LAST HAUL BEFORE LUNCH

Begin of Lecture 5: Last Session — Perdurant Entities

Behaviours, Discussion Entities

FM 2012 Tutorial, Dines Bjørner, Paris, 28 August 2012

Tutorial Schedule

• **Lectures 1–2**	9:00–9:40 + 9:50–10:30	
1 **Introduction**		Slides 1–35
2 **Endurant Entities: Parts**		Slides 36–110
• **Lectures 3–5**	11:00–11:15 + 11:20–11:45 + 11:50–12:30	
3 **Endurant Entities: Materials, States**		Slides 111–142
4 **Perdurant Entities: Actions and Events**		Slides 143–174
√ 5 **Perdurant Entities: Behaviours**		**Slides 175–285**
Lunch	**12:30–14:00**	
• **Lectures 6–7**	14:00–14:40 + 14:50–15:30	
6 **A Calculus: Analysers, Parts and Materials**		Slides 286–339
7 **A Calculus: Function Signatures and Laws**		Slides 340–377
• **Lectures 8–9**	16:00–16:40 + 16:50–17:30	
8 **Domain and Interface Requirements**		Slides 378–424
9 **Conclusion: Comparison to Other Work**		Slides 428–460
Conclusion: What Have We Achieved		Slides 425–427 + 461–472

8.4. Discrete Behaviours

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

- Continuous behaviours
  - ⟐ are **continuous** in the sense of the **calculus** of **mathematical**;
  - ⟐ to qualify as a **continuous behaviour time** must be an essential aspect of the **behaviour**.
  - ⟐ We shall treat **continuous behaviours** in Sect. 9.
- **Discrete behaviours** can be modelled in many ways, for example using
  - ⟐ `CSP` [Hoare85+2004].
  - ⟐ `MSC` [MSCall],
  - ⟐ `Petri Nets` [m:petri:wr09] and
  - ⟐ `Statechart` [Harel87].
- We refer to Chaps. 12–14 of [TheSEBook2wo].
- In this tutorial we shall use `RSL/CSP`.

8.4.1. What is Meant by 'Behaviour' ?

- We give two characterisations of the concept of 'behaviour'.
  - a "loose" one and
  - a "slanted one.
- A loose characterisation runs as follows:
  - by a **behaviour** we understand
    - a set of sequences of
    - **action**s, **event**s and **behaviour**s.

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

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

8.4.2. Behaviour Narratives

- Behaviour narratives may take many forms.
  - A behaviour may best be seen as composed from several interacting behaviours.
    - Instead of narrating each of these,
    - as will be done in Example 36,
    - one may proceed by first narrating the interactions of these behaviours.
  - Or a behaviour may best be seen otherwise,
    - for which, therefore, another style of narration may be called for,
    - one that "traverses the landscape" differently.
  - Narration is an art.
  - Studying narrations – and practice – is a good way to learn effective narration.

Example: 35 A Transport Behaviour Narrative.

- Our example is that of a vehicle monitoring system.
- That is, a system of a road net, a fleet of vehicles and a road monitor.
- That is, we take that as a[n existing] domain.
- In other words, it is not a requirements prescription.

28. From a **vehicle monitoring system, VMS** one can observe
    (a) a [road] **net, n:N**,
    (b) a **fleet, f:F** of vehicles and
    (c) a **road monitor, m:M**.
29. From a fleet of vehicles one can observe a set of uniquely identified (vi:VI) vehicles (v:V). We consider vehicles to be atomic parts.
30. We consider the road monitor to be an atomic part.

31. At any one time vehicles are positioned
    (a) at hubs or
    (b) along links —
    (c) where hub positions indicate the link from where the vehicle arrived at the hub and the link to where it is aimed, i.e., atH(fli:LI,hi:HI,tli:LI), and
    (d) where link positions indicate the hub from where the vehicle arrived at the link and the hub to where it is aimed, i.e., onL(fhi:HI,li:LI,frac:FRAC,thi:HI), where frac designates the fraction “down” the link that the vehicle has so far travelled.

32. And at any one time, t, vehicles
    (a) are either standing still
    (b) or moving —
    (c) where vehicle positions at times t and the immediate next times t' are unchanged, respectively
    (d) have changed (where we do not record immediate next time, i.e., incremental hub position changes):
        i. either atH(fli,hi,tli) and atH(fli,hi,tli) or
        ii. onL(fhi,li,f,thi) and onL(fhi,li,f+δ,thi) where δ is a tiny positive increment ($0 < \delta \ll 1$).
33. Whenever a vehicle has or has not moved the road monitor is informed about its new position. ■

8.4.3. An Aside on Agents, Behaviours and Processes

- "In philosophy and sociology, agency is the capacity of an agent (a person or other entity) to act in a world."
- "In philosophy, the agency is considered as belonging to that agent even if that agent represents a fictitious character, or some other non-existent entity."
- That is, we consider agents to be those persons or other entities that
  - are in the domain and
  - observes the domain
  - evaluates what is being observed
  - and invokes actions.
- We describe agents by describing behaviours.

- A behaviour description denotes a process, that is, a set of
  - actions,
  - events and
  - processes.
- We shall not enter into any further speculations on
  - agency,
  - agents and
  - how agents observe, including
    - what they know and believe (**epistemic logic**),
    - what is necessary and possible (**deontic logic**) and
    - what is true at some tie and what is always true (**temporal logic**).
  - A proper domain science and engineering must, however, eventually examine these (**modal logic**) issues.

8.4.4. On Behaviour Description Components

- When narrating plus, at the same time, formalising,
  - i.e., textually alternating between
  - narrative texts and
  - formal texts,
- one usually starts with what seems to be the most important behaviour concepts of the given domain:
  - which are the important part types characterising the domain;
  - which of these parts will become a basis for behaviour processes;
  - how are these behaviour processes to interact,
  - that is, which channels and what messages may possibly be communicated.

Example: 36 A Transport Behaviour Formalisation.

We continue Example 35.

- We refer to narrative Items 28–28(c) (Page 182).

type

28. VMS, N, F, M

value

28(a). obs_N: VMS $\rightarrow$ N
28(b). obs_F: VMS $\rightarrow$ F
28(c). obs_M: VMS $\rightarrow$ M

34. Vehicles are here considered atomic parts

35. with unique identifiers.

type
34. V, VI
value
35. uni_Π: V → VI

- We refer to Items 29–28(c) (Slide 182).
- We introduce a number of values of the vehicle monitoring system.

36. A net.
37. The set of hubs.
38. The set of links.
39. The vehicle fleet observer function.
40. The fleet.
41. The set of vehicles of that fleet.
42. The set of unique identifiers of those vehicles.
43. The monitor.

value

36. n:N = obs_N(VMS)
37. hs:H-**set** = obs_Hs(n)
38. ls:L-**set** = obs_Ls(n)
39. obs_Vs: F → V-**set**
40. f:F
41. vs:V-**set** = obs_Fs(f)
42. vis:VI-**set**={uid_Π(v)|v:V·v∈vs}
43. m:M

- We refer to narrative Items 31–31(d) (Page 183).

type

31. VPos = atHub | onLnk

31(c). AtHub = atH(fli:LI,hi:HI,tli:LI)

31(d). onLnk = onL(fhi:HI,li:LI,frac:FRAC,thi:HI)

31(d). FRAC = **Real axiom** $\forall$ frac:FRAC $\cdot$ $0 <$ frac < 1

- We refer to narrative Item 33 (Page 184).
- It assumes the below.

44. To communicate vehicle movements vehicles communicate their positions to the monitor by offering outputs on a vehicle to monitor channel.

44. **channel** { vm[vi] | vi:VI · vi ∈ vis } VePos

45. A global variable, vps, records all possible initial vehicle positions (i.e., in an infinite set due to infinitisimality of any vehicle's "down link fractional position":
46. for all possible "at hub" positions, and
47. for all possible "on link" positions

45. **variable** vps:VPos-**infset** :=
46. $\{$atH(fi,hi,ti)$|$fi,ti:LI,hi:HI•mereo_H(get_H(n)(hi))$\supseteq\{$fi,ti$\}\subseteq$lis$\wedge$hi $\in$ his$\}$
47. $\cup$ $\{$onL(fi,li,f,ti)$|$fi,ti:HI,li:LI,f:FRAC•mereo_L(get_L(n)(li))=$\{$fi,ti$\}\subseteq$his$\wedge$li $\in$ lis$\}$

48. The monitor keeps track of vehicle movements — as lists of vehicle positions.

48. **type** TBL = VI $\overrightarrow{m}$ VPos*

49. Initial positions are obtained by arbitrary selection, get_VPos(), from the global vps variable.

49. **value** table:TBL = [vi↦⟨get_VPos()⟩|vi>VI•vi ∈ vis]

50. The get_VPos() function applies to the meta state variable (hence the argument type **Unit**) component vps and yields a vehicle position, vp:VPos.
51. That vehicle position is arbitrarily chosen from the contents of the global variable
52. from which that position is removed in order to avoid that two or more vehicles are initially piled at the same position;
53. "finally" vp is yielded.

value

```
50. get_VPos: Unit → VPos
50. get_VPos() ≡
51.     let vp:VPos•vp ∈ vps in
52.     vps := vps \ {vp} ;
53.     vp end
```

54. We consider the

    (a) the **vehicle monitoring system, vms**,
    (b) the **veh**icles, and
    (c) the **mon**itor

    to be processes.

The Overall System Behaviour

54(a). vms: **Unit** → **Unit**
54(a). vms() ≡
54(b). ‖ {veh(uid_Π(v))(v)(**hd** tbl(uid_Π(v)))|v:V•v ∈ vs}
54(c). ‖ mon(m)(table)

55. A **veh**icle process

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

The vehicle process communicates

- with the **mon**itor process on channel vm[vi]
- (sends, but receives no messages), and
- otherwise evolves "infinitely" (hence **Unit**).

56. We define here the vehicle behaviour **at** a **Hub** (**hi**).

   (a) Either the vehicle remains at that hub informing the monitor of this (cf. Items 32(a), 32(c), 32((d))i and 33 on page 184),

   (b) or, internally non-deterministically,

   (c) moves (cf. Items 32(b) on page 184, 32(d) and 32((d))ii on page 184) onto a link, **tli**, whose "next" hub, identified by **thi**, is obtained from the mereology of the link identified by **tli**;

   (d) informs the monitor, on channel **vm[vi]**, that it is now on the link identified by **tli** (cf. Item 33 on page 184),

   (e) whereupon the vehicle resumes the vehicle behaviour positioned at the very beginning (**0**) of that link,

   (f) or, again internally non-deterministically,

   (g) the vehicle "disappears — off the radar" !

The Vehicle Behaviour At Hubs

55. veh: vi:VI → v:V → vp:VPos
55. → **out** vm[vi] **Unit**, **pre**: uid_Π(v)=vi
56. veh(vi)(v)(vp:atH(fli,hi,tli)) ≡
56(a). vm[vi]!vp ; veh(vi)(v)(vp)
56(b). ⨅
56(c). **let** {hi′,thi}=mereo_L(get_L(tli)(n)) **in assert:** hi′=hi
56(d). vm[vi]!onL(tli,hi,0,thi) ;
56(e). veh(vi)(v)(onL(tli,hi,0,thi)) **end**
56(f). ⨅
56(g). **stop**

57. Either
    (a) the vehicle remains at that link position informing the monitor of this (cf. Item 33 on page 184),
    (b) or, internally non-deterministically,
    (c) if the vehicle's position on the link has not yet reached the hub,
        i. then the vehicle moves an arbitrary increment δ along the link informing the monitor of this (cf. Item 33 on page 184), or
        ii. else, while obtaining a "next link" from the mereology of the hub (where that next link could very well be the same as the link the vehicle is about to leave),
            A. the vehicle informs the monitor (cf. Item 33 on page 184) that it is now at the hub identified by thi,
            B. whereupon the vehicle resumes the vehicle behaviour positioned at that hub.
58. or, internally non-deterministically,
59. the vehicle "disappears — off the radar" !

The Vehicle Behaviour Along Links

55. veh(vi)(v)(vp:onL(fhi,li,f,thi)) $\equiv$
57(a). vm[vi]!vp ; veh(vi)(v)(vp)
57(b). $\prod$
57(c). **if** f + δ<1
57((c))i. **then** vm[vi]!onL(fhi,li,f+δ,thi) ;
57((c))i. veh(vi)(v)(onL(fhi,li,f+δ,thi))
57((c))ii. **else let** li′:LI•li′ $\in$ mereo_H(get_H(thi)(n)) **in**
57((c))iiA. vm[vi]!atH(li,thi,li′);
57((c))iiB. veh(vi)(v)(atH(li,thi,li′)) **end end**
58. $\prod$
59. **stop**

60. The monitor behaviour evolves around the attributes of an own "state", m:M, a table of traces of vehicle positions, while accepting messages about vehicle positions and otherwise progressing "infinitely".

61. Either the monitor "does own work"

62. or, internally non-deterministically accepts messages from vehicles.
    (a) A message, msg, may arrive from the vehicle identified by vi.
    (b) That message is appended to that vehicle's movement trace,
    (c) whereupon the monitor resumes its behaviour —
    (d) where the communicating vehicles range over all identified vehicles.

The Monitor Behaviour

```
60.     mon: M → TBL → in {vm[vi]|vi:VI·vi ∈ vis} Unit
60.     mon(m)(tbl) ≡
61.         let m′ = own_mon_work(m)(tbl) i mon(m′)(tbl) end
62.         ⌈⌉
62(a).          ⌷ { let msg = vm[vi]? in
62(b).                let tbl′ = tbl † [vi ↦ tbl(vi)⌢⟨msg⟩] in
62(c).                mon(m)(tbl′) end
62(d).              end | vi:VI · vi ∈ vis }
61.     own_mon_work: M → TBL → M
```

- **Discussion:**
  - We have modelled behaviours as co-operating sequences of actions and events.
    - Actions included the movement or decisions, of vehicles, not to move.
    - Events were (just) modelled by vehicles "disappearing off the 'radar'".

- The reader is kindly asked to compare the
  - narrative of the vehicle monitoring system (Items 28–33, Pages 182–184) with its
  - formalisation (Items 34–62(d), Pages 189–203).
- The former is brief and is independent of a particular understanding of "the nature" of the processes which model the system behaviour.
- The latter is less brief and
  - appears to require narrative descriptions
  - that pertain to the specific set-up necessary to
  - "explain the nature" of the processes which model the system behaviour. ■

8.4.5. A Model of Parts and Behaviours

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

A Model of Parts

63. The *whole* contains a set of *parts*.
64. *Parts* are either *atomic* or *composite*.

type
63. W, P, A, C
64. P = A | C
value
65. obs_Ps: (W|C) → P-**set**

65. From *composite parts* one can observe a set of *parts*.
66. All *parts* have *unique identifiers*

type
66. PI
value
66. uid_Π: P → Π

67. From a *whole* and from any *part* of that *whole* we can e**xtr**act all contained *part*s.

68. Similarly one can e**xtr**act the *unique identifier*s of all those contained *parts*.

69. Each part may have a *mereology* which may be "empty".

70. A *mereology*'s *unique part identifier*s must refer to some other parts other than the part itself.

value
67. $\mathrm{xtr\_Ps}$: (W|P) → P-**set**
67. $\mathrm{xtr\_Ps}(w) \equiv$
67. $\{\mathrm{xtr\_Ps}(p)|p{:}P{\cdot}p \in \mathrm{obs\_Ps}(p)\}$
67. **pre**: $\mathrm{is\_W}(p)$
67. $\mathrm{xtr\_Ps}(p) \equiv$
67. $\{\mathrm{xtr\_Ps}(p)|p{:}C{\cdot}p\in \mathrm{obs\_Ps}(p)\}\cup\{p\}$
67. **pre**: $\mathrm{is\_P}(p)$
68. $\mathrm{xtr\_\Pi s}$: (W|P) → Π-**set**
68. $\mathrm{xtr\_\Pi s}(wop) \equiv$
68. $\{\mathrm{uid\_P}(p)|p \in \mathrm{xtr\_Ps}(wop)\}$
69. $\mathrm{mereo\_P}$: P → Π-**set**

axiom
70. $\forall$ w:W
70. **let** ps = $\mathrm{xtr\_Ps}(w)$ **in**
70. $\forall$ p:P $\cdot$ p $\in$ ps $\cdot$
70. $\forall \pi{:}\Pi \cdot \pi \in \mathrm{mereo\_P}(p) \Rightarrow$
70. $\pi \in \mathrm{xtr\_\Pi s}(p)$ **end**

71. An **attribute map** of a *part* associates with *attribute names*, i.e., *type names*, their *values*, whatever they are.

72. From a *part* one can extract its attribute map.

73. Two *parts share attributes* if their respective **attribute maps** share *attribute names*.

74. Two *parts share properties* if the y

    (a) either *share attributes*

    (b) or the *unique identifier* of one is in the *mereology* of the other.

type
71. AttrNm, AttrVAL,
71. AttrMap = AttrNm $\underset{m}{\rightarrow}$ AttrVAL
value
72. attr_AttrMap: P → AttrMap
73. share_Attributes: P×P → **Bool**
73. share_Attributes(p,p′) ≡
73. **dom** attr_AttrMap(p) ∩
73. **dom** attr_AttrMap(p′) ≠ {}
74. share_Properties: P×P → **Bool**
74. share_Properties(p,p′) ≡
74(a). share_Attributes(p,p′)
74(b). ∨ uid_P(p) ∈ mereo_P(p′)
74(b). ∨ uid_P(p′) ∈ mereo_P(p)

Conversion of Parts into CSP Programs

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

type
75. K = Π-**set axiom** ∀ k:K•**card** k=2
value
75. xtr_Ks: (W|P) → K-**set**
75. xtr_Ks(wop) ≡
75.   **let** ps = xtr_Ps(w) **in**
75.   {{uid_P(p),π}|p:P,π:Π•p∈ ps
75.   ∧ ∃ p′:P•p′≠p∧π=uid_P(p′)
75.   ∧ uid_P(p)∈uid_P(p′)} **end**
76. w:W
77. **channel** {ch[k]|k:xtr_Ks(w)}:M

78. Now the 'whole' *behaviour* **whole** is the parallel composition of *part processes*, one for each of the immediate parts of the *whole*.

79. A *part process* is

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

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

78. whole: W → **Unit**
78. whole(w) ≡
78. ‖ {part(uid_P(p))(p) |
78. p:P•p ∈ xtr_Ps(w)}

79. part: π:Π → P → **Unit**
79. part(π)(p) ≡
79(b). is_A(p) → atom(π)(p),
79(b). _ → comp(π)(p)

80. A *composite process*, part, consists of
    (a) a *composite core process*, comp_core, and
    (b) the parallel composition of *part processes* one for each *contained part* of part.
81. An *atomic process* consists of just an *atomic core process*, *atom_core*.

value
80. comp: π:Π $\rightarrow$ p:P $\rightarrow$
80. **in**,**out** {ch[{π,π'}|{$\pi'$$\in$ mereo_P(p)}]}
80. **Unit**
80. comp(π)(p) $\equiv$
80(a). comp_core(π)(p) $\|$
80(b). $\|$ {part(uid_P(p′))(p′) |
80(b). p′:P•p′ $\in$ obs_Ps(p)}
81. atom: π:Π $\rightarrow$ p:P $\rightarrow$
81. **in**,**out** {ch[{π,π'}|{$\pi'$$\in$ mereo_P(p)}]}
81. **Unit**
81. atom(π)(p) $\equiv$ atom_core(π)(p)

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

We leave the update action undefined.

value
82. core: π:Π $\to$ p:P $\to$
82. **in**,**out** {ch[{π,π'}|{$\pi'$$\in$ mereo_P(p)}]}
82. **Unit**

82. core(π)(p) $\equiv$
82(a). **let** p$'$ = update(π)(p)
82(b). **in** core(π)(p$'$) **end**
82(b). **assert:** uid_P(p)=π=uid_P(p$'$)

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

8.4.6. Sharing Properties ≡ Mutual Mereologies

- In the model of the tight relationship between parts and behaviours
  - we “equated” two-element set of **unique identifiers** of parts that **share properties**
  - with the concept of **connectors**, and these again with **channels**.
- We need secure that this relationship,
  - between the two-element **connector** sets of **unique identifiers** of parts that **share properties**
  - and the **channels**

with the following **theorem**:

83. For every *whole*, i.e., domain,
84. if two distinct *parts* share properties
85. then their respective mereologies refer to one another,
86. and vice-versa
    - if two distinct *parts*
    - have their respective mereologies refer to one another,
    - then they share properties.

theorem:

83. $\forall$ w:W,p,p′:P•p≠p′∧{p,p′}⊆xtr_Ps(w) ⇒
84. share_Properties(p,p′)
86. ≡
85. uid_P(p)∈mereo_P(p′)∧uid_P(p′)∈mereo_P(p)

8.4.7. Behaviour Signatures

- By a **behaviour signature** we shall understand the combination of three clauses:
  - a **message type clause**,
    - **type** M,
  - possibly a **channel index type** clause,
    - **type** Idx,
  - a **channel declaration** clause
    - **channel** ch:M or
      channel {ch[i]|i:Idx•i ∈is}:M

    where is is a set of Idx values (defined somehow, e.g., **value** is:Idx-**set** = ... where ... is an expression of Idx values), and, finally,
  - a **behaviour function signature**:
    - **value** beh: Π → P → **out** ch **Unit** or
      value beh: Π → P → **out** ch **Unit** or
      value beh: Π → P → **in, out** ch **Unit** or
      value beh: Π → P → **in, out** {ch[i]|i:Idx• ∈is′} **Unit** or
      value beh: Π → P → **in** {ch[i]|i:Idx• ∈is′} **out** {ch[j]|j:Idx• ∈is′} **Unit**,
      etc.

- The **Conversion of Parts into CSP Programs** "story" gives the general idea:
  - To associate, in **principle**, with every **part** an own **behaviour**.
  - (Example 36 (Slides 188–**??**) did not do that:
    - in **principle** it did, but then it omitted describing
    - behaviours of "un-interesting" parts!)
  - Tentatively each **behaviour signature**, that is, each **part behaviour**, is
    - specified having a **unique identifier type**, respectively
    - given a **unique identifier argument**.

  Whether this tentative provision
    - for **unique identifiers** is necessary
    - will soon be revealed by further **domain analysis**.

- Before defining the **behaviour process signatures**
  - the **domain analyser** examines each of the chosen behaviours
  - with respect to its interaction with other chosen behaviours
  - in order to decide on
    - interaction message types and
    - "dimensionality" of channels,
    - whether singular or an array.
- Then the
  - **message types** can be *defined*,
  - the **channels** *declared*, and
  - the **behaviour function signature** can be *defined*,

  i.e., the full **behaviour signature** can be *defined*.

8.4.8. Behaviour Definitions

- We observe from the 'Conversion of Parts into CSP Programs' section, Slide 211,
  - that the "generation" of the core processes was syntax directed,
  - yet "delivered" a "flat" structure of parallel processes,
  - that is, no processes "running", *embedded*, within other processes.
- We make this remark since parts did not follow that prescription:
  - parts can, indeed, be *embedded* within one another.

- So our first "conclusion"[25], with respect to the structure of **domain behaviours**, is
  - that we shall model all behaviours of the "whole" domain
  - as a flat structure of **concurrent behaviours** —
    - one for each part contained in the whole —
  - which, when they need refer to properties of
  - behaviours of parts within which the part
    - on which "their" behaviour

    is embedded
  - then they interact with the behaviours of those parts,
  - that is, communicate messages.

[25] We put double quotes around the term 'conclusion' (above) since that conclusion was and is a choice, that is, not governed by necessity.

- The '**Conversion of Parts into CSP Programs**' section, Slide 211,
  - then suggested that there be
    - one **atom core** behaviour for each **atomic part**, and
    - one **composite core** behaviour for each **composite part**

    of the domain.
- The **domain analyser** may find that some of these **core behaviours**
  - are not necessary,
  - that is, that they — for the chosen scope of the domain model —
  - do not play a meaningful rôle.

Example: 37 "Redundant" Core Behaviours. We refer to the series of examples around the transport net domain.

- Transport nets, n:N, consist of
  - sets, hs:HS, of hubs and
  - sets, ls:LS, of links.
- Yet we may decide, for one domain scope,
  - to model only
    - hub,
    - link and
    - vehicle

    behaviours,
- and not 'set of hubs' and 'set of links' behaviours. ■

- Then the **domain analyser** can focus on exploring each individual **process behaviour**.
- Again the **Conversion of Parts into CSP Programs** "story" gives the general ideas that motivate the following:
- For each of the **parts**, **p**, a **behaviour expression** can be "generated":
  - beh_p(uid_P(p))(p).

  The idea is
  - that (uid_P(p)) uniquely identifies the **part behaviour** and
  - that the **part properties** of (p) serve as the **local state** for **beh_p**.

- Now we present an **analysis** of **part behaviours** around three 'alternatives':
  - (i) a **part behaviour** which basically represents a **proactive behaviour**;
  - (ii) one which basically represents a **reactive behaviour**; and
  - (iii) one which, so-to-speak alternates between **proactive** and **reactive behaviours.**
- What we are doing now is to examine
  - the form of the **core behaviours**,
  - cf. Item 82 (Slide 214).

- (i) A **proactive behaviour** is characterised by three facets.
  - (i.1) taking the initiative to interact with other **part behaviours** by offering output,
  - (i.2) **internally non-deterministically** ($\sqcap$) ranging interactions over several alternatives, and
  - (i.3) **externally non-deterministically** ($\square$) selecting which other behaviour to interact with, i.e., to offer output to.
- (i.1) A **proactive behaviour** takes the initiative to interact by expressing **output clauses**:

87. $\mathcal{O}_P$: ch ! val or ch[i] ! val or ch[i,j] ! val etc.

- (i.2) The **proactive behaviour** interaction request
  - may range over either of a finite number of alternatives,
  - one for each alternative, a_i, "kind" of interaction.
  - We may express such a non-deterministic (alternative) choice *either* as follows:

88. $\mathcal{NI}_P$: **type** Choice = $\mathrm{a}_1 \sqcap \mathrm{a}_2 \sqcap ... \sqcap \mathrm{a}_n$
    value let c:Choice **in**
    case c **of** $\mathrm{a}_1 \to \mathcal{E}_1$, $\mathrm{a}_2 \to \mathcal{E}_2$, ..., $\mathrm{a}_n \to \mathcal{E}_n$ **end end**

  - *or*, which is basically the same,

89. $\mathcal{NI}_P$: **value** ... $\mathcal{E}_1 \sqcap ... \sqcap \mathcal{E}_n$ ...

  - where each $\mathcal{E}_i$ usually contains an input clause, for example, ch ?.

- (i.3) The **proactive external non-deterministic choice** is directed at either of a number of other **part behaviours**.
  - This **proactive** selection is expressed

    90. $\mathcal{NX}_P$: $\mathcal{C}_i$ ⌷ $\mathcal{C}_j$ ⌷ ... ⌷ $\mathcal{C}_k$
    - where each of the $\mathcal{C}$lauses
    - express respective output clauses
    - (usually) directed at different **part behaviours**,
    - say ch[i] ! val. ch[j] ! val, etc., ch[k] ! val.
  - Another way of expressing **external non-deterministic choice** selection is

    91. $\mathcal{NX}_P$: ⌷ { ...; ch[i] ! fct(i) ; ... | i:Idx·i ∈ is }
- $\mathcal{O}$utput clauses [(i.1)], Item 88 $\mathcal{O}_P$,
  - may [(i.2)] occur in the $\mathcal{E}_i$ clauses of $\mathcal{NI}_P$, Items 89 and 90 and
  - must [(i.3)] occur in each of the $\mathcal{C}_i$ clauses of $\mathcal{NX}_P$, Item 91.

- (ii) A **reactive behaviour** is characterised by three
  - (ii.1) offering to interact with other **part behaviours** by offering to accept input,
  - (ii.2) **internally non-deterministically** ($\sqcap$) ranging interactions over several alternatives, and
  - (ii.3) **externally non-deterministically** ($\square$) selecting which other behaviour to interact with, i.e., to accept input from.
- (ii.1) A **reactive behaviour** expresses **input clauses**:

92. $\mathcal{I}_R$: ch ? or ch[i] ? or ch[i,j] ? etc.

- (ii.2) The reactive behaviour
  - may range over either of a finite number of alternatives,
  - one for each alternative, a_i, "kind" of interaction.
  - We may express such a non-deterministic (alternative) choice *either* as follows:

93. $\mathcal{NI}_R$: **value let** c:Choice **in**
    case c **of** $a_1 \rightarrow \mathcal{E}_1$, ..., $a_n \rightarrow \mathcal{E}_n$ **end end**

  where each of the expressions, $\mathcal{E}_i$, may, and usually contains a input clause ($\mathcal{I}$, Item 92 on the facing page).
  - Thus the $\mathcal{NI}_R$ clause is almost identical to the $\mathcal{NI}_P$ clause, Item 89 on page 228.
  - Hence another way of expressing external non-deterministic choice is

94. $\mathcal{NX}_R$: $\prod$ { ...; ch[i]!fct(i) ; ... | i:Idx·i ∈ is }.

- (ii.3) The **reactive behaviour** selection is directed at either of a number of other **part behaviours**.
  - This **external non-deterministic choice** is expressed

    95. $\mathcal{NX}_R$: $\mathcal{C}_i$ ⌷ $\mathcal{C}_j$ ⌷ ... ⌷ $\mathcal{C}_k$

    - where each of the $\mathcal{C}$lauses
    - express respective input clauses
    - (usually) directed at different **part behaviours**,
    - say ch[i] ?. ch[j] ?, etc., ch[k] ?.
  - Another way of expressing **external non-deterministic choice** selection is

    96. $\mathcal{NX}_R$: ⌷ { ...; ch[i] ? ; ... | i:Idx·i ∈ is }

  - Thus the $\mathcal{NX}_R$ clauses are almost identical to the $\mathcal{NX}_P$ clauses, Items 90–91.

- $\mathcal{I}$nput clauses [(ii.1)], Item 92 $\mathcal{I}_R$,
  - may [(ii.2)] occur in the $\mathcal{E}_i$ clauses of $\mathcal{NI}_R$, Items 93–94 and
  - must [(ii.3)] occur in each of the $\mathcal{C}_i$ clauses of $\mathcal{NX}_R$, Items 95–96.

- (iii) An alternating **proactive behaviour** and **reactive behaviour**
  - is characterised by expressing both
    - **reactive behaviour** and
    - **proactive behaviour**s

    combined by either
    - **non-deterministic internal choice** ($\sqcap$) or
    - **non-deterministic external choice** ($\square$) combinators.

    For example:

    97. $(\mathcal{NI}_{P_i}[\sqcap\text{or}\square]\mathcal{NX}_{P_j})[\sqcap\text{or}\square](\mathcal{NI}_{R_k}[\sqcap\text{or}\square]\mathcal{NX}_{R_\ell})$.

- The meta-clause $[\sqcap\text{or}\square]$ stands for either $\sqcap$ or $\square$.
- Here there usually is a disciplined use of input/output clauses.

Example: 38 A Pipeline System Behaviour.

- We refer to Examples
  - 14 (Slide 90) and
  - 21–23 (Slides
  - 117–125)
  - and especially Examples 24–25 (Slides 127–131).

- We consider (cf. Example 22) the pipeline system units to represent also the following behaviours:
  - pls:PLS, Item 4(a) on page 119, to also represent the system process, pipeline_system, and for each kind of unit, cf. Example 14, there are the unit processes:
    - unit,
    - well (Item 3(c) on page 91),
    - pipe (Item 3(a)),
    - pump (Item 3(a)),
    - valve (Item 3(a)),
    - fork (Item 3(b)),
    - join (Item 3(b)) and
    - sink (Item 3(d) on page 91).

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

- We illustrate essentials of just one of these behaviours.

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

- The core_fork_behaviour(ui)(u) distributes
  - what oil (or gas) in receives,
    - on the one input sel_UIs_in(u) = {iui},
    - along channel u_u_ch[iui]
  - to its two outlets
    - sel_UIs_out(u) = {oui_1,oui_2},
    - along channels u_u_ch[oui_1], u_u_ch[oui_2].

- The core_fork_behaviour(ui)(u) also communicates with the pipeline_system behaviour.
  - What we have in mind here is to model a traditional supervisory control and data acquisition, SCADA system.

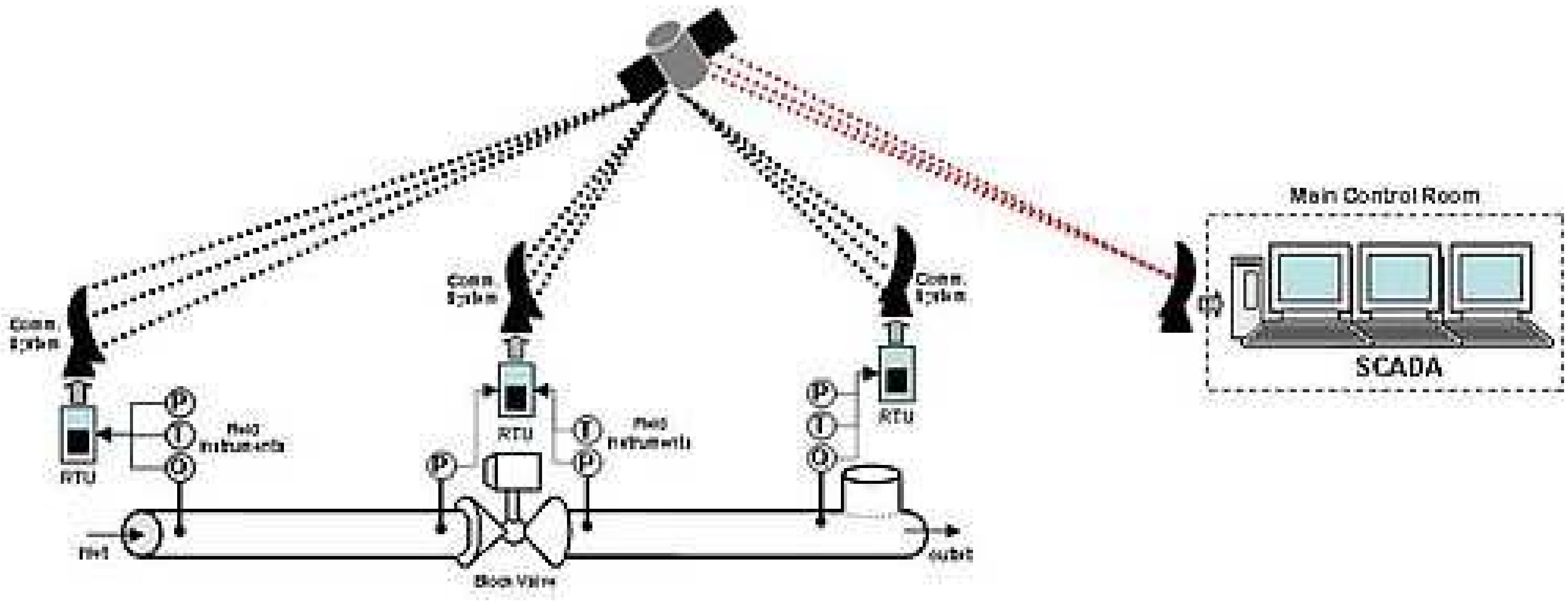

Figure 1: A supervisory control and data acquisition system

- SCADA is then part of the **pipeline_system** behaviour.

98.

98. pipeline_system: PLS → **in**,**out** { pls_u_ch[ui]:ui:UI•i ∈ UIs(pls) } **Unit**
98.   pipeline_system(pls) ≡ scada(props(pls)) ∥ ∥{ unit(u)|u:U•u ∈ obs_Us(pl

- props was defined on Slide 133.

99. scada non-deterministically (internal choice, $\lceil\rceil$), alternates between continually
    (a) doing own work,
    (b) acquiring data from pipeline units, and
    (c) controlling selected such units.

type
99. Props
value
99. scada: Props → **in**,**out** { pls_ui_ch[ui] | ui:UI•ui ∈ ∈ uis } **Unit**
99. scada(props) ≡
99(a). scada(scada_own_work(props))
99(b). $\lceil\rceil$ scada(scada_data_acqui_work(props))
99(c). $\lceil\rceil$ scada(scada_control_work(props))

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

100. The **scada_data_acqui_work**
   (a) non-deterministically, external choice, $\lceil\rceil$, offers to accept data,
   (b) and **scada_input_update**s the scada state —
   (c) from any of the pipeline units.

value

100. scada_data_acqui_work: Props → **in**,**out** { pls_ui_ch[ui] | ui:UI•ui ∈ ∈

100. scada_data_acqui_work(props) ≡

100(a). ⌈⌉ { **let** (ui,data) = pls_ui_ch[ui] ? **in**

100(b). scada_input_update(ui,data)(props) **end**

100(c). | ui:UI · ui ∈ uis }

100(b). scada_input_update: UI × Data → Props → Props

type

100(a). Data

101. The scada_control_work

   (a) analyses the scada state (props) thereby selecting a pipeline unit, ui, and the controls, ctrl, that it should be subjected to;
   (b) informs the units of this control, and
   (c) scada_output_updates the scada state.

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

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

Modelling Behaviours, I/II

- The **domain describer** has decided that an **entity** is a **perdurant** and is, or represents a **behaviour**.
  - ◈ The domain describer has further decided that the observed behaviour is of a class of behaviours — of the "same kind" — that need be described.
  - ◈ By behaviours of the 'same kind' is meant that these can be described by the same **channel declarations**, **function signature** and **function definition**.

Modelling Behaviours, II/II

- First the domain describer must decide on the underlying function signature.
  - It must be decided which synchronisation and communication
    - inputs and
    - outputs

    this behaviour requires, i.e., the **in,out** clause of the signature,
  - that also includes the "discovery" of necessary channel declarations.
- Finally the function definition must be decided upon.

9. Continuous Perdurants

- By a continuous perdurant we shall understand a continuous behaviour.
- This section serves two purposes:
  - to point out that believable system descriptions must entail both
    - a discrete phenomena domain description and
    - a continuous phenomena mathematical model.
  - and this poses some semantics problems:
    - the formal semantics of the discrete phenomena description language and
    - the meta-mathematics of, for example, differential equations,

    at least as of today, July 31, 2012, are not commensurable!
  - That is, we have a problem — as will be outlined later in this lecture.

9.1. Some Examples

Example: 39 **Continuous Behaviour: The Weather.** We give a familiar example of continuous behaviour.

- The *weather* — understood as the time-wise evolution of a number of attributes of the *weather* material:
  - *temperature*,
  - *wind direction*,
  - *wind force*,
  - *atmospheric pressure*,
  - *humidity*,
  - *sky formation* (`clear, cloudy, ...`),
  - *precipitation*,
  - etcetera.
- That is, *weather* is seen as the state of the *atmosphere* as it evolves over time.■

Example: 40 Continuous Behaviour: Road Traffic. We give another familiar example of continuous behaviour.

- The *automobile traffic* is the time-wise evolution of cars along a net has the following additional attributes:
  - *car identity* (CI),
  - *position* (P, on the net),
  - *direction* (D),
  - *velocity* (V),
  - *acceleration* (A),
  - etcetera (...).
- The equation below captures this:

$$\mathsf{TF} = \mathsf{T} \rightarrow (\mathsf{CI} \underset{m}{\rightarrow} (\mathsf{P} \times \mathsf{D} \times \mathsf{V} \times \mathsf{A} \times \ldots))$$

- We refer to Example 36
  - specifically the veh, hub and mon behaviours.
  - These "mimic" a discretised version of the above:

$$\mathsf{TF} = \mathsf{T} \underset{m}{\rightarrow} (\mathsf{CI} \underset{m}{\rightarrow} (\mathsf{P} \times \mathsf{D} \times \mathsf{V} \times \mathsf{A} \times \ldots))$$

■

Example: 41 Pipeline Flows. A last example of continuous behaviour.

- We refer to Examples 12, 14, 21–25, 41–45 and 49.
- These examples focused on
  - the **atomic**parts and the **composite parts** of pipelines,
  - and dealt with the liquid or gas materials as they related to pipeline units.
- In the present example we shall focus on
  - the overall material flow "across" a pipeline.
  - in particular the **continuity** as
  - as contrasted with the pipeline unit **discrete**
  - aspects of flow.

- Which, then, are these pipeline system **continuity** concerns ?
  - In general we are interested in
    1. *whether the **flow** is **laminar** or **turbulent**:*
       (a) *within a **unit**, or*
       (b) *within an entire, possibly intricately **networked pipeline**;*
    2. *what the **shear stresses** are;*
    3. *whether there are undesirable **pressures**;*
    4. *whether there are **leaks** above normal values;*

    etcetera.
- To answer questions like those posed in
  - Items 1(a) and 2, we need not build up the models sketched in Examples 12, 14, 24, 25, 41–45 and 49.
  - But for questions like those posed in Items 1(b), 3 and 4 we need such models.

- To answer any of the above questions, and many others, we need establish, in the case of pipelines, *fluid dynamics models* [Batchelor1967,Thorley1991,Wendt1992,Coulbeck2010].
- These models involve such mathematical as are based, for example, on
  - *Newtonian Fluid Behaviours*,
  - *Bernoulli Equations*,
  - *Navier–Stokes Equations*,
  - etcetera.
- Each of these **mathematical models**
  - capture the dynamics of one specific pipeline unit,
  - not assemblies of two or more. ■

9.2. Two Kinds of Continuous System Models

- There are at least two different kinds of mathematical models for continuous systems.
  - There are the models which are based on physics models mentioned above, for example
    - the dynamics of flows in networks,
  - and there are the models which builds on **control theory** to express **automatic control** solutions to the monitoring & control of pipelines, for example:
    - the opening, closing and setting of pumps, and
    - the opening, closing and setting of valves

    depending on monitored values of dynamic well, pipe, pump, valve, fork, join and sink attributes.

- Example 41 on page 249 assumes
  - the fluid mechanics domain models
  - to complement the discrete domain model of Example 38 on page 235,

  whereas
- Example 44 on page 272
  - builds on Examples 41 and 38
  - but assumes that automatic monitoring & control requirements prescriptions
  - have been derived, in the usual way from the former fluid mechanics domain models.

9.3. Motivation for Consolidated Models

- By a consolidated model
  - we shall understand a formal description
  - that brings together both
    - discrete
      - for example TripTych style domain description

    and
    - continuous
      - for example classical mathematical description
  - models of a system.

- We shall **motivate** the need for consolidated models, that is **for building both**
  - the novel **domain descriptions,**
    - such as this tutorial suggests,
    - with its many aspects of discreteness,

    and the
  - the **classical mathematical models**,
    - as this section suggests,
    - including, for example, as in the case of Example 41, fluid dynamics mathematics.

- This motivation really provides the justification for bringing the two disciplines together:
  - discrete system domain modelling with
  - continuous system physics modelling

  in this tutorial.

- The classical mathematical models of, for example, pipelines,
  - model physical phenomena within parts or within materials;
  - and also combinations of *neighbouring*,
    - parts with parts and
    - parts with materials.
  - But classical mathematical modelling
    - cannot model continuous phenomena
    - for other than definite concrete, specific combinations of parts and/or materials.

- The kind of domain modelling,
  - that is brought forward in this tutorial can,
  - within one domain description
  - model a whole class,
  - indeed an indefinite,
  - class of systems.

9.4. Generation of Consolidated Models

- The idea is therefore this
  - create a **domain description**
    for a whole, the indefinite class of "alike" systems, to wit
    - for an indefinite class of pipelines,
    - for an indefinite class of container lines,
    - for an indefinite class of health care systems,
  - and then "adorn" such a description
    - first with **classical mathematical model**s
      of simple **part**s of such systems; and
    - then "replicate" these **mathematical model**s across the indefinite class of **discrete model**s
    - by "pairing"
      - each **definite classical concrete mathematical model**
      - with an, albeit **abstract general discrete model**.

9.4.1. The Pairing Process

- The "pairing process" depends on a notion of **boundary condition**.
  - The **boundary conditions** for **mereology**-related **parts** are, yes,
    - expressed by their **mereology**,
    - that is, by how the **parts** fit together.
  - The **boundary conditions** for **continuous models** are understood as
    - the set of conditions specified for the solution
    - to a set of differential equations at the boundary between the **parts** being individually modelled.

- In pairing we take the “cue”, i.e., directives, from
  - the **discrete domain model** for the generic **part** and its related **material**
  - since it is the more general, and
  - “match” its **mereology** with
  - the **continuous mathematics model** of a **part** and its related **material**

9.4.2. Matching

- Matching now means the following.
  - Let $\mathcal{D}_{\mathsf{P},\mathsf{M}}$
    - designate a $\mathcal{D}$omain $\mathcal{D}$escription
    - for a part and/or a material, of type P, respectively M,
    - zero or one part type and zero or one material type(s).
  - Let $\mathcal{M}_{\mathsf{P},\mathsf{M}}$
    - designate a $\mathcal{M}$athematical $\mathcal{M}$odel
    - for a part and/or a material of type P, respectively M,
    - zero or one part type and zero or one material type(s).

Example: 42 A Transport Behaviour Consolidation.

- An example $\mathcal{D}_{\mathsf{P,M}}$ could be
  - the one, for vehicles, shown in Example 36 (Slides 188–206)
  - as specifically expressed in the two frames:
    - 'The Vehicle Behaviour at Hubs' on Slide 200 and
    - 'The Vehicle Behaviour along Links' on Slide 202.
- On Slide 200 of Example 36 notice vehicle vi movement at hub in formula line
  - 56(a) — apparently not showing any movement and
  - 56(e) — showing movement from hub onto link.
- On Slide 202 notice vehicle vi movements along link in formula lines
  - 57(a) — no movement (stopped or parked),
  - 57((c))i — incremental movement along link, and
  - 57((c))iiB — movement from link into hub.

- The corresponding example $\mathcal{M}_{\mathsf{P,M}}$ might then be
  - modelling these movements and no movements
  - requiring access to such attributes as
    - link length,
    - vehicle position,
    - vehicle velocity,
    - vehicle acceleration,

    etcetera.
- This model would need to abstract the non-deterministic behaviour of the driver:
  - accelerating,
  - decelerating or
  - steady velocity.
- Example 36's model of vehicles' link position in terms of a fragment (δ) can be expected to appear in $\mathcal{M}_{\mathsf{P,M}}$ as an x, viewing the link as an x-axis. ■

Example: 43 **A Pipeline Behaviour Consolidation.** We continue the line of exemplifying formalisations of pipelines, cf. Examples 14 (Slide 90) and 21–23 (Slides 117–125) and especially Examples 24–25 (Slides 127–131).

- Let the $\mathcal{D}_{\mathsf{P,M}}$ model be focused on the flows and leaks of pipeline units, cf. Examples 24 and 25.
- The $\mathcal{M}_{\mathsf{P,M}}$ model would then $\mathcal{M}$athematically model the fluid dynamics of the pipeline material per pipeline unit: flow and part actions and reactions for any of the corresponding $\mathcal{D}$omain models:
  - *wells*, $\mathcal{D}^{\mathsf{well}}_{\mathsf{U,O}} \to \mathcal{M}^{\mathsf{well}}_{\mathsf{U,O}}$,
  - *pipes*, $\mathcal{D}^{\mathsf{pipe}}_{\mathsf{U,O}} \to \mathcal{M}^{\mathsf{pipe}}_{\mathsf{U,O}}$,
  - *pumps*, $\mathcal{D}^{\mathsf{pump}}_{\mathsf{U,O}} \to \mathcal{M}^{\mathsf{pump}}_{\mathsf{U,O}}$,
  - *valves*, $\mathcal{D}^{\mathsf{valve}}_{\mathsf{U,O}} \to \mathcal{M}^{\mathsf{valve}}_{\mathsf{U,O}}$,
  - *forks*, $\mathcal{D}^{\mathsf{fork}}_{\mathsf{U,O}} \to \mathcal{M}^{\mathsf{fork}}_{\mathsf{U,O}}$,
  - *joins*, $\mathcal{D}^{\mathsf{join}}_{\mathsf{U,O}} \to \mathcal{M}^{\mathsf{join}}_{\mathsf{U,O}}$, and
  - *sinks* $\mathcal{D}^{\mathsf{sink}}_{\mathsf{U,O}} \to \mathcal{M}^{\mathsf{sink}}_{\mathsf{U,O}}$.

■

- Some more model annotations,
  - ◈ reflecting the match between $\mathcal{D}_{\mathsf{P,M}}$ and $\mathcal{M}_{\mathsf{P,M}}$,

  seem relevant.
  - ◈ Thus we further subscript $\mathcal{D}_{\mathsf{P,M}}$ optionally with
    - ⊚ a **unique identifier variable**, π, and
    - ⊚ the properties $p_i, p_j, ..., p_k$ where
      - ∗ p_i is a **property name** of **part type** P or of **material type** M,
      - ∗ and where these property names typically are the distinct attribute names of P and/or M,

    to arrive at $\mathcal{D}^{\pi}_{\mathsf{P,M}_{p_i,p_j,...,p_k}}$.
  - ◈ Here π is a variable name for p:P, i.e., π is uid_P(p).
  - ◈ Do not confuse **property names**, p_i etc., with **part names**, p.

- And we likewise adorn $\mathcal{M}_{\mathsf{P,M}}$ optionally with
  - ◈ superscripts $p_i, p_j, ..., p_k$ and
  - ◈ subscripts $x_i, x_j, ..., x_k$ where
    - ⚭ $p_i, p_j, ..., p_k$ are as for $\mathcal{D}^{\pi}_{\mathsf{P,M}_{p_i,p_j,...,p_k}}$ and
    - ⚭ $x_i, x_j, ..., x_k$ are the names of the variables occurring in $\mathcal{M}_{\mathsf{P,M}}$
      - ∗ possibly in its **partial differential equation**s,
      - ∗ possibly in its **difference equation**s,
      - ∗ possibly in its other mathematical expressions of the $\mathcal{M}_{\mathsf{P,M}}$ model.

  to arrive at $\mathcal{M}^{\pi}_{\mathsf{P,M}^{p_i,p_j,...,p_k}_{x_i,x_j,...,x_k}}$

- The "adornments" are the result of an **analysis** which
  - identifies the variables of $\mathcal{M}_{\mathsf{P,M}}$
  - with the properties of $\mathcal{D}_{\mathsf{P,M}}$.
- Common to all **conventional mathematical model**s
  - is that they all operate with a very **simple type concept**:
    - **Real**s, **Int**egers,
    - **array**s (**vector**s, **matrices**, and **tensor**s),
    - **set**s of the above and sets.
- Common to all **domain model description**s
  - is that they all operate with a rather **sophisticated type concept**:
    - **abstract type**s and **concrete type**s,
    - union $(\mathsf{T}_i|\mathsf{T}_j...)$ of these,
    - **set**s, **Cartesian**s, **list**s, **map**s, and **partial function**s and **total function**s over these, etcetera.

9.4.3. Model Instantiation

- The above models, $\mathcal{D}_{\mathsf{P,M}}$ and $\mathcal{M}_{\mathsf{P,M}}$, differ as follows.
  - The $\mathcal{D}_{\mathsf{P,M}}$ models (are claimed to) hold for indefinite sets of domains "of the same kind":
    - The **axioms** and **invariants**, cf.
      - Example 11 on page 82,
      - Examples 24–25 (Slides 127–130) and
      - Example 27 on page 138,

      are universally quantified over all transport nets.
- The $\mathcal{M}_{\mathsf{P,M}}$ models express no such logic.

- The above difference can, however, be ameliorated.
  - For a given, that is, an instantiated domain,
    - we can “compile” the $\mathcal{D}_{\mathsf{P,M}}$ models
    - into a set of models,
    - one per **part** of that domain;
  - similarly, with the **binding** of model $\mathcal{M}_{\mathsf{P,M}}$ variables to instantiated model $\mathcal{D}_{\mathsf{P,M}}$ attributes,
    - we can “compile” the $\mathcal{M}_{\mathsf{P,M}}$ models
    - into as set of — instantiated $\mathcal{M}_{\mathsf{P,M}}$ models,
    - one per **part** of that domain.

9.4.3.1 Model Instantiation – in Principle

- Since this **partial evaluation compilation** can be (almost) automated,
  - there is really no reason to actually perform it;
  - all necessary theorems should be derivable from the annotated models.
    - $\mathcal{D}^{\pi}_{\mathsf{P},\mathsf{M}_{p_i,p_j,\ldots,p_k}}$ and
    - $\mathcal{M}^{\pi}_{\mathsf{P},\mathsf{M}^{p_i,p_j,\ldots,p_k}_{x_i,x_j,\ldots,x_k}}$.
- That is, as far as a **domain understanding** concerns
  - we might, with
    - **continuous mathematical modelling** and
    - mostly **discrete domain modelling**
  - very well have achieved all we can possibly, today, achieve.

9.4.3.2 Model Instantiation – in Practice

- We continue Example 38 (Slides 235–243).
  - The definition of pipeline_system function (Slide 240) indicates the basis for an instantiation.

Example: 44 An Instantiated Pipeline System.

- Figure 2 indicates an instantiation.

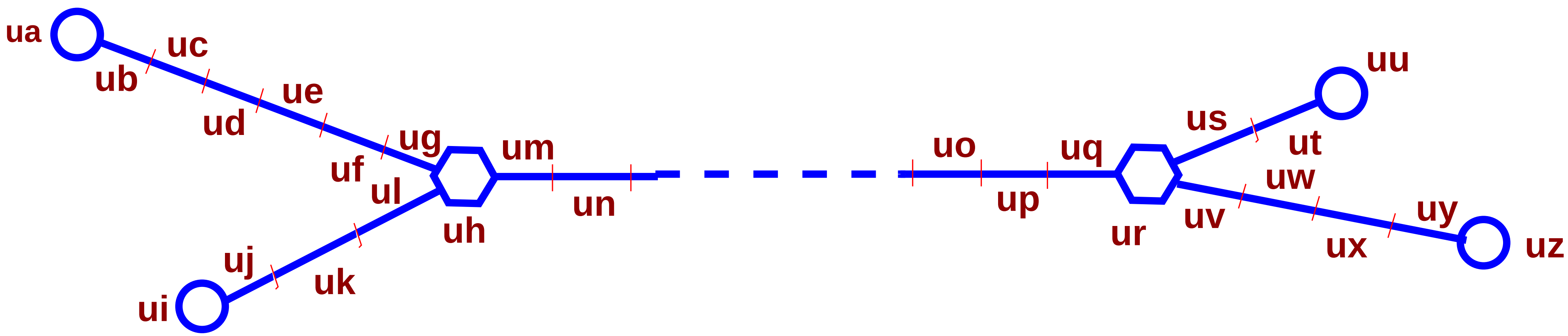

Figure 2: A specific pipeline

- That pipeline system gives rise to the following instantiation.

  scada(pro)‖
  unit(ua)‖unit(ub)‖unit(uc)‖unit(ud)‖unit(ue)‖unit(uf)‖unit(ug)‖
  unit(uh)‖
  unit(ui)‖unit(uj)‖unit(uk)‖unit(ul)‖
  unit(um)‖unit(un)‖...‖unit(uo)‖unit(up)‖unit(uq)‖
  unit(ur)‖
  unit(us)‖unit(ut)‖unit(uu)‖
  unit(uv)‖unit(uw)‖unit(ux)‖unit(uy)‖unit(uz)

- It is in the scada behaviour, that each of the $\mathcal{M}_{U,O}^{uid\_U(u)}$ models are 'instantiated'.
- The above instantiated model
  - is not a domain model of a generic pipeline system
  - but is a requirements model for the monitoring & control of a specific pipeline system.

9.5. An Aside on Time

- An important aspect of **domain modelling** is the description of **time phenomena**:
  - **absolute time** (or just **time**) and
  - **time intervals**.
- We shall, regrettably, not cover this facet in this tutorial, but refer to
  - a number of specifications expressed in combined uses of
    - the `RAISE` [RaiseMethod] combined with
    - the `DC: Duration Calculus` [zcc+mrh2002].
  - We could also express these specifications using `TLA+` [Lamport-TLA+02]: Lamport's `Temporal Logic of Actions`.
- We otherwise refer to [TheSEBook2wo] (Chap. 15.).

9.6. A Research Agenda

- This section opens two main lines of research problems;
  - methodology problems cum computing science problems and
  - computer science cum mathematics problems.

9.6.1. Computing Science cum Programming Methodology Problems

- Some of the **methodology problems** are
  - **techniques** for developing **continuous mathematics** models — which we leave to the relevant fields of
    - physics and
    - control theory

    to "deliver";
  - contained in this are more detailed **techniques** for **matching** $\mathcal{D}_{\mathsf{D,M}}$ and $\mathcal{M}_{\mathsf{D,M}}$ models,
    - that is, for identifying and pairing the p_is and x_is in
      - $\mathcal{D}^{\pi}_{\mathsf{P,M}_{p_i,p_j,\ldots,p_k}}$ and
      - $\mathcal{M}^{\pi}_{\mathsf{P,M}^{p_i,p_j,\ldots,p_k}_{x_i,x_j,\ldots,x_k}}$

      and
    - for **instantiating** these.

- A problem of current programming methodology in
  - that it has for most of its “existence”
  - relied on discrete mathematics
  - and not sufficiently educated and trained
  - its candidates in continuous mathematics.

9.6.2. Mathematical Modelling Problems

- Some of the open mathematics problems are
  - the lack of well-understood interfaces between
    - discrete mathematics models and
    - continuous mathematics models;
  - and the lack of proof systems across the two modes of expression.

- By well-understood interfaces between the two modes of expression,
  - the discrete mathematics models and
  - the continuous mathematics models;

  we mean that the semantics models of
  - the discrete mathematics formal specification languages and
  - the continuous mathematics specification notations,

  at this time, July 31, 2012, are not commensurate, that is, do not "carry over":
  - a variable, a of some, even abstract type, say A,
  - cannot easily be related to what it has to be related to, namely
  - a variable, x of some concrete, mathematical type, say **Real** or **Int**eger, or arrays of these, etc.

- Lack of proof systems across the two modes of expression.
  - the discrete mathematics models and
  - the continuous mathematics models;

  we mean,
  - firstly, that the former problem of lack of clear a↔x relations is taken to prevent such proof systems,
  - secondly, that mathematics essentially does not embody a "formal language".
- But nobody is really looking into, that is, researching possible "solutions" to these problems.

10. Discussion of Entities

- We have examined the concepts of **entities**, **endurant** and **perdurant**.
- We have not examined those "things" (of a domain) which "fall outside" this categorisation.
  - That would lead to a rather lengthy discourse.
  - In the interest of "really understanding" what can be described such a computer science study should be made.
  - Philosophers have clarified the issues in centuries of studies.
    - Their interest is in
      - identifying the issues and
      - clarifying the questions.
    - Computer scientists are interested in answers.

- We see entities as either
  - **endurants** or
  - **perdurants**

  or as either
  - **discrete** or
  - **continuous**.
- We analyse **discrete endurants** into **atomic** and **composite parts** with
  - **observers**,
  - **unique identifiers**,
  - **mereology** and
  - **attributes**.
- And we analyse **perdurants** into **actions**, **events** and **behaviours**.

- This **domain ontology** is entirely a pragmatic one:
  - it appears to work;
  - it has been used in the description of numerous cases;
  - it leads to descriptions which in a straightforward manner lend
    - themselves to the "derivation"
    - of significant fragments of requirements;
  - and appears not to stand in the way of obtaining remaining requirements.

- Most convincingly to us is that the concepts of our approach
  - endurants and perdurants,
  - atomic and composite parts,
  - mereology and attributes,
  - actions, events and behaviours

  fit it with major categories of philosophically analyses.

End of Lecture 5: Last Session — Perdurant Entities

Behaviours, Discussion Entities

FM 2012 Tutorial, Dines Bjørner, Paris, 28 August 2012

HAVE A GOOD LUNCH – SEE YOU BACK AT 2 PM