Dines Bjørner's MAP-i Lecture # 5

Perdurants: Actions, Events and Behaviours

Tuesday, 26 May 2015: 10:00–10:45

1.3. Perdurant Entities

- We shall give only a cursory overview of perdurants.
- That is, 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, through this cursory overview, we can justify our detailed study of endurants,
    - their part and subparts,
    - their unique identifiers, mereology and attributes.

- This justification is manifested
  - (i) in expressing the types of **signature**s,
  - (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**.

1.3.1. 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_material` *s* ■

Example 53 . States: Some examples of states are:

- A road hub can be a state,
  cf. Hub State, HΣ, Example 38 on Slide 181.
- A road net can be a state – since its hubs can be.
- Container stowage areas, CSA, Example 22 on Slide 135, 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 potentially carry components

■

1.3.2. Actions, Events and Behaviours

- To us perdurants are further 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.

On Action, Event and Behaviour Distinctions:

- The distinction into action, event and behaviour perdurants is pragmatic.

1.3.2.1. 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].
- For a seminal book on "time in computing" we refer to the eclectic *Modeling Time in Computing, Springer 2012*.
- And for seminal book on time at the epistemology level we refer to *The Logic of Time, Kluwer 1991*.

1.3.2.2. 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 54 . 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 Examples 20 on Slide 123 and 43 on Slide 198 can be considered actors;
- so can the net and its links and hubs.
- The pipeline monitor and each pipeline unit of the *Pipeline System*, Example 27 on Slide 140 and Examples 27 on Slide 140 and 33 on Slide 162 will be considered actors.
- The bank general ledger and each bank customer of the *Shared Passbooks* example, Example 44 on Slide 201, will be considered actors ■

1.3.2.3. Parts, Attributes and Behaviours

- Example 54 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 55 . 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 ■

1.3.3. Discrete Actions

Definition 13 . Discrete Action: *By a* **discrete action** *[54] 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 56 . Road Net Actions:

- Examples of *Road Net* actions initiated by the net actor are:
  - **ins**ertion of **h**ubs,
  - **ins**ertion of **l**inks,
  - **rem**oval of **h**ubs,
  - **rem**oval of **l**inks,
  - **set**ting of **h**ub states.
- Examples of *Traffic System* actions initiated by vehicle actors are:
  - **mov**ing a vehicle along a link,
  - **stop**ping a vehicle,
  - **start**ing a vehicle,
  - **enter**ing a hub and
  - **leav**ing a hub ■

1.3.4. Discrete Events

Definition 14 . Event: *By an* **event** *we shall understand*

- *some unforeseen thing,*
- *that is, some ‘not-planned-for’ “action”, one*
- *which surreptitiously, non-deterministically changes a well-formed state*
- *into another, but usually not a well-formed state,*
- *and for which no particular domain actor can be made responsible* ■

- Events can be characterised by
  - a pair of (before and after) states,
  - a predicate over these
  - and, optionally, a **time** or **time interval**.
- The notion of event continues to puzzle philosophers [36, 51, 49, 35] [41, 2, 47, 34] [50, 33].
- We note, in particular, [35, 2, 47].

Example 57 . Road Net and Road Traffic Events:

- Some road net events are:
  - “disappearance” of a hub or a link,
  - failure of a hub state to change properly when so requested, and
  - occurrence of a hub state leading traffic into “wrong-way” links.
- Some road traffic events are:
  - the crashing of one or more vehicles (whatever ‘crashing’ means),
  - a car moving in the wrong direction of a one-way link, and
  - the clogging of a hub with too many vehicles ■

1.3.5. Discrete Behaviours

Definition 15 . Discrete Behaviour: *By a* **discrete behaviour** *we shall understand*

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

Example 58 . Behaviours:

- Examples of behaviours:
  - **Road Nets:** A sequence of hub and link insertions and removals, link disappearances, etc.
  - **Road Traffic:** A sequence of movements of vehicles along links, entering, circling and leaving hubs, crashing of vehicles, etc.
  - **Pipelines:** A sequence of pipeline pump and valve openings and closings, and failures to do so (events), etc.
  - **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 ■

1.3.5.1. Channels and Communication

- Behaviours
  - sometimes synchronise
  - and usually communicate.
- We use CSP to 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]|ide:IDE}:M

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

1.3.5.2. 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,
  - part components and
  - part materials

  with behaviour states.

- Thus, for every pair of parts, p_{ik}:P_i and $\mathsf{p}_{j\ell}$: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 $\mathsf{p}_{j\ell}$:P_j, of these sorts, then just a simple channel, say ch_{P_i,P_j}.

      channel 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 a *vector* of channels.
      - Let $\{p_{j1}, p_{j2}, ..., p_{jn}\}$ be all the part of the domain of 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 $\{\mathsf{ch}[\,\{\pi_i,\pi_j\}\,]|\pi_i{=}\mathbf{uid\_}\mathsf{P}_i(\mathsf{p}_i)\wedge\pi_j \in \mathsf{uids}\}$:A.

Example 59 . Bus System Channels:

- We extend Examples 20 on Slide 123 and 43 on Slide 198.
- We consider the **fleet** and the **vehicle**s to be behaviours.

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

91 the **fleet** and

92 the **vehicles**.

93 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
90.     δ:Δ,
91.     f:F = obs_part_F(δ),
92.     vs:V-set = obs_part_Vs(obs_part_VC((obs_part_F(δ))))
    channel
93.     {fch[{uid_F(f),uid_V(v)}]|v:V·v ∈ vs}:BT
```

Example 60 . Bank System Channels:

- We extend Example 44 on Slide 201.
- We consider the **general ledger** and the **customers** to be behaviours.

94 We assume some **bank system**. From the **bank system**

95 we observe the **general ledger**.

96 and the set of **customers**.

97 We consider **passbooks** to be the only message communicated between the **general ledger** and the **customer** behaviours.

```
value
94.  bs:BS
95.  gl=obs_part_GL(obs_part_AD(bs)):GL
96.  cs=obs_part_Cs(obs_part_CS(bs)):C-set
channel
97.  {bsch[{uid_GL(gl),uid_C(c)}]|c:C•c ∈ cs}:PB ■
```

1.3.6. 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 61 . Flow in Pipelines:

- We refer to Examples 33, 48, 49, 50 and 51.
- 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).

1.3.7. Attribute Value Access

- We can distinguish between three kinds of attributes:
  - the **constant attributes** which are those whose values are **static**;
  - the **programmable attributes** which are those dynamic values are exclusively set by part processes; and
  - the remaining **dynamic attributes**

  are here seen as **individual behaviours**.

1.3.7.1. Access to Static Attribute Values

- The **constant attributes** can be "copied" **attr_A(p)** (and retain their values).

1.3.7.2. Access to External Attribute Values

- By the **external behaviour attributes**
  - we shall thus understand the
    - inert,
    - reactive,
    - autonomous and the
    - biddable

    attributes ■

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

99 Let $\Pi_{\xi A}$ be the set of indexes into the **external attribute channel**, say **attr_A_ch**, one for each distinct attribute name, A, in ξA.

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

101 External attribute values are then accessed by the input, from channel **attr_A_ch**[π]-accessible external attribute behaviours.

102 The **type** of **attr_A_ch**[π] is considered to be **Unit**$\overset{\sim}{\rightarrow}$A.

98. **value**

98. ξA: $\{\eta$A$|$A is any external attribute name$\}$

99. $\Pi_{\xi A}$: Π**-set**

100. **channel**

100. $\{$attr_A_ch$[\pi]|\pi \in \Pi_{\xi A}\}$

101. **value**

101. attr_A_ch$[\pi]$?

101. **type**

101. attr_A_ch$[\pi]$: **Unit**$\overset{\sim}{\rightarrow}$A $[$abbrv.:$\mathbb{U}$A$]$

- We shall omit the η prefix in actual descriptions.
- The choice of representing **external behaviour attribute**s as behaviours is a technical one.
- See Items 187c. and 187a. Slide 426 for a use of the concept of **external behaviour attribute** channels.

1.3.7.3. Access to Programmable Attribute Values

- The **programmable attributes** are treated as function arguments.
- This is a technical choice. It is motivated as follows.
  - We find that programmable attribute values are set (i.e., updated) by part processes.
  - 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:

$$\textbf{value} \;\; \mathsf{f}\text{: } \mathsf{Arg} \to \mathsf{Arg}; \;\; \mathsf{f(a)} \equiv (\ldots \; \textbf{let} \; \mathsf{a}' = \mathsf{F}(\ldots)(\mathsf{a}) \; \textbf{in} \; \mathsf{f}(\mathsf{a}') \; \textbf{end})$$

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

1.3.8. Perdurant Signatures and Definitions

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

Definition 16 . Function Signature: *By a* **function signature** *we shall understand*

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

Definition 17 . 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* $\overset{\sim}{\rightarrow}$ *(partial function)* ■

- The type expressions
  - are usually part sort or type, material sort or attribute type names,
  - but may, occasionally be expressions over respective type names involving **-set**, $\times$, *, $\underset{m}{\rightarrow}$ and $|$ type constructors.

1.3.9. Action Signatures and Definitions

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

$$\begin{array}{l} \mathsf{action}\colon\ \mathsf{VAL} \to \Sigma \overset{\sim}{\to} \Sigma \\ \mathsf{action(v)}(\sigma)\ \textbf{as}\ \sigma' \\ \quad \textbf{pre}\colon\ \ \mathcal{P}(\mathsf{v},\sigma) \\ \quad \textbf{post}\colon\ \mathcal{Q}(\mathsf{v},\sigma,\sigma') \end{array}$$

  - 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 $\overset{\sim}{\rightarrow}$
  - 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 $\mathcal{P}$(v,σ).
- The post condition $\mathcal{Q}$(v,σ,σ') characterises the “after” state, σ':Σ, with respect to the “before” state, σ:Σ, and possible arguments (v:VAL).

Example 62 . Insert Hub Action Formalisation: We formalise aspects of the above-mentioned hub and link actions:

103 Insertion of a hub requires

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

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

value

103. $\mathsf{insert\_H}: \mathsf{H} \to \mathsf{N} \overset{\sim}{\to} \mathsf{N}$

103. $\mathsf{insert\_H(h)(n)}$ **as** $\mathsf{n'}$

104. **pre**: $\sim\exists\ \mathsf{h'}:\mathsf{H}\cdot\mathsf{h'} \in \mathbf{obs\_part}\mathsf{\_Hs}(\mathbf{obs\_part}\mathsf{\_HS(n)})\cdot\mathbf{uid}\mathsf{\_H(h)}=\mathbf{uid}\mathsf{\_H(h')}$

105. **post**: $\mathbf{obs\_part}\mathsf{\_Hs}(\mathbf{obs\_part}\mathsf{\_HS(n')})=\mathbf{obs\_part}\mathsf{\_Hs}(\mathbf{obs\_part}\mathsf{\_HS(n)})\cup\{\mathsf{h}\}$ ■

- Which could be the argument values, v:VAL, of actions ?
  - Well, there can basically be only two 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[18],
  - ⟐ ascribing signatures to action functions, and
  - ⟐ determining
    - ⊚ action pre-conditions and
    - ⊚ action post-conditions.
  - ⟐ Of these, ascribing signatures is, perhaps, the most crucial:
    - ⊚ In the process of determining the action signature
    - ⊚ one oftentimes discovers
    - ⊚ that part or material attributes have been left “undiscovered”.

---

[18] 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 63 shows examples of signatures whose arguments are
    - either parts,
    - or parts and unique identifiers,
    - or parts and unique identifiers and attributes.

Example 63 . Some Function Signatures:

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

  $$\textbf{value}\ \mathsf{insert\_L}\colon \mathsf{L} \times (\mathsf{HI} \times \mathsf{HI}) \to \mathsf{N} \stackrel{\sim}{\to} \mathsf{N}$$

- Removing a hub and removing a link:

  $$\begin{aligned}\textbf{value}\ \mathsf{remove\_H}&\colon \mathsf{HI} \to \mathsf{N} \stackrel{\sim}{\to} \mathsf{N}\\ \mathsf{remove\_L}&\colon \mathsf{LI} \to \mathsf{N} \stackrel{\sim}{\to} \mathsf{N}\end{aligned}$$

- Changing a hub state.

  $$\textbf{value}\ \mathsf{change\_H\Sigma}\colon \mathsf{HI} \times \mathsf{H\Sigma} \to \mathsf{N} \stackrel{\sim}{\to} \mathsf{N}$$ ■

1.3.10. Event Signatures and Definitions

- Events are usually characterised by
  - the absence of known actors and
  - the absence of explicit "external" arguments.
- Hence the schematic function (event) signature:

$$\begin{array}{l}\textbf{value}\\ \quad \mathsf{event}\text{: } \Sigma \times \Sigma \rightarrow \textbf{Bool}\\ \quad \mathsf{event}(\sigma,\sigma') \textbf{ as true} \sqcap \textbf{false}\\ \qquad \textbf{pre}\text{: } P(\sigma)\\ \qquad \textbf{post}\text{: } Q(\sigma,\sigma')\end{array}$$

- The event signature expresses
  - that a selection of the domain
  - as provided by the Σ type expression
  - is “acted” upon, by unknown actors, and possibly changed.
- The partial function type operator $\overset{\sim}{\to}$
  - shall indicate that $\mathsf{event}(\sigma,\sigma')$
  - may not be defined for some states σ.
- The resulting state may, or may not, satisfy axioms and well-formedness conditions over Σ — as expressed by the post condition $Q(\sigma,\sigma')$.

- Events may thus cause well-formedness of states to fail.
- Subsequent actions,
  - once actors discover such “disturbing events”,
  - are therefore expected to remedy that situation, that is,
  - to restore well-formedness.
- We shall not illustrate this point.

Example 64 . Link Disappearence Formalisation: We formalise aspects of the above-mentioned link disappearance event:

106 The result net is not well-formed.

107 For a link to disappear there must be at least one link in the net;

108 and such a link may disappear such that

109 it together with the resulting net makes up for the “original” net.

```
value
106.  link_diss_event: N × N′ × Bool
106.  link_diss_event(n,n′) as tf
107.     pre:  obs_part_Ls(obs_part_LS(n))≠{}
108.     post: ∃ l:L·l ∈ obs_part_Ls(obs_part_LS(n)) ⇒
109.                l ∉ obs_part_Ls(obs_part_LS(n′))
109.                ∧ n′ ∪ {l} = obs_part_Ls(obs_part_LS(n))  ■
```

1.3.11. Discrete Behaviour Signatures and Definitions

- We shall only cover behaviour signatures when expressed in `RSL/CSP` [39].
- 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 is "revealed" in the function signature as follows:

  behaviour: ... → **in** 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

- As shown in [21] 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.
- The process evolves around a state:
  - its unique identity, $\pi : \Pi$,,
  - its possibly changing mereology, mt:MT[19],
  - the possible components and materials of the part[20], and
  - the constant, the external and the programmable attributes of the part.

---

[19] For MT see footnote 12 on Slide 158.

[20] — we shall neither treat components nor materials further in this document

- A behaviour signature is therefore:

  behaviour: π:Π × me:MT × sa:SA × ea:EA → pa:PA → **out** ochs **in** ichns **Unit**

  where

  - (i) π:Π 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 behaviour,
  - (iv) ea:EA lists the external attribute channels of the part behaviour,
  - (v) ps:PA lists the programmable attribute values of the part behaviour, and where
  - (vi) ochs and ichns refer to the shared attributes of the behaviours.

- We focus, for a little while, on the expression of
  - sa:SA,
  - ea:EA and
  - pa:PA,
- that is, on the concrete types of SA, EA and PA.
  - $\mathcal{S}_{\mathcal{A}}$: SA simply lists the static value types: $svT_1, svT_2, ..., svT_s$ where s is the number of static attributes of parts p:P.
  - $\mathcal{E}_{\mathcal{A}}$ EA simply lists the channel indexes to the external attribute values: $((eA_1, \pi_{eA_1}), (eA_2, \pi_{eA_2}), ..., (eA_x, \pi_{eA_x}))$[21] where x is the number, 0 or more, of external attributes of parts p:P.
  - $\mathcal{P}_{\mathcal{A}}$ PA simply lists appropriate programmable value expression type: $(pvT_1, pvT_2, ..., pvT_q)$ where q is the number of programmable attributes of parts p:P

---

[21] See paragraph *Access to External Attribute Values* on Slide 274.

- Let P be a composite sort defined in terms of sub-sorts PA, PB, . . . , PC.
  - ◈ The process compiled from cp:P, is composed from
    - ⊚ a process, $\mathcal{M}_{cP_{\text{CORE}}}$, relying on and handling the unique identifier, mereology and attributes of process p as defined by P
    - ⊚ operating in parallel with processes $p_a, p_b, \ldots, p_c$ where
      - ∗ p_a is “derived” from PA,
      - ∗ p_b is “derived” from PB,
      - ∗ ..., and
      - ∗ p_c is “derived” from PC.
- 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) ≡
    $\mathcal{M}_{cP_{\text{CORE}}}$(**uid**_P(p),**obs_mereo**_P(p),$\mathcal{S}_{\mathcal{A}}$(p),$\mathcal{E}_{\mathcal{A}}$(p))($\mathcal{P}_{\mathcal{A}}$(p))
    ‖ compile_process(**obs_part**_PA(p))
    ‖ compile_process(**obs_part**_PB(p))
    ‖ ...
    ‖ compile_process(**obs_part**_PC(p))

- The text macros: $\mathcal{S}_{\mathcal{A}}$, $\mathcal{E}_{\mathcal{A}}$ and $\mathcal{P}_{\mathcal{A}}$ were informally explained above.
- Part sorts PA, PB, ..., PC are obtained from the `observe_part_sorts` prompt, Slide 122.

- Let P be a composite sort defined in terms of the concrete type **Q-set**.
  - ⟐ The process compiled from p:P, is composed from
    - ⊚ a process, $\mathcal{M}_{cP_{\mathrm{CORE}}}$, 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) ≡
          $\mathcal{M}_{cP_{\mathrm{CORE}}}$(uid_P(p),obs_mereo_P(p),$\mathcal{S}_{\mathcal{A}}$(p),$\mathcal{E}_{\mathcal{A}}$(p))($\mathcal{P}_{\mathcal{A}}$(p))
        ‖ ‖{compile_process(q)|q:Q•q ∈ qs}
```

Process Schema III: `is_atomic(p)`

```
value
    compile_process: P → RSL-Text
    compile_process(p) ≡
        $\mathcal{M}_{aP_{\mathrm{CORE}}}$(uid_P(p),obs_mereo_P(p),$\mathcal{S}_{\mathcal{A}}$(p),$\mathcal{E}_{\mathcal{A}}$(p))($\mathcal{P}_{\mathcal{A}}$(p))
```

Example 65 . Bus Timetable Coordination:

- We refer to Examples 20 on Slide 123, 21 on Slide 130, 43 on Slide 198 and 59 on Slide 265.

110 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.

111 The **fleet** process is compiled as per Process Schema II (Slide 297)

```
    type
        $\Delta$, F, VC         [Example 20 on Slide 123]
        V, Vs=V-set    [Example 21 on Slide 130]
        FI, VI, BT     [Example 43 on Slide 198]
    channel
        {fch...}       [Example 59 on Slide 265]
    value
110.        $\delta:\Delta$,
110.        f:F = obs_part_F($\delta$),
110.        vs:V-set = obs_part_Vs(obs_part_VC(f)),
110.        bt:BT = attr_BT(f)
    axiom
110.        $\forall$ v:V•v $\in$ vs $\Rightarrow$ bt = attr_BT(v) [Example 43 on Slide 198]
    value
111.        fleet: fi:FI$\times$BT $\rightarrow$ in,out {fch[{fi,uid_V(v)}]|v:V•v $\in$ vs} process
111.        fleet(fi,bt) $\equiv$
111.            $\mathcal{M}_F$(fi,bt)
111.            $\|$ $\|$ {vehicle(uid_V(v),fi:FI,bt)|v:V•v $\in$ vs}
111.        vehicle: vi:VI$\times$fi:FI$\times$bt:BT $\rightarrow$ in,out fch[{fi,vi}] process
111.        vehicle(vi,fi,bt) $\equiv$ $\mathcal{M}_V$(vi,fi,bt)
```

- Fleet and vehicle processes
  - $\mathcal{M}_F$ and
  - $\mathcal{M}_V$
- are both "never-ending" processes:

value

$\mathcal{M}_F$: $\mathsf{fi{:}FI}\times\mathsf{bt{:}BT} \rightarrow$ **in**,**out** $\{\mathsf{fch}[\,\{\mathsf{fi},\mathbf{uid}\_\mathsf{V}(\mathsf{v})\}\,]|\mathsf{v{:}V}\cdot\mathsf{v} \in \mathsf{vs}\}$ **process**

$\mathcal{M}_F(\mathsf{fi,bt}) \equiv$ **let** $\mathsf{bt}' = \mathcal{F}(\mathsf{fi,bt})$ **in** $\mathcal{M}_F(\mathsf{fi,bt}')$ **end**

$\mathcal{M}_V$: $\mathsf{vi{:}VI}\times\mathsf{fi{:}FI}\times\mathsf{bt{:}BT} \rightarrow$ **in**,**out** $\mathsf{fch}[\,\{\mathsf{fi,vi}\}\,]$ **process**

$\mathcal{M}_V(\mathsf{vi,fi,bt}) \equiv$ **let** $\mathsf{bt}' = \mathcal{V}(\mathsf{vi,bt})$ **in** $\mathcal{M}_V(\mathsf{vi,fi,bt}')$ **end**

- The "core" processes,
  - $\mathcal{F}$ and
  - $\mathcal{V}$,

are simple actions.

- In this example we simplify them to change only bus timetables.
- The expression of actual synchronisation and communication between the **fleet** and the **vehicle** processes are contained in $\mathcal{F}$ and $\mathcal{V}$.

value

$\mathcal{F}$: fi:FI×bt:BT → **in**,**out** {fch[{fi,**uid**_V(v)|v:V·v ∈ vs}]} BT
$\mathcal{F}$(fi,bt) ≡ ...

$\mathcal{V}$: vi:VI×fi:FI×bt:BT → **in**,**out** fch[{fi,vi}] BT
$\mathcal{V}$(vi,fi,bt) ≡ ...

- What the synchronisation and communication between the **fleet** and the **vehicle** processes consists of we leave to the reader ! ■

Process Schema IV: Core Process (I)

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

$\mathcal{M}_{cP_{\mathrm{CORE}}}$: π:Π×me:MT×sa:SA×ea:EA→pa:PA→**in** inchs **out** ochs **Unit**
$\mathcal{M}_{cP_{\mathrm{CORE}}}(\pi$,me,sa,ea)(pa) $\equiv$
    let (me′,pa′) = $\mathcal{F}(\pi$,me,sa,ea)(pa) **in**
    $\mathcal{M}_{cP_{\mathrm{CORE}}}(\pi$,me′,sa,ea)(pa′) **end**

$\mathcal{F}$: π:Π×me:MT×sa:SA×ea:EA→PA→**in** inchs **out** ochs → MT×PA

- $\mathcal{F}$
  - potentially communicates with all those part processes (of the whole domain)
  - with which it shares attributes, that is, has connectors.
  - $\mathcal{F}$ 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.
  - $\mathcal{F}$ also contains expressions, attr_ch[(A,π)] ?, to external attributes.
- An example of the update of programmable attributes is shown in the vehicle definitions in Sect. 6.2.3, Slides 344 and 346.

- The $\mathcal{F}$ action non-deterministically internal choice chooses between
  - either [1,2,3,4]
    - [1] accepting input from
    - [4] another part process,
    - [2] then optionally offering a reply to that other process, and
    - [3] finally delivering an updated state;
  - or [5,6,7,8] offering
    - [5] an output,
    - [6] val,
    - [8] to another part process,
    - [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:Π → me:MT → sa:SA × ea:EA → pa:PA → in,out 𝓔(π,me)  MT × PA
    𝓕(π,me,sa,ea)(pa) ≡
[1]      ⊔⊓{ let val = ch[π′] ? in
[2]            ch[π′] ! in_reply(sa,ea,pa)(val) ;
[3]            in_update(me,sa,ea,pa)(π′,sa,ea,pa) end
[4]          | π′ ∈ 𝓔(π,me)}
[5]    ⊓⊔⊓{ let (π′,val) = await_reply(me,sa,ea,pa) in
[6]            ch[π′] ! out_reply(val,sa,ea,pa) ;
[7]            out_update(me,sa,ea,pa) end
[8]          | π′ ∈ 𝓔(π,me)}
[9]    ⊓       (me,own_work(sa,ea,pa))

    in_reply:  SA×EA×PA × VAL → VAL
    in_update: (MT×SA×EA×PA) → (MT×PA)
    await_reply: (MT×SA×EA×PA) → Π×VAL
    out_reply: (SA×EA×PA×VAL) → VAL
    out_update: (MT×SA×EA×PA) → (MT×PA)
    own_work: SA×EA×PA → (MT×PA)
```

1.3.12. Concurrency: Communication and Synchronisation

- Process Schemas I, II and IV (Slides 295, 297 and 305), 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**.

1.3.13. 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.

1.3.13.1. 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.

1.3.13.2. Discussion

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

Dines Bjørner's MAP-i Lecture # 5

End of MAP-i Lecture # 5:
Perdurants: Actions, Events and Behaviours

Tuesday, 26 May 2015: 10:00–10:45