIs = HI-set
    axiom ∀ his:HIs·card his=2
  LIs = LI-set
  HΣ = (LI×LI)-set
  LΣ = (HI×HI)-set
  HΩ = HΣ-set
  LΩ = LΣ-set
value
  mereo_L: L → HIs,
  mereo_H: H → LIs
  attr_HΣ: H → HΣ
  attr_LΣ: L → LΣ
  attr_HΩ: H → HΩ
  attr_LΩ: L → LΩ
axiom
  ∀ h:H·attr_HΣ(h)⊆attr_HΩ(h)
  ∀ l:L·attr_LΣ(l)⊆attr_LΩ(l)
```

●

5.1.7. Examples

- We repeat some examples, but now "formalised".

Example 48 (The Root Domain Category $\Delta_{\langle\Delta\rangle}$) When observing the very essence of the transport domain "at the $\Delta_{\langle\Delta\rangle}$ level" one observes:

type
  N, F, Monitor
value
  obs_N: $\Delta \to$ N
  obs_F: $\Delta \to$ F
  obs_Monitor: $\Delta \to$ Monitor

●

Example 49 (The Net Domain Category $\Delta_{\langle \Delta, N \rangle}$) When observing the very essence of the Net domain, "at the $\Delta_{\langle \Delta, N \rangle}$ level" one observes:

type
  H
  HS = H-**set**
  L
  LS = L-**set**

value
  obs_HS: N → HS
  obs_LS: N → LS
  empty_N: **Unit** → N
  insert_H: N × H $\xrightarrow{\sim}$ H

●

Example 50 (The Fleet Domain Category $\Delta_{\langle \Delta, F \rangle}$) When observing the very essence of the Fleet domain, "at the $\Delta_{\langle \Delta, F \rangle}$ level" one observes:

type	**value**
V	obs_VS: F → VS
VS = V-**set**	attr_VI: V → VI, attr_... → ...

where ... stand for attributes that we may wish to associate with vehicles.

•

Example 51 (The Hub Domain Category $\Delta_{\langle \Delta,N,HS,H \rangle}$) When observing the very essence of the Fleet domain, "at the $\Delta_{\langle \Delta,N,HS,H \rangle}$ level" one observes:

```
type                          value
  HI                            uid_HI: H → HI, attr_...: H → ...
```

where ... stand for *LOC*ation, etc.

•

Example 52 (The Link Domain Category $\Delta_{\langle \Delta,N,LS,L \rangle}$) When observing the very essence of the Fleet domain, "at the $\Delta_{\langle \Delta,N,LS,L \rangle}$ level" one observes:

type	**value**
LI	uid_LI: L → LI, attr_...: L → ...

where ... stand for *LOC*ation, *LEN*gth, etc.

●

Example 53 (The Compound Hub and Link Domain Category $\Delta_{\{\langle\Delta,N,HS,H\rangle,\langle N,LS,L\rangle\}}$)
When observing the very essence of the Fleet domain, "at the $\Delta_{\{\langle\Delta,N,HS,H\rangle,\langle\Delta,N,LS,L\rangle\}}$ level" one observes:

type
  HΣ = (LI×LI)-**set**, HΩ = HΣ-**set**,
  LΣ = (HI×HI)-**set**, LΩ = LΣ-**set**
value
  attr_HΣ: H → HΣ, attr_HΩ: H → HΩ
  attr_LΣ: L → LΣ, attr_LΩ: L → LΩ
  mereo_L: L → HIs (= HI-**set**) **axiom** ∀ l:L:**card** mereo_L(l)=2
  mereo_H: H → LIs (= LI-**set**)
  remove_H: HI → N ~→ N
  insert_L: L → N ~→ N
  remove_L: LI → N ~→ N

•

5.1.8. Discussion

-
-
-
-

5.2. Proposed Type and Signature 'Discoverers'

- By a 'domain discoverer' we shall understand
  - a tool and a set of principles and techniques
  - for using this tool
  - in the discovery of the entities of a domain.
- In this section we shall put forward a set of type and signature discoverers.
  - Each discoverer is indexed by a simple or a compound domain or sub-domain index.
  - And each discoverer is dedicated to some aspect of some entities.
- Together the proposed discoverers should cover the most salient aspects of domains.
- Our presentation of type and signature discoverer does not claim to help analyse "all" of a domain.

type
  Index = Smpl_Idx | Cmpd_Idx
  Smpl_Idx = {| ⟨Δ⟩^idx | idx:Domain_Type* |}
  Cmpd_Idx′ = Smpl_Idx-**set**
  Cmpd_Idx = {| sis:Cmpd_Idx′ · wf_Cmpd_Idx(sis) |}
value
  wf_Cmpd_Idx: Cmpd_Idx′ → **Bool**
  wf_Cmpd_Idx(sis) ≡ ∀ si,si′:Smpl_Idx · {si,si′}⊆sis ∧ si≠si′

  $\mathcal{DISCOVER\_KIND}$: Index → **Text**
  $\mathcal{DISCOVER\_KIND}$(ℓ^⟨t⟩) **as** text
    pre: ℓ^⟨t⟩ is a valid index beginning with Δ
    post: text is some, in our case, RSL text

- The idea of the ℓ^⟨t⟩ index is that it identifies a sub-domain, t, of Δ

– where *DISCOVERER_KIND* is any of the several different "kinds" of domain forms:

[86 (Slide 257)] PART_SORTS,
[87 (Slide 259)] HAS_A_CONCRETE_TYPE,
[88 (Slide 262)] PART_TYPES,
[89 (Slide 266)] UNIQUE_ID,
[92 (Slide 268)] MEREOLOGY,
[92(a) (Slide 268)] ATTRIBUTES,
[96 (Slide 279)] ACTION_SIGNATURES,
[97 (Slide 280)] EVENT_SIGNATURES and
[99 (Slide 284)] BEHAVIOUR_SIGNATURES.

- In a domain analysis (i.e., discovery) the domain description emerges "bit-by-bit".
  - Initially types are discovered and hence texts which define
    - unique identifier types and functions,
    - mereology types and functions, and
    - attribute types and functions.
  - Then the signatures of actions, events and behaviours.

5.2.1. Analysing Domain Parts

- The two most important aspects of an algebra are those of
  - its parts and
  - its operations.
- Rather than identifying, that is, discovering or analysing individual parts
  - we focus on discovering their types —
  - initially by defining these as sorts.
- And rather than focusing on defining what the operations achieve
  - we concentrate on the signature, i.e., the types of the operations.

- It (therefore) seems wise to start with the discovery of parts, and hence of their types.
  - Part types are present in the signatures of all actions, events and behaviours.
  - When observing part types we also observe a variety of part type analysers:
    * possible unique identities of parts,
    * the possible mereologies of composite parts, and
    * the types of the attributes of these parts.

5.2.1.1. Domain Part Sorts and Their Observers

- Initially we "discover" parts —
  - by deciding upon their types,
  - in the form, first of sorts,
  - subsequently and possibly in the form of concrete types.

5.2.1.1.1. A Domain Sort Discoverer

86. A part type discoverer applies to a simply indexed domain, *index*, and yields
    (a) a set of type names
    (b) each paired with a part (sort) observer.

value

86. $\mathbb{PART\_SORTS}$: Index $\xrightarrow{\sim}$ **Text**
86. $\mathbb{PART\_SORTS}(\ell^\frown\langle \mathrm{T}\rangle)$:
86(a). tns:$\{\mathrm{T}_1,\mathrm{T}_2,...,\mathrm{T}_m\}$:TN-**set** $\times$
86(b). $\{$ obs_T_j: T $\rightarrow$ T_j | T_j:tns$\}$

Example 54 (Some Part Sort Discoveries) We apply a concrete version of the above sort discoverer to the road-pricing transport domain Δ:

$\mathbb{PART\_SORTS}(\langle\Delta\rangle)$:
  type
    N, F, Monitor
  value
    obs_N: $\Delta \rightarrow$ N
    obs_F: $\Delta \rightarrow$ F
    obs_Monitor: $\Delta \rightarrow$ Monitor

$\mathbb{PART\_SORTS}(\langle\Delta,\text{N}\rangle)$:
  type
    HS, LS
  value
    obs_HS: N $\rightarrow$ HS
    obs_LS: N $\rightarrow$ LS

●

5.2.1.2. Domain Part Types and Their Observers

5.2.1.2.1. Do a Sort Have a Concrete Type ?

- Sometimes we find it expedient
  - to endow a "discovered" sort with a concrete type expression, that is,
  - "turn" a sort definition into a concrete type definition.

87. Thus we introduce the "discoverer":

-

87 HAS_A_CONCRETE_TYPE: Index → **Bool**
87 HAS_A_CONCRETE_TYPE(ℓ⌢⟨t⟩):**true**|**false**

Example 55 (Some Type Definition Discoveries) We exemplify two true expressions:

$$\mathbb{HAS\_A\_CONCRETE\_TYPE}(\langle \Delta, \mathrm{N}, \mathrm{HS} \rangle)$$
$$\mathbb{HAS\_A\_CONCRETE\_TYPE}(\langle \Delta, \mathrm{N}, \mathrm{LS} \rangle)$$

●

5.2.1.2.2. A Domain Part Type Observer

- The $\mathbb{PART\_TYPES}(\ell^\frown\langle \mathsf{t} \rangle)$ invocation yields one or more sort definitions of part types together with their observer functions.
  - The domain analyser can decide that some parts can be immediately analysed into concrete types.
  - Thus, together with yielding a type name, the $\mathbb{PART\_TYPES}$ can be expected to yield also a type definition, that is, a type expression (paired with the type name).
  - Not all type expressions make sense.
  - We suggest that only some make sense.

88. The $\mathbb{PART\_TYPES}$ discoverer applies to a composite type, t, and yields
    (a) a type definition, $\mathsf{T} = \mathsf{TE}$,
    (b) together with the sort and/or type definitions of so far undefined type names of TE.
    (c) The $\mathbb{PART\_TYPES}$ discoverer is not defined if the designated sort is judged to not warrant a concrete type definition.

88. $\mathbb{PART\_TYPES}$: Index $\xrightarrow{\sim}$ **Text**
88. $\mathbb{PART\_TYPES}(\ell^\frown\langle \mathrm{t} \rangle)$:
88(a). **type** t = te,
88(b). T_1 or $\mathrm{T}_1 = \mathrm{TE}_1$
88(b). T_2 or $\mathrm{T}_2 = \mathrm{TE}_2$
88(b). ...
88(b). T_n or $\mathrm{T}_n = \mathrm{TE}_n$
88(c). **pre**: $\mathbb{HAS\_A\_CONCRETE\_TYPE}(\ell^\frown\langle \mathrm{t} \rangle)$

Example 56 (Some Part Type Discoveries) We exemplify two discoveries:

$\mathbb{PART\_TYPES}(\langle \Delta, \mathrm{N}, \mathrm{HS} \rangle)$:
  type
    H, HS = H-**set**

$\mathbb{PART\_TYPES}(\langle \Delta, \mathrm{N}, \mathrm{LS} \rangle)$:
  type
    L, LS = L-**set**

●

5.2.1.2.3. Concrete Part Types

- In Example 56 on the previous page we illustrated one kind of concrete part type: sets.
- Practice shows that sorts often can be analysed into sets.
- Other analyses of part sorts are
  - Cartesians,
  - list, and
  - simple maps:

86(b). te: tn1 × tn2 × ... × tnm
86(b). te: tn*
86(b). te: Token $\xrightarrow[m]{}$ tn

- where
  - tn's are part type – usually sort – names
  - some of which may have already been defined,
  - and where Token is some simple atomic (non-part) type.

5.2.1.3. Part Type Analysers

- There are three kinds of analysers:
  - *unique identity*,
  - *mereology analysers* and
  - *general attribute analysers*. and

5.2.1.3.1. Unique Identity Analysers

- We associate with every part type T, a unique identity type TI.

89. So, for every part type T we postulate a unique identity analyser function uid_TI.

```
value
89. UNIQUE_ID: Index → Text
89. UNIQUE_ID(ℓ^⟨T⟩):
89.     type
89.         TI
89.     value
89.         uid_TI: T → TI
```

5.2.1.3.2. Mereology Analysers

- Give a part, p, of type T, the mereology, MEREOLOGY, of that part
  - is the set of all the unique identifiers
    of the other parts to which part p is partship-related
  - as "revealed" by the mereo_TI$_i$ functions applied to p.
-
- "cuts across" all indices
-

90. Let types T_1, T_2, ..., T_n be the types of all parts of a domain.
91. Let types TI_1, TI_2, ..., TI_m, $m \leq n$, be the types of the unique identifiers of all parts of that domain.
92. The mereology analyser MEREOLOGY is a generic function which applies to an index and yields the set of
    (a) zero,
    (b) one or
    (c) more

    mereology observers.

type

90. $\mathrm{T} = \mathrm{T}_1 \mid \mathrm{T}_2 \mid ... \mid \mathrm{T}_n$
91. $\mathrm{T}_{idx} = \mathrm{TI}_1 \mid \mathrm{TI}_2 \mid ... \mid \mathrm{TI}_m$ (m≤n)
92. $\mathbb{MEREOLOGY}$: Index → **Text**
92. $\mathbb{MEREOLOGY}(\ell^\frown\langle \mathrm{T} \rangle)$:
92(a). **either:** {}
92(b). **or:** $\mathrm{mereo\_TI}_x$: $\mathrm{T} \to (\mathrm{TI}_x | \mathrm{TI}_x$**-set**$)$
92(c). **or:** { $\mathrm{mereo\_TI}_x$: $\mathrm{T} \to (\mathrm{TI}_x | \mathrm{TI}_x$**-set**$)$,
92(c). $\mathrm{mereo\_TI}_y$: $\mathrm{T} \to (\mathrm{TI}_y | \mathrm{TI}_y$**-set**$)$,
92(c). ...,
92(c). $\mathrm{mereo\_TI}_z$: $\mathrm{T} \to (\mathrm{TI}_z | \mathrm{TI}_z$**-set**$)$ }

- where none of TI_x, TI_y, ..., TI_z are equal to TI and each is some T_{idx}.

-
- MORE TO COME
-

5.2.1.3.3. General Attribute Analysers

- A general attribute analyser analyses parts beyond their unique identities and possible mereologies.

93. Part attributes have names. We consider these names to also abstractly name the corresponding attribute types, that is, the names function both as attribite names and sort names. Finally we allow attributes of two or more otherwise distinct part types to be the same.

94. ATTRIBUTES applies to parts of any part type **t** and yields

95. the set of attribute observer functions **attr_at**, one for each attribute sort **at** of **t**.

```
type
93.  AT = AT_1 | AT_2 | ... | AT_n
value
94.  ATTRIBUTES: Index → Text
94.  ATTRIBUTES(ℓ^⟨T⟩):
95.      type
95.          AT_1, AT_2, ..., AT_m
95.      value
95.          attr_AT_1: T → AT_1
95.          attr_AT_2: T → AT_2
95.          ...,
95.          attr_AT_m: T → AT_m, m≤n
```

Example 57 (Example Part Attributes) We exemplify attributes of composite and of atomic parts:

$\mathbb{ATTRIBUTES}(\langle\Delta\rangle)$:
  type
    Domain_Name, ...
  value
    attr_Name: $\Delta \to$ Domain_Name
    ...

- where
  - Domain_Name could include *State Roads* or *Rail Net*.
  - etcetera.

```
ATTRIBUTES(⟨Δ,N⟩):
    type
        Sub_Domain_Location, Sub_Domain_Owner, Kms, ...
    value
        attr_Location: N → Sub_Domain_Location
        attr_Owner: N → Sub−Domain_Owner
        attr_Length: N → Kms
        ...
```

- where
  - Sub_Domain_Location could include *Denmark*,
  - Sub_Domain_Owner could include *The Danish Road Directorate*[34], respectively *BaneDanmark*[35],
  - etcetera.

[34] http://www.vejdirektoratet.dk/roaddirectorate.asp?page=dept&objno=1024
[35] http://uk.bane.dk/default_eng.asp?artikelID=931

ATTRIBUTES(⟨Δ,N,HS,L⟩):
  type
    LOC, LEN, LΣ, LΩ, ...
  value
    attr_LOC: L → LOC
    attr_LEN: L → LEN
    attr_LΣ: L → LΣ
    attr_LΩ: L → LΩ

- where ···
  - LOC might reveal some Bezier curve[36] representation of the possibly curved three dimensional location of the link in question,
  - LEN might designate length in meters,
  - LΣ designates the state of the link (which directions are open),
  - LΩ designates the state space (i.e., set of all allowed state of the link), etcetera.

●

[36] http://en.wikipedia.org/wiki/Bézier_curve

⋆ *Attribute Sort Exploration* ⋆

- Once the attribute sorts of a part type have been determined
  - there remains to be "discovered" the concrete types of these sorts.
  - We omit treatment of this point in the present version of these these lectures.

5.2.2. Discovering Action Signatures
5.2.2.1. General

- We really should discover actions, but actually analyse function definitions.
- And we focus, in these lectures, on just "discovering" the function signatures of these actions.
- By a function signature, to repeat, we understand
  - a functions name, say **fct**, and
  - a function type expression (**te**), say **dte**$\xrightarrow{\sim}$**rte** where
    - **dte** defines the type of the function's **d**efinition set
    - and **rte** defines the type of the function's image, or **r**ange set.

5.2.2.2. Function Signatures Usually Depend on Compound Domains

- We use the term 'functions' to cover actions, events and behaviours.
- We shall in general find that the signatures of actions, events and behaviours depend on types of more than one domain.
  - Hence the schematic index set $\{\ell_1\widehat{\ }\langle \mathsf{t}_1\rangle,\ell_2\widehat{\ }\langle \mathsf{t}_2\rangle,...,\ell_n\widehat{\ }\langle \mathsf{t}_n\rangle\}$
  - is used in all actions, events and behaviours discoverers.

5.2.2.3. The ACTION_SIGNATURES Discoverer

96. The ACTION_SIGNATURES meta-function applies to an index set and yields
    (a) a set of action signatures each consisting of an action name and a pair of definition set and range type expressions where
    (b) the type names that occur in these type expressions are defined by in the domains indexed by the index set.

96 ACTION_SIGNATURES: Index $\xrightarrow{\sim}$ **Text**
96 ACTION_SIGNATURES($\{\ell_1\widehat{\ }\langle \mathrm{T}_1\rangle,\ell_2\widehat{\ }\langle \mathrm{T}_2\rangle,...,\ell_n\widehat{\ }\langle \mathrm{T}_n\rangle\}$):
96(a) act_fct$_i$: te$_{i_d}$ [37] $\xrightarrow{\sim}$ te$_{i_r}$,
96(a) act_fct$_j$: te$_{j_d}$ $\xrightarrow{\sim}$ te$_{j_r}$,
96(a) ... ,
96(a) act_fct$_k$: te$_{k_d}$ $\xrightarrow{\sim}$ te$_{k_r}$
96(b) **where:**
96(b) type names in (te$_{(i|j|...|k)_d}$) and in (te$_{(i|j|...|k)_r}$) are
96(b) type names defined by the indices which are prefixes of
96(b) $\ell_m\widehat{\ }\langle T_m\rangle$ and where T_m is in some signature **act_fct**$_{i|j|...|k}$.

[37] The sub-sub-scripts d and r designate $d$efinition set, respectively $r$ange.

5.2.3. Discovering Event Signature

- Events are from the point of view of signatures very much like actions.

97. The EVENT_SIGNATURES meta-function applies to an index set and yields
   (a) a set of action signatures each consisting of an action name and a pair of definition set and range type expressions where
   (b) the type names that occur in these type expressions are defined either by in the domains indexed by the index set or by the environment (i.e., "outside" the domain Δ).

97 EVENT_SIGNATURES: Index $\xrightarrow{\sim}$ **Text**
97 EVENT_SIGNATURES($\{\ell_1\hat{}\langle \mathrm{T}_1\rangle,\ell_2\hat{}\langle \mathrm{T}_2\rangle,...,\ell_n\hat{}\langle \mathrm{T}_n\rangle\}$):
97(a) evt_fct$_i$: te$_{i_d}$ $\xrightarrow{\sim}$ te$_{i_r}$,
97(a) evt_fct$_j$: te$_{j_d}$ $\xrightarrow{\sim}$ te$_{j_r}$,
97(a) ... ,
97(a) evt_fct$_k$: te$_{k_d}$ $\xrightarrow{\sim}$ te$_{k_r}$
97(b) **where:**
97(b) type names of $\mathsf{te}_{(i|j|...|k)_d}$ and $\mathsf{te}_{(i|j|...|k)_r}$ are type names
97(b) defined by the indices which are prefixes of $\ell_m\hat{}\langle t_m\rangle$
97(b) and where t_m is in some signature $\mathsf{act\_fct}_{i|j|...|k}$ or may
97(b) refer to types definable only "outside" Δ

5.2.4. Discovering Behaviour Signatures

- We choose, in these lectures, to model behaviours in CSP[38].
- This means that we model (synchronisation and) communication between behaviours by means of message-typed, m:M, CSP channels (**channel** ch:M) and CSP

  output: ch!e [offer to deliver value of expression e on channel ch], and
  input: ch? [offer to accept value of expression e on channel ch].

- We allow for the declaration of single channels as well as of one, two, ..., n dimensional arrays of channels with indexes ranging over channel index types

  type Idx, CIdx, RIdx ...:
  channel ch:M, { ch_v[vi]:M′|vi:Idx }, { ch_m[ci,ri]:M″|ci:CIdx,ri:RIdx }, ...

  etcetera.

- We assume some familiarity with RSL/CSP.

[38] Other behaviour modelling languages are Petri Nets, MSCs: Message Sequence Charts, Statechart etc.

- A behaviour usually involves two or more distinct sub-domains.

Example 58 (The Involved Subdomains of a Vehicle Behaviour)
Let us illustrate that behaviours usually involve two or more distinct sub-domains.

- A vehicle behaviour, for example, involves
  - the vehicle subdomain,
  - the hub subdomain (as vehicles pass through hubs),
  - the link subdomain (as vehicles pass along links) and,
  - for the road pricing system, also the monitor subdomain.

●

98. The BEHAVIOUR_SIGNATURES is a meta function.
99. It applies to a set of indices and results in a text,
100. The text contains
    (a) a set of zero, one or more message types,
    (b) a set of zero, one or more channel index types,
    (c) a set of zero, one or more channel declarations, and
    (d) 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
    (e) an input/output clause which refers to channels over which the signatured behaviour may interact with its environment.

99. BEHAVIOUR_SIGNATURES: Index $\xrightarrow{\sim}$ **Text**
99. BEHAVIOUR_SIGNATURES($\{\ell_1\widehat{\ }\langle T_1\rangle,\ell_2\widehat{\ }\langle T_2\rangle,...,\ell_n\widehat{\ }\langle T_n\rangle\}$):
100(a). **type** $M = M_1 \mid M_2 \mid ... \mid M_m$, $m \geq 0$
100(b). $I = I_1 \mid I_2 \mid ... \mid I_n$, $n \geq 0$
100(c). **channel** ch,vch[i],{vch[i]:M|i:I_a},{mch[j,k]:M|j:I_b,k:I_c},...
100(d). **value**
100(d). bhv$_1$: ate$_1$ $\rightarrow$ *inout*$_1$ rte$_1$,
100(d). bhv$_2$: ate$_2$ $\rightarrow$ *inout*$_2$ rte$_2$,
100(d). ... ,
100(d). bhv$_m$: ate$_m$ $\rightarrow$ *inout*$_m$ rte$_m$,
100(d). **where** type expressions **atei**$_i$ and **rte**$_i$ for all **i** involve at least two
100(d). types t'_i and t''_j of respective indexes $\ell_i\widehat{\ }\langle t_i\rangle$ and $\ell_j\widehat{\ }\langle t_j\rangle$
100(e). **where** *inout*$_i$: **in** k | **out** k | **in**,**out** k
100(e). **where** k: ch | ch[i] | {ch[i]|i $\in$ I_a} | {mch[j,k]:M|i:I_b,j:I_c} | ...

Example 59 (A Vehicle Behaviour Signature Discovery) We refer, for example, to Examples 36 (Slides 192–206) and 40 (Slides 223–227).

let ih=⟨Δ.N.LS,H⟩,il=⟨Δ.N.HS,L⟩,iv=⟨Δ,F,V⟩,im=⟨Δ,Monitor⟩ **in**
𝔹𝔼ℍ𝔸𝕍𝕀𝕆𝕌ℝ_𝕊𝕀ℊℕ𝔸𝕋𝕌ℝ𝔼𝕊({iv,ih,iv,im}) **as** text
  let n:N, hs=obs_HS(n), ls=obs_LS(n), vs=obs_F(ℙ𝔸ℝ𝕋_𝕊𝕆ℝ𝕋𝕊)(⟨Δ⟩) **in**
  where text:
    type
      VL_Msg, VH_Msg, VM_Msg
    channel
58(a). {vh_ch[attr_VI(v),attr_HI(h)]|v:V,h:H•v ∈ vd∧h ∈ hs}:VH_Msg
58(b). {vl_ch[attr_VI(v),attr_LI(h)]|v:V,l:L•v ∈ vs∧h ∈ ls}:VL_Msg
58(c). m_ch:VM_Msg
    value
60. vehicle: VI → V → VP →
79. **out**,**in** {vl_ch[vi,li]|li:LI•li ∈ xtr_LIs(ls)}
79. {vh_ch[vi,hi]|hi:HI•hi ∈ xtr_HIs(hs)} **out** m_ch,... **Unit**
end end

●

5.3. What Does Application Mean ?

- Now what does it actually mean "to apply" a discover function ?
- We repeat our list of discoverers.

  [86 (Slide 257)] PART_SORTS,
  [87 (Slide 259)] HAS_A_CONCRETE_TYPE,
  [88 (Slide 262)] PART_TYPES,
  [89 (Slide 266)] UNIQUE_ID,
  [92 (Slide 268)] MEREOLOGY,
  [92(a) (Slide 268)] ATTRIBUTES,
  [96 (Slide 279)] ACTION_SIGNATURES,
  [97 (Slide 280)] EVENT_SIGNATURES and
  [99 (Slide 284)] BEHAVIOUR_SIGNATURES.

- It is the domain engineer cum scientist who "issues" the "commands".

- The first "formal" domain inquiry is that of $\mathbb{PART\_SORTS}(\langle\Delta\rangle)$.
  - We refer to Item 86 on page 257, for example as captured by the formulas, Items 86–86(b) (Slide 289).
- For the domain engineer to 'issue' one of the 'discovery commands' means that that person
  - has prepared his mind to study the domain and is open to impressions,
  - has decided which $\mathcal{DISCOVERER\_KIND}$ to focus on, and
  - has studied the "rules of engagement" of that command, that is
    - which pre-requisite discoverers must first have been applied,
    - with which index, that is, in which context the command invocation should be placed,
    - and which results the invocation is generally expected to yield.

5.3.1. PART_SORTS

- Let us review the PART_SORTS discoverer:

value

86. PART_SORTS: Index $\xrightarrow{\sim}$ **Text**
86. PART_SORTS($\ell^\frown\langle$T$\rangle$):
86(a). tns:{T_1,T_2,...,T_m}:TN-**set** $\times$
86(b). { obs_T_j: T $\rightarrow$ T_j | T_j:tns}

- The domain analyser has decided to "position" the search at domain index $\ell^\frown\langle$T$\rangle$
  - where T $=\Delta$ if $\ell = \langle\rangle$ and
  - where T is some "previously discovered part type.

- From Item 86(a) the domain analyser is guided (i.e., advised) to analyse the domain "at position $\ell^\frown\langle \mathsf{T}\rangle$:
  - is the domain type T a type of one or more subpart types ?
    - If so then decide which they are, that is: $\mathsf{T}_1,\mathsf{T}_2,...,\mathsf{T}_m$, that is, the "generation" of the text **type** $\mathsf{T}_1,\mathsf{T}_2,...,\mathsf{T}_m$,
    - if not then $\mathsf{tns}=\{\}$ and no text is "generated".
- Item 86(b), and given the domain analyser's resolution of Item 86(a), then directs
  - the "generation" of m observers $\mathsf{obs\_T}_j$: $\mathsf{T} \to \mathsf{T}_j$ (for $j : \{1..m\}$).

- The decision as to which subpart types to choose is $X\%$ "art" where X is typically 60-80 !.
  - Well, in general 'abstraction is an art' !
  - One can learn a number of abstraction principles.
  - Mostly these abstraction principles can be "taught" in two steps.
    - In a first step,
      ***
    - In a second step,
      ***

5.3.2. HAS_A_CONCRETE_TYPE

- Let us review the HAS_A_CONCRETE_TYPE analyser:

87 HAS_A_CONCRETE_TYPE: Index → **Bool**
87 HAS_A_CONCRETE_TYPE($\ell^\frown\langle$T$\rangle$):**true**|**false**

- Item 87 directs the domain analyser to decide
  - whether the domain type T at "position" $\ell^\frown\langle$t$\rangle$
  - should be given a concrete type definition.
- It is a decision sôlely at the discretion of the domain analyser
  - whether domain type T should be given a concrete type definition,
  - and which concrete type it should then be "given",
  - that is, how it should be "concretely abstractly" modelled.

5.3.3. PART_TYPES

- Let us review the PART_TYPES analyser:

88. PART_TYPES: Index $\xrightarrow{\sim}$ **Text**
88. PART_TYPES(ℓ^⟨t⟩):
88(a). **type** T = TE,
88(b). T_1 or T_1 = TE_1
88(b). T_2 or T_2 = TE_2
88(b). ...
88(b). T_n or T_n = TE_n
88(c). **pre**: HAS_A_CONCRETE_TYPE(ℓ^⟨t⟩)

- The domain analyser has decided to "position" the search at domain index $\ell^\frown\langle \mathsf{T} \rangle$
  - where $\mathsf{T} = \Delta$ if $\ell = \langle\rangle$ and
  - where T is some "previously discovered part type.
- From Item 88 the domain analyser is guided (i.e., advised) to analyse the domain "at position $\ell^\frown\langle \mathsf{T} \rangle$:
  - can an abstract, yet concrete type definion be given for T ?
  - If so then decide which it should be, that is, should it be an atomic type
    - a number type, **Intg, Rat, Real**,
    - a Boolean type, **Bool**, or
    - a token type, f.ex. TOKEN;

  - or should it be a composite type
    * either a set type: TE: T_s**-set** of te: T_s**-infset**,
    * or a Cartesian type: TE: $\mathsf{T}_1 \times \mathsf{T}_2 \times ... \mathsf{T}_m$,
    * or a list type: TE: $\mathsf{T}^\star_\ell$ or te: T^ω_ℓ
    * or a map type: TE: $\mathsf{T}_d \xrightarrow[m]{} \mathsf{T}_t$?

- In either case the text
  - type T = TE,
  - T_1 or T_1=TE_1,
  - T_2 or T_1=TE_2,
  - ...,
  - T_n or T_n=TE_n

  is generated
- where TE (TE_x) is a type expression whose
- so far undefined type names T_1, T_2, ..., T_n must be defined, either as sorts, or a concrete types.

- The decision as to which concrete types to choose is $Y\%$ "art" where Y is typically 60-80 !
  - Well, in general 'abstraction is an art' !
  - One can learn a number of abstraction principles.
  - Mostly these abstraction principles can be "taught" in two steps.
    * In a first step,
      "this teaching" is done by showing a number of examples.
      .
      .
      .
    * In a second step ***
    *

5.3.4. UNIQUE_ID

- Let us review the UNIQUE_ID analyser:

value
89. UNIQUE_ID: Index → **Text**
89.a UNIQUE_ID(ℓ⌢⟨T⟩):
89.b **type**
89.c TI
89.d **value**
89.e uid_TI: T → TI

- Item 89.a inquires as to the
- Line 89.b **type** name
- Line 89.c of the inquired part type's unique identifiers
- Line 89.d and the function signature **value**
- Line 89.e of
  - the observer, uid_TI, name,
  - the definition set type (T, of course) and
  - the range set type (TI — obviously).
  - Thus, the only real "new"
  - "discovery" here is the name,
  - TI, of the unique identifier type.

5.3.5. MEREOLOGY

- Let us review the MEREOLOGY analyser (of that last page reference):

type
90. $\mathrm{T}_k = \mathrm{T}_1 \mid \mathrm{T}_2 \mid ... \mid \mathrm{T}_n$
91. $\mathrm{T}_{idx} = \mathrm{TI}_1 \mid \mathrm{TI}_2 \mid ... \mid \mathrm{TI}_m$ (m≤n)
92. MEREOLOGY: Index → **Text**
92. MEREOLOGY($\ell^\frown\langle \mathrm{T}\rangle$):
92(a). **either:** {}
92(b). **or:** mereo_TI_x: T → (TI_x|TI_x-**set**)
92(c). **or:** { mereo_TI_x: T → (TI_x|TI_x-**set**),
92(c). mereo_TI_y: T → (TI_y|TI_y-**set**),
92(c). ...,
92(c). mereo_TI_z: T → (TI_z|TI_z-**set**) }

- where none of TI_x, TI_y, ..., TI_z are equal to TI and each is some T_{idx}.

5.3.6. ATTRIBUTES

type

93. $\mathrm{AT} = \mathrm{AT}_1 \mid \mathrm{AT}_2 \mid ... \mid \mathrm{AT}_n$

value

92(a). $\mathbb{ATTRIBUTES}$: Index → **Text**

92(a). $\mathbb{ATTRIBUTES}(\ell^\frown\langle \mathrm{T}\rangle)$:

92(b). **type**

92(b). $\mathrm{AT}_1, \mathrm{AT}_2, ..., \mathrm{AT}_m$

92(b). **value**

92(b). $\mathrm{attr\_AT}_1$: $\mathrm{T} \to \mathrm{AT}_1$

92(b). $\mathrm{attr\_AT}_2$: $\mathrm{T} \to \mathrm{AT}_2$

92(b). ...,

92(b). $\mathrm{attr\_AT}_m$: $\mathrm{T} \to \mathrm{AT}_m$, $\mathrm{m} \leq \mathrm{n}$

5.3.7. ACTION_SIGNATURES

96 ACTION_SIGNATURES: Index $\xrightarrow{\sim}$ **Text**

96 ACTION_SIGNATURES($\{\ell_1\hat{}\langle \mathrm{T}_1\rangle,\ell_2\hat{}\langle \mathrm{T}_2\rangle,...,\ell_n\hat{}\langle \mathrm{T}_n\rangle\}$):

96(a) $\quad \mathrm{act\_fct}_i$: te_{i_d} $\overset{39}{\xrightarrow{\sim}}$ te_{i_r},

96(a) $\quad \mathrm{act\_fct}_j$: te_{j_d} $\xrightarrow{\sim}$ te_{j_r},

96(a) $\quad$ ... ,

96(a) $\quad \mathrm{act\_fct}_k$: te_{k_d} $\xrightarrow{\sim}$ te_{k_r}

96(b) **where:**

96(b) $\quad$ type names in $(\mathrm{te}_{(i|j|...|k)_d})$ and in $(\mathrm{te}_{(i|j|...|k)_r})$ are

96(b) $\quad$ type names defined by the indices which are prefixes of

96(b) $\quad \ell_m\hat{}\langle T_m\rangle$ and where T_m is in some signature $\mathsf{act\_fct}_{i|j|...|k}$.

5.3.8. EVENT_SIGNATURES

97 EVENT_SIGNATURES: Index $\xrightarrow{\sim}$ **Text**

97 EVENT_SIGNATURES($\{\ell_1\hat{}\langle T_1\rangle,\ell_2\hat{}\langle T_2\rangle,...,\ell_n\hat{}\langle T_n\rangle\}$):

97(a) $\quad$ evt_fct$_i$: te$_{i_d} \xrightarrow{\sim}$ te$_{i_r}$,

97(a) $\quad$ evt_fct$_j$: te$_{j_d} \xrightarrow{\sim}$ te$_{j_r}$,

97(a) $\quad$ ... ,

97(a) $\quad$ evt_fct$_k$: te$_{k_d} \xrightarrow{\sim}$ te$_{k_r}$

5.3.9. BEHAVIOUR_SIGNATURES

99. BEHAVIOUR_SIGNATURES: Index $\xrightarrow{\sim}$ **Text**
99. BEHAVIOUR_SIGNATURES($\{\ell_1\hat{}\langle T_1\rangle,\ell_2\hat{}\langle T_2\rangle,...,\ell_n\hat{}\langle T_n\rangle\}$):
100(a). **type** M = M_1 | M $_2$| ... | M_m, m≥0
100(b). I = I_1 | I_2 | ... | I_n, n≥0
100(c). **channel** ch,vch[i],{vch[i]:M|i:I_a},{mch[j,k]:M|j:I_b,k:I_c},...
100(d). **value**
100(d). bhv_1: ate_1 → *inout*$_1$ rte_1,
100(d). bhv_2: ate_2 → *inout*$_2$ rte_2,
100(d). ... ,
100(d). bhv_m: ate_m → *inout*$_m$ rte_m,

5.4. ***

5.5. Discussion

-
-
-

End Lecture 6: Discovering Domain Entities

Lecture 7: Conclusion

6. Conclusion

6.1. General

6.2. What Have We Achieved ?

6.3. Other Formal Models

6.4. Research Issues

6.5. Engineering Issues

6.6. Comparable Work

6.7. Acknowledgements

End Lecture 7: Conclusion