1. Agenda

Manifest Domains

Analysis & Description

Dines Bjørner

Institute of Software, CAS, Beijing

Monday–Wednesday November 16–19, 2017

http://www.imm.dtu.dk/~dibj/2017/summer/peking-slides.pdf

Compiled: October 9, 2017: 15:51

Lecture 1 — 1–77

1. Summary

- We show that manifest domains,
  - an understanding of which are
  - a prerequisite for software requirements prescriptions,

  can be precisely described:
  - narrated and
  - formalised.

- We show that such manifest domains can be understood as a collection of
  - **endurant**, that is, basically spatial entities:
    - **part**s,
    - **component**s and
    - **material**s,

    and
  - **perdurant,** that is, basically temporal entities:
    - **action**s,
    - **event**s and [We shall skip treatment of events.]
    - **behaviour**s.

- We show that parts can be modeled in terms of
  - **external qualities** whether:
    - **atomic** or
    - **composite**

    parts,
- having **internal qualities**:
  - **unique identifications**,
  - **mereologies**, which model relations between parts, and
  - **attributes**.

- We show that the manifest domain analysis endeavour can be supported by a calculus of manifest domain analysis prompts:
  - is_entity,
  - is_endurant,
  - is_perdurant,
  - is_part,
  - is_component,
  - is_material,
  - is_atomic,
  - is_composite,
  - has_components,
  - has_materials,
  - has_concrete_type,
  - attribute_names,
  - is_stationary, etcetera;

- and show how the manifest domain description endeavour can be supported by a calculus of manifest domain description prompts:
  - `observe_part_sorts`,
  - `observe_part_type`,
  - `observe_components`,
  - `observe_materials`,
  - `observe_unique_identifier`,
  - `observe_mereology`,
  - `observe_attributes`.

- We show how to model attributes, essentially following Michael Jackson, [Jac95]:
  - The attribute model introduces the attribute analysis prompts
    - is_static_attribute,
    - is_dynamic_attribute,
    - is_inert_attribute,
    - is_reactive_attribute,
    - is_active_attribute,
    - is_autonomous_attribute,
    - is_biddable_attribute and
    - is_programmable_attribute.

- We show how to model essential aspects of perdurants in terms of their signatures based on the concepts of endurants.
- And we show how one can "compile"
  - descriptions of endurant parts into
  - descriptions of perdurant behaviours.
- We do not show prompt calculi for perdurants.
- The above contributions express a method
  - with principles, techniques and tools
  - for constructing domain descriptions.
- It is important to realise that we do not wish to nor claim that the method can describe all that it is interesting to know about domains.

1. Introduction

- The broader subject of this paper is that of software development.
- The narrower subject is that of manifest domain engineering.
- We shall see software development
  in the context of the `TripTych` approach (next section).

- The contribution of these lectures are twofold:
  - the propagation of manifest domain engineering
    - as a first phase of the development of
    - a large class of software —

  and
    - a set of principles, techniques and tools
    - for the engineering of the analysis & descriptions
    - of manifest domains.

- These principles, techniques and tools
  are embodied in a set of analysis and description prompts.
  - We claim that this embodiment
  - — in the form of prompts —
  - is novel.

1.1. The TripTych Approach to Software Engineering

- We suggest a TripTych view of software engineering:
  - *before hardware and software systems can be designed and coded*
  - *we must have a reasonable grasp of "its" requirements;*
  - *before requirements can be prescribed*
  - *we must have a reasonable grasp of "the underlying" domain.*

- To us, therefore, software engineering contains the three sub-disciplines:
  - domain engineering,
  - requirements engineering and
  - software design.

- This seminar contributes, we claim, to a methodology for **domain analysis** &[1] **domain description**.
- Reference [Bjø16c]
  - show how to "refine" **domain descriptions** into **requirements prescriptions**,

  and reference [Bjø16b]
  - indicates more general relations between **domain descriptions** and
    - **domain demos**,
    - **domain simulators** and
    - more general domain specific software.

[1]When, as here, we write *A* & *B* we mean *A* & *B* to be one subject.

- The concept of **systems engineering** arises naturally in the TripTych approach.
  - First: *domains can be claimed to be systems.*
  - Secondly: *requirements are usually not restricted to software, but encompasses all the human and technological "assists" that must be considered.*
  - Other than that we do not wish to consider domain analysis & description principles, techniques and tools specific to "systems engineering".

1.2. Method and Methodology

1.2.1. Method

- By a **method** we shall understand
  - a “structured” set of `principles`
  - for `selecting` and `applying`
  - a number of `techniques` and `tools`
  - for `analysing` problems and `synthesizing` solutions
  - for a given domain ■[2]

[2]Definitions and examples are delimited by ■ symbols.

- The 'structuring' amounts,
  - in this treatise on domain analysis & description,
  - to the techniques and tools being related to a set of
  - domain analysis & description "prompts",
  - "issued by the method",
  - prompting the domain engineer,
  - hence carried out by the **domain analyser** & describer[3] —
  - conditional upon the result of other prompts.

[3] We shall thus use the term **domain engineer** to cover both the analyser & the describer.

1.2.2. Discussion

- There may be other ‘definitions’ of the term ‘method’.
- The above is the one that will be adhered to in this seminar.
- The main idea is that
  - there is a clear understanding of what we mean by, as here,
    - a software development method,
    - in particular a *domain analysis & description method.*

- The **main principles** of the `TripTych` domain analysis and description approach are those of
  - abstraction and both
    - narrative and
    - formal
  - modeling.
  - This means that evolving domain descriptions
    - necessarily limit themselves to a subset of the domain
    - focusing on what is considered relevant, that is,
    - abstract “away” some domain phenomena.

- The **main techniques** of the `TripTych` domain analysis and description approach are
  - besides those techniques which are in general associated with formal descriptions,
  - focus on the techniques that relate to the deployment of of the individual prompts.

- And the **main tools** of the `TripTych` domain analysis and description approach are
  - the analysis and description prompts and the
  - description language, here the Raise Specification Language RSL.

- A main contribution of this seminar is therefore
  - that of "painstakingly" elucidating the
    - principles,
    - techniques and
    - tools

    of the domain analysis & description method.

1.2.3. Methodology

- By **methodology** we shall understand
  - the study and knowledge
  - about one or more methods[4] ■

[4]Please note our distinction between method and methodology. We often find the two, to us, separate terms used interchangeably.

1.3. Computer and Computing Science

- By **computer science** we shall understand
  - the study and knowledge of
    - the conceptual phenomena
    - that "exists" inside computers
  - and, in a wider context than just computers and computing,
    - of the theories "behind" their
    - formal description languages ■
- Computer science is often also referred to as theoretical computer science.

- By **computing science** we shall understand
  - the study and knowledge of
    - how to construct
    - and describe

    those phenomena ■
- Another term for computing science is programming methodology.

- These lectures are about computing science.
  - They are concerned with the construction of domain descriptions.
  - They put forward a calculus for analysing and describing domains.
  - They do not theorize about this calculus.
  - There are no theorems about this calculus and hence no proofs.
  - We leave that to another study and paper.

1.4. What Is a Manifest Domain ?

- By ‘domain’ we mean the same as ‘problem domain’ [JHJ07].
- We offer a number of complementary delineations of what we mean by a manifest domain.
- But first some examples, “by name” !

Example 1 Names of Manifest Domains:
Examples of suggestive names of manifest domains are:

- *air traffic,*
- *banks,*
- *container lines,*
- *documents,*
- *hospitals,*
- *pipelines,*
- *railways* and
- *road nets* ■

- A **manifest domain** is a
  - human- and
  - artifact-assisted
  - arrangement of
    - endurant, that is spatially "stable", and
    - perdurant, that is temporally "fleeting"

    entities.
  - Endurant entities are
    - either parts
    - or components
    - or materials.
  - Perdurant entities are
    - either actions
    - or events
    - or behaviours ■

Example 2 Manifest Domain Endurants:
Examples of (names of) endurants are

- **Air traffic:** *aircraft, airport, air lane.*
- **Banks**: *client, passbook.*
- **Container lines:** *container, container vessel, terminal port.*
- **Documents:** *document, document collection.*
- **Hospitals:** *patient, medical staff, ward, bed, medical journal.*
- **Pipelines:** *well, pump, pipe, valve, sink, oil.*
- **Railways:** *simple rail unit, point, crossover, line, track, station.*
- **Road nets:** *link (street segment), hub (street intersection)* ■

Example 3 Manifest Domain Perdurants:
Examples of (names of) perdurants are

- **Air traffic:** *start (ascend) an aircraft, change aircraft course.*
- **Banks:** *open, deposit into, withdraw from, close (an account).*
- **Container lines:** *move container off or on board a vessel.*
- **Documents:** *open, edit, copy, shred.*
- **Hospitals:** *admit, diagnose, treat (patients).*
- **Pipelines:** *start pump, stop pump, open valve, close valve.*
- **Railways:** *switch rail point, start train.*
- **Road nets:** *set a hub signal, sense a vehicle* ■

Example 4 **Endurant Entity Qualities**:
Examples of (names of) endurant qualities:

- **Pipeline:**
  - *unique identity of a pipeline unit,*
  - *mereology (connectedness) of a pipeline unit,*
  - *length of a pipe,*
  - *(pumping) height of a pump,*
  - *open/close status of a valve.*
- **Road net:**
  - *unique identity of a road unit (hub or link),*
  - *road unit mereology:*
    - *identity of neighbouring hubs of a link,*
    - *identity of links emanating from a hub,*
  - *and state of hub (traversal) signal* ■

Example 5 **Perdurant Entity Qualities**:
Examples of (names of) perdurant qualities:

- **Pipeline:**
  - *the signature of an open (or close) valve action,*
  - *the signature of a start (or stop) pump action,*
  - *etc.*
- **Road net:**
  - *the signature of an insert (or remove) link action,*
  - *the signature of an insert (or remove) hub action,*
  - *the signature of a vehicle behaviour,*
  - *etc.* ■

We shall in the rest of this paper just write
'domain' instead of 'manifest domain'.

1.5. What Is a Domain Description ?

- By a **domain description** we understand
  - a collection of pairs of
  - narrative and commensurate
  - formal

  texts, where each pair describes
  - either aspects of an endurant entity
  - or aspects of a perdurant entity ■

- What does it mean that some text describes a domain entity ?
- For a text to be a **description text** it must be possible
  - to either, if it is a narrative,
    - to reason, informally, that the *designated* entity
    - is described to have some properties
    - that the reader of the text can observe
    - that the described entities also have;
  - or, if it is a formalisation
    - to prove, mathematically,
    - that the formal text
    - *denotes* the postulated properties ■

- By a **domain description** we shall thus understand a text which describes
  - the entities of the domain:
    - whether endurant or perdurant,
    - and when endurant whether
      - discrete or continuous,
      - atomic or composite;
    - or when perdurant whether
      - actions,
      - events or
      - behaviours.
  - as well as the qualities of these entities.

- So the task of the domain analyser cum describer is clear:
  - There is a domain: right in front of our very eyes,
  - and it is expected that that domain be described.

1.6. Towards a Methodology of Manifest Domain Analysis & Description

1.6.0.1 Practicalities of Domain Analysis & Description.

- How does one go about analysing & describing a domain ?
  - Well, for the first,
    - one has to designate one or more **domain analysers** cum
    - **domain describers**,
    - i.e., trained **domain scientists** cum **domain engineers**.
  - How does one get hold of a **domain engineer** ?
    - One takes a **software engineer** and *educates* and *trains* that person in
      - **domain science** &
      - **domain engineering**.
    - A derivative purpose of this seminar is to unveil aspects of **domain science** & **domain engineering**.

- The education and training consists in bringing forth
  - a number of scientific and engineering issues
    - of **domain analysis** and
    - of **domain description**.
  - Among the engineering issues are such as:
    - *what do I do when confronted*
      - *with the task of domain analysis ?* and
      - *with the task of description ?* and
    - *when, where and how do I*
      - `select` *and* `apply`
      - *which* `techniques` *and which* `tools` ?

- Finally, there is the issue of
  - *how do I, as a domain describer, choose appropriate*
    - *abstractions and*
    - *models ?*

1.6.0.2 The Four Domain Analysis & Description “Players”.

- We can say that there are four ‘players’ at work here.
  - (i) the domain,
  - (ii) the domain analyser & describer,
  - (iii) the domain analysis & description method, and
  - (iv) the evolving domain analysis & description (document).

- The domain is there.
  - The domain analyser & describer cannot change the domain.
  - Analysing & describing the domain does not change it[5].
  - During the analysis & description process
    - the domain can be considered inert.
    - (It changes with the installation of such software
    - as has been developed from the
    - requirements developed from the
    - domain description.)
  - In the physical sense the domain will usually contain
    - entities that are static (i.e., constant), and
    - entities that are dynamic (i.e., variable).

[5]Observing domains, such as we are trying to encircle the concept of domain, is not like observing the physical world at the level of subatomic particles. The experimental physicists' instruments of observation change what is being observed.

- The domain analyser & domain describer is a human,
  - preferably a scientist/engineer[6],
  - well-educated and trained in domain science & engineering.
  - The domain analyser & describer
    - observes the domain,
    - analyses it according to a method and
    - thereby produces a domain description.

[6] At the present time domain analysis appears to be partly an artistic, partly a scientific endeavour. Until such a time when domain analysis & description principles, techniques and tools have matured it will remain so.

- As a concept the method is here considered “fixed”.
  - By ‘fixed’ we mean that its principles, techniques and tools do not change during a domain analysis & description.
  - The domain analyser & describer
    - may very well apply these principles, techniques and tools
    - more-or-less haphazardly,
    - flaunting the method,
    - but the method remains invariant.
  - The method, however, may vary
    - from one domain analysis & description (project)
    - to another domain analysis & description (project).
  - Domain analysers & describers do become wiser from a project to the next.

- Finally there is the evolving *domain analysis & description*.
  - That description is a text, usually both informal and formal.
  - Applying a *domain description prompt* to the domain
    - yields an *additional domain description text*
    - which is added to the thus evolving *domain description*.

- One may speculate of the rôle of the "input" domain description.
  - Does it change ?
  - Does it help determine the additional domain description text ?
  - Etcetera.
- Without loss of generality we can assume
  - that the "input" domain description is changed and
  - that it helps determine the added text.

- Analysis & description is a trial-and-error, iterative process.
  - During a sequence of analyses,
  - that is, analysis prompts,
  - the analyser "discovers"
  - either more pleasing abstractions
  - or that earlier analyses or descriptions were wrong,
  - or that an entity either need be abstracted or made less abstract.
  - So they are corrected.

1.6.0.3 An Interactive Domain Analysis & Description Dialogue.

- We see domain analysis & description
  - as a process involving the above-mentioned four 'players',
  - that is, as a dialogue
  - between the domain analyser & describer and the domain,
  - where the dialogue is guided by the method
  - and the result is the description.
- We see the method as a 'player' which issues prompts:
  - alternating between:
  - "analyse this" (analysis prompts) and
  - "describe that" (synthesis or, rather, description prompts).

1.6.0.4 Prompts

- In this seminar we shall suggest
  - a number of *domain analysis prompts* and
  - a number of *domain description prompts*.
- The **domain analysis prompt**s
  - (schematically: `analyse_named_condition(e)`)
  - directs the analyser to inquire
  - as to the truth of whatever the prompt "names"
  - at wherever part (component or material), e, in the domain the prompt so designates.

- Based on the truth value of an analysed entity the domain analyser may then be prompted to describe that part (or material).
- The **domain description prompt**s
  - (schematically: `observe_type_or_quality(e)`)
  - directs the (analyser cum) describer to formulate
  - both an informal and a formal description
  - of the type or qualities of the entity designated by the prompt.
- The prompts form languages, and there are thus two languages at play here.

1.6.0.5 A Domain Analysis & Description Language.

- The 'Domain Analysis & Description Language' thus consists of a number of meta-functions, the prompts.
  - The meta-functions have names (say `is_endurant`) and types,
  - but have no formal definition.
  - They are not computable.
  - They are "performed" by the domain analysers & describers.
  - These meta-functions are
    - systematically introduced and
    - informally explained

    in Sects. 2–4.

1.6.0.6 The Domain Description Language.

- The 'Domain Description Language' is RSL [GHH+92], the RAISE Specification Language [GHH+95].
- With suitable, simple adjustments it could also be either of
  - Alloy [Jac06],
  - Event B [Abr09],
  - VDM-SL [BJ78, BJ82, FL98],
  - Z [WD96] or
  - CafeOBJ [FNT00] or
  - Magnolia (!?).
- We have chosen RSL because of its simple provision for
  - defining sorts,
  - expressing axioms, and
  - postulating observers over sorts.

1.6.0.7 Domain Descriptions: Narration & Formalisation

- Descriptions
  - *must* be readable and
  - *must* be mathematically precise.[7]
- For that reason we decompose domain description fragments into clearly identified "pairs" of
  - narrative texts and
  - formal texts.

[7] One must insist on formalised domain descriptions in order to be able to verify that domain descriptions satisfy a number of properties not explicitly formulated as well as in order to verify that requirements prescriptions satisfy domain descriptions.

1.7. One Domain – Many Models ?

- Will two or more domain engineers cum scientists arrive at "the same domain description" ?
- No, almost certainly not !
- What do we mean by "the same domain description" ?
  - To each proper description we can associate a mathematical meaning, its semantics.
  - Not only is it very unlikely that the syntactic form of the domain descriptions are the same or even "marginally similar".
  - But it is also very unlikely that the two (or more) semantics are the same;
  - that is, that all properties that can be proved for one domain model can be proved also for the other.

- Why will different domain models emerge ?
  - Two different domain describers will, undoubtedly,
  - when analysing and describing independently,
  - focus on different aspects of the domain.
    - One describer may focus attention on certain phenomena,
    - different from those chosen by another describer.
    - One describer may choose some abstractions
    - where another may choose more concrete presentations.
    - Etcetera.

- We can thus expect that a set of domain description developments lead to a set of distinct models.
  - ◈ As these domain descriptions
    - ◎ are communicated amongst domain engineers cum scientists
    - ◎ we can expect that iterated domain description developments
    - ◎ within this group of developers
    - ◎ will lead to fewer and more similar models.
  - ◈ Just like physicists,
    - ◎ over the centuries of research,
    - ◎ have arrived at a few models of nature,
    - ◎ we can expect there to develop some consensus models of "standard" domains.

- We expect, that sometime in future, software engineers,
  - when commencing software development
    for a "standard domain", that is,
  - one for which there exists one or more "standard models",
  - will start with the development of a domain description
  - based on "one of the standard models" —
  - just like control engineers of automatic control
  - "repeat" an essence of a domain model for a control problem.

Example 6 One Domain – Three Models:

- In this paper we shall bring many examples from a domain containing automobiles.
  - (i) One domain model may focus on roads and vehicles, with roads being modeled in terms of atomic hubs (road intersections) and atomic links (road sections between immediately neighbouring hubs), and with automobiles being modeled in terms of atomic vehicles.
  - (ii) Another domain model considers hubs of the former model as being composite, consisting, in addition to the "bare" hub, also of a signaling part — with automobiles remaining atomic vehicles,
  - (iii) A third model focuses on vehicles, now as composite parts consisting of composite and atomic sub-parts such as they are relevant in the assembly-line manufacturing of cars[8] ■

[8] The road nets of the first two models can be considered a zeroth model.

1.8. Structure of Seminar

- Sections 2.–4. are the main sections of this seminar.
  - They cover the analysis and description of
  - endurants and perdurants.
- Section 2. introduce the concepts of
  - entities,
  - endurant entities and
  - perdurant entities.

- Section 3. introduces
  - the external qualities of
    - parts,
    - components and
    - materials,

    and
    - the internal qualities of
      - unique part identifiers,
      - part mereologies and
      - part attributes.

- Section 4. complements Sect. 3.
  - It covers analysis and description of perdurants.
  - We consider the "compilation", Sect. , of part descriptions, i.e., endurants, into behaviour descriptions to be a separate contribution.
- Section 5. concludes the seminar.

2. Entities

2.1. General

Definition 1 Entity:

- *By an* **entity** *we shall understand a* **phenomenon**, *i.e., something*
  - *that can be* observed, *i.e., be*
    - *seen or touched*

    by humans,
  - *or that can be* conceived
    - *as an* abstraction
    - *of an entity.*
  - *We further demand that an entity can be objectively described* ■

Analysis Prompt 1 *is_entity:*

- *The domain analyser analyses "things" (θ) into either entities or non-entities.*
- *The method can thus be said to provide the* **domain analysis prompt***:*
  - *is_entity — where is_entity(θ) holds if θ is an entity* ■[9]
- `is_entity` is said to be a *prerequisite prompt* for all other prompts.

[9]Analysis prompt definitions and description prompt definitions and schemes are delimited by ■.

Whither Entities:

- The "demands" that entities
  - be observable and objectively describable

  raises some philosophical questions.
- Can sentiments, like feelings, emotions or "hunches" be objectively described ?
- This lecturer thinks not.
- And, if so, can they be other than artistically described ?
- It seems that
  - psychologically and
  - aesthetically

  "phenomena" appears to lie beyond objective description.
- We shall leave these speculations for later.

2.2. Endurants and Perdurants

Definition 2 Endurant:

- *By an* **endurant** *we shall understand an entity*
  - *that can be observed or conceived and described*
  - *as a "complete thing"*
  - *at no matter which given snapshot of time.*

  Were we to "freeze" time
  - *we would still be able to observe the entire endurant* ■

- That is, endurants "reside" in space.
- Endurants are, in the words of Whitehead (1920), continuants.

Example 7 **Traffic System Endurants**:
Examples of traffic system endurants are:

- traffic system,
- road nets,
- fleets of vehicles,
- sets of hubs,
- sets of links,
- hubs,
- links and
- vehicles ■

Definition 3 Perdurant:

- *By a* **perdurant** *we shall understand an entity*
  - *for which only a fragment exists*
    if we look at or touch them
    at any given snapshot in time, that is,
  - *were we to freeze time we would only see or touch*
    a fragment of the perdurant ■
- That is, perdurants "reside" in space and time.
- Perdurants are, in the words of Whitehead(1920), occurrents.

Example 8 Traffic System Perdurants:
Examples of road net perdurants are:

- *insertion* and *removal* of hubs or links (actions),
- *disappearance* of links (events),
- vehicles *entering* or *leaving* the road net (actions),
- vehicles *crashing* (events) and
- *road traffic* (behaviour) ■

Analysis Prompt 2 *is_endurant:*

- *The domain analyser analyses an entity, ϕ, into an endurant as prompted by the* **domain analysis prompt***:*
  - *is_endurant — ϕ is an endurant if is_endurant(ϕ) holds.*
- *is_entity is a* prerequisite prompt *for is_endurant* ■

Analysis Prompt 3 *is_perdurant:*

- *The domain analyser analyses an entity ϕ into perdurants as prompted by the* **domain analysis prompt***:*
  - *is_perdurant — ϕ is a perdurant if is_perdurant(ϕ) holds.*
- *is_entity is a* prerequisite prompt *for is_perdurant* ■

- In the words of Whitehead(1920)
  - an endurant has stable qualities that enable its various appearances at different times to be recognised as the same individual;
  - a perdurant is in a state of flux that prevents it from being recognised by a stable set of qualities.

Necessity and Possibility:

- It is indeed possible to make the endurant/perdurant distinction.
- But is it necessary ?
- We shall argue that it is 'by necessity' that we make this distinction.
  - Space and time are fundamental notions.
  - They cannot be dispensed with.
  - So, to describe manifest domains
    without resort to space and time
    is not reasonable.

2.3. Discrete and Continuous Endurants

Definition 4 Discrete Endurant:

- *By a* **discrete endurant** *we shall understand an endurant which is*
  - *separate,*
  - *individual or*
  - *distinct*

  in form or concept ■

Example 9 Discrete Endurants:

- Examples of discrete endurants are
  - a road net,
  - a link,
  - a hub,
  - a vehicle,
  - a traffic signal,
  - etcetera ■

Definition 5 Continuous Endurant:

- *By a* **continuous endurant** *we shall understand an endurant which is*
  - *prolonged, without interruption,*
  - *in an unbroken series or pattern* ■

Example 10 Continuous Endurants:

- Examples of continuous endurants are
  - water,
  - gas,
  - grain,
  - oil,
  - sand,
  - etcetera ■

- Continuity shall here not be understood in the sense of mathematics.
  - Our definition of ‘continuity’ focused on
    - *prolonged,*
    - *without interruption,*
    - *in an unbroken series or*
    - *pattern.*
  - In that sense
    materials and components
    shall be seen as ‘continuous’,

Analysis Prompt 4 *is_discrete:*

- *The domain analyser analyses endurants e into discrete entities as prompted by the* **domain analysis prompt***:*
  - *is_discrete — e is discrete if is_discrete(e) holds* ■

Analysis Prompt 5 *is_continuous:*

- *The domain analyser analyses endurants e into continuous entities as prompted by the* **domain analysis prompt***:*
  - *is_continuous — e is continuous if is_continuous(e) holds* ■

2.4. An Upper Ontology Diagram of Domains

- Figure 1 on the facing slide shows a so-called upper ontology for manifest domains.
  - So far we have covered only a fraction of this ontology, as noted.
  - By ontologies we shall here understand
  - *"formal representations of a set of concepts within a domain and the relationships between those concepts"*.

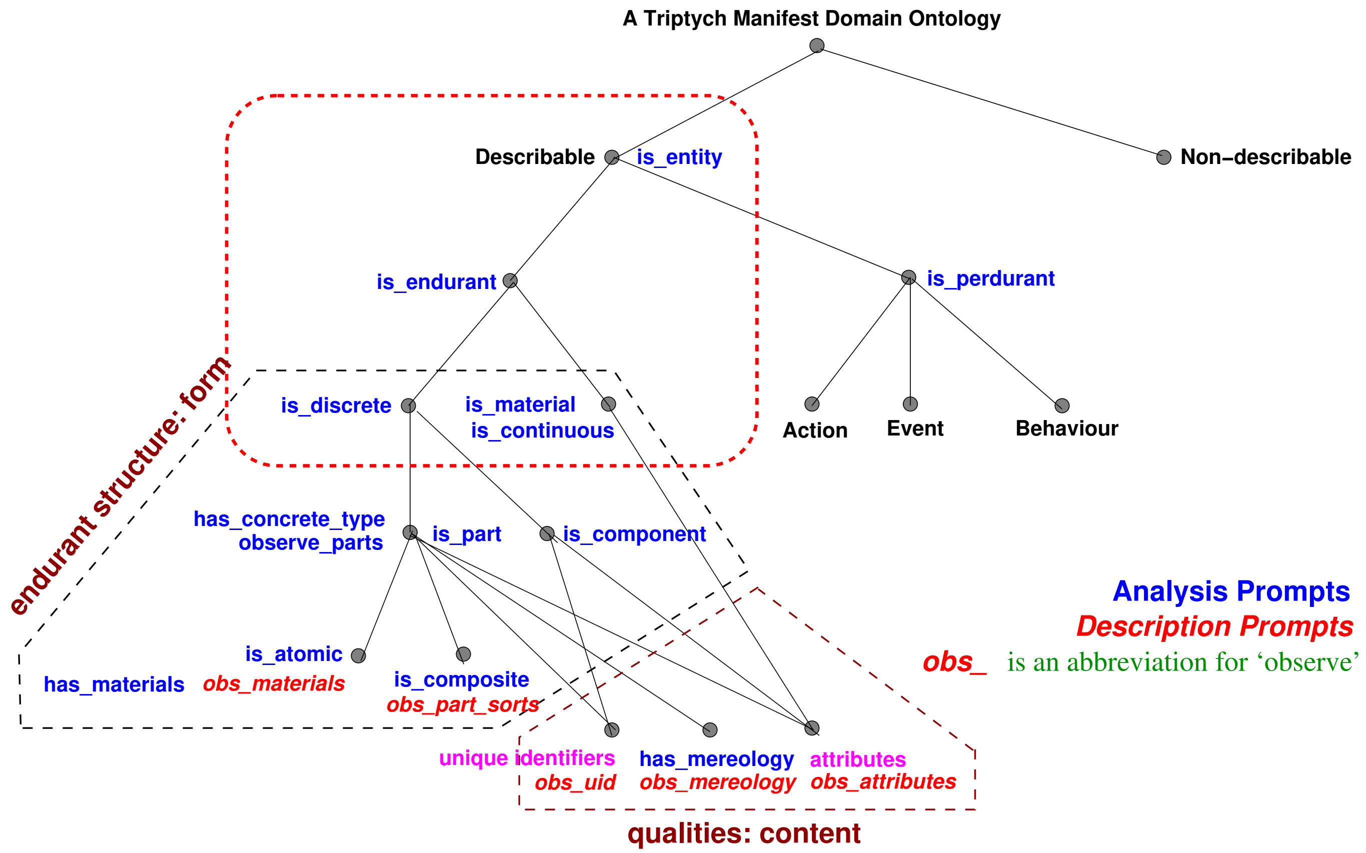

Figure 1: An Upper Ontology for Domains – **We've just covered the red-dashed concepts**

Lecture 2 79–166

3. Endurants

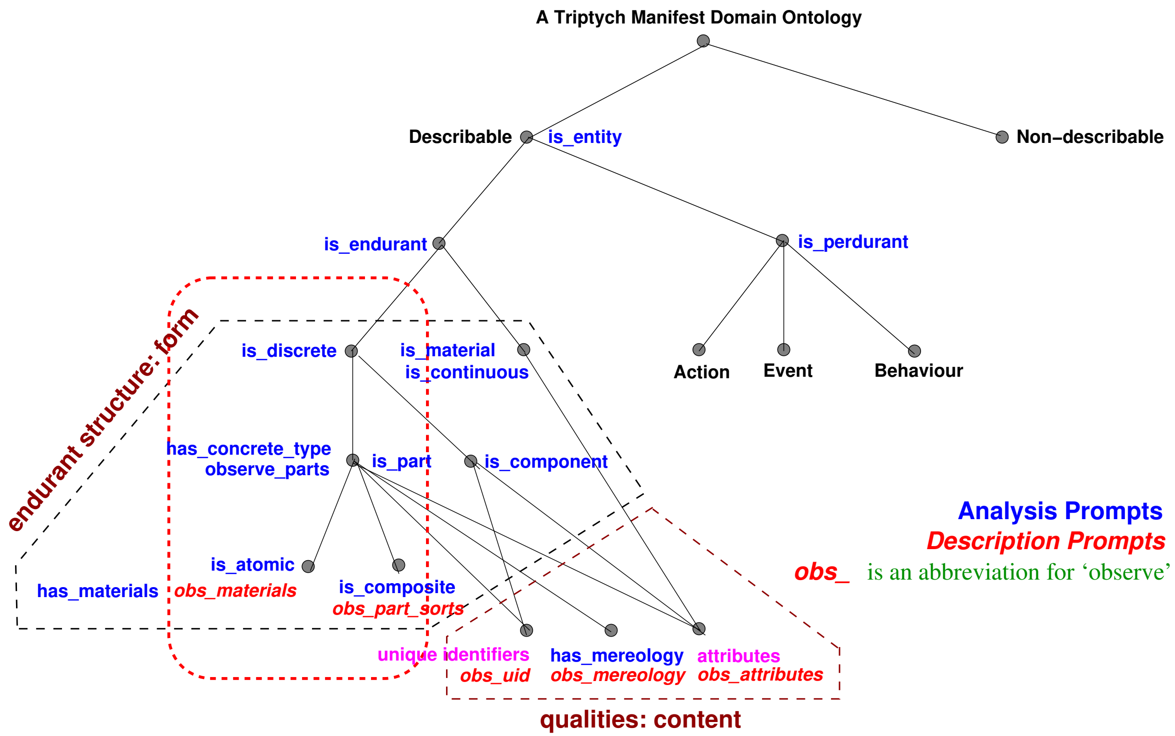

Figure 2: An Upper Ontology for Domains: **Parts** – **We're now going to cover these red-dashed concepts**

- This section brings a comprehensive treatment of the analysis and description of endurants.

3.1. Parts, Components and Materials

3.1.1. General

Definition 6 Part:

- *By a* **part** *we shall understand*
  - *a discrete endurant*
  - *which the domain engineer chooses*
  - *to endow with* **internal qualities** *such as*
    - *unique identification,*
    - *mereology, and*
    - *one or more attributes* ■

We shall soon define the terms
'unique identification', 'mereology', and 'attributes'.

Example 11 Parts: Example

- 7 on Slide 64 illustrated
  - traffic systems,
  - road nets,
  - fleets of vehicles,
  - set of hubs,
  - set of links,
  - hubs and
  - links

parts,

and examples

- 15 on Slide 95 and
- 16 on Slide 97

shall illustrate parts ■

Definition 7 Component:

- *By a* **component** *we shall understand*
  - *a discrete endurant*
  - *which we, the domain analyser cum describer chooses*
  - *to* **not** *endow with* **internal qualities** ■

Example 12 Components:

- Examples of components are:
  - chairs, tables, sofas and book cases in a living room,
  - letters, newspapers, and small packages in a mail box,
  - machine assembly units on a conveyor belt,
  - boxes in containers of a container vessel,
  - etcetera ■

"At the Discretion of the Domain Engineer":

- We emphasise the following analysis and description aspects:
  - (a) The domain is full of observable phenomena.
    - It is the decision of the domain analyser cum describer
    - whether to analyse and describe some such phenomena,
    - that is, whether to include them in a domain model.
  - (b) The borderline between an endurant
    - being (considered) discrete or
    - being (considered) continuous
    - is fuzzy.
    - It is the decision of the domain analyser cum describer
    - whether to model an endurant as discrete or continuous.

- (c) The borderline between a discrete endurant
  - being (considered) a part or
  - being (considered) a component
  - is fuzzy.
  - It is the decision of the domain analyser cum describer
  - whether to model a discrete endurant as a part or as a component.
- (d) We shall later show how to “compile” parts into processes.
  - A factor, therefore, in determining whether
  - to model a discrete endurant as a part or as a component
  - is whether we may consider a discrete endurant as also representing a process.

Definition 8 Material:

- *By a* **material** *we shall understand a continuous endurant* ■

Example 13 Materials: Examples of material endurants are:

- air of an air conditioning system,
- grain of a silo,
- gravel of a barge,
- oil (or gas) of a pipeline,
- sewage of a waste disposal system, and
- water of a hydro-electric power plant. ■

Example 14 **Parts Containing Materials**:

- Pipeline units are here considered discrete, i.e., parts.
- Pipeline units serve to convey material ■

3.1.2. Part, Component and Material Analysis Prompts

Analysis Prompt 6 *is_part:*

- *The domain analyser analyse endurants, e, into part entities as prompted by the* **domain analysis prompt***:*
  - *is_part — e is a part if is_part(e) holds* ■

- We remind the reader that the outcome of is_part(e)
- is very much dependent on the domain engineer's intention
- with the domain description, cf. Slide 84.

Analysis Prompt 7 *is_component:*

- *The domain analyser analyse endurants e into component entities as prompted by the* **domain analysis prompt***:*
  - *is_component — e is a component if is_component(e) holds* ■
- We remind the reader that the outcome of `is_component(`*e*`)`
- is very much dependent on the domain engineer's intention
- with the domain description, cf. Slide 84.

Analysis Prompt 8 *is_material*:

- *The domain analyser analyse endurants e into material entities as prompted by the* **domain analysis prompt**:
  - *is_material — e is a material if is_material(e) holds* ■
- We remind the reader that the outcome of `is_material`(e)
- is very much dependent on the domain engineer's intention
- with the domain description, cf. Slide 84.

3.1.3. Part, Component and Material Qualities

- To us
  - parts have unique identifiers, mereology and attributes;
  - components have unique identifiers and attributes;
  - materials have attributes
- [The above “restrictions” are pragmatic.]
- [Other “divisions” of “labour” could be formulated.]

3.1.4. Atomic and Composite Parts

- A distinguishing quality
  - of parts
  - is whether they are
    - atomic or
    - composite.
- Please note that we shall,
  - in the following,
  - examine the concept of parts
  - in quite some detail.

- That is,
  - parts become the domain endurants of main interest,
  - whereas components and materials become of secondary interest.
- This is a choice.
  - The choice is based on pragmatics.
  - It is still the domain analyser cum describers' choice
    - whether to consider a discrete endurant
    - a part
    - or a component.
  - If the domain engineer wishes to investigate
    - the details of a discrete endurant
    - then the domain engineer choose to model
    - the discrete endurant as a part
    - otherwise as a component.

Definition 9 **Atomic Part:**

- **Atomic parts** *are those which,*
  - *in a given context,*
  - *are deemed to not consist of meaningful, separately observable proper* sub-parts ■
- A **sub-part** is a part ■

Example 15 Atomic Parts:
Examples of atomic parts of the above mentioned domains are:

- aircraft[10] (of air traffic),
- demand/deposit accounts (of banks),
- containers (of container lines),
- documents (of document systems),
- hubs, links and vehicles (of road traffic),
- patients, medical staff and beds (of hospitals),
- pipes, valves and pumps (of pipeline systems), and
- rail units and locomotives (of railway systems) ■

[10] Aircraft from the point of view of airport management are atomic. From the point of view of aircraft manufacturers they are composite.

Definition 10 Composite Part:

- **Composite parts** *are those which,*
  - *in a given context,*
  - *are deemed to indeed consist of meaningful, separately observable proper* sub-parts ■

Example 16 Composite Parts:
Examples of composite parts of the above mentioned domains are:

- airports and air lanes (of air traffic),
- banks (of a financial service industry),
- container vessels (of container lines),
- dossiers of documents (of document systems),
- routes (of road nets),
- medical wards (of hospitals),
- pipelines (of pipeline systems), and
- trains, rail lines and train stations (of railway systems). ■

Analysis Prompt 9 *is_atomic:*

- *The domain analyser analyses a discrete endurant, i.e., a part p into an atomic endurant:*
  - *is_atomic(p): p is an atomic endurant if is_atomic(p) holds* ■

Analysis Prompt 10 *is_composite:*

- *The domain analyser analyses a discrete endurant, i.e., a part p into a composite endurant:*
  - *is_composite(p): p is a composite endurant if is_composite(p) holds* ■

- `is_discrete` is a **prerequisite prompt is_discrete** of both `is_atomic` and `is_composite`.

Whither Atomic or Composite:

- If we are analysing & describing vehicles
  in the context of a road net, cf. the Traffic System Example Slide 64,
  - then we have chosen to abstract vehicles
  - as atomic;
- if, on the other hand, we are analysing & describing vehicles
  in the context of an automobile maintenance garage
  - then we might very well choose to abstract vehicles
  - as composite —
  - the sub-parts being the object of diagnosis
  - by the auto mechanics.

3.1.5. On Observing Part Sorts and Types

- We use the term 'sort'
  - when we wish to speak of an abstract type,
  - that is, a type for which we do not wish to express a model[11].
  - We shall use the term 'type' to cover both
    - abstract types and
    - concrete types.

[11] for example, in terms of the concrete types:

- sets,
- lists,
- Cartesians,
- maps,

or other.

3.1.6. On Discovering Part Sorts

- We “equate” a formal concept with a type (i.e., a sort).
  - Thus, to us, a part sort is a set of all those entities
  - which all have exactly the same qualities.
- Our aim now
  - is to present the basic principles that let
  - the domain analyser decide on **part sort**s.

- We observe parts one-by-one.
- (α) Our analysis of parts concludes when we have
  - "lifted" our examination of a particular part instance
  - to the conclusion that it is of a given sort,
  - that is, reflects a formal concept.
- Thus there is, in this analysis, a "eureka",
  - a step where we shift focus
  - from the concrete to the abstract,
  - from observing specific part instances
  - to postulating a sort:
    - from one to the many.

Analysis Prompt 11 *observe_parts:*

- *The* **domain analysis prompt***:*
  - ◈ *observe_parts*(p)
- *directs the domain analyser to observe the sub-parts of p* ■

Let us say the sub-parts of p are: $\{p_1,p_2,\ldots,p_m\}$.

- (β) The analyser analyses, for each of these parts, p_{i_k},
  - ◈ which formal concept, i.e., sort, it belongs to;
  - ◈ let us say that it is of sort P_k;
  - ◈ thus the sub-parts of p are of sorts $\{P_1,P_2,\ldots,P_m\}$.
- Some P_k may be atomic sorts, some may be composite sorts.

- The domain analyser continues to examine a finite number of other composite parts: $\{p_j, p_\ell, \ldots, p_n\}$.
  - It is then "discovered", that is, decided, that they all consists of the same number of sub-parts
    - $\{p_{i_1},p_{i_2},\ldots,p_{i_m}\}$,
    - $\{p_{j_1},p_{j_2},\ldots,p_{j_m}\}$,
    - $\{p_{\ell_1},p_{\ell_2},\ldots,p_{\ell_m}\}$,
    - ...,
    - $\{p_{n_1},p_{n_2},\ldots,p_{n_m}\}$,

    of the same, respective, part sorts.

- (γ) It is therefore concluded, that is, decided, that $\{p_i, p_j, p_\ell, \ldots, p_n\}$ are all of the same part sort P with observable part sub-sorts $\{P_1,P_2,\ldots,P_m\}$.

- Above we have *type-font-highlighted* three sentences: (α, β, γ).
- When you analyse what they "prescribe" you will see that they entail a "depth-first search" for part sorts.
  - The β sentence says it rather directly:
  - *"The analyser analyses, for each of these parts, p_k, which formal concept, i.e., part sort it belongs to."*
  - To do this analysis in a proper way, the analyser must ("recursively") analyse the parts "down" to their atomicity,
  - and from the atomic parts decide on their part sort,
  - and work ("recurse") their way "back",
  - through possibly intermediate composite parts,
  - to the p_ks.

- Of course, when the analyser starts by examining atomic parts
  - then the analysis "recursion" is not necessary;
  - as it is never necessary when the analyser proceeds "bottom-up":
  - analysing only such composite parts whose sub-parts have already been analysed

3.1.7. Part Sort Observer Functions

- The above analysis amounts to the analyser
  - first "applying" the domain analysis prompt
  - `is_composite`(p) to a discrete endurant,
  - where we now assume that the obtained truth value is **true**.
  - Let us assume that parts p:P consists of sub-parts of sorts $\{P_1, P_2, \ldots, P_m\}$.
  - Since we cannot automatically guarantee that our domain descriptions secure that
    - P and each P_i ($1 \leq i \leq$ m)
    - denotes disjoint sets of entities

    we must prove it.

Domain Description Prompt 1 *observe_part_sorts*:

- *If is_composite(p) holds, then the analyser "applies" the* **domain description prompt**
  - *observe_part_sorts(p)*

  resulting in the analyser writing down the part sorts *and* part sort observers
  domain description text
  according to the following schema:

1. observe_part_sorts schema

Narration:

[s] ... narrative text on sorts ...
[o] ... narrative text on sort observers ...
[i] ... narrative text on sort recognisers ...
[p] ... narrative text on proof obligations ...

Formalisation:

type
[s] P,
[s] P_i $[1 \leq i \leq m]$ **comment:** P_i $[1 \leq i \leq m]$ abbreviates P_1, P_2, ..., P_m
value
[o] **obs_part**_P_i: $P \to P_i$ $[1 \leq i \leq m]$
[i] **is**_P_i: $(P_1|P_2|...|P_m) \to \mathbf{Bool}$ $[1 \leq i \leq m]$

proof obligation [Disjointness of part sorts]
[p] $\forall\ p{:}(P_1|P_2|...|P_m) \cdot$
[p] $\bigwedge \{$**is**_P_i(p) $\equiv \bigwedge \{\sim$**is**_P_j(p) $|$ j $\in$ {1..m} \ {i}} | i $\in$ {1..m}}

Example 17 **Composite and Atomic Part Sorts of Transportation**:

- The following example illustrates the multiple use of the `observe_part_sorts` function:
  - first to δ:Δ, a specific transport domain, Item 1,
  - then to an $n : N$, the net of that domain, Item 2, and
  - then to an $f : F$, the fleet of that domain, Item 3.

1 A transportation domain is composed from a net, a fleet (of vehicles) and a monitor.

2 A transportation net is composed from a collection of hubs and a collection of links.

3 A fleet is a collection of vehicles.

- The monitor is considered an atomic part.

type
1. Δ, N, F, M
value
1. **obs_part**_N: $\Delta \rightarrow$ N,
1. **obs_part**_F: $\Delta \rightarrow$ F,
1. **obs_part**_M: $\Delta \rightarrow$ M
type
2. HS, LS
value
2. **obs_part**_HS: N $\rightarrow$ HS,
2. **obs_part**_LS: N $\rightarrow$ LS
type
3. VS
value
3. **obs_part**_VS: F $\rightarrow$ VS

- *A* ***proof obligation*** *has to be discharged,*
  - *one that shows disjointedness of sorts N, F and M.*
  - *An informal sketch is:*
    - *entities of sort N are composite and consists of two parts:*
    - *aggregations of hubs, HS, and aggregations of links, LS.*
    - *Entities of sort F consists of an aggregation, VS, of vehicles.*
    - *So already that makes N and F disjoint.*
    - *M is an atomic entity — where N and F are both composite.*
    - *Hence the three sorts N, F and M are disjoint* ■

3.1.8. On Discovering Concrete Part Types

Analysis Prompt 12 *has_concrete_type:*

- *The domain analyser*
  - *may decide that it is expedient, i.e., pragmatically sound,*
  - *to render a part sort, P, whether atomic or composite, as a concrete type, T.*
  - *That decision is prompted by the holding of the* **domain analysis prompt***:*
    - *has_concrete_type(p).*
  - *is_discrete is a* **prerequisite prompt** *of has_concrete_type* ■

- The reader is reminded that
  - the decision as to whether an abstract type is (also) to be described concretely
  - is entirely at the discretion of the domain engineer.

Domain Description Prompt 2 *observe_part_type*:

- *Then the domain analyser applies the* **domain description prompt***:*
  - *observe_part_type*(*p*)[12]
- *to parts p:P which then yield the* part type *and* part type observers *domain description text according to the following schema:*

[12] has_concrete_type is a **prerequisite prompt** of observe_part_type.

2. `observe_part_type` schema

Narration:

[t$_1$] ... narrative text on sorts and types S_i ...

[t$_2$] ... narrative text on types T ...

[o] ... narrative text on type observers ...

Formalisation:

type

[t$_1$] S_1, S_2, ..., S_m, ..., S_n,

[t$_2$] $T = \mathscr{E}(S_1, S_2, \ldots, S_n)$

value

[o] **obs_part**_T: P → T

- The type name,
  - T, of the concrete type,
  - as well as those of the auxiliary types, S_1,S_2,...,S_m,
  - are chosen by the domain describer:
    - they may have already been chosen
    - for other sort–to–type descriptions,
    - or they may be new.

Example 18 **Concrete Part Types of Transportation**:

We continue Example 17 on Slide 110:

4 A collection of hubs is a set of hubs and
a collection of links is a set of links.

5 Hubs and links are, until further analysis, part sorts.

6 A collection of vehicles is a set of vehicles.

7 Vehicles are, until further analysis, part sorts.

type

4. Hs = H-**set**, Ls = L-**set**
5. H, L
6. Vs = V-**set**
7. V

value

4. **obs_part**_Hs: HS → Hs, **obs_part**_Ls: LS → Ls
6. **obs_part**_Vs: VS → Vs ■

3.1.9. Forms of Part Types

- Usually it is wise to restrict the part type definitions, $\mathsf{T}_i = \mathscr{E}_i(\mathsf{Q},\mathsf{R},\ldots,\mathsf{S})$, to simple type expressions.

  - T=A-**set** or
  - T=A* or
  - T=ID $\overrightarrow{m}$ A or
  - T=$\mathsf{A}_t|\mathsf{B}_t|\ldots|\mathsf{C}_t$

  where

  - ID is a sort of unique identifiers,
  - T=$\mathsf{A}_t|\mathsf{B}_t|\ldots|\mathsf{C}_t$ defines the disjoint types
    - A_t==mkA_t(s:A_s),
    - B_t==mkB_t(s:B_s), ...,
    - C_t==mkC_t(s:C_s),

    and where
  - A, A_s, B_s, ..., C_s are sorts.
  - Instead of A_t==mkA_t(a:A_s), etc., we may write A_t::A_s etc.

3.1.10. Part Sort and Type Derivation Chains

- Let P be a composite sort.
- Let P_1, P_2, ..., P_m be the part sorts "discovered" by means of observe_part_sorts(p) where p:P.
- We say that P_1, P_2, ..., P_m are (immediately) **derived** from P.
- If P_k is derived from P_j and P_j is derived from P_i, then, by transitivity, P_k is **derived** from P_i.

3.1.10.1 No Recursive Derivations

- We "mandate" that
  - if P_k is derived from P_j
  - then there
    - can be no P derived from P_j
    - such that P is P_j,
    - that is, P_j cannot be derived from P_j.
- That is, we do not allow recursive domain sorts.
- It is not a question, actually of allowing recursive domain sorts.
  - It is, we claim to have observed,
  - in very many domain modeling experiments,
  - that there are no recursive domain sorts !

3.1.11. Names of Part Sorts and Types

- The domain analysis and domain description text prompts
  - observe_part_sorts,
  - observe_material_sorts and
  - observe_part_type

  — as well as the
  - attribute_names,
  - observe_material_sorts,
  - observe_unique_identifier,
  - observe_mereology and
  - observe_attributes

  prompts introduced below — “yield” type names.
  - That is, it is as if there is
    - a reservoir of an indefinite-size set of such names
    - from which these names are “pulled”,
    - and once obtained are never “pulled” again.

- There may be domains for which two distinct part sorts may be composed from identical part sorts.
- *In this case the domain analyser indicates so by prescribing a part sort already introduced.*

Example 19 **Container Line Sorts**:

- Our example is that of a container line
  - with container vessels and
  - container terminal ports.

8 A container line contains a number of container vessels and a number of container terminal ports, as well as other parts.

9 A container vessel contains a container stowage area, etc.

10 A container terminal port contains a container stowage area, etc.

11 A container stowage areas contains a set of uniquely identified container bays.

12 A container bay contains a set of uniquely identified container rows.

13 A container row contains a set of uniquely identified container stacks.

14 A container stack contains a stack, i.e., a first-in, last-out sequence of containers.

15 Containers are further undefined.

- After a some slight editing we get:

type
CL
VS, VI, V, Vs = VI $\overrightarrow{m}$ V,
PS, PI, P, Ps = PI $\overrightarrow{m}$ P
value
obs_part_VS: CL → VS
obs_part_Vs: VS → Vs
obs_part_PS: CL → PS
obs_part_Ps: CTPS → CTPs
type
CSA
value
obs_part_CSA: V → CSA
obs_part_CSA: P → CSA

type
BAYS, BI, BAY, Bays=BI $\overrightarrow{m}$ BAY
ROWS, RI, ROW, Rows=RI $\overrightarrow{m}$ ROW
STKS, SI, STK, Stks=SI $\overrightarrow{m}$ STK
C
value
obs_part_BAYS: CSA → BAYS,
obs_part_Bays: BAYS → Bays
obs_part_ROWS: BAY → ROWS,
obs_part_Rows: ROWS → Rows
obs_part_STKS: ROW → STKS,
obs_part_Stks: STKS → Stks
obs_part_Stk: STK → C*

- Note that `observe_part_sorts`(v:V) and `observe_part_sorts`(p:P) both yield CSA ■

3.1.12. More On Part Sorts and Types

- The above "experimental example" motivates the below.
  - ◈ We can always assume that composite parts p:P abstractly consists of a definite number of sub-parts.
    - ◎ **Example 20**. We comment on Example 17, Page 110: Parts of type Δ and N are composed from three, respectively two abstract sub-parts of distinct types ■
  - ◈ Some of the parts, say p_{i_z} of $\{p_{i_1},p_{i_2},\ldots,p_{i_m}\}$, of p:P, may themselves be composite.
    - ◎ **Example 21**. We comment on Example 17: Parts of type N, F, HS, LS and VS are all composite ■

- There are, pragmatically speaking, two cases for such compositionality.
  - Either the part, p_{i_z}, of type t_{i_z}, is is composed from a definite number of abstract or concrete sub-parts of distinct types.
    - **Example 22**. We comment on Example 17: Parts of type N are composed from three sub-parts ■
  - Or it is composed from an indefinite number of sub-parts of the same sort.
    - **Example 23**. We comment on Example 17: Parts of type HS, LS and VS are composed from an indefinite numbers of hubs, links and vehicles, respectively ■

Example 24 Pipeline Parts:

16 A pipeline consists of an indefinite number of pipeline units.

17 A pipeline units is either a well, or a pipe, or a pump, or a valve, or a fork, or a join, or a sink.

18 All these unit sorts are atomic and disjoint.

type
16. PL, U, We, Pi, Pu, Va, Fo, Jo, Si
16. Well, Pipe, Pump, Valv, Fork, Join, Sink
value
16. **obs_part**_Us: PL → U-**set**
type
17. U == We | Pi | Pu | Va | Fo | Jo | Si
18. We::Well, Pi::Pipe, Pu::Pump, Va::Valv, Fo:Fork, Jo::Join, Si::Sink ■

3.2. External and Internal Qualities of Parts

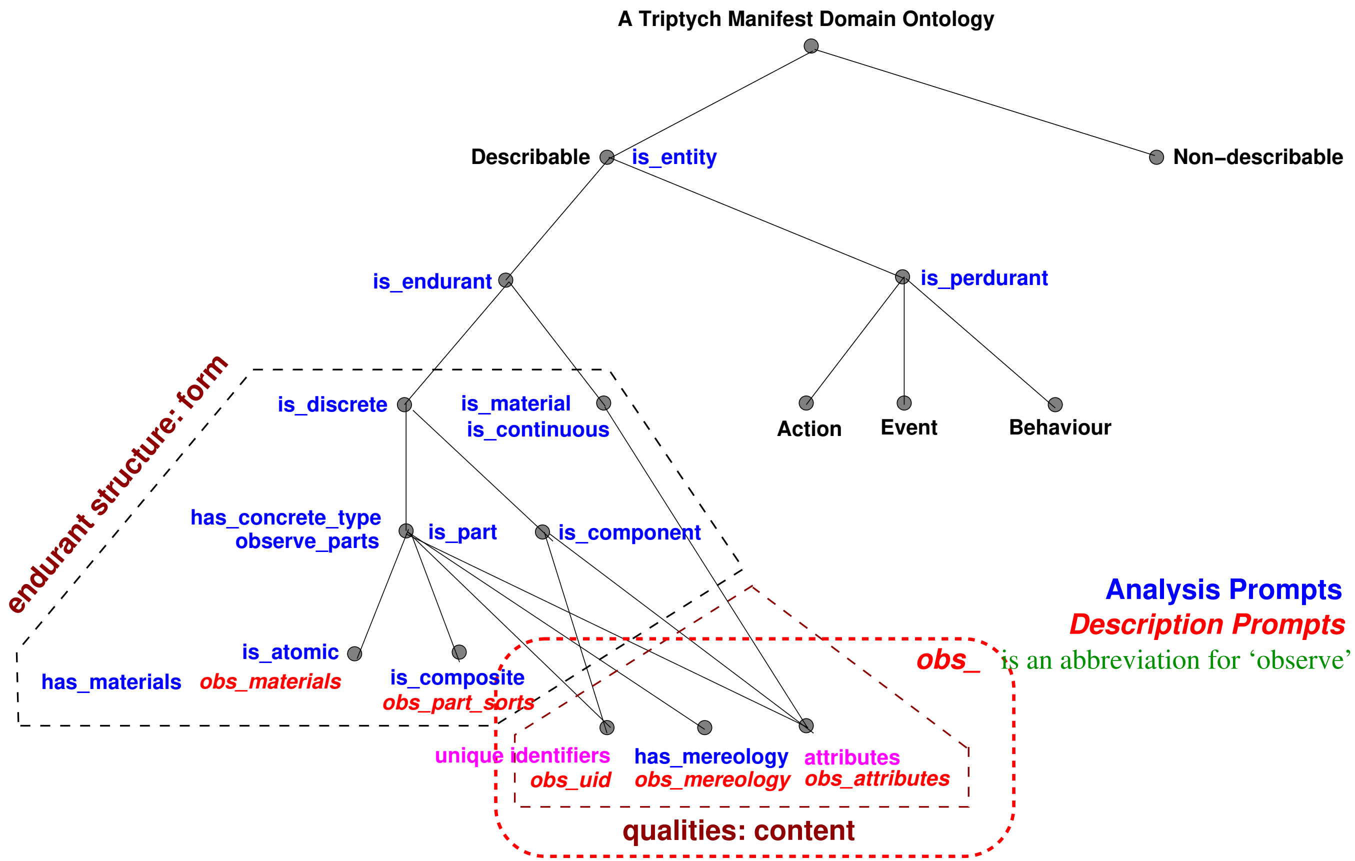

Figure 3: An Upper Ontology for Domains — **Internal Qualities**

- By an **external part quality** we shall understand the
  - is_atomic,
  - is_composite,
  - is_discrete and
  - is_continuous

  qualities ■
- By an **internal part quality** we shall understand the part qualities to be outlined in the next sections:
  - unique ids,
  - mereology and
  - attributes ■
- By **part qualities** we mean the sum total of
  - external endurant and
  - internal endurant

  qualities ■

3.3. Three Categories of Internal Qualities

- We suggest that the internal qualities of parts be analysed into three categories:
  - (i) a category of unique part identifiers,
  - (ii) a category of mereological quantities and
  - (iii) a category of general attributes.

- Part mereologies are about sharing qualities between parts.
  - Some such sharing expresses spatio-topological properties of how parts are organised.
  - Other part sharing aspects express relations (like equality) of part attributes.
  - We base our modeling of mereologies on the notion of unique part identifiers.
  - Hence we cover **internal qualities** in the order (i–ii–iii).

3.4. Unique Part Identifiers

- We introduce a notion of unique identification of parts.
- We assume
  - (i) that all parts, p, of any domain P, have **unique identifiers**,
  - (ii) that **unique identifiers** (of parts p:P) are **abstract values** (of the **unique identifier** sort PI of parts p:P),
  - (iii) such that distinct part sorts, P_i and P_j, have distinctly named **unique identifier** sorts, say PI_i and PI_j,
  - (iv) that all π_i:PI_i and π_j:PI_j are distinct, and
  - (v) that the observer function **uid**_P applied to p yields the unique identifier, say π:PI, of p.

Representation of Unique Identifiers:

- Unique identifiers are abstractions.
  - When we endow two parts (say of the same sort) with distinct unique identifiers
  - then we are simply saying that these two parts are distinct.
  - We are not assuming anything about how these identifiers otherwise come about.

Domain Description Prompt 3 *observe_unique_identifier*:

- *We can therefore apply the* **domain description prompt***:*
  - *observe_unique_identifier*
- *to parts p:P*
  - *resulting in the analyser writing down*
  - *the* unique identifier type and observer *domain description text according to the following schema:*

3. observe_unique_identifier schema

Narration:

[s] ... narrative text on unique identifier sort PI ...

[u] ... narrative text on unique identifier observer **uid**_P ...

[a] ... axiom on uniqueness of unique identifiers ...

Formalisation:

type

[s] PI

value

[u] **uid**_P: P $\rightarrow$ PI

axiom

[a] $\mathscr{U}$

Example 25 **Unique Transportation Net Part Identifiers**:
We continue Example 17 on Slide 110.

19 Links and hubs have unique identifiers

20 and unique identifier observers.

type
19. LI, HI
value
20. **uid**_LI: L → LI
20. **uid**_HI: H → HI
axiom [Well−formedness of Links, L, and Hubs, H]
19. $\forall$ l,l′:L · **uid**_LI(l)=**uid**_LI(l′) ⇒ l=l′,
19. $\forall$ h,h′:H · **uid**_HI(h)=**uid**_HI(h′) ⇒ h=h′ ■

- Axiom 19, although expressed for links and hubs of road nets, applies in general:
  - Two parts with the same unique part identifiers
  - are indeed one and the same part.

3.5. Mereology

- Mereology is the study and knowledge of parts and part relations.
  - Mereology, as a logical/philosophical discipline,
    can perhaps best be attributed to the
    Polish mathematician/logician
    Stanisław Leśniewski [CV99, Bjø14a].

3.5.1. Part Relations

- Which are the relations that can be relevant for part-hood ?
- We give some examples.
  - Two otherwise distinct parts may share attribute values.

  Example 26 Shared Timetable Mereology (I):
  - Two or more distinct public transport busses
    - may "run" according to the (identically) same,
    - thus "shared", bus time table ■

- Two otherwise distinct parts may be said to, for example, be topologically "adjacent" or one "embedded" within the other.

  Example 27 Topological Connectedness Mereology:
  - (i) two rail units may be connected (i.e., adjacent);
  - (ii) a road link may be connected to two road hubs;
  - (iii) a road hub may be connected to zero or more road links;
  - (iv) distinct vehicles of a road net may be monitored by one and the same road pricing sub-system ■

- The above examples are in no way indicative of the "space" of part relations that may be relevant for part-hood.
- The domain analyser is expected to do a bit of experimental research in order to discover necessary, sufficient and pleasing "mereology-hoods" !

3.5.2. Part Mereology: Types and Functions

Analysis Prompt 13 *has_mereology:*

- *To discover necessary, sufficient and pleasing "mereology-hoods" the analyser can be said to endow a truth value,* **true***, to the* **domain analysis prompt***:*
  - *has_mereology*
- When the domain analyser decides that
  - some parts are related in a specifically enunciated mereology,
  - the analyser has to decide on suitable
    - **mereology type**s and
    - **mereology observer**s (i.e., part relations).

- We can define a **mereology type** as a type $\mathscr{E}$xpression over unique [part] identifier types.
  - We generalise to unique [part] identifiers over a definite collection of part sorts, P1, P2, ..., Pn,
  - where the parts p1:P1, p2:P2, ..., pn:Pn are not necessarily (immediate) sub-parts of some part p:P.

type
  PI1, PI2, ..., PIn
  MT = $\mathscr{E}$(PI1, PI2, ..., PIn),

Domain Description Prompt 4 observe_mereology:

- *If* has_mereology*(p) holds for parts p of type* P*,*
  - *then the analyser can apply the* **domain description prompt***:*
    - observe_mereology
  - *to parts of that type*
  - *and write down the* mereology types and observer *domain description text according to the following schema:*

4. observe_mereology schema

Narration:

[t] ... narrative text on mereology type ...
[m] ... narrative text on mereology observer ...
[a] ... narrative text on mereology type constraints ...

Formalisation:

type
[t] MT[13]= $\mathscr{E}$(PI1,PI2,...,PIm)
value
[m] **obs_mereo**_P: P $\rightarrow$ MT
axiom [Well−formedness of Domain Mereologies]
[a] $\mathscr{A}$(MT)

[13] MT will be used several times in Sect. .

- *Here $\mathscr{E}$(PI1,PI2,...,PIm) is a type expression over possibly all unique identifier types of the domain description,*
- *and $\mathscr{A}$(MT) is a predicate over possibly all unique identifier types of the domain description.*
- *To write down the concrete type definition for MT requires a bit of analysis and thinking.*
- has_mereology *is a* **prerequisite prompt** *for* observe_mereology ■

Example 28 **Road Net Part Mereologies**:
We continue Example 17 on Slide 110 and Example 25 on Slide 136.

21 Links are connected to exactly two distinct hubs.

22 Hubs are connected to zero or more links.

23 For a given net the link and hub identifiers
of the mereology of hubs and links
must be those of links and hubs, respectively, of the net.

type

21. LM′ = HI-**set**, LM = {|his:HI-**set** · **card**(his)=2|}
22. HM = LI-**set**

value

21. **obs_mereo**_L: L → LM
22. **obs_mereo**_H: H → HM

axiom [Well−formedness of Road Nets, N]

23. ∀ n:N,l:L,h:H·
23. l ∈ **obs_part**_Ls(**obs_part**_LS(n))
23. ∧ h ∈ **obs_part**_Hs(**obs_part**_HS(n))
23. ⇒ **obs_mereo**_L(l) ⊆ ∪{**uid**_H(h) | h ∈ **obs_part**_Hs(**obs_part**_HS(n))}
23. ∧ **obs_mereo**_H(h) ⊆ ∪{**uid**_H(l) | l ∈ **obs_part**_Ls(**obs_part**_LS(n))} ■

Example 29 Pipeline Parts Mereology:

- We continue Example 24 on Slide 127.
- Pipeline units serve to conduct fluid or gaseous material.
- The flow of these occur in only one direction: from so-called input to so-called output.

24 Wells have exactly one connection to an output unit.

25 Pipes, pumps and valves have exactly one connection from an input unit and one connection to an output unit.

26 Forks have exactly one connection from an input unit and exactly two connections to distinct output units.

27 Joins have exactly two connections from distinct input units and one connection to an output unit.

28 Sinks have exactly one connection from an input unit.

29 Thus we model the mereology of a pipeline unit as a pair of disjoint sets of unique pipeline unit identifiers.

type
29. UM′=(UI-**set**×UI-**set**)
29. UM={|(iuis,ouis):UM′·iuis ∩ ouis={}|}
value
29. **obs_mereo**_U: UM
axiom [Well−formedness of Pipeline Systems, PLS (0)]
∀ pl:PL,u:U · u ∈ **obs_part**_Us(pl) ⇒
let (iuis,ouis)=**obs_mereo**_U(u) **in**
case (**card** iuis,**card** ouis) **of**
24. (0,1) → **is**_We(u),
25. (1,1) → **is**_Pi(u)∨**is**_Pu(u)∨**is**_Va(u),
26. (1,2) → **is**_Fo(u),
27. (2,1) → **is**_Jo(u),
28. (1,0) → **is**_Si(u), _ → **false**
end end ■

3.5.3. Formulation of Mereologies

- The `observe_mereology` domain descriptor, Slide 143,
  - may give the impression that the mereo type MT can be described
  - "at the point of issue" of the `observe_mereology` prompt.
  - Since the MT type expression may depend on any part sort
  - the mereo type MT can, for some domains,
  - "first" be described when all part sorts have been dealt with.

3.6. Part Attributes

- To recall: there are three sets of **internal qualities**:
  - unique part identifiers,
  - part mereology and
  - attributes.
- Unique part identifiers and part mereology
  are rather definite kinds of internal endurant qualities.
- Part attributes form more "free-wheeling" sets of **internal qualities**.

3.6.1. Inseparability of Attributes from Parts

- Parts are
  - typically recognised because of their spatial form
  - and are otherwise characterised by their intangible, but measurable attributes.
- We learned from our exposition of *formal concept analysis* that
  - a formal concept, that is, a type, consists of all the entities
  - which all have the same qualities.
- Thus removing a quality from an entity makes no sense:
  - the entity of that type
  - either becomes an entity of another type
  - or ceases to exist (i.e., becomes a non-entity) !

3.6.2. Attribute Quality and Attribute Value

- We distinguish between
  - an attribute, as a logical proposition, and
  - an attribute value, as a value in some value space.

Example 30 Attribute Propositions and Other Values:

- A particular street segment (i.e., a link), say ℓ,
  - satisfies the proposition (attribute) has_length, and
  - may then have value length 90 meter for that attribute.
- A particular road transport domain, δ,
  - has three immediate sub-parts: net, n, fleet, f, and monitor m;
  - typically nets `has_net_name` and `has_net_owner` proposition attributes
  - with, for example, `US Interstate Highway System` respectively `US Department of Transportation` as values for those attributes

■

3.6.3. Endurant Attributes: Types and Functions

- Let us recall that attributes cover qualities other than unique identifiers and mereology.
- Let us then consider that parts have one or more attributes.
  - These attributes are qualities
  - which help characterise "what it means" to be a part.
- Note that we expect every part to have at least one attribute.

Example 31 **Atomic Part Attributes**:

- Examples of attributes of atomic parts such as a human are:
  - *name,*
  - *gender,*
  - *birth-date,*
  - *birth-place,*
  - *nationality,*
  - *height,*
  - *weight,*
  - *eye colour,*
  - *hair colour,*

  etc.
- Examples of attributes of transport net links are:
  - *length,*
  - *location,*
  - *1 or 2-way link,*
  - *link condition,*

  etc. ■

Example 32 Composite Part Attributes:

- Examples of attributes of composite parts such as a road net are:
  - *owner,*
  - *public or private net,*
  - *free-way or toll road,*
  - *a map of the net,*

  etc.
- Examples of attributes of a group of people could be: *statistic distributions of*
  - *gender,*
  - *age,*
  - *income,*
  - *education,*
  - *nationality,*
  - *religion,*

  etc. ■

- We now assume that all parts have attributes.
- The question is now, in general, how many and, particularly, which.

Analysis Prompt 14 *attribute_names:*

- *The* **domain analysis prompt** *attribute_names*
  - *when applied to a part p*
  - *yields the set of names of its attribute types:*
  - *attribute_names(p):* $\{\eta A_1, \eta A_2, ..., \eta A_n\}$.
- η *is a type operator. Applied to a type A it yields is name*[14] ■

[14]Normally, in non-formula texts, type A is referred to by ηA. In formulas A denote a type, that is, a set of entities. Hence, when we wish to emphasize that we speak of the name of that type we use ηA. But often we omit the distinction

- We cannot automatically, that is, syntactically, guarantee that our domain descriptions secure that
  - the various attribute types
  - for an emerging part sort
  - denote disjoint sets of values.

  Therefore we must prove it.

3.6.3.1 The Attribute Value Observer

- The "built-in" description language operator
  - **attr**_A
- applies to parts, p:P, where ηA∈`attribute_names`(p).
- It yields the value of attribute A of p.

Domain Description Prompt 5 *observe_attributes*:

- *The domain analyser experiments, thinks and reflects about part attributes.*
- *That process is initated by the* **domain description prompt***:*
  - *observe_attributes.*
- *The result of that* **domain description prompt** *is that the domain analyser cum describer writes down the* attribute (sorts or) types and observers *domain description text according to the following schema:*

5. `observe_attributes` schema

Narration:

[t] ... narrative text on attribute sorts ...

[o] ... narrative text on attribute sort observers ...

[i] ... narrative text on attribute sort recognisers ...

[p] ... narrative text on attribute sort proof obligations ...

Formalisation:

type

[t] A_i $[1 \leq i \leq n]$

value

[o] **attr**_A_i:P→A_i $[1 \leq i \leq n]$

[i] **is**_A_i:(A_1|A_2|...|A_n)→**Bool** $[1 \leq i \leq n]$

proof obligation [Disjointness of Attribute Types]

[p] $\forall\ \delta:\Delta$

[p] **let** P be any part sort **in** [the Δ domain description]

[p] **let** a:(A_1|A_2|...|A_n) **in** **is**_A_i(a) $\neq$ **is**_A_j(a) **end end** $[i \neq j,\ 1 \leq i,j \leq n]$

- *The* **type** *(or rather sort) definitions:* A_1, A_2, ..., A_n, *inform us that the domain analyser has decided to focus on the distinctly named* A_1, A_2, ..., A_n *attributes.*
- *And the* **value** *clauses*
  - **attr**_A_1:$P \to A_1$,
  - **attr**_A_2:$P \to A_2$,
  - ...,
  - **attr**_A_n:$P \to A_n$

  are then "automatically" given:
  - *if a part,* p:P, *has an attribute* A_i
  - *then there is postulated, "by definition" [eureka] an attribute observer function* **attr**_A_i:$P \to A_i$ *etcetera* ■

- The fact that, for example, A_1, A_2, ..., A_n, are attributes of p:P, means that the propositions
  - has_attribute_$A_1(p)$,
    has_attribute_$A_2(p)$,
    ..., and
    has_attribute_$A_n(p)$

  holds.
- Thus the observer functions **attr**_A_1, **attr**_A_2, ..., **attr**_A_n
  - can be applied to p in P
  - and yield attribute values a_1:A_1, a_2:A_2, ..., a_n:A_n respectively.

Example 33 **Road Hub Attributes**: After some analysis a domain analyser may arrive at some interesting hub attributes:

30 hub state:
from which links (by reference) can one reach which links (by reference),

31 hub state space:
the set of all potential hub states that a hub may attain,

32 such that

a. the links referred to in the state are links of the hub mereology

b. and the state is in the state space.

33 Etcetera — i.e., there are other attributes not mentioned here.

```
type
30     HΣ = (LI×LI)-set
31     HΩ = HΣ-set
value
30     attr_HΣ:H→HΣ
31     attr_HΩ:H→HΩ
axiom [Well−formedness of Hub States, HΣ]
32     ∀ h:H · let hσ = attr_HΣ(h) in
32a.       {li,li′|li,li′:LI·(li,li′)∈ hσ}⊆obs_mereo_H(h)
32b.       ∧ hσ ∈ attr_HΩ(h)
32         end ■
```

Lecture 3 168–246

3.6.4. Attribute Categories

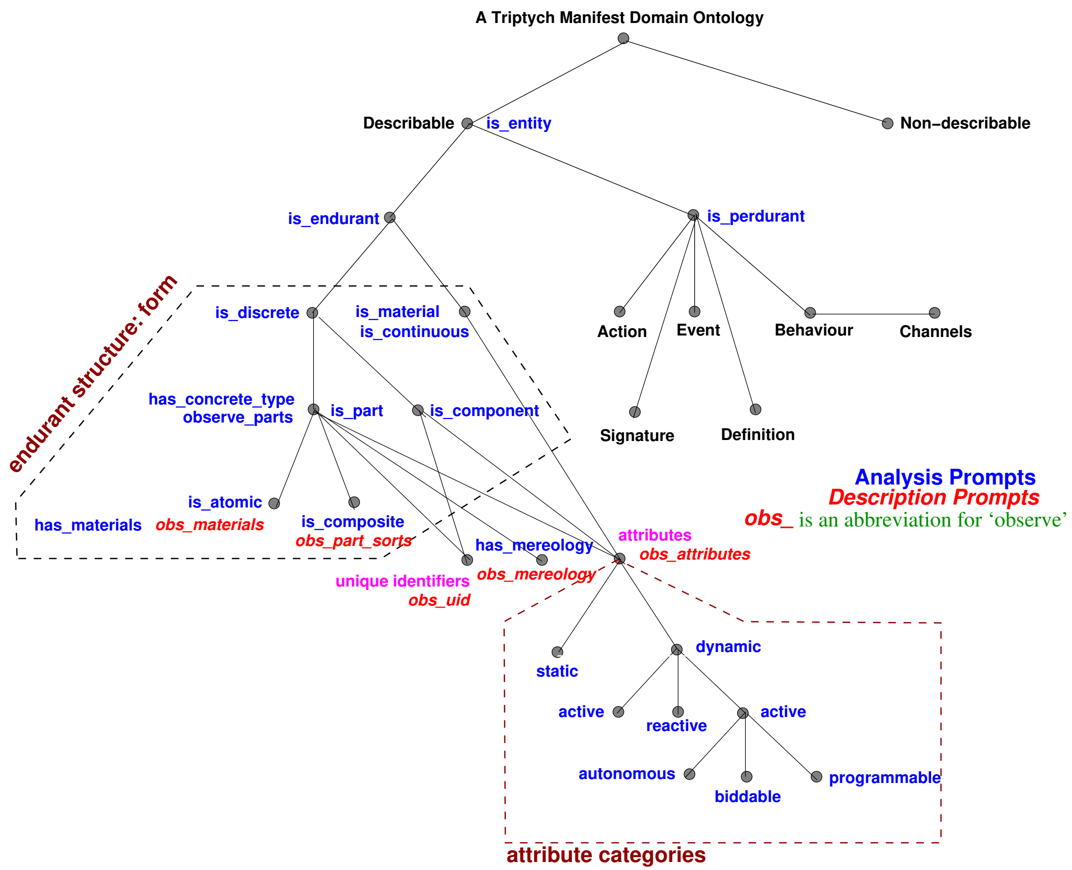

Figure 4: An Upper Ontology for Domains: **Attribute Categories**

- One can suggest a hierarchy of part attribute categories:
  - static or
  - dynamic values — and within the dynamic value category:
    - inert values or
    - reactive values or
    - active values — and within the dynamic active value category:
      - autonomous values or
      - biddable values or
      - programmable values.
- We now review these attribute value types.
  The review is based on [Jac95, M.A. Jackson].

Part attributes are either constant or varying, i.e., **static** or **dynamic** attributes.

- By a **static attribute**, a:A, `is_static_attribute`(a), we shall understand an attribute whose values
  - are constants,
  - i.e., cannot change.
- By a **dynamic attribute**, a:A, `is_dynamic_attribute`(a), we shall understand an attribute whose values
  - are variable,
  - i.e., can change.

Dynamic attributes are either inert, reactive or active attributes.

- By an **inert attribute**, a:A, `is_inert_attribute`(a),
  we shall understand a dynamic attribute whose values
  - only change as the result of external stimuli where
  - these stimuli prescribe properties of these new values.
- By a **reactive attribute**, a:A, `is_reactive_attribute`(a),
  we shall understand a dynamic attribute whose values,
  - if they vary, change value in response to
  - the change of other attribute values.
- By an **active attribute**, a:A, `is_active_attribute`(a),
  we shall understand a dynamic attribute whose values
  - change (also) of its own volition.

Active attributes are either autonomous, biddable or programmable attributes.

- By an **autonomous attribute**, a:A, is_autonomous_attribute(a), we shall understand a dynamic active attribute
  - whose values change value only "on their own volition".[15]
- By a **biddable attribute**, a:A, is_biddable_attribute(a), (of a part) we shall understand a dynamic active attribute whose values
  - are prescribed
  - but may fail to be observed as such.
- By a **programmable attribute**, a:A, is_programmable_attribute(a), we shall understand a dynamic active attribute whose values
  - can be prescribed.

[15] The values of an autonomous attributes are a "law onto themselves and their surroundings".

Example 34 Static and Dynamic Attributes:

- Link lengths can be considered **static.**
- Buses (i.e., vehicles) have a *timetable* attribute which is **inert**, i.e., can change, only when the bus company decides so.
- The weather can be considered **autonomous.**
- Pipeline valve units include the two attributes of *valve opening* (`open`, `close`) and *internal flow* (measured, say `gallons` per `second`).
  - The valve opening attribute is of the **biddable** attribute category.
  - The flow attribute is **reactive** (flow changes with valve opening/closing).
- Hub states (red, yellow, green) can be considered **biddable:** one can "try" set the signals but the electro-mechanics may fail.
- Bus companies **program** their own timetables, i.e., bus company timetables are **programmable** — are computers ■

- **External Attributes:** By an **external attribute** we shall understand
  - ◈ a dynamic attribute
  - ◈ which is not a biddable or a programmable attribute ■
- The idea of external attributes is this:
  - ◈ They are the attributes whose values are set by factors "outside" the part of which they are an attribute.
  - ◈ In contrast, the programmable (and biddable) attributes have their values determininistically (non-deterministically) set by the part [behaviour] of which they are an attribute.
- **Controllable Attributes:** By a **controllable attribute** we shall understand
  - ◈ either a biddable or a programmable attribute ■

• Figure 5 captures an attribute value ontology.

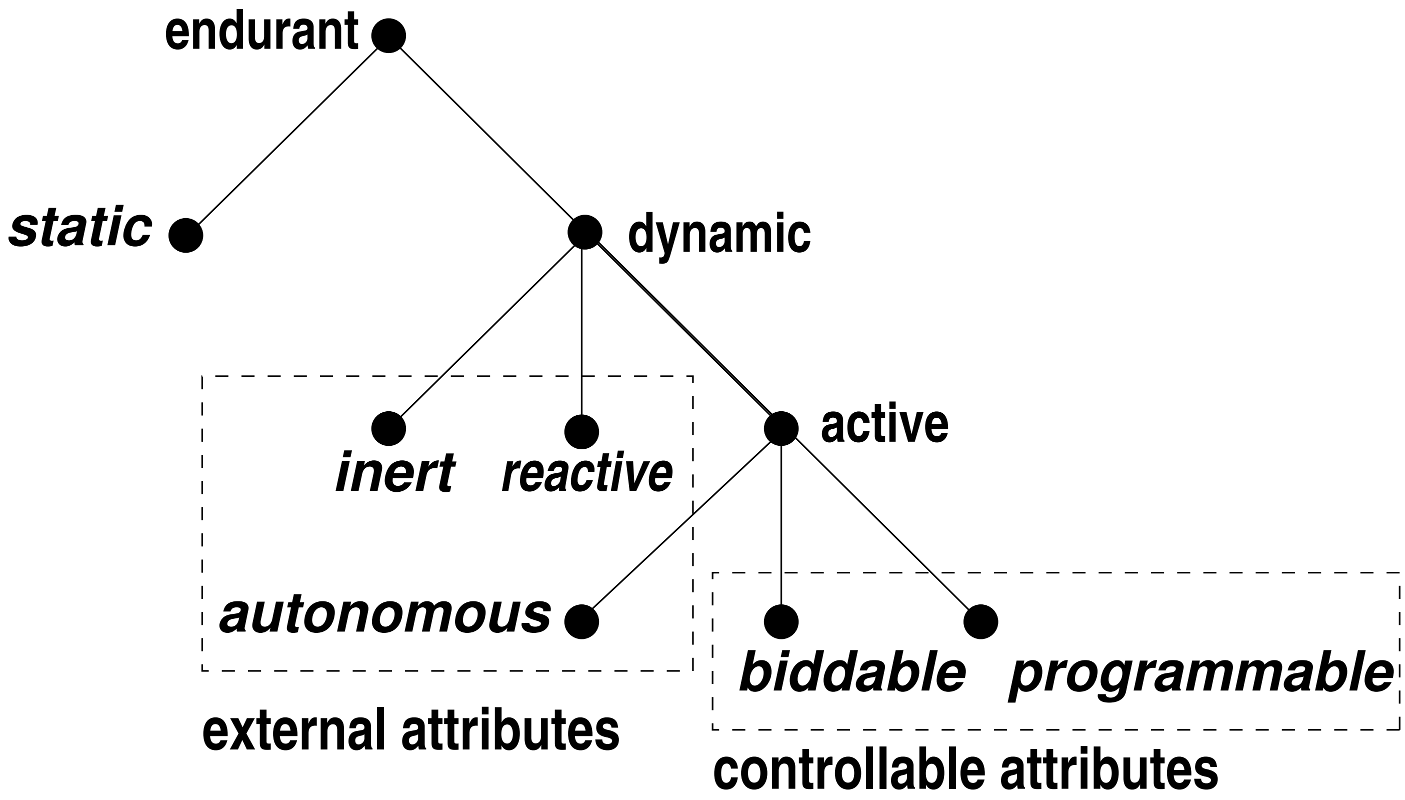

Figure 5: Attribute Value Ontology

3.6.5. Access to Attribute Values

- In an action, event or a behaviour description
  - **static values** of parts, p, (say of type A)
  - can be "copied", **attr**_A(p),
  - and still retain their (static) value.
- But, for action, event or behaviour descriptions,
  - external dynamic values of parts, p,
  - cannot be "copied", but **attr**_A(p) must be "performed"
  - every time they are needed.
- That is:
  - **static values** require at most one domain access,
  - whereas external attribute values require repeated domain accesses.
- We shall return to the issue of attribute value access in Sect. 1.3.8.

3.6.6. Event Values

- Among the external attribute values we observe a new kind of value: the **event values.**
  - ⋄ We may optionally ascribe ordinarily typed, say A, values, a:A, with **event attribute**s.
  - ⋄ By an **event attribute** we shall understand
    - ◎ an attribute whose values are
      - ∗ either ”nil” ([f]or “absent”),
      - ∗ or are some more definite value (a:A) ■
  - ⋄ Event values *occur* instantaneously.
    - ◎ They can be thought of as the raising of a signal
    - ◎ followed immediately by the lowering of that signal.

Example 35 Event Attributes:

- (i) The passing of a vehicle past a tollgate is an event.
  - It occurs at a usually unpredictable time.
  - It otherwise “carries” no specific value.
- (ii) The identification of a vehicle by a tollgate sensor is an event.
  - It occurs at a usually unpredictable time.
  - It specifically “carries” a vehicle identifier value ■

- Event attributes are not to be confused with event perdurants.
- External attributes are either event attributes or are not.
- More on access to event attribute values in Sect. 4.7.4 [as from Slide 243].

3.6.7. Shared Attributes

- Normally part attributes of different part sorts are distinctly named.
- If, however, observe_attributes(p_{ik}:P_i) and observe_attributes($p_{j\ell}$:P_j),
  - for any two distinct part sorts, P_i and P_j, of a domain,
  - "discovers" identically named attributes, say A,
  - then we say that parts p_i:P_i and p_j:P_j **share** attribute A.
  - that is, that a:**attr**_A(p_i) (and a′:**attr**_A(p_j)) is a **shared attribute**
  - (with a=a′ always (□) holding).

Attribute Naming:

- Thus the domain describer has to exert great care when naming attribute types.
  - If P_i and P_j are two distinct types of a domain,
  - then if and only if an attribute of P_i is to be shared with an attribute of P_j
  - that attribute must be identically named in the description of P_i and P_j and
  - otherwise the attribute names of P_i and P_j must be distinct.

Example 36. **Shared Attributes.** Examples of shared attributes:

- Bus timetable attributes have the same value
  as the fleet timetable attribute.
- A link incident upon or emanating from a hub
  shares the connection between that link and the hub as an attribute.
- Two pipeline units[16], p_i with unique identifier π_i,
  and p_j with unique identifier π_j, that are connected,
  such that an outlet marked π_j of p_i
  "feeds into" inlet marked π_i of p_j,
  are said to share the connection (modeled by, e.g., $\{(\pi_i, \pi_j)\}$) ■

[16] See Example 29 on Slide 147

Example 37 Shared Timetables:

- The fleet and vehicles of Example 17 on Slide 110 and Example 18 on Slide 117 is that of a bus company.

34 From the fleet and from the vehicles we observe unique identifiers.

35 Every bus mereology records the same one unique fleet identifier.

36 The fleet mereology records the set of all unique bus identifiers.

37 A bus timetable is a shared fleet and bus attribute.

type
34. FI, VI, BT
value
34. **uid**_F: F $\rightarrow$ FI
34. **uid**_V: V $\rightarrow$ VI
35. **obs_mereo**_F: F $\rightarrow$ VI-**set**
36. **obs_mereo**_V: V $\rightarrow$ FI
37. **attr**_BT: (F|V) $\rightarrow$ BT
axiom
$\Box$ $\forall$ f:F $\Rightarrow$
$\forall$ v:V $\cdot$ v $\in$ **obs_part**_Vs(**obs_part**_VC(f)) $\cdot$ **attr**_BT(f) = **attr**_BT(v)

- The simple identical attribute name-sharing first outlined above may be generalised.
  - If P_i and P_j are two distinct types of a domain,
  - then if an attribute, A, of P_i
    is to be shared with an attribute, B, of P_j,
  - attribute B must be expressed in terms of A.

3.7. Components

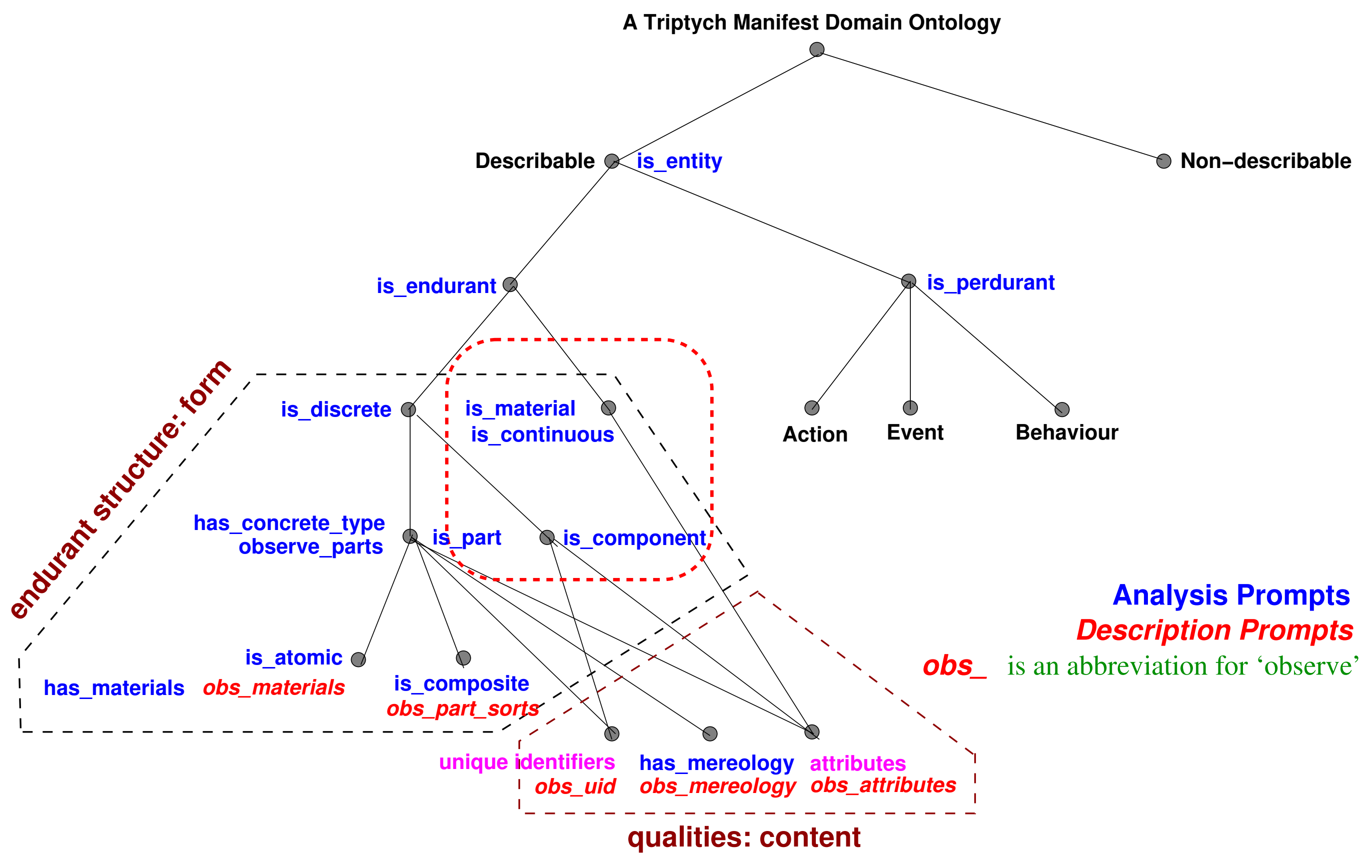

Figure 6: An Upper Ontology for Domains — **Components and Materials**

- Components are
  - discrete endurants
  - which the domain analyser & describer has chosen to **not** endow
  - with **mereology**
  - but with **unique identifiers** and **attributes**.
- We associate components with atomic parts,
  - such that an atomic part may or may not have components,
  - and if they potentially have components,
  - they are all of the same sort
  - and there may be a finite set of zero, one or more such components.

Example 38 **Parts and Components**:

- We observe components as associated with atomic parts:
  - The contents, that is, the collection of zero, one or more boxes, of a container are the components of the container part.
  - Conveyor belts transport machine assembly units and these are thus considered the components of the conveyor belt ■

- We now complement the `observe_part_sorts` (of earlier).
- We assume, without loss of generality, that only atomic parts may contain components.
- Let *p*:*P* be some atomic part.

Analysis Prompt 15 *has_components:*

- *The* **domain analysis prompt***:*
  - *has_components*(*p*)
- *yields* **true** *if atomic part p may contain zero, one or more components otherwise false* ■

- Let us assume that parts $p{:}P$ embody components of sort K.

Domain Description Prompt 6 *observe_component_sort*:

- *The* **domain description prompt**:
  - *observe_component_sort_P(p)*
  - *yields the* component sorts *and* component sort observer *domain description text according to the following schema –*
  - *whether or not the actual part p contains any components:*

6. observe_component_sort_P schema

Narration:

[s] ... narrative text on component sort ...

[o] ... narrative text on component observer ...

Formalisation:

type

[s] K

value

[o] **obs_comp**_K: P → K-**set**

Example 39 **Container Components**:
We continue Example 19 on Slide 122.

38 When we apply `obs_component_sorts_C` to any container c:C we obtain

a. a type clause stating the sort of the various components, ck:CK, of a container, and

b. the component observer function signature.

type
38a. CK
value
38b. **obs_comp**_CKs: C → CK-**set** ■

- We have presented one way of tackling the issue of describing components.
  - There are other ways.
  - We leave those ‘other ways’ to the reader.
- We are not going to suggest techniques and tools for analysing, let alone ascribing qualities to components.
  - We suggest that conventional
    abstract modeling techniques
    and tools be applied.

3.8. Materials

- Continuous endurants (i.e., **materials**) are entities, *m*, which satisfy:
  - `is_material`(m) $\equiv$ `is_endurant`(m)$\wedge$`is_continuous`(m)

Example 40 Parts and Materials:

- We observe materials as associated with atomic parts:
  - Thus liquid or gaseous materials are observed in pipeline units ■

- We shall in this seminar not cover
  the case of parts being immersed in materials.

- We assume, without loss of generality, that only atomic parts may contain materials.
- Let *p*:*P* be some atomic part.

Analysis Prompt 16 *has_materials:*

- *The* **domain analysis prompt***:*
  - *has_materials(p)*
- *yields* **true** *if the atomic part p:P potentially may contain materials otherwise false* ■

- Let us assume that parts p:P embody materials of sorts $\{M_1, M_2, \ldots, M_n\}$.
- Since we cannot automatically guarantee that our domain descriptions secure that
  - each M_i ($[1 \leq i \leq$ n$]$)
  - denotes disjoint sets of entities

  we must prove it.

Domain Description Prompt 7 *observe_material_sorts_P*:

- *The* **domain description prompt**:
  - *observe_material_sorts_P(e)*

  yields the material sort and material sort observer domain description text according to the following schema whether or not part p actually contains materials:

7. observe_material_sorts_P schema

Narration:

[s] ... narrative text on material sort ...

[o] ... narrative text on material sort observer ...

Formalisation:

type

[s] M

value

[o] **obs_mat**_M: P → M

Example 41 **Pipeline Material**: We continue Example 24 on Slide 127 and Example 29 on Slide 147.

39 When we apply obs_material_sorts_U to any unit u:U we obtain

a. a type clause stating the material sort LoG
for some further undefined liquid or gaseous material, and

b. a material observer function signature.

type

39a. LoG

value

39b. **obs_mat**_LoG: U → LoG

has_materials(u) is a prerequisite for **obs_mat**_LoG(u) ■

3.8.1. Materials-related Part Attributes

- It seems that the "interplay" between parts and materials
  - is an area where domain analysis
  - in the sense of this paper
  - is relevant.

Example 42 Pipeline Material Flow:
We continue Examples 24, 29 and 41.

- Let us postulate a[n attribute] sort Flow.
- We now wish to examine the flow of liquid (or gaseous) material in pipeline units.
- We use two types

40 **type** F, L.

- Productive flow, F, and wasteful leak, L,
  is measured, for example, in terms of volume of material per second.
- We then postulate the following unit attributes
  - "measured" at the point of in- or out-flow
  - or in the interior of a unit.

41 current flow of material into a unit input connector,

42 maximum flow of material into a unit input connector while maintaining laminar flow,

43 current flow of material out of a unit output connector,

44 maximum flow of material out of a unit output connector while maintaining laminar flow,

45 current leak of material at a unit input connector,

46 maximum guaranteed leak of material at a unit input connector,

47 current leak of material at a unit input connector,

48 maximum guaranteed leak of material at a unit input connector,

49 current leak of material from "within" a unit, and

50 maximum guaranteed leak of material from "within" a unit.

type
40. F, L
value
41. **attr**_cur_iF: U → UI → F
42. **attr**_max_iF: U → UI → F
43. **attr**_cur_oF: U → UI → F
44. **attr**_max_oF: U → UI → F
45. **attr**_cur_iL: U → UI → L
46. **attr**_max_iL: U → UI → L
47. **attr**_cur_oL: U → UI → L
48. **attr**_max_oL: U → UI → L
49. **attr**_cur_L: U → L
50. **attr**_max_L: U → L

- The maximum flow attributes are static attributes and are typically provided by the manufacturer as indicators of flows below which laminar flow can be expected.
- The current flow attributes may be considered either reactive or biddable attributes ■

3.8.2. Laws of Material Flows and Leaks

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

Example 43 **Pipelines: Intra Unit Flow and Leak Law**:

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

52 The flows into a unit equal

a. the leak at the inputs

b. plus the leak within the unit

c. plus the flows out of the unit

d. plus the leaks at the outputs.

```
axiom [Well−formedness of Pipeline Systems, PLS (1)]
51.  ∀ pls:PLS,b:B\We\Si,u:U ·
51.       b ∈ obs_part_Bs(pls)∧u=obs_part_U(b)⇒
51.       let (iuis,ouis) = obs_mereo_U(u) in
52.       sum_cur_iF(u)(iuis) =
52a..         sum_cur_iL(u)(iuis)
52b..       ⊕ attr_cur_L(u)
52c..       ⊕ sum_cur_oF(u)(ouis)
52d..       ⊕ sum_cur_oL(u)(ouis)
51.       end
```

53 The sum_cur_iF (cf. Item 52) sums current input flows over all input connectors.

54 The sum_cur_iL (cf. Item 52a.) sums current input leaks over all input connectors.

55 The sum_cur_oF (cf. Item 52c.) sums current output flows over all output connectors.

56 The sum_cur_oL (cf. Item 52d.) sums current output leaks over all output connectors.

53. sum_cur_iF: U → UI-**set** →F
53. sum_cur_iF(u)(iuis) ≡ ⊕ {**attr**_cur_iF(u)(ui)|ui:UI·ui ∈ iuis}
54. sum_cur_iL: U → UI-**set** → L
54. sum_cur_iL(u)(iuis) ≡ ⊕ {**attr**_cur_iL(u)(ui)|ui:UI·ui ∈ iuis}
55. sum_cur_oF: U → UI-**set** → F
55. sum_cur_oF(u)(ouis) ≡ ⊕ {**attr**_cur_iF(u)(ui)|ui:UI·ui ∈ ouis}
56. sum_cur_oL: U → UI-**set** → L
56. sum_cur_oL(u)(ouis) ≡ ⊕ {**attr**_cur_iL(u)(ui)|ui:UI·ui ∈ ouis}
⊕: (F|L) × (F|L) → F ■

Example 44 **Pipelines: Inter Unit Flow and Leak Law**:

57 For every pair of connected units of a pipeline system the following law apply:

  a. the flow out of a unit directed at another unit minus the leak at that output connector

  b. equals the flow into that other unit at the connector from the given unit plus the leak at that connector.

```
   axiom [Well−formedness of Pipeline Systems, PLS (2)]
57.     ∀ pls:PLS,b,b′:B,u,u′:U·
57.        {b,b′}⊆obs_part_Bs(pls)∧b≠b′∧u′=obs_part_U(b′)
57.        ∧ let (iuis,ouis)=obs_mereo_U(u),(iuis′,ouis′)=obs_mereo_U(u′),
57.              ui=uid_U(u),ui′=uid_U(u′) in
57.           ui ∈ iuis ∧ ui′ ∈ ouis′ ⇒
57a..             attr_cur_oF(u′)(ui′) − attr_leak_oF(u′)(ui′)
57b..          = attr_cur_iF(u)(ui) + attr_leak_iF(u)(ui)
57.           end
57.     comment: b′ precedes b ■
```

- From the above two laws one can prove the **theorem:**
  - what is pumped from the wells equals
  - what is leaked from the systems plus what is output to the sinks.

3.9. "No Junk, No Confusion"

- Domain descriptions are, as we have already shown, formulated,
  - both informally
  - and formally,

  by means of abstract types,
  - that is, by sorts
  - for which no concrete models are usually given.
- Sorts are made to denote
  - possibly empty,
  - possibly infinite,
  - rarely singleton,
  - sets of entities on the basis of the qualities defined for these sorts, whether external or internal.

- By **junk** we shall understand
  - that the domain description
  - unintentionally denotes undesired entities.
- By **confusion** we shall understand
  - that the domain description
  - unintentionally have two or more identifications
  - of the same entity or type.
- The question is
  - *can we formulate a [formal] domain description*
  - *such that it does not denote junk or confusion* ?
- The short answer to this is no !

- So, since one naturally wishes "no junk, no confusion" what does one do ?
- The answer to that is
  - *one proceeds with great care !*
- To avoid **junk** we have stated a number of **sort well-formedness axiom**s, for example:[17]
  - Slide 136 for *wf* links and hubs,
  - Slide 143 for *wf* road net mereologies,
  - Slide 146 for *wf* pipeline mereologies,
  - Slide 166 for *wf* hub states,
  - Slide 204 for *wf* pipeline systems,
  - Slide 206 for *wf* pipeline systems,
- To avoid **confusion** we have stated a number of **proof obligation**s:
  - Slide 109 for *Disjointness of Part Sorts* and
  - Slide 162 for *Disjointness of Attribute Types.*

[17] Let *wf* abbreviate *well-formed.*

3.10. Discussion of Endurants

- In Sect. 4.2.2 a "depth-first" search for part sorts was hinted at, but only in the sequence of examples, as given.
- That sequence of examples essentially expressed
  - that we discover domains epistemologically[18]
  - but understand them ontologically.[19]
- The Danish philosopher Søren Kirkegaard (1813–1855) expressed it this way:
  - *Life is lived forwards,*
  - *but is understood backwards.*

[18] **Epistemology**: the theory of knowledge, especially with regard to its methods, validity, and scope. Epistemology is the investigation of what distinguishes justified belief from opinion.

[19] **Ontology**: the branch of metaphysics dealing with the nature of being.

- The presentation of the of the **domain analysis prompt**s and the **domain description prompt**s results in domain descriptions which are ontological.
- The "depth-first" search recognizes the epistemological nature of bringing about understanding.
- This "depth-first" search
  - that ends with the analysis of atomic part sorts
  - can be guided, i.e., hastened (shortened),
  - by postulating composite sorts
  - that "correspond" to vernacular nouns:
  - everyday nouns that stand for classes of endurants.

4. Perdurants

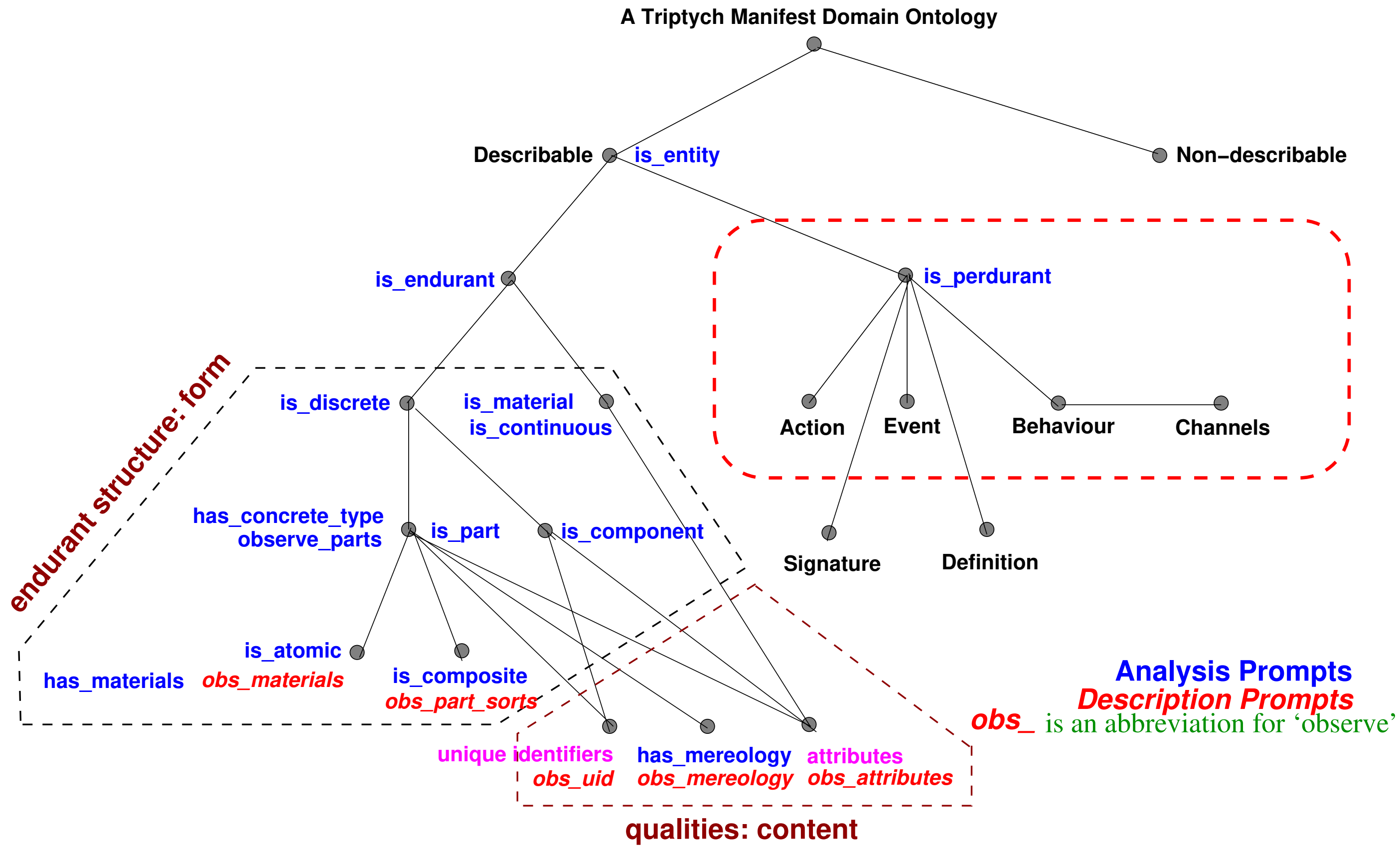

Figure 7: An Upper Ontology for Domains — **Perdurants**

- As regards **perdurants**, we shall **not** present
  - a set of **domain analysis prompt**s and
  - a set of **domain description prompt**s
  - leading to description language, i.e., RSL texts
  - describing perdurant entities.
- The reason for giving this albeit cursory overview of perdurants
  - is that we can justify our detailed study of endurants,
    - their part and sub parts,
    - their unique identifiers, mereology and attributes.

- This justification is manifested
  - (i) in expressing the types of signatures,
  - (ii) in basing behaviours on parts,
  - (iii) in basing the for need for
    CSP-oriented inter-behaviour communications
    on shared part attributes,
  - (iv) in indexing behaviours as are parts, i.e., on unique identifiers,

  and
  - (v) in directing inter-behaviour communications across channel arrays indexed as per the mereology of the part behaviours.

- These are all notions related to endurants
  and are now justified by their use in describing perdurants.
- Perdurants can perhaps best be explained in terms of
  - a notion of state and
  - a notion of time.
- We shall, in this seminar, not detail notions of time.

4.1. States

Definition 11 **State:** *By a* **state** *we shall understand*

- *any collection of* **parts**
- *each of which has*
- *at least one dynamic attribute*
- *or* `has_components` *or* `has_materials` ■

Example 45 States:

- A road hub can be a state, cf. Hub State, HΣ, Example 33 on Slide 165.
- A road net can be a state – since its hubs can be.
- Container stowage areas, CSA, Example 19 on Slide 122, of container vessels and container terminal ports can be states as containers can be removed from and put on top of container stacks.
- Pipeline pipes can be states as they potentially carry material.
- Conveyor belts can be states as they may carry components ■

4.2. Actions, Events and Behaviours

- To us perdurants are further, pragmatically, analysed into
  - actions,
  - events, and
  - behaviours.
- We shall define these terms below.
- Common to all of them is that they potentially change a state.
- Actions and events are here considered atomic perdurants.
- For behaviours we distinguish between
  - discrete and
  - continuous

  behaviours.

4.2.1. Time Considerations

- We shall, without loss of generality, assume
  - that actions and events are atomic
  - and that behaviours are composite.
- Atomic perdurants may "occur" during some time interval,
  - but we omit consideration of and concern
    for what actually goes on during such an interval.
- Composite perdurants can be analysed into "constituent"
  - actions,
  - events and
  - "sub-behaviours".
- We shall also omit consideration of temporal properties of behaviours.

- Instead we shall refer to two seminal monographs:
  - Specifying Systems [Leslie Lamport, 2002] and
  - Duration Calculus: A Formal Approach to Real-Time Systems [Zhou ChaoChen and Michael Reichhardt Hansen, 2004] (and [Bjø06, Chapter 15]).

- For a seminal book on "time in computing" we refer to the eclectic [FMMR12, Mandrioli et al., 2012].
- And for seminal book on time at the epistemology level we refer to [van91, J. van Benthem, 1991].

4.2.2. Actors

Definition 12 Actor: *By an* **actor** *we shall understand*

- *something that is capable of initiating and/or carrying out*
  - *actions,*
  - *events or*
  - *behaviours* ■
- We shall, in principle, associate an actor with each part.
  - These actors will be described as behaviours.
  - These behaviours evolve around a state.
  - The state is
    - the set of qualities,
      in particular the dynamic attributes,
      of the associated parts
    - and/or any possible components or materials of the parts.

Example 46 **Actors**: We refer to the road transport and the pipeline systems examples of earlier.

- The fleet, each vehicle and the road management of the *Transportation System* of Example 17 on Slide 110 can be considered an actor;
- so can the net and its links and hubs.
- The pipeline monitor and each pipeline unit of the *Pipeline System*, Example 24 on Slide 127 and Examples 24 on Slide 127 and 29 on Slide 147 will be considered actors ■

4.2.3. Parts, Attributes and Behaviours

- Example 46 on the preceding slide focused on what shall soon become a major relation within domains:
  - that of parts being also considered actors,
  - or more specifically, being also considered to be behaviours.

Example 47 Parts, Attributes and Behaviours:

- Consider the term 'train'.
- It has several possible "meanings".
  - the train as a part, viz., as standing on a platform;
  - the train as listed in a timetable (an attribute of a transport system part),
  - the train as a behaviour: speeding down the rail track ■

4.3. Discrete Actions

Definition 13 Discrete Action:
By a **discrete action** *[WS12, Wilson and Shpall] we shall understand*

- *a foreseeable thing*
- *which deliberately*
- *potentially changes a well-formed state, in one step,*
- *usually into another, still well-formed state,*
- *and for which an actor can be made responsible* ■

- An action is what happens when a function invocation changes, or potentially changes a state.

Example 48 Road Net Actions:

- Examples of *Road Net* actions initiated by the net actor are:
  - insertion of hubs,
  - insertion of links,
  - removal of hubs,
  - removal of links,
  - setting of hub states.
- Examples of *Traffic System* actions initiated by vehicle actors are:
  - moving a vehicle along a link,
  - stopping a vehicle,
  - starting a vehicle,
  - entering a hub and
  - leaving a hub ■

4.4. Discrete Events

- In the Bergen lectures I shall skip treatment of events.

4.5. Discrete Behaviours

Definition 14 Discrete Behaviour:
By a **discrete behaviour** *we shall understand*

- *a set of sequences of potentially interacting sets of discrete*
  - *actions,*
  - *events and*
  - *behaviours* ■

Example 49 Behaviours:

- (i) Road Nets: A sequence of hub and link insertions and removals, link disappearances, etc.
- (ii) Road Traffic: A sequence of movements of vehicles along links, entering, circling and leaving hubs, crashing of vehicles, etc.
- (iii) Pipelines: A sequence of pipeline pump and valve openings and closings, and failures to do so (events), etc.
- (iv) Container Vessels and Ports: Concurrent sequences of movements (by cranes) of containers from vessel to port (unloading), with sequences of movements (by cranes) from port to vessel (loading), with dropping of containers by cranes, etcetera ■

4.5.1. Channels and Communication

- Behaviours
  - sometimes synchronise
  - and usually communicate.
- We use the CSP [Hoa85] notation (adopted by RSL) to introduce and model behaviour communication.
  - Communication is abstracted as
    - the sending (ch ! m) and
    - receipt (ch ?)
    - of messages, m:M,
    - over channels, ch.

type M
channel ch:M

◇ Communication

- between (unique identifier) indexed behaviours
- have their channels modeled as similarly indexed channels:

out: ch[idx]!m
in: ch[idx]?
channel {ch[ide]:M|ide:IDE}

where IDE typically is some type expression
over unique identitifer types.

4.5.2. Relations Between Attribute Sharing and Channels

- We shall now interpret
  - the syntactic notion of attribute sharing with
  - the semantic notion of channels.
- This is in line with the above-hinted interpretation of
  - parts with behaviours, and, as we shall soon see,
  - part attributes with behaviour states.

- Thus, for every pair of parts, p_{ik}:P_i and $p_{j\ell}$:P_j, of distinct sorts, P_i and P_j which share attribute values in A
  - we are going to associate a channel.
    - If there is only one pair of parts, p_{ik}:P_i and $p_{j\ell}$:P_j, of these sorts, then we associate just a simple channel, say attr_A_ch$_{P_i,P_j}$, with the shared attribute.

      channel attr_A_ch$_{P_i,P_j}$:A.
    - If there is only one part, p_i:P_i, but a definite set of parts p_{jk}:P_j, with shared attributes, then we associate a *vector* of channels with the shared attribute.
      - Let $\{p_{j1}, p_{j2}, ..., p_{jn}\}$ be all the parts of the domain sort P_j.
      - Then $uids : \{\pi_{p_{j1}}, \pi_{p_{j2}}, ..., \pi_{p_{jn}}\}$ is the set of their unique identifiers.
      - Now a schematic channel array declaration can be suggested:

        channel {attr_A_ch[{π_i,π_j}]:A|π_i=**uid**_P_i(p_i)$\wedge\pi_j \in$ uids}.

Example 50 Bus System Channels:

- We extend Examples 17 on Slide 110.
- We consider the fleet and the vehicles to be behaviours.

58 We assume some transportation system, δ. From that system we observe

59 the fleet and

60 the vehicles.

61 The fleet to vehicle channel array is indexed by the 2-element sets of the unique fleet identifier and the unique vehicle identifiers. We consider bus timetables to be the only message communicated between the fleet and the vehicle behaviours.

value

58. δ:Δ,

59. f:F = **obs_part**_F(δ),

60. vs:V-**set** = **obs_part**_Vs(**obs_part**_VC((**obs_part**_F(δ))))

channel

61. {attr_BT_ch[{**uid**_F(f),**uid**_V(v)}]|v:V·v ∈ vs}:BT ■

4.6. Continuous Behaviours

- By a **continuous behaviour** we shall understand
  - a **continuous time**
  - sequence of **state changes**.
- We shall not go into what may cause these **state changes**.

Example 51 **Flow in Pipelines**:

- We refer to Examples 29, 41, 42, 43 and 44.
- Let us assume that oil is the (only) material of the pipeline units.
- Let us assume that there is a sufficient volume of oil in the pipeline units leading up to a pump.
- Let us assume that the pipeline units leading from the pump (especially valves and pumps) are all open for oil flow.
- Whether or not that oil is flowing, if the pump is pumping (with a sufficient head) then there will be oil flowing from the pump outlet into adjacent pipeline units ■

- To describe the flow of material (say in pipelines) requires knowledge about a number of material attributes — not all of which have been covered in the above-mentioned examples.
- To express flows one resorts to the mathematics of fluid-dynamics using such second order differential equations as first derived by Bernoulli (1700–1782) and Navier–Stokes (1785–1836 and 1819–1903).
- There is, as yet, no notation that can serve to integrate formal descriptions (like those of `Alloy, B, The B Method,RSL,VDM` or `Z`) with first, let alone second order differential equations. But some progress has been made [LWZ13, ZWZ13] since [WYZ94].

4.7. Attribute Value Access

- We distinguish between four kinds of attributes:
  - the **static attribute**s which are those whose values are fixed, i.e., does not change,
  - the **programmable attribute**s or **biddable attribute**s, i.e., the **controllable attribute**s, which are those dynamic values are exclusively set by part processes, and
  - the remaining **dynamic attribute**s which here, technically speaking, are seen as separate **external processes**s.
  - The **event attribute**s are those external attributes whose value occur for an instant of time.

4.7.1. Access to Static Attribute Values

- The **static attributes** can be "copied", **attr**_A(p), and retain their values.

4.7.2. Access to External Attribute Values

- By the **external attributes**, to repeat,
  - we shall understand the
    - inert, the
    - autonomous and the
    - reactive

  attributes ■

62 Let ξA be the set of names, ηA, of all **external attributes**.

63 Each **external attribute**, A, is seen as an individual behaviour, each "accessible" by means of unique channel, attr_A_ch.

64 External attribute values are then the value, a, of, i.e., accessed by, the input, attr_A_ch ?.

62. **value** ξA = {ηA|A is any external attribute name}
63. **channel** {attr_A_ch:A | ηA $\in$ ξA}
64. **value** a = attr_A_ch ?

- We shall omit the η prefix in actual descriptions.
- The choice of representing external attribute values as CSP processes[20] is a technical one.

[20]— not to be confused with domain behaviours

4.7.3. Access to Controllable Attribute Values

- The **controllable attributes** are treated as function arguments.
- This is a technical choice. It is motivated as follows.
  - We find that
    - these values are a function of other part attribute values, including at least one controllable attribute value, and
    - that the values are set (i.e., updated) by part behaviours.
  - That is, to each part, whether atomic or composite, we associate a behaviour.
  - That behaviour is (to be) described as we describe functions.
  - These functions (normally) *"go on forever"*.
  - Therefore these functions are described basically by a "tail" recursive definition:

$$\textbf{value}\ \ \mathsf{f}\colon\ \mathsf{Arg} \rightarrow \mathsf{Arg};\ \ \mathsf{f(a)} \equiv (\ldots\ \textbf{let}\ \mathsf{a}' = \mathscr{F}(\ldots)(\mathsf{a})\ \textbf{in}\ \mathsf{f(a')}\ \textbf{end})$$

  - where $\mathscr{F}$ is some expression based on values defined within the function definition body of f and on f's "input" argument a, and
  - where a can be seen as a **controllable attribute**.

4.7.4. Access to Event Values

- Event values reflect a stage change in a part behaviour.
  - We therefore model events as messages
  - communicated over a channel, **attr_A_ch**,
  - that is, **attr_A_ch ! a**,
  - where **A** is the event attribute, i.e., message type.
  - Thus fulfillment of **attr_A_ch ?** expresses
    - both that the event has taken place
    - and its value, if relevant.
  - Example 55 on Slide 277 illustrates the concept of event attributes and event values.

4.8. Perdurant Signatures and Definitions

- We shall treat perdurants as function invocations.
- In our cursory overview of perdurants
  - we shall focus on one perdurant quality:
  - function signatures.

Definition 15 Function Signature:
By a **function signature** *we shall understand*

- *a* **function name** *and*
- *a function type expression* ■

Definition 16 Function Type Expression:
By a **function type expression** *we shall understand*

- *a pair of* **type expressions***.*
- *separated by a function type constructor*
  - *either* $\rightarrow$ *(total function)*
  - *or* $\stackrel{\sim}{\rightarrow}$ *(partial function)* ■

- The **type expressions** are
  - part or
  - material or
  - component
  - sort or type, or
  - attribute type

  names,
- but may, occasionally be expressions over respective type names involving

  -set, $\times$, *, $\overrightarrow{m}$ and $|$

  type constructors.

Lecture 4 247–299

4.9. Action Signatures and Definitions

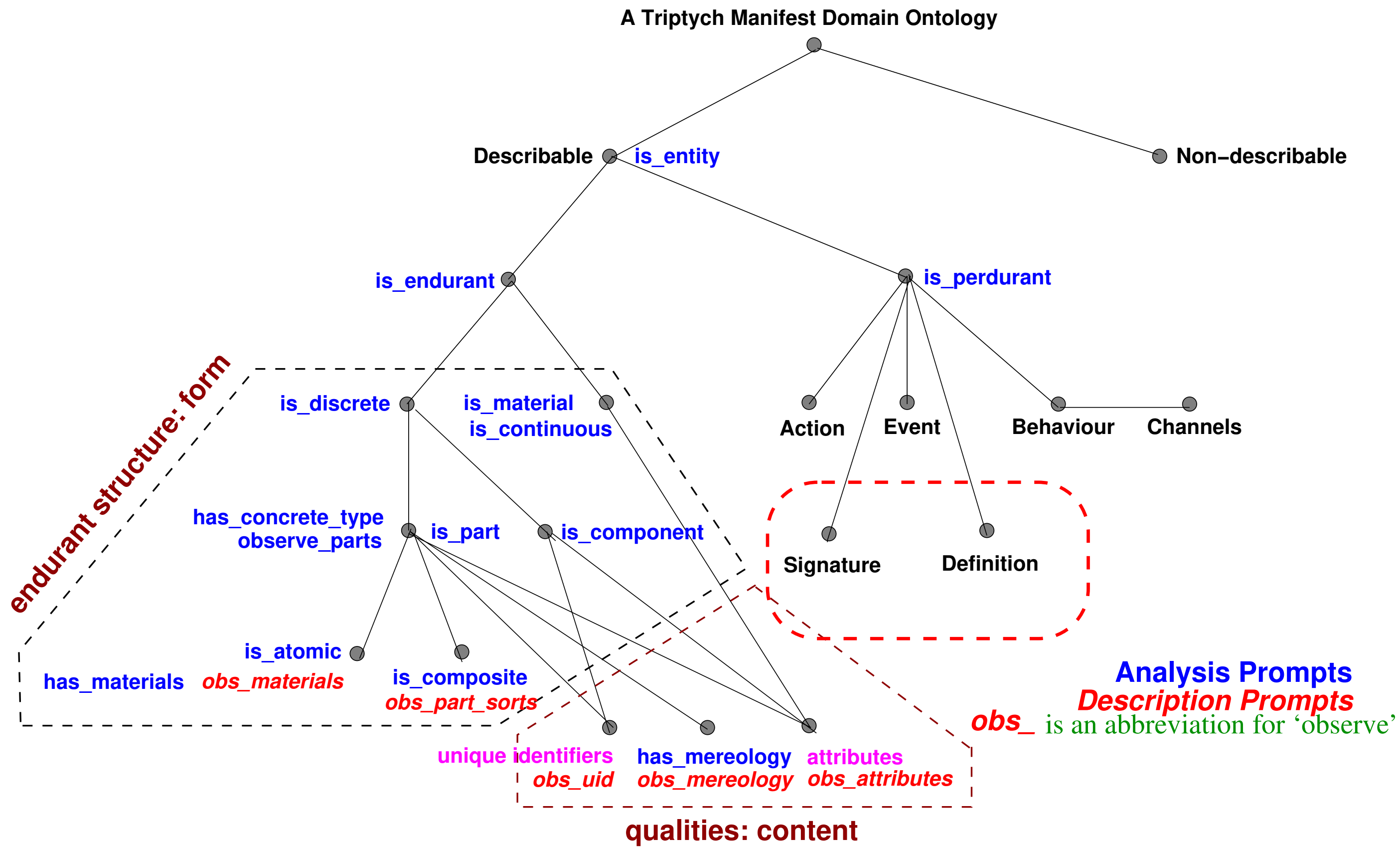

Figure 8: An Upper Ontology for Domains — **Perdurants: Signatures, Definitions,**

- Actors usually provide their initiated actions with arguments, say of type VAL.
  - Hence the schematic function (action) signature and schematic definition:

$$\mathsf{action}\colon\ \mathsf{VAL} \to \Sigma \stackrel{\sim}{\to} \Sigma$$
$$\mathsf{action}(\mathsf{v})(\sigma)\ \mathbf{as}\ \sigma'$$
$$\quad \mathbf{pre}\colon\ \mathscr{P}(\mathsf{v},\sigma)$$
$$\quad \mathbf{post}\colon\ \mathscr{Q}(\mathsf{v},\sigma,\sigma')$$

  - expresses that a selection of the domain,
  - as provided by the Σ type expression,
  - is acted upon and possibly changed.

- The partial function type operator $\stackrel{\sim}{\rightarrow}$
  - shall indicate that action(v)(σ)
  - may not be defined for the argument, i.e., initial state σ
  - and/or the argument v:VAL,
  - hence the precondition $\mathscr{P}$(v,σ).
- The post condition $\mathscr{Q}$(v,σ,σ') characterises the "after" state, σ':Σ, with respect to the "before" state, σ:Σ, and possible arguments (v:VAL).

Example 52 **Insert Hub Action Formalisation**: We formalise aspects of the above-mentioned hub action:

65 Insertion of a hub requires

66 that no hub exists in the net with the unique identifier of the inserted hub,

67 and then results in an updated net with that hub.

value

65. $\mathsf{insert\_H}$: $\mathsf{H} \rightarrow \mathsf{N} \stackrel{\sim}{\rightarrow} \mathsf{N}$
65. $\mathsf{insert\_H(h)(n)}$ **as** n'
66. **pre**: $\sim\exists\ \mathsf{h}':\mathsf{H}\cdot\mathsf{h}' \in$ **obs_part**_Hs(**obs_part**_HS(n))·**uid**_H(h)=**uid**_H(h′)
67. **post**: **obs_part**_Hs(**obs_part**_HS(n′))=**obs_part**_Hs(**obs_part**_HS(n))

- Which could be the argument values, v:VAL, of actions ?
  - Well, there can basically be only the following kinds of argument values:
    - parts, components and materials, respectively
    - unique part identifiers, mereologies and attribute values.
  - It basically has to be so
    - since there are no other kinds of values in domains.
  - There can be exceptions to the above
    - (Booleans,
    - natural numbers),

    but they are rare !

- **Perdurant (action) analysis thus proceeds as follows:**
  - identifying relevant actions,
  - assigning names to these,
  - delineating the "smallest" relevant state[21],
  - ascribing signatures to action functions, and
  - determining
    - action pre-conditions and
    - action post-conditions.
  - Of these, ascribing signatures is the most crucial:
    - In the process of determining the action signature
    - one oftentimes discovers
    - that part or component or material attributes have been left ("so far") "undiscovered".

[21] By "smallest" we mean: containing the fewest number of parts. Experience shows that the domain analyser cum describer should strive for identifying the smallest state.

- Example 52 showed example of a signature with only a part argument.
- Example 53 shows examples of signatures whose arguments are
  - parts and unique identifiers, or
  - parts, unique identifiers and attribute values.

Example 53 **Some Function Signatures**:

- Inserting a link between two identified hubs in a net:

  value insert_L: $\mathsf{L} \times (\mathsf{HI} \times \mathsf{HI}) \rightarrow \mathsf{N} \stackrel{\sim}{\rightarrow} \mathsf{N}$

- Removing a hub and removing a link:

  value remove_H: $\mathsf{HI} \rightarrow \mathsf{N} \stackrel{\sim}{\rightarrow} \mathsf{N}$
  remove_L: $\mathsf{LI} \rightarrow \mathsf{N} \stackrel{\sim}{\rightarrow} \mathsf{N}$

- Changing a hub state.

  value change_HΣ: $\mathsf{HI} \times \mathsf{H\Sigma} \rightarrow \mathsf{N} \stackrel{\sim}{\rightarrow} \mathsf{N}$ ■

4.10. Event Signatures and Definitions

- In the Bergen lectures we drop treatment of Events.

4.11. Discrete Behaviour Signatures and Definitions

4.11.1. Behaviour Signatures

- The behaviour functions are now called processes.
- That a behaviour function is a never-ending function, i.e., a process, is "revealed" in the function signature by the "trailing" **Unit**:

  behaviour: ... → ... **Unit**

- That a process takes no argument is "revealed" by a "leading" Unit:

  behaviour: **Unit** → ...

- That a process accepts channel, viz.: ch, inputs, including accesses an external attribute A, is "revealed" in the function signature as follows:

  behaviour: ... → **in** ch ... , resp. **in** attr_A_ch

- That a process offers channel, viz.: ch, outputs is “revealed” in the function signature as follows:

  behaviour: ... → **out** ch ...

- That a process accepts other arguments is “revealed” in the function signature as follows:

  behaviour: ARG → ...

- where ARG can be any type expression:

  T, T→T, T→T→T, etcetera

  where T is any type expression.

4.11.1.1 Part Behaviours:

- We can, without loss of generality, associate with each part a behaviour;
  - parts which share attributes
  - (and are therefore referred to in some parts' mereology),
  - can communicate (their "sharing") via channels.

- Processes are named, and part process names have indexes, namely the unique part identifier: π:Π.
  - The p be the part and let $part_\pi$ be the name of the process associated with part p.
  - The process named $part_\pi$ shall have the process name $part_\pi$ mean the following.
    - Let $part_\pi(args) \equiv \mathscr{B}$ be the definition of process $part_\pi$.
    - Occurrences of π in the definition body $\mathscr{B}$ shall be considered bound to the π of the process name $part_\pi$.
    - Thus, if the process named $part_i$ has π bound to i both in the process name $part_\pi$ and in the body $\mathscr{B}$.

- The process evolves around a state, or, rather, a set of values:
  - its possibly changing mereology, mt:MT[22],
  - the possible components and materials of the part, and
  - the attributes of the part.

[22] For MT see footnote 13 on Slide 143.

- A behaviour signature is therefore:

  $\mathsf{beh}_{\pi:\Pi}$: me:MT × sa:SA → ca:CA → **in** *ichns*(ea:EA) **in**,**out** *iochs*(me)

  where

  - (i) $\pi:\Pi$ is the unique identifier of part p, i.e., π=**uid**_P(p),
  - (ii) me:ME is the mereology of part p, me = **obs_mereo**_P(p),
  - (iii) sa:SA lists the static attribute values of the part,
  - (iv) ca:CA lists the controllable and attribute values of the part,
  - (v) *ichns*(ea:EA) refer to the external attribute *i*nput *chan*nel*s*, and where
  - (vi) *iochs*(me) are the input/output channels serving the attributes shared between the part p and the parts designated in its mereology me, cf. Sect. .

- We focus, for a little while, on the expression of
  - sa:SA,
  - ea:EA and
  - ca:CA,
- that is, on the concrete types of SA, EA and CA.
  - $\mathscr{S}_{\mathscr{A}}(p)$: sa:SA lists the static value types, $(svT_1, ..., svT_s)$, where s is the number of *s*tatic attributes of parts p:P.
  - $\mathscr{E}_{\mathscr{A}}(p)$: ea:EA lists the external attribute value channels of parts p:P in the behaviour signature and as input channels, *ichns*, see 9 lines above.
  - $\mathscr{C}_{\mathscr{A}}(p)$: ca:CA lists the controllable value expression types of parts p:P.
    - A **controllable attribute value expression** is an expression involving one or more attribute value expressions of the type of the biddable or programmable attribute ■

4.11.2. Behaviour Definitions

- Let P be a composite sort defined in terms of sub-sorts P_1, P_2, ..., P_n.
  - The process definition compiled from p:P, is composed from
    - a process description, $\mathcal{M}cP_{\textsf{uid\_}P(p)}$, relying on and handling the unique identifier, mereology and attributes of part p
    - operating in parallel with processes $p_1, p_2, \ldots, p_n$ where
      - p_1 is compiled from p_1:P_1,
      - p_2 is compiled from p_2:P_2,
      - ..., and
      - p_n is compiled from p_n:P_n.
- The domain description "compilation" schematic below "formalises" the above.

Process Schema I: Abstract `is_composite(p)`

value

compile_process: P → RSL-**Text**

compile_process(p) ≡

$\mathscr{M}P_{\mathbf{uid\_}P(p)}$(**obs_mereo**_P(p),$\mathscr{S}_{\mathscr{A}}$(p))($\mathscr{C}_{\mathscr{A}}$(p))

‖ compile_process(**obs_part**_P_1(p))

‖ compile_process(**obs_part**_P_2(p))

‖ ...

‖ compile_process(**obs_part**_P_n(p))

- The text macros: $\mathscr{S}_{\mathscr{A}}$ and $\mathscr{C}_{\mathscr{A}}$ were informally explained above.
- Part sorts P_1, P_2, ..., P_n are obtained from the `observe_part_sorts` prompt, Slide 109.

- Let P be a composite sort defined in terms of the concrete type Q-**set**.
  - The process definition compiled from p:P, is composed from
    - a process, $\mathscr{M}P$, relying on and handling the unique identifier, mereology and attributes of process p as defined by P
    - operating in parallel with processes q:**obs_part**_Qs(p).
- The domain description "compilation" schematic below "formalises" the above.

Process Schema II: Concrete `is_composite(p)`

type
  Qs = Q-**set**
value
  qs:Q-**set** = **obs_part**_Qs(p)
  compile_process: P → RSL-**Text**
  compile_process(p) ≡
    $\mathscr{M}P_{\textsf{uid\_}P(p)}$(**obs_mereo**_P(p),$\mathscr{S}_{\mathscr{A}}$(p))($\mathscr{C}_{\mathscr{A}}$(p))
   ‖ ‖{compile_process(q)|q:Q·q ∈ qs}

Process Schema III: is_atomic(p)

value

compile_process: P → RSL-**Text**

compile_process(p) ≡

$\mathscr{M}P_{\textsf{uid\_}P(p)}$(**obs_mereo**_P(p),$\mathscr{S}_{\mathscr{A}}$(p))($\mathscr{C}_{\mathscr{A}}$(p))

Example 54 **Bus Timetable Coordination**:

- We refer to Examples 17 on Slide 110, 18 on Slide 117 and 50 on Slide 234.

68 δ is the transportation system; f is the fleet part of that system; vs is the set of vehicles of the fleet; bt is the shared bus timetable of the fleet and the vehicles.

69 The fleet process is compiled as per Process Schema II (Slide 266).

- The definitions of the fleet and vehicle processes
  - are simplified
  - so as to emphasize the master/slave, programmable/inert
  - relations between these processes.

type
 Δ, F, VS [Example 17 on Slide 110]
 V, Vs=V-**set** [Example 18 on Slide 117]
 FI, VI, BT
value
68. δ:Δ,
68. f:F = **obs_part**_F(δ),
68. fi:FI = **uid**_F(f)
68. vs:V-**set** = **obs_part**_Vs(**obs_part**_VS(f))
axiom
68. ∀ v:V·v ∈ vs ⇒ □ **attr**_BT(f) = **attr**_BT(v)
value
69. fleet$_{fi}$: BT → **out** attr_BT_ch **Unit**
69. fleet$_{fi}$(bt) ≡ $\mathscr{M}F_{fi}$(bt) ∥ ∥ {vehicle$_{\textbf{uid}_V(v)}$()|v:V·v ∈ vs}
69. vehicle$_{vi}$: **Unit** → **in** attr_BT_ch **Unit**
69. vehicle$_{vi}$ ≡ $\mathscr{M}V_{vi}$(attr_BT_ch) ; vehicle$_{vi}$()

- The fleet process
  - $\mathscr{M}_F$
- is a "never-ending" processes:

value

$\mathscr{M}F_{fi}$: BT → **out** attr_BT_ch **Unit**

$\mathscr{M}F_{fi}(\mathsf{bt}) \equiv$ **let** $\mathsf{bt}' = \mathscr{F}_{fi}(\mathsf{bt})$ **in** $\mathscr{M}F_{fi}(\mathsf{bt}')$ **end**

- Function $\mathscr{F}_{fi}$ is a simple action.

- The expression of actual synchronisation and communication between the fleet and the vehicle processes
- is contained in $\mathscr{F}_{fi}$.

value

$\mathscr{F}_{fi}$: bt:BT → **out** attr_BT_ch BT
$\mathscr{F}_{fi}$(bt) ≡ (**let** bt′ = f_{fi}(bt)(...) **in** bt′ **end**) ⌈⌉ (attr_BT_ch ! bt ; bt)
f_{fi}: BT → ... → BT

- The auxiliary function f_{fi} "embodies" the programmable nature of the timetable attribute ■

- Please note a master part's programmable attribute can be reflected in two ways:
  - as a programmable attribute and
  - as an output channel to the behaviour specification of slave parts.
- This is illustrated, in Example 54 where
  - the fleet behaviour has programmable attribute BT
  - and output channel attr_BT_ch to vehicle behaviours.

Process Schema IV: Core Process (I)

- The core processes can be understood as never ending, "tail recursively defined" processes:

$\mathscr{M}P_{\pi:\Pi}$: me:MT×sa:SA → ca:CA →
  in *ichns*(ea:EA) **in**,**out** *iochs*(me) **Unit**
$\mathscr{M}P_{\pi:\Pi}$(me,sa)(ca) ≡
  let (me′,ca′) = $\mathscr{F}_{\pi:\Pi}$(me,sa)(ca) **in**
  $\mathscr{M}P_{\pi:\Pi}$(me′,sa)(ca′) **end**

$\mathscr{F}_{\pi:\Pi}$: me:MT×sa:SA → CA →
  in *ichns*(ea:EA) **in**,**out** *iochs*(me) → MT×CA

- $\mathscr{F}_\pi$
  - potentially communicates with all those part processes (of the whole domain)
  - with which it shares attributes, that is, has connectors.
  - $\mathscr{F}_\pi$ is expected to contain input/output clauses referencing the channels of the in ... out ... part of their signatures.
  - These clauses enable the sharing of attributes.
  - $\mathscr{F}_\pi$ also contains expressions, attr_A_ch ?, to external attributes.

- We present a rough sketch of $\mathscr{F}_\pi$.
- The $\mathscr{F}_\pi$ action non-deterministically internal choice chooses between
  - either [1,2,3,4]
    - [1] accepting input from
    - [4] a "offering" part process,
    - [2] optionally offering a reply, and
    - [3] finally delivering an updated state;
  - or [5,6,7,8]
    - [5] finding a suitable "order" (val)
    - [8] to a "inquiring" behaviour (π'),
    - [6] offering that value (on channel ch[π']
    - [7] and then delivering an updated state;
  - or [9] doing own work resulting in an updated state.

Process Schema V: Core Process (II)

```
value
    𝓕_π: me:MT × sa:SA → ca:CA → in ichns(ea:EA) in,out iochs(me)  MT×CA
    𝓕_π(me,sa)(ca) ≡
          [] { let val = ch[π′] ? in
               ( ch[π′] ! in_reply(val)(me,sa)(ca) ⌈⌉ skip ) ;
               in_update(val)(me,sa)(ca) end
             | π′: Π · π′ ∈ ℰ(π,me)}
       ⌈⌉ [] { let val = await_reply(π′)(me,sa)(ca) in
               ch[π′] ! val ;
               out_update(val)(me,sa)(ca) end
             | π′: Π · π′ ∈ ℰ(π,me)}
       ⌈⌉     (me,own_work(sa)(ca))
    channels ch[π′] are defined in in ichns(ea:EA) in,out iochs(me)
    in_reply:  VAL → SA×EA → CA → in ichns(ea:EA) in,out iochs(me)  VAL
    in_update:  VAL → MT×SA → CA → in,out iochs(me)  MT×CA
    await_reply:  Π → MT×SA → CA → in,out iochs(me)  VAL
    out_update:  VAL → MT×SA → CA → in,out iochs(me)  MT×CA
    own_work:  SA×EA → CA → in,out iochs(me)  CA
```

Example 55 Tollgates: Part and Behaviour:

- Figure 9 abstracts essential features of a tollgate.

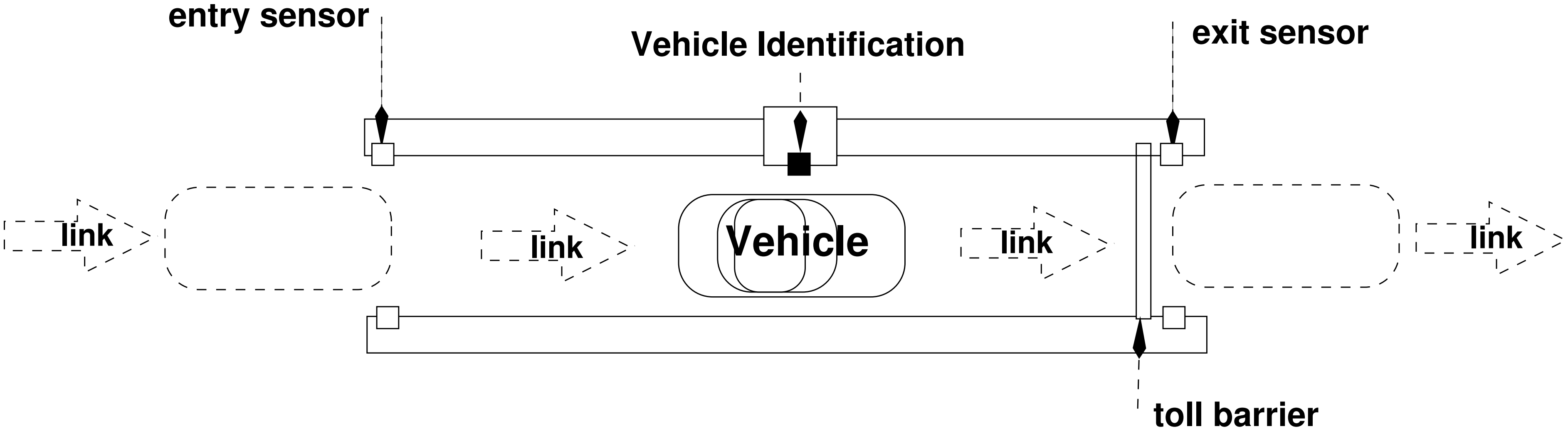

Figure 9: A tollgate

70 A tollgate is a composite part.
It consists of

71 an entry sensor ($_{ES}$),
a vehicle identity sensor ($_{IS}$),
a barrier ($_{B}$), and
an exit sensor ($_{XS}$).

72 The sensors function as follows:

a. When a vehicle first starts passing the entry sensor then it sends an appropriate (event) message to the tollgate.
b. When a vehicle's identity is recognised by the identity sensor then it sends an appropriate (event) message to the tollgate.
c. When a vehicle ends passing the exit sensor then it sends an appropriate (event) message to the tollgate.

73 We therefore model these sensors as shared dynamic event attributes.

a. For the sensors these are master attributes.

b. For the tollgate they are slave attributes.

c. In all three cases they are therefore modeled as channels.

74 A vehicle passing the gate

a. first “triggers” the entry sensor ("Enter"),
b. which results in the lowering ("Lower") of the barrier,
c. then the vehicle identity sensor ("vi:VI"),
d. with the tollgate “mysteriously”[23] handling that identity, and, simultaneously
e. raising ("Raise") the barrier, and
f. finally the output sensor ("Exit") is triggered as the vehicle leaves the tollgate,
g. and the barrier is lowered.

75 whereupon the tollgate resumes being a tollgate.

76 TGI is the type unique tollgate identifiers.

[23]... that is, passes vi on to the road pricing monitor — where we omit showing relevant channels.

- Instead of one tollgate we may think of a number of tollgates:
  - Each with their unique identifier — together with a finite set of two or more such identifiers, tgis:TGI-**set**.

```
type
70.    TG
71.    ES, IS, B, XS
74a..  En = {|"Enter"|}
74b..  Ba = {|"Lower","Raise"|}
74c..  Id = VI
74e..  Ex = {|"Exit"|}
76.    TGI
value
71.    obs_part_ES: TG → ES
71.    obs_part_IS: TG → IS
71.    obs_part_B: TG → B
71.    obs_part_XS: TG → XS
76.    uid_TGI: TG → TGI
74a..  attr_Enter: TG|ES → {|"Enter"|}
74c..  attr_Identity: TG|IS → VI
74e..  attr_Exit: TG|XS → {|"Exit"|}
channel
74.    {attr_En_ch[tgi]|tgi:TGI·tgi∈tgis}: En
74.    {attr_Id_ch[tgi]|tgi:TGI·tgi∈tgis}: VI
74.    {attr_Ba_ch[tgi]|tgi:TGI·tgi∈tgis}: BA
74.    {attr_Ex_ch[tgi]|tgi:TGI·tgi∈tgis}: Ex
value
74.    gate_{tgi:TGI}: Unit →
74.       in attr_En_ch[tgi],attr_Id_ch[tgi],attr_Ex_ch[tgi]
74.       out attr_Ba_ch[tgi] Unit
74.    gate_{tgi:TGI}() ≡
74a..      attr_En_ch[tgi] ? ;
74b..      attr_Ba_ch[tgi] ! "Lower" ;
74c..      let vi = attr_Id_ch[tgi] ? in
74d..      ( handle(vi) ‖
74e..      attr_Ba_ch[tgi] ! "Raise" ) ;
74f..      attr_Ex_ch[tgi] ? ;
74g..      attr_Ba[tgi] ! "Lower" ;
75.        gate_{tgi:TGI}() end
```

- The enter, identity and exit events are
  - slave attributes of the tollgate part and
  - master attributes of respectively
    - the entry sensor,
    - the vehicle identity sensor, and
    - the exit sensor sub-parts.
- We do not define the behaviours of these sub-parts.
  - We only assume that they each issue appropriate
  - attr_A_ch ! **out**put messages
  - where A is either Enter, Identity, or Exit and where event values en:Enter and ex:Exit are ignored ■

4.12. Concurrency: Communication and Synchronisation

- Process Schemas I, II and IV (Slides 264, 266 and 273), reveal
  - that two or more parts, which temporally coexist (i.e., at the same time),
  - imply a notion of concurrency.
  - Process Schema IV, through the RSL/CSP language expressions ch ! v and ch ?,
  - indicates the notions of communication and synchronisation.
  - Other than this we shall not cover these crucial notion related to parallelism.

4.13. Summary and Discussion of Perdurants

- The most significant contribution of this section has been to show that
  - for every domain description
  - there exists a normal form behaviour —
  - here expressed in terms of a CSP process expression.

4.13.1. Summary

- We have proposed to analyse perdurant entities into actions, events and behaviours — all based on notions of state and time.
- We have suggested modeling and abstracting these notions in terms of functions with signatures and pre-/post-conditions.
- We have shown how to model behaviours in terms of CSP (communicating sequential processes).
- It is in modeling function signatures and behaviours that we justify the endurant entity notions of parts, unique identifiers, mereology and shared attributes.

4.13.2. Discussion

- The analysis of perdurants into actions, events and behaviours represents a choice.
- We suggest skeptical readers to come forward with other choices.

5. Closing

- In Sect. we emphasised that in order to develop software the designers *must have a reasonable grasp of the "underlying" domain.*
- That means that when we design software, its requirements, to us, must be based on such a "grasp", that is, that the domain description must cover that *"underlying" domain.*
- We are not claiming that the domain descriptions (for software development) must cover more than the *"underlying" domain.*
- But what that *"underlying" domain* then is, is an open question which we do not speculate on in this paper.

- Domain descriptions are not “cast in stone !”
  - It is to be expected that domains are
    - researched
    - and their descriptions are developed

    as research projects — typically in universities.
  - It is also to be expected
    - that several domain descriptions coexist “simultaneously”,
    - that they may converge,
    - that some whither away, are rejected, and
    - that new descriptions are developed “on top of”, that is, on the basis of existing ones, which they replace,
    - descriptions that enlarge on, or restrict previous descriptions.

- It is finally to be expected that
  - when requirements are to be “derived” from a domain description, see, for example, [Bjø16d],
  - that the requirements cum domain engineers
  - redevelop a projected domain description
  - having some existing domain descriptions “at hand”.

5.1. Analysis & Description Calculi for Other Domains

- The analysis and description calculus of this paper appears suitable for manifest domains.
- For other domains other calculi may be necessary.
  - There is the introvert, composite domain(s) of systems software:
    - operating systems, compilers, database management systems, Internet-related software, etcetera.
    - The classical computer science and software engineering disciplines related to these components of systems software appears to have provided the necessary analysis and description "calculi."

- There is the domain of financial systems software
  - accounting & bookkeeping,
  - banking systems,
  - insurance,
  - financial instruments handling (stocks, etc.),
  - etcetera.
- Etcetera.
- For each domain characterisable by a distinct set of analysis & description calculus prompts such calculi must be identified.

5.2. On Domain Description Languages

- We have in this seminar expressed the domain descriptions in the `RAISE` [GHH+95] specification language `RSL` [GHH+92].
- With what is thought of as minor changes, one can reformulate these domain description texts in either of
  - `Alloy` [Jac06] or
  - `The B-Method` [Abr09] or
  - `VDM` [BJ78, BJ82, FL98] or
  - `Z` [WD96].
- One could also express domain descriptions algebraically, for example in `CafeOBJ` [FN97, FGO12].
  - The analysis and the description prompts remain the same.
  - The description prompts now lead to `Alloy`, `B-Method`, `VDM`, `Z` or `CafeOBJ` texts.

- We did not go into much detail with respect to perdurants.
  - For all the very many domain descriptions, covered elsewhere, RSL (with its CSP sub-language) suffices.
  - It is favoured here because of its integrated CSP sub-language which both facilitates
    - the 'compilation' of part descriptions into "the dynamics" of parts in terms of CSP processes, and
    - the modeling of external attributes in terms of CSP process input channels.
  - But there are cases, not documented in this seminar, where, [BGH+in], we have conjoined our RSL domain descriptions with descriptions in
    - Petri Nets [Rei10] or
    - MSC [IT99] or
    - StateCharts [Har87].

5.3. Open Problems

- The present paper has outlined a great number of
  - principles,
  - techniques and
  - tools

  of domain analysis & description.
- They give rise, now, to the investigation of further
  - principles,
  - techniques and
  - tools

  as well as underlying theories.

- We list some of these “to do” items:
  - (1) *a mathematical model of prompts;*
  - (2) *a sharpened definition of “what is a domain”;*
  - (3) *laws of description prompts;*
  - (4) *an understanding of domain facets* [Bjø16a];
  - (5) *a prompt calculus for perdurants;*
  - (6) *commensurate discrete and continuous models [WYZ94, ZWZ13];*
  - (7) *a study of the interplay between parts, materials and components;*
  - (8) *a closer study of external attributes and their variety of access forms and of biddable attributes;* and
  - (9) *specific domain theories;* etcetera.

5.4. Tony Hoare's Summary on 'Domain Modeling'

- In a 2006 e-mail, in response, undoubtedly to my steadfast, perhaps conceived as stubborn insistence, on domain engineering,
- Tony Hoare summed up his reaction to domain engineering as follows, and I quote[24]:

"There are many unique contributions that can be made by domain modeling.

1 *The models describe all aspects of the real world that are relevant for any good software design in the area. They describe possible places to define the system boundary for any particular project.*

2 *They make explicit the preconditions about the real world that have to be made in any embedded software design, especially one that is going to be formally proved.*

[24] E-Mail to Dines Bjørner, July 19, 2006

3 They describe the whole range of possible designs for the software, and the whole range of technologies available for its realisation.

4 They provide a framework for a full analysis of requirements, which is wholly independent of the technology of implementation.

5 They enumerate and analyse the decisions that must be taken earlier or later in any design project, and identify those that are independent and those that conflict. Late discovery of feature interactions can be avoided."

- All of these issues are covered, to some extent, in [Bjø06, Part IV].
- Tony Hoare's list pertains to a wider range that just the Manifest Domains treated in this paper.

5.5. Beauty Is Our Business

It's life that matters, nothing but life –
the process of discovering, the everlasting and perpetual process,
not the discovery itself, at all.[25]

- I find that quote appropriate in the following, albeit rather mundane, sense:
  - It is the process of analysing and describing a domain
  - that exhilarates me:
  - that causes me to feel very happy and excited.
- There is beauty [E.W. Dijkstra] not only in the result but also in the process.

[25] Fyodor Dostoyevsky, The Idiot, 1868, Part 3, Sect. V

6. Bibliography

6.1. Bibliographical Notes

6.1.1. Published Papers

- Web page **www.imm.dtu.dk/˜dibj/domains/** lists the published papers and reports mentioned below.
- I have thought about domain engineering for more than 25 years.
- But serious, focused writing only started to appear since [Bjø06, Part IV] — with [Bjø03, Bjø97] being exceptions:
  - [Bjø07, 2007] suggests a number of domain science and engineering research topics;
  - [Bjø10a, 2008] covers the concept of domain facets;
  - [BE10, 2008] explores compositionality and Galois connections.
  - [Bjø08, Bjø10c, 2008,2009] show how to systematically, but, of course, not automatically, "derive" requirements prescriptions from domain descriptions;

- [Bjø11a, 2008] takes the triptych software development as a basis for outlining principles for believable software management;
- [Bjø09, Bjø14a, 2009,2013] presents a model for Stanisław Leśniewski's [CV99] concept of mereology;
- [Bjø10b, Bjø11b] present an extensive example and is otherwise a precursor for the present paper;
- [Bjø11c, 2010] presents, based on the `TripTych` view of software development as ideally proceeding from domain description via requirements prescription to software design, concepts such as software demos and simulators;

- [Bjø13, 2012] analyses the TripTych, especially its domain engineering approach, with respect to Maslow's [26] and Peterson's and Seligman's [27] notions of humanity: how can computing relate to notions of humanity;
- the first part of [Bjø14b, 2014] is a precursor for the present paper with its second part presenting a first formal model of the elicitation process of analysis and description based on the prompts more definitively presented in the current paper; and
- [Bjø14c, 2014] focus on domain safety criticality.

The present paper basically replaces the domain analysis and description section of all of the above reference — including [Bjø06, Part IV, 2006].

[26] *Theory of Human Motivation.* Psychological Review 50(4) (1943):370-96; and *Motivation and Personality,* Third Edition, Harper and Row Publishers, 1954.
[27] *Character strengths and virtues: A handbook and classification.* Oxford University Press, 2004

6.1.2. Reports

We list a number of reports all of which document descriptions of domains. These descriptions were carried out in order to research and develop the domain analysis and description concepts now summarised in the present paper. These reports ought now be revised, some slightly, others less so, so as to follow all of the prescriptions of the current paper. Except where a URL is given in full, please prefix the web reference with: `http://www2.compute.dtu.dk/~dibj/`.

1 *A Railway Systems Domain:* `racosy/domains.ps` (2003)

2 *Models of IT Security. Security Rules & Regulations:* `it-security.pdf` (2006)

3 *A Container Line Industry Domain:* `container-paper.pdf` (2007)

4 *The "Market": Consumers, Retailers, Wholesalers, Producers:* `themarket.pdf` (2007)

5 *What is Logistics ?:* `logistics.pdf` (2009)

6 *A Domain Model of Oil Pipelines:* `pipeline.pdf` (2009)

7 *Transport Systems:* `comet/comet1.pdf` (2010)

8 *The Tokyo Stock Exchange:* `todai/tse-1.pdf` and `todai/tse-2.pdf` (2010)

9 *On Development of Web-based Software. A Divertimento:* `wfdftp.pdf` (2010)

10 *Documents (incomplete draft):* `doc-p.pdf` (2013)

11 *A Credit Card System:* `/2016/uppsala/accs.pdf` (2016)

6.2. References

[Abr09] Jean-Raymond Abrial. The B Book: Assigning Programs to Meanings *and* Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge, England, 1996 and 2009.

[BE10] Dines Bjørner and Asger Eir. Compositionality: Ontology and Mereology of Domains. Some Clarifying Observations in the Context of Software Engineering in July 2008, eds. Martin Steffen, Dennis Dams and Ulrich Hannemann. In *Festschrift for Prof. Willem Paul de Roever Concurrency, Compositionality, and Correctness*, volume 5930 of *Lecture Notes in Computer Science*, pages 22–59, Heidelberg, July 2010. Springer.

[BGH+in] Dines Bjørner, Chris W. George, Anne Eliabeth Haxthausen, Christian Krog Madsen, Steffen Holmslykke, and Martin Pěnička. "UML"–ising Formal Techniques. In *INT 2004: Third International Workshop on Integration of Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 423–450. Springer–Verlag, 28 March 2004, ETAPS, Barcelona, Spain. Final Version.

[BJ78] Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.

[BJ82] Dines Bjørner and Cliff B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.

[Bjø97] Dines Bjørner. Michael Jackson's Problem Frames: Domains, Requirements and Design. In Li ShaoYang and Michael Hinchley, editors, *ICFEM'97: International Conference on Formal Engineering Methods*, Los Alamitos, November 12–14 1997. IEEE Computer Society. Final Version.

[Bjø03] Dines Bjørner. Domain Engineering: A "Radical Innovation" for Systems and Software Engineering ? In *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, Heidelberg, October 7–11 2003. Springer–Verlag. The Zohar Manna International Conference, Taormina, Sicily 29 June – 4 July 2003. .

[Bjø06] Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.

[Bjø07] Dines Bjørner. Domain Theory: Practice and Theories, Discussion of Possible Research Topics. In *ICTAC'2007*, volume 4701 of *Lecture Notes in Computer Science (eds. J.C.P. Woodcock et al.)*, pages 1–17, Heidelberg, September 2007. Springer.

[Bjø08] Dines Bjørner. From Domains to Requirements. In *Montanari Festschrift*, volume 5065 of *Lecture Notes in Computer Science (eds. Pierpaolo Degano, Rocco De Nicola and José Meseguer)*, pages 1–30, Heidelberg, May 2008. Springer.

[Bjø09] Dines Bjørner. On Mereologies in Computing Science. In *Festschrift: Reflections on the Work of C.A.R. Hoare*, History of Computing (eds. Cliff B. Jones, A.W. Roscoe and Kenneth R. Wood), pages 47–70, London, UK, 2009. Springer.

[Bjø10a] Dines Bjørner. Domain Engineering. In Paul Boca and Jonathan Bowen, editors, *Formal Methods: State of the Art and New Directions*, Eds. Paul Boca and Jonathan Bowen, pages 1–42, London, UK, 2010. Springer.

[Bjø10b] Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics, Part I of II: The Engineering Part*. *Kibernetika i sistemny analiz*, (4):100–116, May 2010.

[Bjø10c] Dines Bjørner. The Rôle of Domain Engineering in Software Development. Why Current Requirements Engineering Seems Flawed! In *Perspectives of Systems Informatics*, volume 5947 of *Lecture Notes in Computer Science*, pages 2–34, Heidelberg, Wednesday, January 27, 2010. Springer.

[Bjø11a] Dines Bjørner. Believable Software Management. *Encyclopedia of Software Engineering*, 1(1):1–32, 2011.

[Bjø11b] Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics Part II of II: The Science Part*. *Kibernetika i sistemny analiz*, (2):100–120, May 2011.

[Bjø11c] Dines Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. In *Rainbow of Computer Science, Festschrift for Hermann Maurer on the Occasion of His 70th Anniversary.*, Festschrift (eds. C. Calude, G. Rozenberg and A. Saloma), pages 167–183. Springer, Heidelberg, Germany, January 2011.

[Bjø11d] Dines Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. In *Rainbow of Computer Science, Festschrift for Hermann Maurer on the Occasion of His 70th Anniversary.*, Festschrift (eds. C. Calude, G. Rozenberg and A. Saloma), pages 167–183. Springer, Heidelberg, Germany, January 2011.

[Bjø13] Dines Bjørner. *Domain Science and Engineering as a Foundation for Computation for Humanity*, chapter 7, pages 159–177. Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC [Francis & Taylor], 2013. (eds.: Justyna Zander and Pieter J. Mosterman).

[Bjø14a] Dines Bjørner. *A Rôle for Mereology in Domain Science and Engineering*. Synthese Library (eds. Claudio Calosi and Pierluigi Graziani). Springer, Amsterdam, The Netherlands, October 2014.

[Bjø14b] Dines Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model. In Shusaku Iida, José Meseguer, and Kazuhiro Ogata, editors, *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi*. Springer, May 2014.

[Bjø14c] Dines Bjørner. Domain Engineering – A Basis for Safety Critical Software. Invited Keynote, ASSC2014: Australian System Safety Conference, Melbourne, 26–28 May, December 2014.

[Bjø16a] Dines Bjørner. Domain Facets: Analysis & Description. 2016. Extensive revision of [Bjø10a]. http://www.imm.dtu.dk/~dibj/2016/facets/faoc-facets.pdf.

[Bjø16b] Dines Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. Technical report, Fredsvej 11, DK–2840 Holte, Denmark, 2016. Extensive revision of [Bjø11d]. http://www.imm.dtu.dk/~dibj/2016/demos/faoc-demo.pdf.

[Bjø16c] Dines Bjørner. From Domain Descriptions to Requirements Prescriptions – A Different Approach to Requirements Engineering. 2016. Extensive revision of [Bjø08].

[Bjø16d] Dines Bjørner. From Domain Descriptions to Requirements Prescriptions – A Different Approach to Requirements Engineering. 2016.

[CV99] R. Casati and A. Varzi. *Parts and Places: the structures of spatial representation*. MIT Press, 1999.

[FGO12] Kokichi Futatsugi, Daniel Gâlinâ, and Kazuhiro Ogata. Principles of proof scores in CafeOBJ. *Theor. Comput. Science*, 464:90–112, 2012.

[FL98] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.

[FMMR12] Carlo A. Furia, Dino Mandrioli, Angelo Morzenti, and Matteo Rossi. *Modeling Time in Computing*. Monographs in Theoretical Computer Science. Springer, 2012.

[FN97] Kokichi Futatsugi and Ataru Nakagawa. An overview of CAFE specification environment - an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In *Proc. of 1st International Conference on Formal Engineering Methods (ICFEM '97), November 12-14, 1997, Hiroshima, JAPAN*, pages 170–182. IEEE, 1997.

[FNT00] K. Futatsugi, A.T. Nakagawa, and T. Tamai, editors. *CAFE: An Industrial–Strength Algebraic Formal Method*, Sara Burgerhartstraat 25, P.O. Box 211, NL–1000 AE Amsterdam, The Netherlands, 2000. Elsevier. Proceedings from an April 1998 Symposium, Numazu, Japan.

[GHH+92] Chris W. George, Peter Haff, Klaus Havelund, Anne Elisabeth Haxthausen, Robert Milne, Claus Bendix Nielsen, Søren Prehn, and Kim Ritter Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.

[GHH+95] Chris W. George, Anne Elisabeth Haxthausen, Steven Hughes, Robert Milne, Søren Prehn, and Jan Storbank Pedersen. *The RAISE Development Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Hoa85] Charles Anthony Richard Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985. Published electronically: http://www.usingcsp.com/cspbook.pdf (2004).

[IT99] ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992, 1996, 1999.

[Jac95] Michael A. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press. Addison-Wesley, Reading, England, 1995.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.

[JHJ07] Cliff B. Jones, Ian Hayes, and Michael A. Jackson. Deriving Specfications for Systems That Are Connected to the Physical World. In Cliff Jones, Zhiming Liu, and James Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems: Essays in Honour of Dines Bjørner and Zhou Chaochen on the Occasion of Their 70th Birthdays*, volume 4700 of *Lecture Notes in Computer Science*, pages 364–390. Springer, 2007.

[LWZ13] Zhiming Liu, J. C. P. Woodcock, and Huibiao Zhu, editors. *Unifying Theories of Programming and Formal Engineering Methods - International Training School on Software Engineering, Held at ICTAC 2013, Shanghai, China, August 26-30, 2013, Advanced Lectures*, volume 8050 of *Lecture Notes in Computer Science*. Springer, 2013.

[Rei10] Wolfang Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Leitfäden der Informatik. Vieweg+Teubner, 1st edition, 15 June 2010. 248 pages; ISBN 978-3-8348-1290-2.

[van91] Johan van Benthem. *The Logic of Time*, volume 156 of *Synthese Library: Studies in Epistemology, Logic, Methhodology, and Philosophy of Science (Editor: Jaakko Hintika)*. Kluwer Academic Publishers, P.O.Box 17, NL 3300 AA Dordrecht, The Netherlands, second edition, 1983, 1991.

[WD96] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.

[WS12] George Wilson and Samuel Shpall. Action. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2012 edition, 2012.

[WYZ94] Ji Wang, XinYao Yu, and Chao Chen Zhou. Hybrid Refinement. Research Report 20, UNU/IIST, P.O.Box 3058, Macau, 1. April 1994.

[ZWZ13] Naijun Zhan, Shuling Wang, and Hengjun Zhao. Formal modelling, analysis and verification of hybrid systems. In *ICTAC Training School on Software Engineering*, pages 207–281, 2013. http://dx.doi.org/10.1007/978-3-642-39721-9_5, DBLP, http://dblp.uni-trier.de.