(B. B.4.)

Start of Lecture 12: MEREOLOGY

C. Mereology
C.1. Opening
C.1.1. Definition

- By mereology we understand
  - the study and knowledge about
  - parts and wholes
  - and the relationships between parts and between parts and holes.

(C. Mereology C.1. Opening C.1.1. Definition)

C.1.2. Examples

Example 51 – Simple and Composite Net Entities:

- We repeat some of the material from Example 1 on page 39.
- [1] A road, train, airlane (air traffic) or sea lane (shipping) net
- [2] consists, amongst other things, of hubs and links.

type
 [1] N
 [2] H, L
value
 [2] ωHs: N $\rightarrow$ H-**set**, ωLs: N $\rightarrow$ L-**set**,

- We can consider nets as composite and, for the time being, hubs and links as simple.

■ End of Example 51

(C. Mereology C.1. Opening C.1.2. Examples)

- Example 51 illustrated that entities can be either atomic of composite.
- But also functions, events and behaviours can be either atomic or composite.

Example 52 – Simple and Composite Net Functions:

- [3] With every link we associate a length.
- [4] A journey is a pair of a link and a continuation.
- [5] A continuation is either "nil" or is a journey.
- [6] Journies have lengths:
  - [6.1] the length of the link of the journey pair,
  - [6.2] and the length of the continuation – where a "nil" continuation has length 0.

type
[3] LEN
[4] Journey = L × C
[5] C = "nil" | Journey
value
[3] zero_LEN:LEN
[3] ωLEN: L → LEN
[6] length: Journey → LEN
[6] length(l,c) ≡
[6.1] **let** ll = ωLEN(l),
[6.2] cl = **if** c="nil" **then** zero_LEN **else** length(c) **end in**
[6] sum(ll,cl) **end**
sum: LEN × LEN → LEN

- Both
  - the journey and continuation entities, *j* and *c* , and
  - the *length* function

  are composite
- Both
  - the link entities, *ll*,
  - the ω*LEN* function

  are atomic.

■ End of Example 52

Example 53 – Simple and Composite Net Events:

- [7] The isolated crash of two vehicles, at time t, in a traffic, at a hub or along a link can be construed as a single atomic event.
- [8] The crash, within a few seconds $(t, t', t \sim t')$, in a traffic, of three or more vehicles,
  - [8.1] in a hub,
  - [8.2] or along a short segment of a link,

  can be considered a composite event.
- We shall model this event by the predicates which holds of vehicles in a traffic at given times.

```
type
  TF = T → (V →m Pos)
  Pos == μatH(hi:HI) | μonL(πhi:HI,πli:LI,πf:F,πhi′:HI)
type
value
  [7] atomic_crash: V × V → TF → T → Bool
  [7] atomic_crash(v,v′)(tf)(t) ≡ (tf(t))(v)=(tf(t))(v′)
  [7] pre t ∈ DOMAIN tf ∧ {v,v′}⊆dom(tf(t))∧v≠v′

  [8] composite_crash: V-set → TF → (T×T) → Bool
  [8] composite_crash(vs)(tf)(t,t′) ≡
  [8.1] ∃ hi:HI · card{v|v:V· ∈ vs∧(tf(t″))(v)=μatH(hi)∧t≤t″≤t′}≥3∨
  [8.2] ∃ hi′,hi″:HI,li:LI,fs:F-set ·
  [8.2]    fs={r..r′} where 0≤r≃r′≤1 ∧
  [8.2]    card{(tf(t″))(v)=μonL(hi′,li,f,hi″)|v:V,f:F·v ∈ vs∧f ∈ fs∧t≤t″≤t′}≥3
  [8]   pre {t,t′}⊆DOMAIN tf ∧ t∼t′ ∧ ∧ vs⊆dom(tf(t)) ∧ card vs≥3
```

■ End of Example 53

- In the next, long example we consider a pipeline system (or either oil or gas pipes).

Example 54 – Simple and Composite Net Behaviours:

Pipeline Systems and Their Units

35. We focus on nets, $n : N$, of pipes, $\pi : \Pi$, valves, $v : V$, pumps, $p : P$, forks, $f : F$, joins, $j : J$, wells, $w : W$ and sinks, $s : S$.
36. Units, $u : U$, are either pipes, valves, pumps, forks, joins, wells or sinks.
37. Units are explained in terms of disjoint types of PIpes, VAlves, PUmps, FOrks, JOins, WEIls and SKs.[12]

[12] This is a mere specification language technicality.


```
type
  35 N, PI, VA, PU, FO, JO, WE, SK
  36 U = Π | V | P | F | J | S| W
  36 Π == mkΠ(pi:PI)
  36 V == mkV(va:VA)
  36 P == mkP(pu:PU)
  36 F == mkF(fo:FO)
  36 J == mkJ(jo:JO)
  36 W == mkW(we:WE)
  36 S == mkS(sk:SK)
```


Unit Identifiers and Unit Type Predicates

38. We associate with each unit a unique identifier, $ui : UI$.
39. From a unit we can observe its unique identifier.
40. From a unit we can observe whether it is a pipe, a valve, a pump, a fork, a join, a well or a sink unit.

```
type
  38 UI
value
  39 obs_UI: U → UI
  40 is_Π: U → Bool, is_V: U → Bool, ..., is_J: U → Bool
     is_Π(u) ≡ case u of mkPI(_) → true, _ → false end
     is_V(u) ≡ case u of mkV(_) → true, _ → false end
      ...
     is_S(u) ≡ case u of mkS(_) → true, _ → false end
```

Unit Connections

- A connection is a means of juxtaposing units.
- A connection may connect two units in which case one can observe the identity of connected units from "the other side".

41. With a pipe, a valve and a pump we associate exactly one input and one output connection.
42. With a fork we associate a maximum number of output connections, m, larger than one.
43. With a join we associate a maximum number of input connections, m, larger than one.
44. With a well we associate zero input connections and exactly one output connection.
45. With a sink we associate exactly one input connection and zero output connections.

value
41 obs_InCs,obs_OutCs: Π|V|P $\rightarrow$ {|1:**Nat**|}
42 obs_inCs: F $\rightarrow$ {|1:**Nat**|}, obs_outCs: F $\rightarrow$ **Nat**
43 obs_inCs: J $\rightarrow$ **Nat**, obs_outCs: J $\rightarrow$ {|1:**Nat**|}
44 obs_inCs: W $\rightarrow$ {|0:**Nat**|}, obs_outCs: W $\rightarrow$ {|1:**Nat**|}
45 obs_inCs: S $\rightarrow$ {|1:**Nat**|}, obs_outCs: S $\rightarrow$ {|0:**Nat**|}
axiom
42 $\forall$ f:F · obs_outCs(f) $\geq$ 2
43 $\forall$ j:J · obs_inCs(j) $\geq$ 2

- If a pipe, valve or pump unit is input-connected [output-connected] to zero (other) units, then it means that the unit input [output] connector has been sealed.
- If a fork is input-connected to zero (other) units, then it means that the fork input connector has been sealed.
- If a fork is output-connected to n units less than the maximum fork-connectability, then it means that the unconnected fork outputs have been sealed.
- Similarly for joins: "the other way around".

Net Observers and Unit Connections

46. From a net one can observe all its units.
47. From a unit one can observe the the pairs of disjoint input and output units to which it is connected:
    (a) Wells can be connected to zero or one output unit — a pump.
    (b) Sinks can be connected to zero or one input unit — a pump or a valve.
    (c) Pipes, valves and pumps can be connected to zero or one input units and to zero or one output units.
    (d) Forks, f, can be connected to zero or one input unit and to zero or n, $2 \leq n \leq$obs_Cs(f) output units.
    (e) Joins, j, can be connected to zero or n, $2 \leq n \leq$obs_Cs(j) input units and zero or one output units.

value

46 obs_Us: N → U-**set**
47 obs_cUIs: U → UI-**set** × UI-**set**
wf_Conns: U → **Bool**
wf_Conns(u) ≡
let (iuis,ouis) = obs_cUIs(u) **in** iuis ∩ ouis = {} ∧
case u **of**
47(a) mkW(_) → **card** iuis ∈ {0} ∧ **card** ouis ∈ {0,1},
47(b) mkS(_) → **card** iuis ∈ {0,1} ∧ **card** ouis ∈ {0},
47(c) mkΠ(_) → **card** iuis ∈ {0,1} ∧ **card** ouis ∈ {0,1},
47(c) mkV(_) → **card** iuis ∈ {0,1} ∧ **card** ouis ∈ {0,1},
47(c) mkP(_) → **card** iuis ∈ {0,1} ∧ **card** ouis ∈ {0,1},
47(d) mkF(_) → **card** iuis ∈ {0,1} ∧ **card** ouis ∈ {0}∪{2..obs_inCs(j)},
47(e) mkJ(_) → **card** iuis ∈ {0}∪{2..obs_inCs(j)} ∧ **card** ouis ∈ {0,1}
end end

Well-formed Nets, Actual Connections

48. The unit identifiers observed by the obs_cUIs observer must be identifiers of units of the net.

axiom

48 ∀ n:N,u:U · u ∈ obs_Us(n) ⇒
48 **let** (iuis,ouis) = obs_cUIs(u) **in**
48 ∀ ui:UI · ui ∈ iuis ∪ ouis ⇒
48 ∃ u′:U · u′ ∈ obs_Us(n) ∧ u′≠u ∧ obs_UI(u′)=ui **end**

Well-formed Nets, No Circular Nets

49. By a route we shall understand a sequence of units.

50. Units form routes of the net.

type

49 R = UI^{ω}

value

50 routes: N → R-**infset**
50 routes(n) ≡
50 **let** us = obs_Us(n) **in**
50 **let** rs = {⟨u⟩|u:U·u ∈ us} ∪ {r⌢r′|r,r′:R· {r,r′}⊆rs∧adj(r,r′)} **in**
50 rs **end end**

51. A route of length two or more can be decomposed into two routes

52. such that the least unit of the first route "connects" to the first unit of the second route.

value

51 adj: R × R → **Bool**
51 adj(fr,lr) ≡
51 **let** (lu,fu)=(fr(**len** fr),**hd** lr) **in**
52 **let** (lui,fui)=(obs_UI(lu),obs_UI(fu)) **in**
52 **let** ((_,luis),(fuis,_))=(obs_cUIs(lu),obs_cUIs(fu)) **in**
52 lui ∈ fuis ∧ fui ∈ luis **end end end**

53. No route must be circular, that is, the net must be acyclic.

value

53 acyclic: N → **Bool**
53 **let** rs = routes(n) **in**
53 ∼∃ r:R·r ∈ rs⇒∃ i,j:**Nat**·{i,j}⊆**inds** r∧i≠j∧r(i)=r(j) **end**

Pipeline Processes

We now add connectors to our model:

54. From an oil pipeline system one can observe units and connectors.
55. Units are either well, or pipe, or pump, or valve, or join, or fork or sink units.
56. Units and connectors have unique identifiers.
57. From a connector one can observe the ordered pair of the identity of the two from-, respectively to-units that the connector connects.

type
54 OPLS, U, K
56 UI, KI
value
54 obs_Us: OPLS → U-**set**, obs_Ks: OPLS → K-**set**
55 is_WeU, is_PiU, is_PuU, is_VaU, is_JoU, is_FoU, is_SiU: U → **Bool** [mutua
56 obs_UI: U → UI, obs_KI: K → KI
57 obs_UIp: K → (UI|{nil}) × (UI|{nil})

- Above, we think of the types OPLS, U, K, UI and KI as denoting semantic entities.
- Below, in the next section, we shall consider exactly the same types as denoting syntactic entities !

58. There is given an oil pipeline system, opls.
59. To every unit we associate a CSP behaviour.
60. Units are indexed by their unique unit identifiers.
61. To every connector we associate a CSP channel.
    Channels are indexed by their unique "k"onnector identifiers.
62. Unit behaviours are cyclic and over the state of their (static and dynamic) attributes, represented by u.
63. Channels, in this model, have no state.
64. Unit behaviours communicate with neighbouring units — those with which they are connected.
65. Unit functions, $\mathcal{U}_i$, change the unit state.
66. The pipeline system is now the parallel composition of all the unit behaviours.

- **Editorial Remark:**
  - Our use of the term unit and the RSL literal **Unit** may seem confusing, and we apologise.
  - The former, unit, is the generic name of a well, pipe, or pump, or valve, or join, or fork, or sink.
  - The literal **Unit**, in a function signature, before the → "announces" that the function takes no argument.
  - The literal **Unit**, in a function signature, after the → "announces", as used here, that the function never terminates.


```
value
58 opls:OPLS
channel
61 {ch[ ki ]|k:KI,k:K•k ∈ obs_Ks(opls)∧ki=obs_KI(k)} M
value
66 pipeline_system: Unit → Unit
66 pipeline_system() ≡
59   ‖ {unit(ui)(u)|u:U•u ∈ obs_Us(opls)∧ui=obs_UI(u)}

60 unit: ui:UI → U →
64      in,out {ch[ ki ]|k:K,ki:KI•k ∈ obs_Ks(opls)∧ki=obs_KI(k)∧
64                let (ui′,ui″)=obs_UIp(k) in ui ∈{ui′,ui″}\{nil} end}  Unit
62 unit(ui)(u) ≡ let u′ = 𝒰ᵢ(ui)(u) in unit(ui)(u′) end

65 𝒰ᵢ: ui:UI → U →
65      in,out {ch[ ki ]|k:K,ki:KI•k ∈ obs_Ks(opls)∧ki=obs_KI(k)∧
65                let (ui′,ui″)=obs_UIp(k) in ui ∈{ui′,ui″}\{nil} end}  U
```

■ End of Example 54

C.1.3. Discussion

- In this lecture
  - we shall mainly cover
  - atomic and
  - composite

  entities.

C.2. A Conceptual Model of Composite Entities

C.2.1. Systems, Assemblies, Units

- We speak of systems as assemblies.
- From an assembly we can immediately observe a set of parts.
- Parts are either assemblies or units.
- We do not further define what assemblies and units are.

```
type
  S = A, A, U, P = A | U
value
  obs_Ps: (S|A) → P-set
```

- Parts observed from an assembly are said to be immediately embedded in, that is, within, that assembly.
- Two or more different parts of an assembly are said to be immediately adjacent to one another.

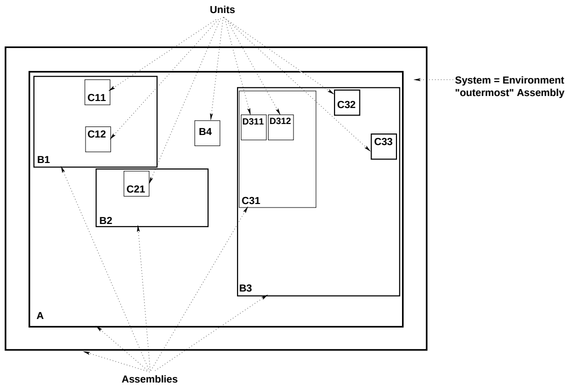

Figure 10: Assemblies and Units "embedded" in an Environment

- A system includes its environment.
- And we do not worry, so far, about the semiotics of all this !

Embeddedness and adjacency generalise to transitive relations.

- Given obs_Ps we can define a function, xtr_Ps,
  - which applies to an assembly a and
  - which extracts all parts embedded in a and including a.
- The functions obs_Ps and xtr_Ps define the meaning of embeddedness.

value

$\text{xtr\_Ps}: (S|A) \rightarrow P\textbf{-set}$

$\text{xtr\_Ps}(a) \equiv$

$\quad \textbf{let } ps = \{a\} \cup \text{obs\_Ps}(a) \textbf{ in } ps \cup \textbf{union}\{\text{xtr\_Ps}(a')|a':A\cdot a' \in ps\} \textbf{ end}$

- **union** is the distributed union operator.
- Parts have unique identifiers.
- All parts observable from a system are distinct.

type

AUI

value

$\text{obs\_AUI}: P \rightarrow \text{AUI}$

axiom

$\forall\, a:A \cdot$

$\quad \textbf{let } ps = \text{obs\_Ps}(a) \textbf{ in}$

$\quad \forall\, p',p'':P \cdot \{p',p''\}\subseteq ps \wedge p'\neq p'' \Rightarrow \text{obs\_AUI}(p')\neq\text{obs\_AUI}(p'') \wedge$

$\quad \forall\, a',a'':A \cdot \{a',a''\}\subseteq ps \wedge a'\neq a'' \Rightarrow \text{xtr\_Ps}(a')\cap \text{xtr\_Ps}(a'')=\{\} \textbf{ end}$

C.2.2. 'Adjacency' and 'Within' Relations

- Two parts, p,p′, are said to be *immediately next to*, i.e., i_next_to(p,p′)(a), one another in an assembly a
  - if there exists an assembly, a' equal to or embedded in a
  - such that p and p′ are observable in that assembly a′.

value

$\text{i\_next\_to}: P \times P \rightarrow A \xrightarrow{\sim} \textbf{Bool}, \textbf{pre } \text{i\_next\_to}(p,p')(a): p\neq p'$

$\text{i\_next\_to}(p,p')(a) \equiv \exists\, a':A \cdot a'=a \vee a' \in \text{xtr\_Ps}(a) \cdot \{p,p'\}\subseteq\text{obs\_Ps}(a')$

- One part, p, is said to be *immediately within* another part, p′in an assembly a
  - if there exists an assembly, a′ equal to or embedded in a
  - such that p is observable in a′.

value

i_within: P × P → A $\xrightarrow{\sim}$ **Bool**
i_within(p,p′)(a) ≡
∃ a′:A • (a=a′ ∨ a′ ∈ xtr_Ps(a)) • p′=a′ ∧ p ∈ obs_Ps(a′)

- We can generalise the immediate 'within' property.
- A part, p, is (transitively) within a part p′, within(p,p′)(a), of an assembly, a,
  - either if p, is immediately within p′ of that assembly, a,
  - or if there exists a (proper) part p′′ of p′
  - such that within(p′′,p)(a).

value

within: P × P → A $\xrightarrow{\sim}$ **Bool**
within(p,p′)(a) ≡
i_within(p,p′)(a) ∨ ∃ p′′:P • p′′ ∈ obs_Ps(p) ∧ within(p′′,p′)(a)

- The function within can be defined, alternatively,
- using xtr_Ps and i_within
- instead of obs_Ps and within :

value

within′: P × P → A $\xrightarrow{\sim}$ **Bool**
within′(p,p′)(a) ≡
i_within(p,p′)(a) ∨ ∃ p′′:P • p′′ ∈ xtr_Ps(p) ∧ i_within(p′′,p′)(a)

lemma: within ≡ within′

- We can generalise the immediate 'next to' property.
- Two parts, p, p′ of an assembly, a, are adjacent if they are
  - either 'next to' one another
  - or if there are two parts p_o, p'_o
    - such that p, p′ are embedded in respectively p_o and p'_o
    - and such that p_o, p'_o are immediately next to one another.

value

adjacent: P × P → A $\xrightarrow{\sim}$ **Bool**
adjacent(p,p′)(a) ≡
i_next_to(p,p′)(a) ∨
∃ p′′,p′′′:P • {p′′,p′′′}⊆xtr_Ps(a) ∧ i_next_to(p′′,p′′′)(a) ∧
((p=p′′)∨within(p,p′′)(a)) ∧ ((p′=p′′′)∨within(p′,p′′′)(a))

C.2.3. Mereology, Part I

- So far we have built a *ground mereology* model, $\mathcal{M}_{\mathcal{G}\mathrm{round}}$.
- Let $\sqsubseteq$ denote *parthood, x is part of y*, $x \sqsubseteq y$.

$$\forall x(x \sqsubseteq x)^{13} \quad (1)$$
$$\forall x, y(x \sqsubseteq y) \wedge (y \sqsubseteq x) \Rightarrow (x = y) \quad (2)$$
$$\forall x, y, z(x \sqsubseteq y) \wedge (y \sqsubseteq z) \Rightarrow (x \sqsubseteq z) \quad (3)$$

- Let $\sqsubset$ denote *proper parthood, x is part of y*, $x \sqsubset y$.
- Formula 4 defines $x \sqsubset y$. Equivalence 5 can be proven to hold.

$$\forall x \sqsubset y =_{\mathrm{def}} x(x \sqsubseteq y) \wedge \neg(x = y) \quad (4)$$
$$\forall\forall x, y(x \sqsubseteq y) \quad \Leftrightarrow \quad (x \sqsubset y) \vee (x = y) \quad (5)$$

[13] Our notation now is not RSL but some conventional first-order predicate logic notation.

- The *proper part* $(x \sqsubset y)$ relation is a strict partial ordering:

$$\forall x \neg(x \sqsubset x) \quad (6)$$
$$\forall x, y(x \sqsubset y) \Rightarrow \neg(y \sqsubset x) \quad (7)$$
$$\forall x, y, z(x \sqsubset y) \wedge (y \sqsubset z) \Rightarrow (x \sqsubset z) \quad (8)$$

- *Overlap*, $\bullet$, is also a relation of parts:
  - Two individuals overlap if they have parts in common:

$$x \bullet y =_{\mathrm{def}} \exists z(z \sqsubset x) \wedge (z \sqsubset y) \quad (9)$$
$$\forall x(x \bullet x) \quad (10)$$
$$\forall x, y(x \bullet y) \Rightarrow (y \bullet x) \quad (11)$$

- Proper overlap, $\circ$, can be defined:

$$x \circ y =_{\mathrm{def}} (x \bullet x) \wedge \neg(x \sqsubseteq y) \wedge \neg(y \sqsubseteq x) \quad (12)$$

- Whereas Formulas (1-11) holds of the model of mereology we have shown so far, Formula (12) does not.
- In the next section we shall repair that situation.
- The *proper part* relation, $\sqsubset$, reflects the *within* relation.
- The *disjoint* relation, $\oint$, reflects the *adjacency* relation.

$$x \oint y =_{\mathrm{def}} \neg(x \bullet y) \quad (13)$$

- Disjointness is symmetric:

$$\forall x, y(x \oint y) \Rightarrow (y \oint x) \quad (14)$$

- The *weak supplementation* relation, Formula 15, expresses
  - that if y is a proper part of x
  - then there exists a part z
  - such that z is a proper part of x
  - and z and y are disjoint
- That is, whenever an individual has one proper part then it has more than one.

$$\forall x, y(y \sqsubset x) \quad \Rightarrow \quad \exists z(z \sqsubset x) \wedge (z \oint y) \quad (15)$$

- Formulas 1–3 and 15 together determine the *minimal mereology*, $\mathcal{M}_{\mathcal{M}\text{inimal}}$.
- Formula 15 does not hold of the model of mereology we have shown so far..
- Formula 15 on the previous page expresses that
  - whenever an individual has one proper part
  - then it has more than one.
- We mentioned there, Slide 402, that we would comment on the fact that our model appears to allow that assemblies may have just one proper part.

- We now do so.
  - We shall still allow assemblies to have just one proper part —
  - in the sense of a sub-assembly or a unit —
  - but we shall interpret the fact that an assembly always have at least one attribute.
  - Therefore we shall "generously" interpret the set of attributes of an assembly to constitute a part.

- In Sect. A.6
  - we shall see how attributes of both units and assemblies of the interpreted mereology
  - contribute to the state components of the unit and assembly processes.

C.2.4. Connectors

- So far we have only covered notions of
  - parts being next to other parts or
  - within one another.
- We shall now add to this a rather general notion of parts being otherwise related.
- That notion is one of connectors.

- Connectors provide for connections between parts.
- A connector is an ability be be connected.
- A connection is the actual fulfillment of that ability.
- Connections are relations between pairs of parts.
- Connections "cut across" the "classical"
  - *parts being part of the (or a) whole* and
  - *parts being related by embeddedness or adjacency*.

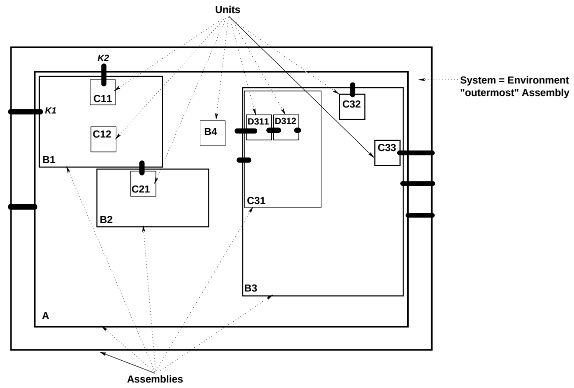

Figure 11: Assembly and Unit Connectors: Internal and External

- For now, we do not "ask" for the meaning of connectors !

- Figure 11 on the preceding page "adds" connectors to Fig. 10 on page 390.
- The idea is that connectors
  - allow an assembly to be connected to any embedded part, and
  - allow two adjacent parts to be connected.
- In Fig. 11 on the preceding page
  - the environment is connected, by *K2*, to part C11;
  - the "external world" is connected, by K1, to B1;
  - etcetera.

- From a system we can observe all its connectors.
- From a connector we can observe
  - its unique connector identifier and
  - the set of part identifiers of the parts that the connector connects.
- All part identifiers of system connectors identify parts of the system.
- All observable connector identifiers of parts identify connectors of the system.

type
 K
value
 obs_Ks: S → K-**set**
 obs_KI: K → KI
 obs_Is: K → AUI-**set**
 obs_KIs: P → KI-**set**
axiom
 ∀ k:K · **card** obs_Is(k)=2,
 ∀ s:S,k:K · k ∈ obs_Ks(s) ⇒
  ∃ p:P · p ∈ xtr_Ps(s) ⇒ obs_AUI(p) ∈ obs_Is(k),
 ∀ s:S,p:P · ∀ ki:KI · ki ∈ obs_KIs(p) ⇒
  ∃! k:K · k ∈ obs_Ks(s) ∧ ki=obs_KI(k)

- This model allows for a rather "free-wheeling" notion of connectors
  - one that allows internal connectors to "cut across" embedded and adjacent parts;
  - and one that allows external connectors to "penetrate" from an outside to any embedded part.
- We need define an auxiliary function.
  - xtr∀KIs(p) applies to a system
  - and yields all its connector identifiers.

value
 xtr∀KIs: S → KI-**set**
 xtr∀Ks(s) ≡ {obs_KI(k)|k:K·k ∈ obs_Ks(s)}

C.2.5. Mereology, Part II

(See Sect. (Slide 398) for Mereology, Part I.)
We shall interpret connections as follows:

- A connection between parts p_i and p_j
  - that enjoy a p_i adjacent to p_j relationship, means $p_i \circ p_j$,
  - that is, although parts p_i and p_j are adjacent
  - they do *share* "something", i.e., have something *in common*.
  - What that "something" is we shall comment on later, when we have "mapped" systems onto parallel compositions of CSP processes.
- A connection between parts p_i and p_j
  - that enjoy a p_i within p_j relationship,
  - does not add other meaning than
  - commented upon later, again when we have "mapped" systems onto parallel compositions of CSP processes.

- With the above interpretation we may arrive at the following, perhaps somewhat "awkward-looking" case:
  - a connection connects two adjacent parts p_i and p_j
    - where part p_i is within part p_{i_o}
    - and part p_j is within part p_{j_o}
    - where parts p_{i_o} and p_{j_o} are adjacent
    - but not otherwise connected.
  - How are we to explain that !
    - Since we have not otherwise interpreted the meaning of parts,
    - we can just postulate that "so it is" !
    - We shall, later, again when we have "mapped" systems onto parallel compositions of CSP processes, give a more satisfactory explanation.

- On Slides 398–401 we introduced the following operators:
  - $\sqsubseteq$, $\sqsubset$, $\bullet$, $\circ$, and $\oint$
- In some of the mereology literature these operators are symbolised with caligraphic letters:
  - $\sqsubseteq$: $\mathcal{P}$: part,
  - $\sqsubset$: $\mathcal{PP}$: proper part,
  - $\bullet$: $\mathcal{O}$: overlap and
  - $\oint$: $\mathcal{U}$: underlap.

C.2.6. Discussion

Summary:

- This ends our first model of a concept of mereology.
- The parts are those of assemblies and units.
- The relations between parts and the whole are,
  - on one hand, those of
    * embeddedness i.e. within, and
    * adjacency, i.e., adjacent,

    and
  - on the other hand, those expressed by connectors: relations
    * between arbitrary parts and
    * between arbitrary parts and the exterior.

Extensions:

- A number of extensions are possible:
  - one can add "mobile" parts and "free" connectors, and
  - one can further add operations that allow such mobile parts to move from one assembly to another along routes of connectors.
- Free connectors and mobility assumes static versus dynamic parts and connectors:
  - a free connector is one which allows a mobile part to be connected to another part, fixed or mobile; and
  - the potentiality of a move of a mobile part introduces a further dimension of dynamics of a mereology.

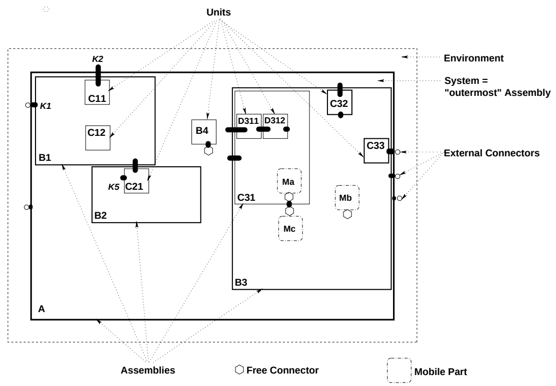

Figure 12: Mobile Parts and Free Connectors

Comments:

- We shall leave the modelling of free connectors and mobile parts to another time.
- Suffice it now to indicate that the mereology model given so far is relevant:
  - that it applies to a somewhat wide range of application domain structures, and
  - that it thus affords a uniform treatment of proper formal models of these application domain structures.

C.3. Functions and Events

-
-
-
-

Example 55 – Pipeline Transport Functions and Events:

- We need introduce a number of auxiliary concepts
- in order to show examples of atomic and composite
- functions and events.

Well-formed Nets, Special Pairs, wfN_SP

67. We define a "special-pairs" well-formedness function.
    (a) Fork outputs are output-connected to valves.
    (b) Join inputs are input-connected to valves.
    (c) Wells are output-connected to pumps.
    (d) Sinks are input-connected to either pumps or valves.

value

67 wfN_SP: N → **Bool**
67 wfN_SP(n) ≡
67 ∀ r:R · r ∈ routes(n) **in**
67 ∀ i:**Nat** · {i,i+1}⊆**inds** r ⇒
67 **case** r(i) **of** ∧
67(a) mkF(_) → ∀ u:U·adj(⟨r(i)⟩,⟨u⟩) ⇒ is_V(u),_→**true end** ∧
67 **case** r(i+1) **of**
67(b) mkJ(_) → ∀ u:U·adj(⟨u⟩,⟨r(i)⟩) ⇒ is_V(u),_→**true end** ∧
67 **case** r(1) **of**
67(c) mkW(_) → is_P(r(2)),_→**true end** ∧
67 **case** r(**len** r) **of**
67(d) mkS(_) → is_P(r(**len** r−1))∨is_V(r(**len** r−1)),_→**true end**

- The **true** clauses may be negated by other **case** distinctions' is_V or is_V clauses.

Special Routes, I

68. A pump-pump route is a route of length two or more whose first and last units are pumps and whose intermediate units are pipes or forks or joins.
69. A simple pump-pump route is a pump-pump route with no forks and joins.
70. A pump-valve route is a route of length two or more whose first unit is a pump, whose last unit is a valve and whose intermediate units are pipes or forks or joins.
71. A simple pump-valve route is a pump-valve route with no forks and joins.
72. A valve-pump route is a route of length two or more whose first unit is a valve, whose last unit is a pump and whose intermediate units are pipes or forks or joins.
73. A simple valve-pump route is a valve-pump route with no forks and joins.
74. A valve-valve route is a route of length two or more whose first and last units are valves and whose intermediate units are pipes or forks or joins.
75. A simple valve-valve route is a valve-valve route with no forks and joins.

value

68-75 ppr,sppr,pvr,spvr,vpr,svpr,vvr,svvr: R → **Bool**
pre {ppr,sppr,pvr,spvr,vpr,svpr,vvr,svvr}(n): **len** n≥2

68 ppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_P(fu) ∧ is_P(lu) ∧ is_πfjr(ℓ)
69 sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
70 pvr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_P(fu) ∧ is_V(r(**len** r)) ∧ is_πfjr(ℓ)
71 sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
72 vpr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_V(fu) ∧ is_P(lu) ∧ is_πfjr(ℓ)
73 sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
74 vvr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_V(fu) ∧ is_V(lu) ∧ is_πfjr(ℓ)
75 sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)

is_πfjr,is_πr: R → **Bool**
is_πfjr(r) ≡ ∀ u:U·u ∈ **elems** r⇒is_Π(u)∨is_F(u)∨is_J(u)
is_πr(r) ≡ ∀ u:U·u ∈ **elems** r⇒is_Π(u)

Special Routes, II

Given a unit of a route,

76. if they exist (∃),
77. find the nearest pump or valve unit,
78. "upstream" and
79. "downstream" from the given unit.

(C. Mereology C.3. Functions and Events)

value

76 $\exists$UpPoV: U $\times$ R $\rightarrow$ **Bool**
76 $\exists$DoPoV: U $\times$ R $\rightarrow$ **Bool**
78 find_UpPoV: U $\times$ R $\stackrel{\sim}{\rightarrow}$ (P|V), **pre** find_UpPoV(u,r): $\exists$UpPoV(u,r)
79 find_DoPoV: U $\times$ R $\stackrel{\sim}{\rightarrow}$ (P|V), **pre** find_DoPoV(u,r): $\exists$DoPoV(u,r)
76 $\exists$UpPoV(u,r) $\equiv$
76 $\exists$ i,j **Nat**$\cdot$\{i,j\}$\subseteq$**inds** r$\wedge$i$\leq$j$\wedge$\{is_V|is_P\}(r(i))$\wedge$u=r(j)
76 $\exists$DoPoV(u,r) $\equiv$
76 $\exists$ i,j **Nat**$\cdot$\{i,j\}$\subseteq$**inds** r$\wedge$i$\leq$j$\wedge$u=r(i)$\wedge$\{is_V|is_P\}(r(j))
78 find_UpPoV(u,r) $\equiv$
78 **let** i,j:**Nat**$\cdot$\{i,j\}$\subseteq$indsr$\wedge$i$\leq$j$\wedge$\{is_V|is_P\}(r(i))$\wedge$u=r(j) **in** r(i) **end**
79 find_DoPoV(u,r) $\equiv$
79 **let** i,j:**Nat**$\cdot$\{i,j\}$\subseteq$indsr$\wedge$i$\leq$j$\wedge$u=r(i)$\wedge$\{is_V|is_P\}(r(j)) **in** r(j) **end**

(C. Mereology C.3. Functions and Events)

State Attributes of Pipeline Units

- By a state attribute of a unit we mean either of the following three kinds:
  - (i) the open/close states of valves and the pumping/not_pumping states of pumps;
  - (ii) the maximum (laminar) oil flow characteristics of all units; and
  - (iii) the current oil flow and current oil leak states of all units.

(C. Mereology C.3. Functions and Events)

80. Oil flow, $\phi : \Phi$, is measured in volume per time unit.
81. Pumps are either pumping or not pumping, and if not pumping they are closed.
82. Valves are either open or closed.
83. Any unit permits a maximum input flow of oil while maintaining laminar flow. We shall assume that we need not be concerned with turbulent flows.
84. At any time any unit is sustaining a current input flow of oil (at its input(s)).
85. While sustaining (even a zero) current input flow of oil a unit leaks a current amount of oil (within the unit).

(C. Mereology C.3. Functions and Events)

type

80 Φ
81 PΣ == pumping | not_pumping
81 VΣ == open | closed

value

−,+: $\Phi \times \Phi \rightarrow \Phi$, <,=,>: $\Phi \times \Phi \rightarrow$ **Bool**
81 obs_PΣ: P $\rightarrow$ PΣ
82 obs_VΣ: V $\rightarrow$ VΣ
83–85 obs_LamiΦ.obs_CurrΦ,obs_LeakΦ: U $\rightarrow$ Φ
is_Open: U $\rightarrow$ **Bool**
case u **of**
mkΠ(_)$\rightarrow$**true**,mkF(_)$\rightarrow$**true**,mkJ(_)$\rightarrow$**true**,mkW(_)$\rightarrow$**true**,mkS(_)$\rightarrow$**true**,
mkP(_)$\rightarrow$obs_PΣ(u)=pumping,
mkV(_)$\rightarrow$obs_VΣ(u)=open
end
acceptable_LeakΦ, excessive_LeakΦ: U $\rightarrow$ Φ

axiom

$\forall$ u:U $\cdot$ excess_LeakΦ(u) > accept_LeakΦ(u)

Flow Laws

- The sum of the current flows into a unit equals the the sum of the current flows out of a unit minus the (current) leak of that unit.
- This is the same as the current flows out of a unit equals the current flows into a unit minus the (current) leak of that unit.
- The above represents an interpretation which justifies the below laws.

86. When, in Item 84, for a unit u, we say that at any time any unit is sustaining a current input flow of oil, and when we model that by obs_CurrΦ(u) then we mean that obs_CurrΦ(u) - obs_LeakΦ(u) represents the flow of oil from its outputs.

value

86 obs_inΦ: U $\rightarrow$ Φ

86 obs_inΦ(u) $\equiv$ obs_CurrΦ(u)

86 obs_outΦ: U $\rightarrow$ Φ

law:

86 $\forall$ u:U $\cdot$ obs_outΦ(u) $=$ obs_CurrΦ(u)$-$obs_LeakΦ(u)

87. Two connected units enjoy the following flow relation:
    (a) If
        i. two pipes, or
        ii. a pipe and a valve, or
        iii. a valve and a pipe, or
        iv. a valve and a valve, or
        v. a pipe and a pump, or
        vi. a pump and a pipe, or
        vii. a pump and a pump, or
        viii. a pump and a valve, or
        ix. a valve and a pump

        are immediately connected
    (b) then
        i. the current flow out of the first unit's connection to the second unit
        ii. equals the current flow into the second unit's connection to the first unit

law:

87(a) $\forall$ u,u$'$:U $\cdot$ {is_Π,is_V,is_P,is_W}(u$'$|u$''$) $\wedge$ adj($\langle$u$\rangle$,$\langle$u$'\rangle$)

87(a) is_Π(u)$\vee$is_V(u)$\vee$is_P(u)$\vee$is_W(u) $\wedge$

87(a) is_Π(u$'$)$\vee$is_V(u$'$)$\vee$is_P(u$'$)$\vee$is_S(u$'$)

87(b) $\Rightarrow$ obs_outΦ(u)=obs_inΦ(u$'$)

- A similar law can be established for forks and joins.
  - For a fork
    - output-connected to, for example, pipes, valves and pumps,
    - it is the case that for each fork output
    - the out-flow equals the in-flow for that output-connected unit.
  - For a join
    - input-connected to, for example, pipes, valves and pumps,
    - it is the case that for each join input
    - the in-flow equals the out-flow for that input-connected unit.
  - We leave the formalisation as an exercise.

Possibly Desirable Properties

88. Let r be a route of length two or more, whose first unit is a pump, p, whose last unit is a valve, v and whose intermediate units are all pipes: if the pump, p is pumping, then we expect the valve, v, to be open.
89. Let r be a route of length two or more, whose first unit is a pump, p, whose last unit is another pump, p' and whose intermediate units are all pipes: if the pump, p is pumping, then we expect pump p'', to also be pumping.
90. Let r be a route of length two or more, whose first unit is a valve, v, whose last unit is a pump, p and whose intermediate units are all pipes: if the valve, v is closed, then we expect pump p, to not be pumping.
91. Let r be a route of length two or more, whose first unit is a valve, v', whose last unit is a valve, v'' and whose intermediate units are all pipes: if the valve, v' is in some state, then we expect valve v'', to also be in the same state.

desirable properties:

88 $\forall$ r:R $\cdot$ spvr(r) $\wedge$
88 **spvr_prop(r):** obs_PΣ(**hd** r)=pumping ⇒ obs_PΣ(r(**len** r))=open

89 $\forall$ r:R $\cdot$ sppr(r) $\wedge$
89 **sppr_prop(r):** obs_PΣ(**hd** r)=pumping⇒obs_PΣ(r(**len** r))=pumping

90 $\forall$ r:R $\cdot$ svpr(r) $\wedge$
90 **svpr_prop(r):** obs_PΣ(**hd** r)=open⇒obs_PΣ(r(**len** r))=pumping

91 $\forall$ r:R $\cdot$ svvr(r) $\wedge$
91 **svvr_prop(r):** obs_PΣ(**hd** r)=obs_PΣ(r(**len** r))

Pipeline Actions

•***Simple Pump and Valve Actions***

92. Pumps may be set to pumping or reset to not pumping irrespective of the pump state.
93. Valves may be set to be open or to be closed irrespective of the valve state.
94. In setting or resetting a pump or a valve a desirable property may be lost.

value

92 pump_to_pump, pump_to_not_pump: P → N → N
93 valve_to_open, valve_to_close: V → N → N

value

92 pump_to_pump(p)(n) **as** n′
92 **pre** p ∈ obs_Us(n)
92 **post let** p′:P•obs_UI(p)=obs_UI(p′) **in**
92 obs_PΣ(p′)=pumping∧else_equal(n,n′)(p,p′) **end**
92 pump_to_not_pump(p)(n) **as** n′
92 **pre** p ∈ obs_Us(n)
92 **post let** p′:P•obs_UI(p)=obs_UI(p′) **in**
92 obs_PΣ(p′)=not_pumping∧else_equal(n,n′)(p,p′) **end**
93 valve_to_open(v)(n) **as** n′
92 **pre** v ∈ obs_Us(n)
93 **post let** v′:V•obs_UI(v)=obs_UI(v′) **in**
92 obs_VΣ(v′)=open∧else_equal(n,n′)(v,v′) **end**
93 valve_to_close(v)(n) **as** n′
92 **pre** v ∈ obs_Us(n)
93 **post let** v′:V•obs_UI(v)=obs_UI(v′) **in**
92 obs_VΣ(v′)=close∧else_equal(n,n′)(v,v′) **end**

value

else_equal: (N×N) → (U×U) → **Bool**
else_equal(n,n′)(u,u′) ≡
obs_UI(u)=obs_UI(u′)
∧ u ∈ obs_Us(n)∧u′ ∈ obs_Us(n′)
∧ omit_Σ(u)=omit_Σ(u′)
∧ obs_Us(n)\{u}=obs_Us(n)\{u′}
∧ ∀ u″:U•u″ ∈ obs_Us(n)\{u} ≡ u″ ∈ obs_Us(n′)\{u′}

omit_Σ: U → $\mathrm{U}_{\mathrm{no\_state}}$ −−− ″`magic`″ function

=: $\mathrm{U}_{\mathrm{no\_state}}$ × $\mathrm{U}_{\mathrm{no\_state}}$ → **Bool**

axiom

∀ u,u′:U•omit_Σ(u)=omit_Σ(u′) ≡ obs_UI(u)=obs_UI(u′)

Events

•*Unit Handling Events*

95. Let n be any acyclic net.
95. If there exists p, p', v, v', pairs of distinct pumps and distinct valves of the net,
95. and if there exists a route, r, of length two or more of the net such that
96. all units, u, of the route, except its first and last unit, are pipes, then
97. if the route "spans" between p and p' and the *simple desirable property*, sppr(r), does not hold for the route, then we have a possibly undesirable event — that occurred as soon as sppr(r) did not hold;
98. if the route "spans" between p and v and the *simple desirable property*, spvr(r), does not hold for the route, then we have a possibly undesirable event;
99. if the route "spans" between v and p and the *simple desirable property*, svpr(r), does not hold for the route, then we have a possibly undesirable event; and
100. if the route "spans" between v and v' and the *simple desirable property*, svvr(r), does not hold for the route, then we have a possibly undesirable event.

events:

95 ∀ n:N · acyclic(n) ∧
95 ∃ p,p′:P,v,v′:V · {p,p′,v,v′}⊆obs_Us(n)⇒
95 ∧ ∃ r:R · routes(n) ∧
96 ∀ u:U · u ∈ **elems**(r)\{**hd** r,r(**len** r)} ⇒ is_Π(i) ⇒
97 p=**hd** r∧p′=r(**len** r) ⇒ ∼sppr_prop(r) ∧
98 p=**hd** r∧v=r(**len** r) ⇒ ∼spvr_prop(r) ∧
99 v=**hd** r∧p=r(**len** r) ⇒ ∼svpr_prop(r) ∧
100 v=**hd** r∧v′=r(**len** r) ⇒ ∼svvr_prop(r)

• *Foreseeable Accident Events*

- A number of foreseeable accidents may occur.

101. A unit ceases to function, that is,
    (a) a unit is clogged,
    (b) a valve does not open or close,
    (c) a pump does not pump or stop pumping.
102. A unit gives rise to excessive leakage.
103. A well becomes empty or a sunk becomes full.
104. A unit, or a connected net of units gets on fire.
105. Or a number of other such “accident”.

Well-formed Operational Nets

106. A well-formed operational net
107. is a well-formed net
    (a) with at least one well, w, and at least one sink, s,
    (b) and such that there is a route in the net between w and s.

value

106 wf_OpN: N → **Bool**
106 wf_OpN(n) ≡
107   satisfies axiom 48 on page 379 ∧ acyclic(n): Item 53 on page 381 ∧
107   wfN_SP(n): Item 67 on page 421 ∧
107   satisfies flow laws, 86 on page 430 and 87 on page 432 ∧
107(a)   ∃ w:W,s:S · {w,s}⊆obs_Us(n) ⇒
107(b)   ∃ r:R· ⟨w⟩^r^⟨s⟩ ∈ routes(n)

Orderly Action Sequences

•*Initial Operational Net*

108. Let us assume a notion of an initial operational net.
109. Its pump and valve units are in the following states
    (a) all pumps are not_pumping, and
    (b) all valves are closed.

value

108 initial_OpN: N → **Bool**
109 initial_OpN(n) ≡ wf_OpN(n) ∧
109(a)   ∀ p:P · p ∈ obs_Us(n) ⇒ obs_PΣ(p)=not_pumping ∧
109(b)   ∀ v:V · v ∈ obs_Us(n) ⇒ obs_VΣ(p)=closed

Oil Pipeline Preparation and Engagement

110. We now wish to prepare a pipeline from some well, $w : W$, to some sink, $s : S$, for flow.
    (a) We assume that the underlying net is operational wrt. w and s, that is, that there is a route, r, from w to s.
    (b) Now, an orderly action sequence for engaging route r is to “work backwards”, from s to w
    (c) setting encountered pumps to pumping and valves to open.

- In this way the system is well-formed wrt. the desirable sppr, spvr, svpr and svvr properties.
- Finally, setting the pump adjacent to the (preceding) well starts the system.

value

110 prepare_and_engage: W × S → N $\stackrel{\sim}{\rightarrow}$ N
110 prepare_and_engage(w,s)(n) ≡
110(a) **let** r:R • ⟨w⟩⁀r⁀⟨s⟩ ∈ routes(n) **in**
110(b) action_sequence(⟨w⟩⁀r⁀⟨s⟩)(**len**⟨w⟩⁀r⁀⟨s⟩)(n) **end**
110 **pre** ∃ r:R • ⟨w⟩⁀r⁀⟨s⟩ ∈ routes(n)

110(c) action_sequence: R → **Nat** → N → N
110(c) action_sequence(r)(i)(n) ≡
110(c) **if** i=1 **then** n **else**
110(c) **case** r(i) **of**
110(c) mkV(_) → action_sequence(r)(i−1)(valve_to_open(r(i))(n)),
110(c) mkP(_) → action_sequence(r)(i−1)(pump_to_pump(r(i))(n)),
110(c) _ → action_sequence(r)(i−1)(n)
110(c) **end end**

Emergency Actions

111. If a unit starts leaking excessive oil
   (a) then nearest up-stream valve(s) must be closed,
   (b) and any pumps in-between this (these) valves and the leaking unit must be set to not_pumping
   (c) — following an orderly sequence.
112. If, as a result, for example, of the above remedial actions, any of the desirable properties cease to hold
   (a) then — a ha !
   (b) Left as an exercise.

■ End of Example 55

C.4. Behaviours: A Semantic Model of a Class of Mereologies

- The model of mereology (Slides 389–349) given earlier focused on the following simple entities (i) the assemblies, (ii) the units and (iii) the connectors.
- To assemblies and units we associate CSP processes, and
- to connectors we associate a CSP channels,
- one-by-one.
- The connectors form the mereological attributes of the model.

C.4.1. Channels

- The CSP channels,
  – are each "anchored" in two parts:
  – if a part is a unit then in "its corresponding" unit process, and
  – if a part is an assembly then in "its corresponding" assembly process.
- From a system assembly we can extract all connector identifiers.
- They become indexes into an array of channels.
  – Each of the connector channel identifiers is mentioned
  – in exactly two unit or assembly processes.

value
 s:S
 $\text{kis:KI-}\mathbf{set} = \text{xtr}\forall\text{KIs(s)}$
type
 $\text{ChMap} = \text{AUI} \xrightarrow[m]{} \text{KI-}\mathbf{set}$
value
 $\text{cm:ChMap} = [\,\text{obs\_AUI(p)}\mapsto\text{obs\_KIs(p)}\,|\,\text{p:P}\cdot\text{p} \in \text{xtr\_Ps(s)}\,]$
channel
 $\text{ch}[\,\text{i}|\text{i:KI}\cdot\text{i} \in \text{kis}\,]\ \text{MSG}$

C.4.2. Process Definitions

value
 $\text{system: S} \to \mathbf{Process}$
 $\text{system(s)} \equiv \text{assembly(s)}$

 $\text{assembly: a:A}\to\mathbf{in},\mathbf{out}\ \{\text{ch}[\,\text{cm(i)}\,]|\text{i:KI}\cdot\text{i} \in \text{cm(obs\_AUI(a))}\}\ \mathbf{process}$
 $\text{assembly(a)} \equiv$
 $\quad\mathcal{M}_\mathcal{A}\text{(a)(obs\_A}\Sigma\text{(a))} \parallel$
 $\quad\parallel \{\text{assembly(a}')|\text{a}'\text{:A}\cdot\text{a}' \in \text{obs\_Ps(a)}\} \parallel$
 $\quad\parallel \{\text{unit(u)}|\text{u:U}\cdot\text{u} \in \text{obs\_Ps(a)}\}$
 $\text{obs\_A}\Sigma\text{: A} \to \text{A}\Sigma$

 $\mathcal{M}_\mathcal{A}\text{: a:A}\to\text{A}\Sigma\to\mathbf{in},\mathbf{out}\ \{\text{ch}[\,\text{cm(i)}\,]|\text{i:KI}\cdot\text{i} \in \text{cm(obs\_AUI(a))}\}\ \mathbf{process}$
 $\mathcal{M}_\mathcal{A}\text{(a)(a}\sigma\text{)} \equiv \mathcal{M}_\mathcal{A}\text{(a)(}A\mathcal{F}\text{(a)(a}\sigma\text{))}$

 $\text{A}\mathcal{F}\text{: a:A} \to \text{A}\Sigma \to \mathbf{in},\mathbf{out}\ \{\text{ch}[\,\text{em(i)}\,]|\text{i:KI}\cdot\text{i} \in$
 $\text{cm(obs\_AUI(a))}\}\times\text{A}\Sigma$

 $\text{unit: u:U} \to \mathbf{in},\mathbf{out}\ \{\text{ch}[\,\text{cm(i)}\,]|\text{i:KI}\cdot\text{i} \in \text{cm(obs\_UI(u))}\}\ \mathbf{process}$
 $\text{unit(u)} \equiv \mathcal{M}_\mathcal{U}\text{(u)(obs\_U}\Sigma\text{(u))}$
 $\text{obs\_U}\Sigma\text{: U} \to \text{U}\Sigma$

 $\mathcal{M}_\mathcal{U}\text{: u:U} \to \text{U}\Sigma \to \mathbf{in},\mathbf{out}\ \{\text{ch}[\,\text{cm(i)}\,]|\text{i:KI}\cdot\text{i} \in \text{cm(obs\_UI(u))}\}\ \mathbf{process}$
 $\mathcal{M}_\mathcal{U}\text{(u)(u}\sigma\text{)} \equiv \mathcal{M}_\mathcal{U}\text{(u)(}U\mathcal{F}\text{(u)(u}\sigma\text{))}$

 $\text{U}\mathcal{F}\text{: U} \to \text{U}\Sigma \to \mathbf{in},\mathbf{out}\ \{\text{ch}[\,\text{em(i)}\,]|\text{i:KI} \cdot \text{i} \in \text{cm(obs\_AUI(u))}\}\ \ \text{U}\Sigma$

C.4.3. Mereology, Part III

- (See Sect. on page 412 for Mereology, Part II.)
- A little more meaning has been added to the notions of parts and connections.
- The within and adjacent to relations between parts (assemblies and units) reflect a phenomenological world of geometry, and
- the connected relation between parts (assemblies and units)
  - reflect both physical and conceptual world understandings:
    - ∗ physical world in that, for example, radio waves cross geometric "boundaries", and
    - ∗ conceptual world in that ontological classifications typically reflect lattice orderings where *overlaps* likewise cross geometric "boundaries".

C.4.4. Discussion

C.4.4.1. Partial Evaluation

- The assembly function "first" "functions" as a compiler.
- The 'compiler' translates an assembly structure into three process expressions:
  - the $\mathcal{M}_{\mathcal{A}}(a)(a\sigma)$ invocation,
  - the parallel composition of assembly processes, a', one for each sub-assembly of a, and
  - the parallel composition of unit processes, one for each unit of assembly a —
  - with these three process expressions "being put in parallel".
  - The recursion in assembly ends when a sub-…-assembly consists of no sub-sub-…-assemblies.
- Then the compiling task ends and the many generated $\mathcal{M}_{\mathcal{A}}(a)(a\sigma)$ and $\mathcal{M}_{\mathcal{U}}(u)(u\sigma)$ process expressions are invoked.

C.4.4.2. Generalised Channel Processes

- That completes our 'contribution':
  - A mereology of systems has been given
  - a syntactic explanation, Sect. 2,
  - a semantic explanation, Sect. 5 and
  - their relationship to classical mereologies.

End of Lecture 12: MEREOLOGY