DOMAIN SCIENCE & ENGINEERING

The TU Wien Lectures, Fall 2022

Dines Bjørner

Technical University of Denmark

The Triptych Dogma

In order to *specify* **software**,
we must understand its requirements.

In order to *prescribe* **requirements**
we must understand the **domain**.

So we must **study, analyse** and **describe** domains.

Day #6: Perdurants, I

- Please consider Fig. 4.1 on Slide 64.
  - The previous two chapters covered the left of Fig. 4.1.
  - This chapter covers the right of Fig. 4.1.

• • •

- This chapter is a rather "drastic" reformulation and simplification of [18, *Chapter 7, i.e., pages 159–196*].
  - Besides, Sect. 6.5 is new.
- In this chapter we transcendentally "morph" manifest
  - **parts** into **behaviours,** that is:
  - **endurants** into **perdurants**.

- We analyse that notion and its constituent notions of
  - **actors**,
  - **channels** and **communication**,
  - **actions** and
  - **behaviours**.
- We shall investigate the, as we shall call them, perdurants of domains.
- That is state and time-evolving domain phenomena.
- The outcome of this chapter is that the student
  - will be able to model the perdurants of domains.
  - Not just for a particular domain instance,
  - but a possibly infinite set of domain instances[1].

[1] By this we mean: You are not just analysing a specific domain, say the one manifested around the corner from where you are, but any instance, anywhere in the world, which satisfies what you have described.

6.1 Part Behaviours – An Analysis

6.1.1 Behaviour Definition Analysis

- Parts co-exist;
  - they do so endurantly as well as perdurantly:
  - endure and perdure.
- Part perdurants, i.e., behaviours, interact with their surroundings, that is, with other behaviours.
- This is true for both natural and man-made parts.
- The present domain modelling method is mainly focused on man-made parts, that is artefacts.
- So our next analysis will take its clues from artefactual parts.
- We can, roughly, analyse part behaviours into three kinds.

- **Proactive Behaviours:** Behaviour B_i offers to synchronise and communicate values – *internal non-deterministically* with either of a definite number of distinct part sort behaviours B_a, B_b, ..., B_c:

```
B(i)(args) ≡
      (... ch[ {i,a} ] ! a_val ; ... ; B(i)(args′))
 ⌈⌉ (... ch[ {i,b} ] ! b_val ; ... ; B(i)(args″))
 ⌈⌉ ...
 ⌈⌉ (... ch[ {i,c} ] ! c_val ; ... ; B(i)(args‴))
```

The tail-recursive invocation of B_i indicates a possible "update" of behaviour B_i arguments. More on this later.

- **Responsive Behaviours:** Behaviour B_i *external non-deterministically* expresses willingness to synchronisation with and accept values from either of a definite number of distinct part sort behaviours B_a, B_b, ..., B_c:

```
B(i)(args) ≡
      (... let av = ch[ {i,a} ] ? in ...  B(i)(args′) end)
 ⌷ (... let bv = ch[ {i,b} ] ? in ... ; B(i)(args″) end)
 ⌷ ...
 ⌷ (... let cv = ch[ {i,c} ] ? in ... ; B(i)(args‴) end)
```

- **Mixed Behaviours:** Or behaviours, more generally, "are" an internal non-deterministic "mix" of the above:

```
B(i)(args) ≡
      ((... ch[ {i,a} ] ! a_val ; ... ; B(i)(args′))
 ⌈⌉  (... ch[ {i,b} ] ! b_val ; ... ; B(i)(args″))
 ⌈⌉  ...
 ⌈⌉  (... ch[ {i,c} ] ! c_val ; ... ; B(i)(args‴)))
 ⌈⌉ ((... let av = ch[ {i,a} ] ? in ...  B(i)(args′) end)
 ▯  (... let bv = ch[ {i,b} ] ? in ... ; B(i)(args″) end)
 ▯  ...
 ▯  (... let cv = ch[ {i,c} ] ? in ... ; B(i)(args‴) end))
```

- The "bodies" of the B_i behaviour definitions, i.e., "...", may contain interactions with [yet other] behaviours.
  Schematically for example:

```
ch[ {i,x} ] ! x_val
{ ch[ {i,z} ] ! z_val | z:{z1,z2,...,zm} }
let yv = ch[ {i,y} ] ? in ... end
let zv = □ { ch[ {i,z} ] ? | z:{z1,z2,...,zm} } in ... end
```

Etcetera. The full force of CSP with RSL is at play!

6.1.2 Channel Analysis

- This is the first of two treatments of the concept of *channels*;
  the present treatment is informal, motivational,
  the second treatment, Sect. 6.2 (right next !), is more formal.
- The CSP concept of *channel*
  is to be our way of expressing the "medium"
  in which behaviours interact.
  - Channels is thus an abstract concept.
  - Please do not think of it as a physical,
    an IT (information technology) device.
  - As an abstract concept it is defined in terms of,
    roughly, the laws, the semantics, of CSP [32].
  - We write 'roughly' since the CSP
    we are speaking of, is "embedded" in RSL.

6.2 Domain Channel Description

- We simplify the general treatment of channel declarations.
  - Basically all we can say, for any domain,
  - is that any two distinct part behaviours
  - may need to communicate.
  - Therefore we declare a vector of channels
  - indexed by sets of two distinct part identifiers.

```
value
  discover_channels: Unit → Unit
  discover_channels() ≡
    “ channel { ch[ {ij,ik} ] | ij,ik:UI · {ij,ik}⊆ uid_σ ∧ ij≠ik } M ”
```

  - Initially we shall leave the type of messages over channels further undefined.
  - As we, laboriously, work through the definition of behaviours, we shall be able to make M precise.

6.3 Behaviour Definition Description

- Behaviours have to be described.
  - Behaviour definitions are in the form of function definitions and are here expressed in RSL
    relying, very much, on its CSP component.
  - Behaviour definitions describe
    the type of the arguments
    the function, i.e., the behaviour, for which it is defined,
    that is, which kind of values it accepts.
  - Behaviour definitions further describe
- Thus there are two elements to a behaviour definition:
  - the behaviour *signature* and
  - the behaviour *body*

  definitions.

6.3.1 Behaviour Signatures

6.3.1.1 General

- Function, F, signatures consists of two textual elements:
  - the function name and
  - the function type:

    value F: A → B, or F: a:A → B

  - where A and B are the types of
    - function ("input") arguments, respectively
    - function ("output") values for such arguments.
- The first form F: A → B is what is normally referred to as the form for function signatures.
- The second form: F: a:A → B "anticipates" the general for for function F invocation: F(a).

6.3.1.2 Domain Behaviour Signatures

- A schematic form of part (p) behaviour signatures is:

b: bi:BI→me:Mer→svl:StaV*→mvl:MonV*→prgl:PrgV* channels **Unit**

- We shall motivate the general form of part behaviour, B, signatures, "step-by-step":

α.	b	the [chosen] name of part p behaviours.
β	U→V→...→W→Z:	The function signature is expressed in the Schönfinkel/Curr style – corresponding to the invocation form F(u)(v)...(w)
γ.	bi:BI:	a general value and the type of part p unique identifier
δ.	me:Mer:	a general value and the type of part p mereology
ϵ.	svl:StaV*:	a general (possibly empty) list of values and types of part p's (possibly empty) list of static attributes
ζ.	mvl:MonV*:	a general list of names of types of part p's (possibly empty) list of monitorable attributes
η.	prgl:PrgV*:	a general list of values and types of part p's (possibly empty) list of programmable attributes
θ.	channels:	are usually of the form: {ch[{i,j}]|(i,j)∈I(me)} and express the su of channels over which behaviour Bs interact with other behav
ι.	**Unit**:	designates the single value ()

[2] Moses Schönfinkel (1888–1942) was a Russian logician and mathematician accredited with having invented combinatory logic [https://en.wikipedia.org/wiki/Moses_Schönfinkel]. Haskell B. Curry (1900–1982) was an American mathematician and logician known for his work in combinatory logic [https://en.wikipedia.org/wiki/Haskell_Curry]

In detail:

α. **Behaviour name:** In each domain description there are many sorts, B, of parts. For each sort there is a generic behaviour, whose name, here b. is chosen to suitably reflect B.

β. **Currying** is here used in the pragmatic sense of grouping "same kind of arguments", i.e., separating these from one-another, by means of the $\rightarrow$s.

γ. The **unique identifier** of part sort B is here chosen to be BI. Its value is a constant.

δ. The **mereology** is a usually constant. For same part sorts it may be a variable.

Example 65 . Variable Mereologies:

- For a road transport system where we focus on the transport the mereology is a constant.
- For a road net where we focus on the development of the road net: building new roads: inserting and removing hubs and links, the mereology is a variable.
- Similar remarks apply to canal systems `www.imm.dtu.dk/~dibj/2021/Graphs/Rivers-and-Canals.pdf`, pipeline systems [8], container terminals [14], assembly line systems [15], etc. ■

ϵ. **Static attribute values** are constants. The use of static attribute values in behaviour body definitions is expressed by an identifier of the stvl list of identifiers.

ζ. **Monitorable attribute values** are generally, ascertainable, i.e., readable, cf. Sect. 5.4.3.1 on Slide 287. Some are *biddable*, can be changed by a, or the behaviour, cf. Sect. 5.4.3.2 on Slide 288, but there is no guarantee, as for programmable attributes, that they remain fixed.

  – The use of a[ny] monitorable attribute value in behaviour body definitions is expressed by a read_A_from_P(mv,bi) where mv is an identifier of the mvl list of identifiers and bi is the unique part identifier of the behaviour definition in which the read occurs.
  – The update of a biddable attribute value in behaviour body definitions is expressed by a update_P_with_A(bi,mv,a).

η. **Programmable attribute values** are just that. They vary as specified, i.e., "programmed", by the behaviour body definition. Tail-recursive invocations of behaviour B_i "replace" relevant programmable attribute argument list elements with "new" values.

θ. **channels:** $I(\mathsf{me})$ expresses a set of unique part identifiers different from bi, hence of behaviours, with which behaviour b(i) interacts.

ι. The **Unit** of the behaviour signature is a short-hand for the behaviour either **read**ing the value of a monitorable attribute, hence global state σ, or performing a **write**, i.e., an *update*, on σ.

6.3.1.3 Action Signatures

- Actions come in any forms:

135. Some take no arguments, say action_a(), but read the global state component σ, and
136. others also take no arguments, say action_b(), but update the global state component σ.
137. Some take an argument, say, action_c(c), but do not "touch" a global state component,
138. while others both take an argument and deliver a value, say action_d(d) and also do not "touch" a global state component.
139. Et cetera !

```
type A, B, C, D, ...
value
135.  action_a: Unit → read σ A
136.  action_b: Unit → write σ B
137.  action_c: C → Unit
138.  action_d: D → E Unit
139.  ...
```

- An example of 137 are the CSP output: ch[...]!c, and
- an example of 138 are the CSP input: **let** e = ch[...]? **in** ... **end**.

6.3.2 Behaviour Invocation

- The general form of behaviour invocation is shown below.
  - The invocation follows the "Currying" of the behaviour type signature.
  - [Normally one would write all this on one line: b(i)(m)(s)(m)(p) ≡.]

```
behaviour_name
  (unique_identifier)
    (mereology)
      (static_values)
        (monitorable_attribute_names)
          (programmable_variables) ≡
... body ...
```

- When first "invoked":

```
value
  discover_ behaviour_signature: P → RSL-Text
  discover_ behaviour_signature(p) ≡
  ❝ behaviour_name:
      UId → Mereo → StaVL → MonVL → ProVL → channels Unit
    behaviour_name
      (uid_B(p))
        (mereo_B(p))
          (types_to_values(static_attribute_types(p)))
            (mon_attribute_types(p))
              (types_to_values(programmable_attribute_types(p))) ≡ ❞
    pre: is_B(p) ∧ is_manifest(p)

discover_ behaviour_signatures: Unit → RSL-Text
discover_ behaviour_signatures() ≡
  { discover_ behaviour_signature(p) | p ∈ σ ∧ is_manifest(p) }
```

6.3.3 Behaviour Definition Bodies

- We remind the student of Sect. 6.1.1 on Slide 373.
- The general, "mixed", form of behaviour definitions was given as:

```
B(i)(args) ≡
      ( ( ... )
∏     ( ... ch[ {i,b} ] ! b_val ; ... ; B(i)(args″) )
∏     ( ... ) )
∏  ( ( ... )
□     (... let bv = ch[ {i,b} ] ? in ... ; B(i)(args″) end )
□     ( ... ) )
```

- We can express the same
  - by separating the alternatives
  - into invocations of separately defined behaviuors.

$$
\begin{array}{l}
\mathsf{B(i)(args)} \equiv \\
\quad (\ \ldots \\
\quad \sqcap \quad \mathsf{Bin}_j\mathsf{(i)(args)} \\
\quad \sqcap \quad \ldots\ ) \\
\quad \sqcap\ (\ \ldots \\
\quad \sqcap \quad \mathsf{Bxn}_k\mathsf{(i)(args)} \\
\quad \sqcap \quad \ldots\ )
\end{array}
$$

- where
  - the internal don-deterministically invoked behaviours $\mathsf{Bin}_j\mathsf{(i)(args)}$ and
  - the external don-deterministically invoked behaviours $\mathsf{Bin}_k\mathsf{(i)(args)}$
- are then separately defined:

$$
\begin{array}{l}
\mathsf{Bin}_j\mathsf{(i)(args)} \equiv (\ \ldots\ \mathsf{Bin}_j\mathsf{(i)(args')}\ ) \\
\mathsf{Bxn}_k\mathsf{(i)(args)} \equiv (\ \ldots\ \mathsf{Bxn}_k\mathsf{(i)(args'')}\ )
\end{array}
$$

6.3.4 Discover Behaviour Definition Bodies

- In other words,
  - for current lack of a more definitive methodology
  - for "discovering" the bodies of behaviour definitions
  - we resort to "..." !

value
  discover_behaviour_definition: P → RSL-**Text**
  discover_behaviour_definition(p) ≡ ...

discover_behaviour_definitions: **Unit** → RSL-**Text**
discover_behaviour_definitions() ≡
  { discover_behaviour_definition(p) | p ∈ σ ∧ is_manifest(p) }

Example 66 . Automobile Behaviour:

Signatures

140. automobile:
   (a) there is the usual "triplet" of arguments: unique identifier, mereology and static attributes;
   (b) then there are two programmable attributes: the automobile position (cf. Item 90 on Slide 267), and the automobile history (cf. Item 127c on Slide 342);
   (c) and finally there are the input/output channel references allowing communication between the automobile and the hub and link behaviours.

141. Similar for
   (a) link and
   (b) hub behaviours.

• We omit the modelling of monitorable attributes (...).

value
140a,140a automobile: ai:AI → ((_,uis):AM) → ...
140b → (apos:APos × ahist:AHist)
140c **in out** {ch[{ai,ui}]|ai:AI,ui:(HI|LI) · ai∈ais ∧ ui ∈ *uis*} **Unit**
141a link: li:LI → (his,ais):LM → LΩ → ...
141a → (LΣ×L_Hist)
141a **in out** {ch[{li,ui}]|li:LI,ui:(AI|HI)-**set** · ai∈ais ∧ li ∈lis∪his} **Unit**
141b hub: hi:HI → ((_,ais):HM) → HΩ ...
141b → (HΣ×H_Host)
141b **in out** {ch[{ai,ui}]|hi:HI,ai:AI · ai∈ais ∧ hi ∈ *uis*} **Unit**

Definitions: Automobile at a Hub

142. We abstract automobile behaviour at a Hub (hi).
   (a) Either the automobile remains in the hub,
   (b) or, internally non-deterministically,
   (c) leaves the hub entering a link,
   (d) or, internally non-deterministically,
   (e) stops.

```
142  automobile(ai)(aai,uis)(...)(apos:atH(fli,hi,tli),ahist) ≡
142a    automobile_remains_in_hub(ai)(aai,uis)(...)(apos:atH(fli,hi,tli),ahist)
142b  ⌈⌉
142c    automobile_leaving_hub(ai)(aai,uis)(...)(apos:atH(fli,hi,tli),ahist)
142d  ⌈⌉
142e    automobile_stop(ai)(aai,uis)(...)(apos:atH(fli,hi,tli),ahist)
```

143. [142a] The automobile remains in the hub:
   (a) the automobile remains at that hub, "idling",
   (b) informing ("first") the hub behaviour.

```
143  automobile_remains_in_hub(ai)(aai,uis)(...)(apos:atH(fli,hi,tli),ahist) ≡
143     let τ = record_𝕋𝕀𝕄𝔼() in
143b     ch[ai,hi] ! τ ;
143a     automobile(ai)(aai,uis)(...)(apos,upd_hist(τ,hi)(ahist))
143     end

143a  upd_hist: (𝕋𝕀𝕄𝔼×I) → (AHist|LHist|HHist) → (AHist|LHist|HHist)
143a  upd_hist(τ,i)(hist) ≡ hist † [ i ↦ ⟨τ⟩⌢hist(i) ]
```

144. [142c] The automobile leaves the hub entering a link:
   (a) tli, whose "next" hub, identified by thi, is obtained from the mereology of the link identified by tli;
   (b) informs the hub it is leaving and the link it is entering,
   (c) "whereupon" the vehicle resumes (i.e., "while at the same time" resuming) the vehicle behaviour positioned at the very beginning (0) of that link.

```
144  automobile_leaving_hub(ai)(aai,uis)(...)(apos:atH(fli,hi,tli),ahist) ≡
144a    (let ({fhi,thi},ais) = mereo_L(retr_L(tli)(σ)) in assert: fhi=hi
144b     ( ch[ ai,hi ] ! τ ‖ ch[ ai,tli ] ! τ ) ;
144c     automobile(ai)(aai,uis)(...)
144c          (onL(tli,(hi,thi),0),upd_hist(τ,tli)(upd_hist(τ,hi)(ahist))) end)
```

145. [142e] Or the automobile "disappears — off the radar" !

145 automobile_stop(ai)(aai,uis),(...)(apos:atH(fli,hi,tli),ahist) ≡ **stop**

- Similar behaviour definitions can be given for *automobiles on a link*, for *link*s and for *hub*s.
- Together they must reflect, amongst other things:
  - the time continuity of automobile flow,
  - that automobiles follow routes,
  - that automobiles, links and hubs together adhere to the intentional pull expressed earlier,
  - et cetera.
- A specification of these aspects must be proved to adhere to these properties.

6.4 Domain Behaviour Initialisation

- For every manifest part it must be described how its behaviour is initialised.

Example 67 . **The Road Transport System Initialisation:** We "wrap up" the main example of this *primer*:

- We omit treatment of monitorable attributes.

146. Let us refer to the system initialisation as an action.
147. All links are initialised,
148. all hubs are initialised,
149. all automobiles are initialised,
150. etc.

```
value
146. rts_initialisation: Unit → Unit
146. rts_initialisation() ≡
147.     || { link(uid_L(l))(mereo_L(l))(attr_LEN(l),attr_LΩ(l))(attr_L_Traffic(l),attr_LΣ(
148.     || || { hub(uid_H(l))(mereo_H(l))(attr_HΩ(l))(attr_H_Traffic(l),attr_HΣ(l))| h:H ·
149.     || || { automobile(uid_A(a))(mereo_A(a))(attr_RegNo(a))(attr_APos(a)) | a:A ·
150.     || ...
```

- We have here omitted possible monitorable attributes.
- We refer to
  - *ls*: Item 36 on Slide 153,
  - *hs*: Item 37 on Slide 153, and
  - *as*: Item 38 on Slide 153 ■

Day #7: Perdurants, II

6.5 Discrete Dynamic Domains

- Up till now our analysis & description of a domain,
  - has, in a sense, been *static:*
  - in analysing a domain we considered its entities
  - to be of a definite number.
- In this section we shall consider the case where the number of entities change:
  - where new entities are *created*
  - and existing entities are *destroyed*,
  - that is:
    - where new parts, and hence behaviours, arise, and
    - existing parts, and hence behaviours, cease to exist.

6.5.1 Create and Destroy Entities

- In the domain we can expect that its behaviours create and destroy entities.

Example 68 . Creation and Destruction of Entities:

- In the *road transport* domain
  - new hubs, links and automobiles may be inserted into the road net, and
  - existing links, hubs and automobiles may be removed from the road net.
- In a *container terminal* domain [5, 14]
  - new containers are introduced, old are discarded;
  - new container vessels are introduced, old are discarded;
  - new ship-to-shore cranes are introduced, old are discarded;
  - et cetera.

- In a *retailer* domain [16]
  - new customers are introduced, old are discarded;
  - new retailers are introduced, old are discarded;
  - new merchandise is introduced, old is discarded;
  - et cetera.
- In a *financial system* domain
  - new customers are introduced, old are discarded;
  - new banks are introduced, old are discarded;
  - new brokers are introduced, old are discarded;
  - et cetera ■

- The issue here is:
  - When hubs and links are inserted or removed
    - the mereologies of "neighbouring" road elements change,
    - and so does the mereology of automobiles.
  - When automobiles are inserted or removed
    - The mereology of road elements
    - have to be changed
    - to take account of the insertions and removals,
    - and so does the mereology of automobiles.
  - And, some domain laws must be re-expressed:
    - The domain part state, σ, must be updated[3],
    - and so must the unique identifier state, uid_σ[4].

[3] Cf. Sect. 4.7.2 on Slide 155
[4] Cf. Sect. 5.2.4 on Slide 194

6.5.1.1 Create Entities

- It is taken for granted here that there are behaviours, one or more, which take the initiative to and carry out the creation of specific entities.
  Let us refer to such a behaviour as the "creator".
- To create an entity implies the following three major steps
  - [A.–C.] the step wise creation of the part and initialisation of the transduced behaviour, and
  - [D.] the adjustment of all such part behaviours that might have their mereologies and attributes updated to accept such requests from creators.

A. To decide on the part sort – in order to create that part – that is
  – to obtain a unique identifier – one hitherto not used;
  – to obtain a mereology, one
    * according to the general mereology for parts of that sort,
    * and how the part specifically is to "fit" into its surroundings;
  – to obtain an appropriate set of attributes:
    * again according to the attribute types for that part sort
    * and, more specifically, choosing initial attribute values.
  – This part is then "joined" to σ[5] and
  – its unique identifier "joined" to uid_σ[6].

[5](the global part state), Cf. Sect. 4.7.2 on Slide 155
[6](the global unique identifier state), Cf. Sect. 5.2.4 on Slide 194

B. Then to transcendentally deduce that part into a behaviour:
   - initialised (according to Sect. 6.3.1) with
     * the unique identifier,
     * the mereology, and
     * the attribute values
   - This behaviour is then invoked and "joined" to the set of current behaviours, cf. Sect. 6.4 on Slide 401 – i.e., just above !

C. Then, finally, to "adjust" the mereologies of topologically or conceptually related parts,
   - that is, for each of these parts to update:
   - their mereology and possibly some
   - state and state space

arguments of their corresponding behaviours.

D. The update of the mereologies
of already "running" behaviours requires the following:
  – that, potentially all, behaviours offers to accept
  – mereology update requests from the "creator" behaviour.

- The latter means, practically speaking,
  – that each part/behaviour
  – which may be subject to mereology changes
  – externally non-deterministically
  – expresses an offer to accept such a change.

Example 69 . Road Net Administrator:

- We introduce the road net behaviour – based on the road net composite part, RN.

151. The road net has a programmable attribute: a *road net (development & maintenance) graph*.[7]
    - The road net graph consists of a quadruple:
        - a map that for each hub identifier records "all" the information that the road net administrator deems necessary[8] for the maintenance and development of road net hubs;
        - a map that for each link identifier records "all" the information[9] that the road net administrator deems necessary for the maintenance and development of road net links;
        - and a map from the hub identifiers to the set of identifiers of the links it is connected to, and
        - the set of all automobile identifiers.

[7] The presentation of the road net Behaviour, rn, is simplified.
[8] We presently abstract from what this information is.
[9] See footnote 8.

152. This graph is commensurate with the actual topology of the road net.

type
151. G = (HI $\overrightarrow{m}$H_Info) × (LI $\overrightarrow{m}$L_Info) × (HI $\overrightarrow{m}$LI-**set**) × AI-**set**
value
151. attr_G: RN → G
axiom
151. ∀ (hi_info,li_info,map,ais):G ·
151. **dom** map = **dom** hi_info = *his* ∧ ∪ **rng** map = **dom** li_info = *lis* ∧
152. ∀ hi:HI · hi ∈ **dom** hi_info ⇒
152. **let** h:H · h ∈ σ ∧ uid_H(h)=hi **in**
152. **let** (lis′,...) = mereo_H(h) **in** lis′ = map(hi)
152. ais ⊆ *ais* ∧ ...
152. **end end**

- Please note the fundamental difference between
  - the *road net (development & maintenance) graph* and
  - the road net.
- The latter pretends to be “the real thing”.
- The former is “just” an abstraction thereof !

153. The road net mereology ("bypasses") the hub and link aggregates, and comprises a set of hub identifiers and a set of link identifiers – of the road net[10].

type
153. H_Mer = AI-**set** × LI-**set**
153. mereo_RN: RN → RNMer
axiom
153. ∀ rts:RTS · **let** (_,lis) = mereo_H(obs_RN(rts)) **in** lis ⊆ *lis* **end**
value

[10] This is a repeat of the hub mereology given in Item 65 on Slide 214.

154. The road net [administrator] behaviour,
155. amongst other activities (. . .)
156. internal non-deterministically decides upon
    (a) either a hub insertion,
    (b) or a link insertion,
    (c) or a hub removal,
    (d) or a link removal;

• These four sub-behaviours each resume being the road net behaviour.

```
value
154.  rn: RNI → RNMer → G → in,out{ch[ {i,j} ]|{i,j}⊆uid_σ}
154.  rn(rni)(rnmer)(g) ≡
155.      ...
156a.     ∏ insert_hub(g)(rni)(rnmer)
156b.     ∏ insert_link(g)(rni)(rnmer)
156c.     ∏ remove_hub(g)(rni)(rnmer)
156d.     ∏ remove_link(g)(rni)(rnmer)
```

157. These road net sub-behaviours require information about
    (a) a hub to be inserted: its initial state, state space and [empty] traffic history, or
    (b) a link to be inserted: its length, initial state, state space and [empty] traffic history, or
    (c) a hub to be removed: its unique identifier, or
    (d) a link to be removed: its unique identifier.

type

157. Info == nHInfo | nLInfo | oHInfo | oLInfo
157. nHInfo :: HΣ × HΩ × H_Traffic
157. nLInfo :: LEN × LΣ × LΩ × L_Traffic
157. oHInfo :: HI
157. oLInfo :: LI ■

Example 70 . Road Net Development: Hub Insertion:

- Road net development alternates between design,
  - based on the *road net (development & maintenance) graph*, and
- actual, "real life", construction
  - taking place in the real surroundings of the road net.

158. If a hub insertion then the road net behaviour,
     based on the hub and link information and the road net layout
     in the *road net (development & maintenance) graph* selects
     (a) an initial mereology for the hub, h_mer,
     (b) an initial hub state, hσ, and
     (c) an initial hub state space, hω, and
     (d) an initial, i.e., empty hub traffic history;
159. updates its *road net (development & maintenance) graph* with
     information about the new hub,
160. and results in a suitable grouping of these.

```
value
158.  design_new_hub: G → (nHInfo×G)
158.  design_new_hub(g) ≡
158a.     let h_mer:HMer = M_ih(g),
158b.         hσ:HΣ = S_ih(g),
158c.         hω:HΩ = O_ih(g),
158d.         h_traffic = [ ],
159.          g′ = MSO_ih(g) in
160.      ((h_mer,hσ,hω,h_traffic),g′) end
```

- We leave open, in Items 158a–158c, as to what the initial hub mereology, state and state space should be initialised, i.e., the $\mathcal{M}_{ih}, \mathcal{S}_{ih}, \mathcal{O}_{ih}$ and $\mathcal{MSO}_{ih}$ functions.

161. To insert a new hub the road net administrator
    (a) first designs the new hub,
    (b) then selects a hub part
    (c) which satisfies the design,
        whereupon it updates the global states
    (d) of parts σ,
    (e) of unique identifiers, and
    (f) of hub identifiers –
        in parallel, and in parallel with
162. initiating a new hub behaviour
163. and resuming being the road net behaviour.

161. insert_hub: G×RNI×RNMer → **Unit**
161. insert_hub(g,rni,rnmer) ≡
161a. **let** ((h_mer,hσ,hω,h_traffic),g′) = design_new_hub(g) **in**
161b. **let** h:H · h∉σ ·
161c. mereo_H(h)=h_mer ∧ hσ=attr_HΣ(h) ∧
161c. hω=attr_HΩ(h) ∧ h_traffic=attr_HTraffic(h) **in**
161d. σ := σ ∪ {h}
161e. ‖ uid$_\sigma$:= uid$_\sigma$ ∪ {uid_H(h)}
161f. ‖ *his* := *his* ∪ {uid_H(h)}
162. ‖ hub(uid_H(h))(attr_HΣ(h),attr_HΩ(h),attr_HΩ(h))
163. ‖ rn(rni)(rnmer)(g′)
161. **end end** ■

Example 71 . Road Net Development: Link Insertion:

164. If a link insertion then the road net behaviour
    based on the hub and link information and the road net layout
    in the *road net (development & maintenance) graph* selects
    (a) the mereology for the link, h_mer[11],
    (b) the (static) length (attribute),
    (c) an initial link state, lσ,
    (d) an initial link state space lω, and
    (e) and initial, i.e., empty, link traffic history;
165. updates its *road net (development & maintenance) graph*
    with information about the new link,
166. and results in a suitable grouping of these.

[11] that is, the two existing hub identifiers between whose hubs the new link is to be inserted

value
164. design_new_link: G → (nLInfo×G)
164. design_new_link(g) ≡
164a. **let** l_mer:LMer = $\mathcal{M}_{il}$(g),
164b. le:LEN = $\mathcal{L}_{il}$(g),
164c. lσ:LΣ = $\mathcal{S}_{il}$(g),
164d. lω:LΩ = $\mathcal{O}_{il}$(g),
164e. l_hist:L_Hist = [ ]
165. g′:G = $\mathcal{MLSO}_{il}$(g) **in**
166. ((l_mer,le,lσ,lω,l_hist),g′) **end**

- We leave open, in Items 164a–164d, as to what the initial link mereology, state and state space should be initialised.

167. To insert a new link the road net administrator
    (a) first designs the new link,
    (b) then selects a link part
    (c) which satisfies the design,
        whereupon it updates the global states
    (d) of parts, σ,
    (e) of unique part identifiers, and
    (f) of link identifiers –
        in parallel, and in parallel with
168. initiating a new link behaviour and
169. updating the mereologies and possibly the state and the state space attributes of the connected hubs.

```
value
167.  insert_link: G → Unit
167.  insert_link(rni,l) ≡
167a.     let ((l_mer,le,lσ,lω,l_traffic_hist),g′) = design_new_link(g) in
167c.     let l:L · l∉σ · mereo_L(l)=l_mer ∧
167c.                le=attr_LEN(l) ∧ lσ=attr_LΣ(l) ∧
167c.                lω=attr_LΩ(l) ∧ l_traffic_hist=attr_HTraffic(l) in
167d.    σ := σ ∪ {l}
167e.  ‖ uid_σ := uid_σ ∪ {uid_L(l)}
167f.  ‖ lis := list ∪ {}
168.   ‖ link(uid_L(l))(l_mer)(le,lω)(lσ,l_traffic)
169.   ‖ ch[ {rni,hi1} ] ! updH(M_il(g),Σ_il(g),Ω_il(g))
169.   ‖ ch[ {rni,hi2} ] !
167.     end end ■
```

- We leave undefined the mereology and the state σ and state space ω update functions.

6.5.1.2 Destroy Entities

- The introduction to Sect. 6.5.1.1 on Slide 408 on the *creation of entities*
  - outlined a number of creation issues ([A, B, C, D]).
- For the *destruction of entities*
  - description matters are a bit simpler.
- It is, almost, simply a matter
  - of designating, by its unique identifier,
  - the entity: part and behaviour to be destroyed.
- Almost !
  - The mereology of the destroyed entity
  - must be such that the destruction
  - does not leave "dangling" references !

Example 72 . Road Net Development: Hub Removal:

170. If a hub removal then the road net design_remove_hub behaviour, based on the *road net (development & maintenance) graph*, calculates the *unique hub identifier* of the "isolated" hub to be removed – that is, is not connected to any links,
171. updates the *road net (development & maintenance) graph*, and
172. results in a pair of these.

```
value
170.  design_remove_hub: G → (HI×G)
170.  design_remove_hub(g) as (hi,g′)
170.    let hi:HI · hi ∈ his ∧ let (_,lis) = mereo_H(retr_part(hi)) in lis={} end in
171.    let g′ = M_rh(hi,g) in
172.    (hi,g′) end end
```

173. To remove a hub the road net administrator
    (a) first designs which old hub is to be removed
    (b) then removes the designated hub,
        whereupon it updates the global states
    (c) of parts σ,
    (d) of unique identifiers, and
    (e) of hub identifiers –

    in parallel, and in parallel with

174. stopping the old hub behaviour
175. and resuming being a road net behaviour.

```
value
173. remove_hub: G → RNI → RNMer → Unit
173. remove_hub(g)(rni)(rnmer) ≡
173a.     let (hi,g') = design_remove_hub(g) in
173b.     let h:H · uid_H(h)=hi ∧ ... in
173c.     σ := σ \ {h}
173d.   ‖ uid_σ := uid_σ \ {hi}
173e.   ‖ his := his \ {hi}
174.    ‖ ch[ {rni,hi} ] ! mkStop()
175.    ‖ rn(rni)(rnmer)(g')
173.      end end ■
```

6.5.2 Adjustment of Creatable and Destructable Behaviours

- When an entity
  - is created or destroyed
  - its creation, respectively destruction
  - affects the neurologically related parts and their behaviours.
    - their mereology
    - and possibly their programmable state attributes
    - need be adjusted.
  - And when entities are destroyed
    their behaviours are **stop**ped !
  - These entities are "informed" so by the creator/destructor entity – as was shown in Examples 70–72.
- The next example will illustrate how such 'affected' entities handle such creator/destructor communication.

Example 73 . Hub Adjustments:

- We have not yet illustrated hub (nor link) behaviours.
- Now we have to !

176. The mereology of a hub is a triple:
    the identification of the set of automobiles that may enter the hub,
    the identification of the set of links that connect to the hub,
    and the identification of the road net.
177. The hub behaviour external non-deterministically (□) alternates
    between
178. doing “own work”,
179. or accepting a stop “command” from the road net administrator,
    or
180. or accepting mereology & state update information,
181. or other.

```
type
176. HMer = AI-set × LI-set × RNI
value
176. mereo_H: H → HMer
177. hub: hi:HI → (auis,lis,rni):HMer → hω:HΩ → (hσ:HΣ×ht:HTraffic) →
177.         {ch[ hi,ui ]|ui:(RNI|AI) · ui=rni∨ui ∈ auis}  Unit
177. hub(hi)(hm:(auis,lis,rni))(hω)(hσ,ht) ≡
178.         ...
179.       □ let mkStop() = ch[ hi,rni ] ? in stop end
180.       □ let mkUpdH(hm′,hσ′,hσ′) = ch[ {rni,hi} ] ? in
180.           hub(hi)(hm′)(hω′)(hσ′,ht) end
181.         ...
```

- Observe from formula Item 179 that the hub behaviour ends,
- whereas "from" Item 180 it tail recurses ! ■

6.5.3 Summary on Creatable & Destructable Entities

- We have sketched how we may model
  the dynamics of creating and destroying entities.
  - It is, but a sketch.
  - We should wish for a more methodological account.
  - So, that is what we are working on – amongst other issues – at the moment.

6.6 Domain Engineering: Description and Construction

- There are two meanings to the term 'Domain Engineering'.
  - the construction of *descriptions* of domains, and
  - the construction of *domains*.
  - Most sections of Chapters 4–6 are "devoted" to the former;
  - the previous section, Sect. 6.5 to the latter.

6.7 Domain Laws

TO BE WRITTEN

6.8 A Domain Discovery Procedure, III

The predecessors of this section are Sects. 4.8.2 on Slide 159 and 5.7 on Slide 361.

6.8.1 Review of the Endurant Analysis and Description Process

- The discover_... functions below were defined in Sects. 4.8.2 on Slide 159 and 5.7 on Slide 361.

```
value
  endurant_analysis_and_description: Unit → Unit
  endurant_analysis_and_description() ≡
      discover_sorts();                                [Page 160]
      discover_internal_endurant_qualities()   [Page 363]
```

- We are now to define a perdurant_analysis_and_description procedure –
- to follow the above endurant_analysis_and_description procedure.

6.8.2 A Domain Discovery Process, III

- We define the perdurant_analysis_and_description procedure
  - in the reverse order of that of Sect. 5.7 on Slide 361,
  - first the full procedure,
  - then its sub-procedures.

A Domain Endurant Analysis and Description Process

```
value
  perdurant_analysis_and_description: Unit → Unit
  perdurant_analysis_and_description() ≡
    discover_state();                          axiom ...   [ Note (a) ]
    discover_channels();                       axiom ...   [ Note (b) ]
    discover_behaviour_signatures(); axiom ...   [ Note (c) ]
    discover_behaviour_definitions(); axiom ...   [ Note (d) ]
    discover_initial_system()            axiom ...   [ Note (e) ]
```

- *Notes*:
  - (a) The States: σ and ui_σ
    - We refer to Sect. 4.7.2 on Slide 155 and Sect. 5.2.4 on Slide 194.
    - The state calculation, as shown on Page 150, must be replicated, i.e., re-discovered, in any separate domain analysis & description.
    - The purpose of the state, i.e., σ, is to formulate appropriate axiomatic constraints and domain laws.
  - (b) The Channels:
    - We refer to Sects. 6.1.2 on Slide 378 and 6.2 on Slide 379.
    - Thus we indiscriminately declare a channel for each pair of distinct unique part identifiers
      whether the corresponding pair of part behaviours,
      if at all invoked, communicate or not.

– (c) **Behaviour Signatures:**
  * We refer to Sect. 6.3.1.2 on Slide 382.
  * We find it more productive to first settle on the signatures of all behaviours – careful thinking has to go into that –
  * before tackling the far more time-consuming work on defining the behaviours:
– (d) **Behaviour Definitions:**
  * We refer to Sect. 6.3.3 on Slide 392.
– (e) **The Running System:**
  * We refer to Sect. 6.4 on Slide 401.

6.9 Summary

Perdurants: Analysis & Description: Method Tools

#	Name	Introduced
	Discovery Functions	
	discover_channels	page 379
	discover_behaviour_signatures	page 391
	discover_behaviour_definitions	page 394
	discover_initial_system	page 401
	perdurant_analysis_and_description	page 439

• • •

- Please consider Fig. 4.1 on Slide 64.
  - This chapter has covered the right of Fig. 4.1.

CHAPTER 7. Summary of the TU Wien Lectures

- Traversal of Analysis & Description Ontology Graph

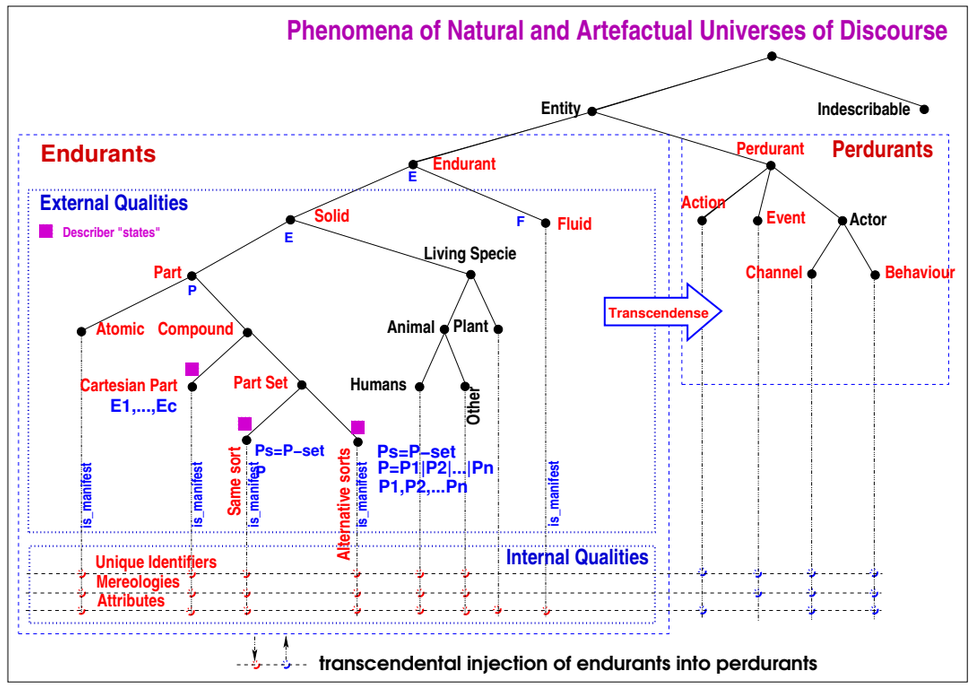

Figure 7.1: Upper Ontology

- Axioms, Well-formedness and Proof Obligations
  - Someone in the audience may have noticed
    that this *primer* hardly mentions the notion of verification,
    - yet domain descriptions,
      as possibly any specification related to software,
    - may require some form of verification.
  - Yet this *primer* appears to skirt the issue.
  - Indeed, we have, regrettably, omitted the issue.
  - So we must refer to student to relevant literature.
  - We cannot, October 20, 2022, point to any definitive book on the topic.
  - The field is under intense research.
  - Instead we refer to such diverse papers as: [23, 31].

- In *endurant description prompts* 2 on Slide 102, 3 on Slide 115 and 4 on Slide 119 we mention the concept of proof obligation. They are also mentioned in *attribute description prompt* 7 on Slide 237.
- In numerous other places we mention the concept of *axiom*: 49 on Slide 198 (uniqueness of unique identifiers), 6 on Slide 211 (mereology), 94 on Slide 279 (disjointness of attribute categories), 121 on Slide 309 (property of time),
- And in some places we mention the concept of *well-formedness*, f.ex., Sect. 5.6.2.4 on Slide 343,

– **Axioms** express properties of endurants, whether external or internal qualities, that holds – as were they laws of the domain.
– **Well-formedness** predicates are defined where external or internal qualities of endurants are defined by concrete types in such ways as to warrant such predicates.
– **Proof obligations** are usually warranted where distinct sort definitions need be separated.

- From Programming Language Semantics to Domain Models
  - Programming Language Semantics
    - The *IBM Vienna Labor* PL/I Definition, 1974.
    - The *Dansk Datamatik Center, DDC* CHILL and Ada Formal Descriptions, 1978-1985.
  - Domain Models give semantics to
    - nouns, endurants, and
    - verbs, perdurants,

of domains.

- Domain Specific Languages: DSL
  - A **DSL** is a language whose "primitives" directly reflects a specific domain's basic entities.
  - **RSL** is not a 'domain specific language'
  - **Domain Models** form the basis for the conception of one or more specific **DSL**s.
  - The **semantics** of these **DSL**s derives, then, from the **Domain Model.**
  - It is suggested, therefore, that **DSL**s be conceived on the basis of domain models.

- RSL vs. RSL$^+$
  - Informal RSL$^+$ is used
    in explaining the domain analysis & description method.
  - RSL is used, independently,
    as the formal specification language
    in which to describe domains.
  - The two are otherwise unrelated !

- Algorithms vs. Domain Descriptions
  - Algorithms are 'the' hallmark of Computing !
  - Clever algorithms (and data structures) are needed to efficiently implement requirements prescriptions.
  - Thus algorithmics enter our concern during software design.

- Domain Facets
  - Intrinsics
  - Technology Support
  - Rules & Regulations
  - Scripts, Contracts
  - Management & Organisation
  - Human Behaviour

- Requirements Engineering
  - 'The Machine'
  - Domain Engineering
    - Projection
    - Instantiation
    - Determination
    - Extension
    - Fitting
  - Interface and Derived Requirements
    - Interface Requirements
      - Shared Endurants
      - Shared Perdurants
    - Derived Requirements
      - Shared with Machine
  - Machine Requirements

- Research/PhD Study Topics
  - Intentional Pull
  - Discrete vs Continuous
  - A Calculus of Perdurants
  - Human Interaction
  - Transcendental Deduction
  - Formal Ontology Models
  - Philosophy

THANKS