Welcome Back — Thanks !

Lecture 3: 14:00–14:40 + 14:50–15:30
Discrete Perdurant and Contiuous Entities

5. Discrete Perdurant Entities

- From Wikipedia:
  - *Perdurant: Also known as occurrent, accident or happening.*
  - *Perdurants are those entities for which only a fragment exists if we look at them at any given snapshot in time.*
  - *When we freeze time we can only see a fragment of the perdurant.*
  - *Perdurants are often what we know as processes, for example 'running'.*
  - *If we freeze time then we only see a fragment of the running, without any previous knowledge one might not even be able to determine the actual process as being a process of running.*
  - *Other examples include an activation, a kiss, or a procedure.*
- A discrete perdurant$_\delta$ is a perdurant which is a discrete entity.

- We shall consider the following **discrete perdurants**.
  - **actions** (Sect. 5.1),
  - **events** (Sect. 5.2), and
  - **discrete behaviours** (Sect. 5.3).
- **Actions** and **events**
  - occur instantaneously,
  - that is, in time, but taking no time, and to therefore be
    - **discrete action**$_\delta$s and
    - **discrete event**$_\delta$s.

5.1. Formal Concept Analysis: Discrete Perdurants

- The domain analyser examines collections of discrete perdurants.
  - In doing so the domain analyser discovers and thus identifies and lists a number of perdurant properties.
  - Each of the discrete perdurants examined usually satisfies only a subset of these properties.
  - The domain analyser now groups discrete perdurant into collections
    - such that each collection have its discrete perdurants satisfy the same set of properties,
    - such that no two distinct collections are indexed, as it were, by the same set of properties, and
    - such that all discrete perdurants are put in some collection.
  - The domain analyser now
    - classify collections as actions, events or behaviours, and
    - assign signatures
  - to distinct collections.
- That is how we assign signatures to discrete perdurants.

5.2. Actions

- By a **function**$_\delta$ we understand a **mathematical concept**,
  - a thing
  - which when **applied** to a **value**, called its **argument**,
  - **yield**s a **value**, called its **result**.
- A **discrete action**$_\delta$ can be understood as
  - a **function**
  - **invoked** on a **state value**
  - and is one that potentially changes that value.
- Other terms for **action** are
  - **function invocation**$_\delta$ and
  - **function application**$_\delta$.

Example: 32 Transport Net and Container Vessel Actions.

- *Inserting* and *removing* hubs and links in a net are considered actions.
- *Setting* the traffic signals for a hub (which has such signals) is considered an action.
- *Loading* and *unloading* containers from or unto the top of a container stack are considered actions. ■

5.2.1. Abstraction: On Modelling Domain Actions

- We claim that we describe domain actions,
  - but we actually describe functions,
  - which are "somewhat far removed" from domains.
- So what are we actually claiming?
  - We are claiming that there is an interesting class of actions
  - and that they can all be abstracted into one, possibly non-deterministic function
  - whose properties are then claimed to "mimic" those of the actions in the interesting class.

5.2.2. Agents: An Aside on Actions

Think'st thou existence doth depend on time?
It doth; but actions are our epochs.
George Gordon Noel Byron,
Lord Byron (1788-1824) Manfred. Act II. Sc. 1.

- *"An action is*
  - *something an agent does*
  - *that was 'intentional under some description'"* [Davidson1980].
- That is, actions are performed by agents.
  - We shall not yet go into any deeper treatment of **agency** or **agents**. We shall do so later.
    - **Agents** will here, for simplicity, be considered **behaviours**,
    - and are treated later in this lecture.

- As to the relation between **intention** and **action**
  - we note that Davidson wrote:
    'intentional under some description'
  - and take that as our cue:
    - the agent follows a script,
    - that is, a behaviour description,
    - and invokes actions accordingly,
    - that is, follow, or honours that script.

5.2.3. Action Signatures

- By an **action signature** we understand a quadruple:
  - a function name,
  - a function definition set type expression,
  - a total or partial function designator ($\rightarrow$, respectively $\stackrel{\sim}{\rightarrow}$), and
  - a function image set type expression:
    fct_name: A $\rightarrow$ Σ ($\rightarrow$|$\stackrel{\sim}{\rightarrow}$) Σ [$\times$ R],

  where $(X \mid Y)$ means either X or Y, and $[Z]$ means that for some signatures there may be a Z component meaning that the action also has the effect of “leaving” a type Z value.

Example: 33 Action Signatures: Nets and Vessels.

insert_Hub: $N \rightarrow H \xrightarrow{\sim} N$;
remove_Hub: $N \rightarrow HI \xrightarrow{\sim} N$;
set_Hub_Signal: $N \rightarrow HI \xrightarrow{\sim} H\Sigma \xrightarrow{\sim} N$
load_Container: $V \rightarrow C \rightarrow StackId \xrightarrow{\sim} V$; and
unload_Container: $V \rightarrow StackId \xrightarrow{\sim} (V \times C)$. ■

5.2.4. Action Definitions

- There are a number of ways in which to characterise an action.
- One way is to characterise its underlying function by a pair of predicates:
  - precondition: a predicate over function arguments — which includes the state, and
  - postcondition: a predicate over function arguments, a proper argument state and the desired result state.
  - If the precondition holds, i.e., is **true**, then the arguments, including the argument state, forms a proper 'input' to the action.
  - If the postcondition holds, assuming that the precondition held, then the resulting state [and possibly a yielded, additional "result" (R)] is as they would be had the function been applied.

Example: 34 Transport Nets Actions.

- In Example 4 we gave an explicit example of an action:
  - ins_H: Items 37–37(d),
- while implicit references to net actions were made in the event predicates
  - link_dis, pre_link_dis: Items 38–39(c),
  - post_link_dis (Items 38–39(c)):
    - rem_L Item 42(a) and
    - ins_L Items 42((c))i–42((c))ii. ■

- What is not expressed, but tacitly assume in the above pre- and post-conditions is
  - that the state, here n, satisfy invariant criteria before (i.e. n) and after (i.e., n') actions,
  - whether these be implied by axioms
  - or by well-formedness predicates.

  over parts.
- This remark applies to any definition of actions, events and behaviours.
- There are other ways of defining functions.
- But the form of these are not germane to the aims of this seminar.

Modelling Actions, I/III

- We refer to the section on Formal Concept Analysis of Discrete Perdurants on Slide 221.
- The **domain describer** has decided that an **entity** is a **perdurant** and is, or represents an **action**: was *"done by an agent and intentionally under some description"* [Davidson1980].
  - ◈ The domain describer has further decided that the observed action is of a class of actions — of the "same kind" — that need be described.
  - ◈ By actions of the 'same kind' is meant that these can be described by the same **function signature** and **function definition**.

Modelling Actions, II/III

- The domain describer must decide on the underlying **function signature**.
  - The **argument type** and the **result type** of the signature are those of either previously identified
    - parts and/or materials,
    - unique part identifiers, and/or
    - attributes.

Modelling Actions, III/III

- Sooner or later the domain describer must decide on the **function definition**.
  - The form must be decided upon.
  - For pre/post-condition forms it appears to be convenient to have developed, "on the side", a **theory of mereology** for the part types involved in the function signature.

5.3. Events

- By an **event**$_\delta$ we understand
  - *a state change*
  - *resulting indirectly from an unexpected application of a function,*
  - *that is, that function was performed "surreptitiously".*
- Events can be characterised by a pair of (before and after) states, a predicate over these and, optionally, a **time** or **time interval**.
- Events are thus like actions:
  - change states,
  - but are usually
    - either caused by "previous" actions,
    - or caused by "an outside action".

Example: 35 Events.

- *Container vessel:* A container falls overboard sometimes between times t and t'.
- *Financial service industry:* A bank goes bankrupt sometimes between times t and t'.
- *Health care:* A patient dies sometimes between times t and t'.
- *Pipeline system:* A pipe breaks sometimes between times t and t'.
- *Transportation:* A link "disappears" sometimes between times t and t'.

5.3.1. An Aside on Events

- We may observe an event, and
  - then we do so at a specific time or
  - during a specific time interval.
- But we wish to describe,
  - not a specific event
  - but a class of events of “the same kind”.
- In this seminar
  - we therefore do not ascribe
  - **time points** or **time intervals**
  - with the occurrences of events.

5.3.2. Event Signatures

- An event signature$_\delta$
  - *is a predicate signature*
  - *having an event name (evt),*
  - *a pair of state types ($\Sigma \times \Sigma$),*
  - *a total function space operator ($\rightarrow$)*
  - *and a* **Bool***ean type constant:*
  - *evt: ($\Sigma\times\Sigma$) $\rightarrow$ **Bool**.*
- Sometimes there may be a good reason
  - for indicating the type, ET, of an event cause value,
  - if such a value can be identified:
  - evt: ET $\times$ ($\Sigma \times \Sigma$) $\rightarrow$ **Bool**.

5.3.3. Event Definitions

- An **event definition**$_\delta$ takes the form of
  - *a predicate definition:*
    - *a predicate name and argument list, usually just a state pair,*
    - *an existential quantification*
      - *over some part (of the state) or*
      - *over some dynamic attribute of some part (of the state)*
      - *or combinations of the above*
    - *a pre-condition expression over the input argument(s),*
    - *an implication symbol ($\Rightarrow$), and*
    - *a post-condition expression over the argument(s):*
  - *evt(σ,σ') $=$ $\exists$ (ev:ET) $\bullet$ pre_evt(ev)(σ) $\Rightarrow$ post_evt(ev)(σ,σ').*

There may be variations to the above form.

Example: 36 Road Transport System Event.

- Example 4,
  - Items 38–42((c))ii
  - (Slides 85–88)

  exemplified an event definition.

Modelling Events I/II

- We refer to the section on **Formal Concept Analysis of Discrete Perdurants** on Slide 221.
- The **domain describer** has decided that an **entity** is a **perdurant** and is, or represents an **event**: occurred surreptitiously, that is, was not an action that was *"done by an agent and intentionally under some description"* [Davidson1980].
  - The domain describer has further decided that the observed event is of a class of events — of the "same kind" — that need be described.
  - By events of the 'same kind' is meant that these can be described by the same **predicate function signature** and **predicate function definition**.

Modelling Events, II/II

- First the domain describer must decide on the underlying **predicate function signature**.
  - The **argument type** and the **result type** of the signature are those of either previously identified
    - parts,
    - unique part identifiers, or
    - attributes.
- Sooner or later the domain describer must decide on the **predicate function definition**.
  - For predicate function definitions it appears to be convenient to have developed, "on the side", a **theory of mereology** for the part types involved in the function signature.

5.4. Discrete Behaviours

- We shall distinguish between
  - **discrete behaviours** (this section) and
  - **continuous behaviours**.
- Roughly **discrete behaviours**
  - proceed in discrete (time) steps —
  - where, in this lecture, 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 analysis**;
  - to qualify as a **continuous behaviour time** must be an essential aspect of the **behaviour**.
- **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 seminar we shall use `RSL/CSP`.

5.4.1. 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$_\delta$ we understand
    - a set of sequences of
    - actions, events and behaviours.

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

- 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
    - `MSC`s, or
    - `Statechart`s, or other.

5.4.2. 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 was done in Example 4,
    - 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.

5.4.3. Channels

- We remind the listener that we are focusing exclusively on domain behaviours.
  - Domain behaviours, as we shall see in Sect. 5.4.6, take their "root" in **parts**.
  - We shall find, even when "parts" take the form of concepts, that these do not "overlap".
    - They may share properties,
    - but we can consider them "disjoint".
  - Hence communication between processes
    - can be thought of as communication between "disjoint parts",
    - and, as such, can be abstracted as taking place
    - in a non-physical medium which we shall refer to as **channels**.

- By a **channel**$_\delta$ we shall understand
  - *a means of communicating entities*
  - *between [two] behaviours.*
- To express channel communications we, at present, make use of RSL [RSL]'s **out**put (ch ! v) / **in**put (ch ?) clauses and **channel** declarations,

type M
channel ch M,
value ch!v, ch?,

- Variations of the above clauses are

type ChIdx, ChJdx
channel $\{$ch[i]$|$i:ChIdx$\cdot\mathcal{P}$(i,...)$\}$:M, $\{$ch[i,j]$|$i:ChIdx,j:ChJdx$\cdot\mathcal{P}$(i,j,...)$\}$:M
value ch[i]!v, ch[i]?, ch[i,j]!v, ch[i,j]?

- where $\mathcal{P}$ is a suitable predicate
  - over channel indices and
  - possibly global domain values.

5.4.4. Behaviour Signatures

- By a behaviour signature$_\delta$ we shall understand *a*
  - *a function signature*
  - *augmented by a clause which declares*
    - *the* **in** *channels on which the function accepts inputs and*
    - *the* **out** *channels on which the function offers output.*

value behaviour: A → **in** in_chs **out** out_chs B

- where (i)
  - the form **in** in_chs **out** out_chs
    - may be just **in** in_chs
    - or **out** out_chs
    - or both **in** in_chs **out** out_chs

  that is, behaviour accepts input(s), or offers output(s), or both;

value behaviour: A $\rightarrow$ **in** in_chs **out** out_chs B

- where (ii)
  - A typically is of the forms
    - **Unit** if the behaviour "takes no arguments",
      - that is: behaviour(),

      or
    - PI×P if the behavior is directly based on a part, p:P, for
      - that is: behaviour(uid_P(p),p);

value behaviour: A → **in** in_chs **out** out_chs B

- where (iii)
- in_chs and out_chs are of the form
  - either ch,
  - or {ch[i]|i:ChIdx·$\mathcal{Q}$(i,...)}
  - or {ch[i,j]|i:ChIdx,j:ChJdx·$\mathcal{R}$(i,j,...)},

  $\mathcal{Q}$, $\mathcal{R}$ are appropriate predicates; and
- where (iv)
  - either
  - B is
    * either just **Unit** when the behaviour is typically a never-ending (i.e., cyclic) behaviours,
    * or is some result type C.

5.4.5. Behaviour Definitions

- This section is about the basic form of **behaviour function definitions**.
  - We shall only be concerned with behaviours which define **part behaviours**.
  - By a **part behaviour**$_\delta$ we shall understand
    - *a behaviour whose state*
    - *is that of the part for which it is the behaviour.*
- There are basically two cases for which we are interested in the form of the behaviour definition:
  - the **atomic part behaviour**, and
  - the **composite part behaviour**.

5.4.5.1 Atomic Part Behaviours

- Let p:P be an atomic part of type P.
- Then the basic form of a cyclic atomic behaviour definition is

value
  atomic_core_part_behaviour(uid_P(p))(p) ≡
    let p′ = $\mathcal{A}$(uid_P(p))(p) **in**
    atomic_core_part_behaviour(uid_P(p))(p′) **end**
    post: uid_P(p) = uid_P(p′),

  $\mathcal{A}$: PI → P → **in** ... **out** ... P,

- where $\mathcal{A}$ usually is a terminating function
  - which synchronises and
  - communicates with other part behaviours.

Example: 37 Atomic Part Behaviours.

- Example 4, Sect. 2.8.6 and Sect. 2.8.7 illustrates cyclic atomic behaviours:
  - vehicle at Hub: Items 65–65(d), on Slide 101,
  - vehicle on Link: Items 64–68, on Slide 103 and
  - monitor: Items 69–71(d), on Slide 105.

■

5.4.5.2 Composite Part Behaviours

- Let p:P be an **atomic part** of type P.
- Then the basic form of a cyclic **atomic behaviour definition** is

value

composite_part_behaviour(uid_P(p))(p) ≡
  composite_core_part_behaviour(uid_P(p))(p)
  ∥ { part_behaviour(uid_P(p′))(p′)|p′:P•p′ ∈ **obs_**(p)}

core_part_behaviour: PI → P → **in** ... **out** ... **Unit**
core_part_behaviour(uid_P(p))(p) ≡
  let p′ = $\mathcal{C}$(uid_P(p))(p) **in**
  composite_core_part_behaviour(uid_P(p))(p′) **end**
  post: uid_P(p) = uid_P(p′)

$\mathcal{C}$: PI → P → **in** ... **out** ... P,

- where $\mathcal{C}$ usually is a terminating function
  - which synchronises and
  - communicates with other **part behaviours**.

Example: 38 Compositional Behaviours.

- Example 4, Sect. 2.8.3
  - illustrated compositionality,
  - cf. Items 59– 59(b) on Slide 95. ■

- The next section
  - illustrates the basic principles
  - that we recommend
  - when modelling behaviours of domains
  - consisting of composite and atomic parts.

5.4.6. A Model of Parts and Behaviours

- How often have you not "confused", linguistically,
  - 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'.

- The example additionally has a more general aim,
  - *namely that of showing*
  - *that to every mereology (or parts)*
  - *there is a λ-expression*
  - *here in the form of basically a* `CSP [Hoare85+2004]` *program.*

Example: 39 Syntax and Semantics of Mereology.

5.4.6.1 A Syntactic Model of Parts

106. The *whole* contains a set of *parts*.
107. *Parts* are either *atomic* or *composite*.
108. From *composite parts* one can observe a set of *parts*.
109. All *parts* have *unique identifiers*

type

106. W, P, A, C

107. P = A | C

value

108. **obs**_Ps: (W|C) → P-**set**

type

109. PI

value

109. **uid**_Π: P → Π

110. From a *whole* and from any *part* of that *whole* we can e**xtr**act all contained *part*s.
111. Similarly one can e**xtr**act the *unique identifier*s of all those contained *parts*.
112. Each part may have a *mereology* which may be “empty”.
113. A *mereology*'s *unique part identifier*s must refer to some other parts other than the part itself.

value

110. xtr_Ps: $(W|P) \rightarrow P$**-set**

110. xtr_Ps(w) $\equiv$ {xtr_Ps(p)|p:P·p $\in$ **obs**_Ps(p)}

110.   **pre**: is_W(p)

110. xtr_Ps(p) $\equiv$ {xtr_Ps(p)|p:C·p$\in$ **obs**_Ps(p)}$\cup${p}

110.   **pre**: is_P(p)

111. xtr_Πs: $(W|P) \rightarrow \Pi$**-set**

111. xtr_Πs(wop) $\equiv$ {**uid**_P(p)|p $\in$ xtr_Ps(wop)}

112. **mereo**_P: $P \rightarrow \Pi$**-set**

axiom

113. $\forall$ w:W

113.   **let** ps = xtr_Ps(w) **in**

113. $\forall$ p:P · p $\in$ ps · $\forall$ π:Π · π $\in$ **mereo**_P(p) $\Rightarrow$ π $\in$ xtr_Πs(p) **end**

114. An **attribute map** of a *part* associates with *attribute names*, i.e., *type names*, their *values*, whatever they are.
115. From a *part* one can extract its attribute map.
116. Two *parts share attributes* if their respective **attribute maps** share *attribute names*.
117. 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
114. AttrNm, AttrVAL,
114. AttrMap = AttrNm $\overrightarrow{m}$ AttrVAL
value
115. **attr_**AttrMap: P → AttrMap
116. share_Attributes: P×P → **Bool**
116. share_Attributes(p,p′) ≡
116. **dom** **attr_**AttrMap(p) ∩
116. **dom** **attr_**AttrMap(p′) ≠ {}
117. share_Properties: P×P → **Bool**
117. share_Properties(p,p′) ≡
117(a). share_Attributes(p,p′)
117(b). ∨ **uid_**P(p) ∈ **mereo_**P(p′)
117(b). ∨ **uid_**P(p′) ∈ **mereo_**P(p)

5.4.6.2 A Semantics Model of Parts

118. 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.
119. Let there be given a 'whole', w:W.
120. 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
118. K = Π-**set axiom** ∀ k:K•**card** k=2
value
118. xtr_Ks: (W|P) → K-**set**
118. xtr_Ks(wop) ≡
118. **let** ps = xtr_Ps(w) **in**
118. {{**uid_**P(p),π}|p:P,π:Π•p∈ ps ∧ ∃ p′:P•p′≠p∧π=**uid_**P(p′) ∧ **uid_**P(p)∈uid_P(p′)} **end**

119. w:W

120. **channel** {ch[k]|k:xtr_Ks(w)}:M

121. Now the 'whole' *behaviour* whole is the parallel composition of *part process*es, one for each of the immediate parts of the *whole*.
122. 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*.

121. whole: W → **Unit**
121. whole(w) ≡ ∥ {part(**uid**_P(p))(p) | p:P·p ∈ xtr_Ps(w)}

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

123. 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.

.

value

123. comp: π:Π $\rightarrow$ p:P $\rightarrow$ **in**,**out** {ch[{π,π'}|{$\pi'$$\in$ **mereo**_P(p)}]} **Unit**

123. comp(π)(p) $\equiv$

123(a). comp_core(π)(p) $\|$

123(b). $\|$ {part(**uid**_P(p′))(p′) | p′:P•p′ $\in$ **obs**_Ps(p)}

124. An *atomic process* consists of just an *atomic core process*, *atom_core*

124. $\text{atom: } \pi{:}\Pi \rightarrow \text{p:P} \rightarrow$ **in**,**out** $\{\text{ch}[\{\pi,\pi'\}] \mid \{\pi' \in$ **mereo**_P(p)$\}\}$ **Unit**

124. $\text{atom}(\pi)(\text{p}) \equiv \text{atom\_core}(\pi)(\text{p})$

125. 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

125. core: π:Π $\rightarrow$ p:P $\rightarrow$ **in**,**out** {ch[{π,π'}|{$\pi'$$\in$ **mereo**_P(p)}]} **Unit**

125. core(π)(p) $\equiv$

125(a). **let** p′ = update(π)(p)

125(b). **in** core(π)(p′) **end**

125(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.

6. Continuous Entities

- There are two kinds of **continuous entities**:
  - **material**s (Slides 278–299) and
  - **continuous behaviours** (Slides 300–314).
- By a **material**$_\delta$ we small mean
  - a **continuous endurant**,
  - a **manifest entity** which typically varies in shape and extent.
- By a **continuous behaviour**$_\delta$ we small mean
  - a **continuous perdurant**,
  - which we may think of as a **function**
    - from continuous $\mathbb{T}$ime
    - to some structure, simple or complicated, of
      - **part**s and
      - **material**s.

6.1. Materials

- Let us start with examples of materials.

Example: 40 Materials. Examples of endurant continuous entities are such as

- coal,
- air,
- natural gas,
- grain,
- sand,
- iron ore,
- minerals,
- crude oil,
- solid waste,
- sewage,
- steam and
- water. ■

The above **materials** are either

- **liquid materials** (crude oil, sewage, water),
- **gaseous materials** (air, gas, steam), or
- **granular materials** (coal, grain, sand, iron ore, mineral, or solid waste).

- **Endurant continuous entities**, or **materials** as we shall call them,
  - ⟐ are the **core endurants** of process domains,
  - ⟐ that is, **domains** in which those **materials** *form the basis* for their *"raison d'être"*.

6.1.1. Materials-based Domains

- By a **materials based domain**$_\delta$ we shall mean a **domain**
  - ⟐ *many of whose parts serve to transport materials, and*
  - ⟐ *some of whose actions, events and behaviours serve to monitor and control the part transport of materials.*

Example: 41 Material Processing.

- Oil or gas materials are ubiquitous to pipeline systems — so pipeline systems are oil or gas-based systems.
- Sewage is ubiquitous to waste management systems — so waste management systems are sewage-based systems.
- Water is ubiquitous to systems composed from reservoirs, tunnels and aqueducts which again are ubiquitous to hydro-electric power plants, irrigation systems or water supply utilities — so hydro-electric power plants, irrigation systems and water supply utilities are water-based systems. ■

- Ubiquitous means 'everywhere'.
- A continuous entity, that is, a material
  - is a core material,
  - if it is "somehow related"
  - to one or more parts of a domain.

6.1.2. "Somehow Related" Parts and Materials

- We explain our use of the term "somehow related".

Example: 42 Somehow Related Materials and Parts. With `teletype font` we designate materials and with *slanted font* we imply parts or part processes.

- `Oil` is pumped from *well*s, runs through *pipe*s, is "lifted" by *pump*s, diverted by *fork*s, "runs together" by means of *join*s, and is delivered to *sink*s.
- `Grain` is delivered to silos by trucks, piped through a network of pipes, forks and valves to vessels, etc.
- `Mineral`s are *mined*, *conveyed* by *belt*s to *lorries* or *trains* or *cargo vessels* and finally *deposited*.
- `Iron ore`, for example, is *'conveyed'* into *smelters*, *'roasted'*, *'reduced'* and *'fluxed'*, *'mixed'* with other mineral ores to produce a molten, pure metal, which is then *'collected'* into *ingots*. ■

6.1.3. Material Observers

- When **analysing domains** a key question,
  - ⟐ in view of the above notion of **core continuous endurants** (i.e., materials)

  is therefore:
  - ⟐ does the **domain** embody a notion of **core continuous endurants** (i.e., materials);
  - ⟐ if so, then identify these "early on" in the **domain analysis**.
- Identifying materials —
  - ⟐ their types and
  - ⟐ attributes —

  is slightly different from identifying **discrete endurants**, i.e., **parts**.

Example: 43 Pipelines: Core Continuous Endurant. We continue Examples 30 on Slide 209 and 31 on Slide 211.

- The core continuous endurant, i.e., material,
- of (say oil) pipelines is, yes, oil:

type
  O **material**
value
  obs_O: PLN → O

- The keyword **material** is a pragmatic. ■

- Materials are "few and far between" as compared to parts,
  - we choose to mark the type definitions which designate materials with the keyword **material**.
  - In contrast, we do not mark the type definitions which designate parts with the keyword **discrete**.

- First we do not associate the notion of atomicity or composition with a material. Materials are continuous.
- Second, amongst the attributes, none have to do with geographic (or cadestral) matters. Materials are moved.
- And materials have no unique identification or mereology. No "part" of a material distinguishes it from other "parts".
- But they do have other attributes when occurring in connection with, that is, related to **parts**, for example,
  - volume or
  - weight.

Example: 44 Pipelines: Parts and Materials. We continue Examples 30 on Slide 209 and 31 on Slide 211.

126. From an oil pipeline system one can, amongst others,
    (a) observe the finite set of all its pipeline bodies,
    (b) units are composite and consists of a unit,
    (c) and the oil, even if presently, at time of observation, empty of oil.
127. Whether the pipeline is an oil or a gas pipeline is an attribute of the pipeline system.
    (a) The volume of material that can be contained in a unit is an attribute of that unit.
    (b) There is an auxiliary function which estimates the volume of a given "amount" of oil.
    (c) The observed oil of a unit must be less than or equal to the volume that can be contained by the unit.

type
126. PLS, B, U, Vol
126. O **material**
value
126(a). **obs_**Bs: PLS → B-**set**
126(b). **obs_**U: B → U
126(c). **obs_**O: B → O
127. **attr_**PLS_Type: PLS → {"oil"|"gas"}
127(a). **attr_**Vol: U → Vol
127(b). vol: O → Vol
axiom
127(c). ∀ pls:PLS,b:B•b ∈ **obs_**Bs(pls)⇒vol(**obs_**O(b))≤**attr_**Vol(**obs_**U(b))

- Notice how bodies are composite and consists of
  - a discrete, atomic part, the unit, and
  - a material endurant, the oil.
- We refer to Example 45 on Slide 291.

6.1.4. Material Properties

- These are some of the key concerns in domains focused on materials:
  - transport, flows, leaks and losses, and
  - input to systems and output from systems,
- Other concerns are in the direction of
  - dynamic behaviours of materials focused domains (mining and production), including
  - stability, periodicity, bifurcation and ergodicity.
- In this seminar we shall, when dealing with systems focused on materials, concentrate on modelling techniques for
  - transport, flows, leaks and losses, and
  - input to systems and output from systems.

- **Formal specification languages** like
  - `Alloy [alloy]`,
  - `Event B [JRAbrial:TheBBooks]` ,
  - `CASL [CoFI:2004:CASL-RM]`
  - `CafeOBJ [futatsugi2000a]`,
  - `RAISE [RaiseMethod]`,
  - `VDM [e:db:Bj78bwo,e:db:Bj82b,JohnFitzge` and
  - `Z [m:z:jd+jcppw96]`

  do not embody the mathematical calculus notions of
  - continuity, hence do not "exhibit"
  - neither differential equations
  - nor integrals.
- Hence cannot formalise **dynamic systems** within these **formal specification languages**.
- We refer to Sect. 9.3.1 where we discuss these issues at some length.

Example: 45 Pipelines: Parts and Material Properties. We refer to Examples 30 on Slide 209, 31 on Slide 211 and 44 on Slide 287.

128. Properties of pipeline units additionally include such which are concerned with flows (F) and leaks (L) of materials:
    (a) current flow of material into a unit input connector,
    (b) maximum flow of material into a unit input connector while maintaining laminar flow,
    (c) current flow of material out of a unit output connector,
    (d) maximum flow of material out of a unit output connector while maintaining laminar flow,
    (e) current leak of material at a unit input connector,
    (f) maximum guaranteed leak of material at a unit input connector,
    (g) current leak of material at a unit input connector,
    (h) maximum guaranteed leak of material at a unit input connector,
    (i) current leak of material from "within" a unit,
    (j) maximum guaranteed leak of material from "within" a unit.

129. There are "the usual" arithmetic and comparison operators of flows and leaks, and there is a smallest detectable (flow and) leak.

type
129. F, L

value
129. $\oplus, \ominus$: (F|L)×(F|L) → (F|L)
129. $<, \leq, =$: (F|L)×(F|L) → **Bool**
129. $\otimes$: (F|L)×**Real** → (F|L)
129. /: (F|L)×(F|L) → **Real**
129. ℓ_0:L

128(a). **attr**_cur_iF: U → UI → F
128(b). **attr**_max_iF: U → UI → F
128(c). **attr**_cur_oF: U → UI → F
128(d). **attr**_max_oF: U → UI → F
128(e). **attr**_cur_iL: U → UI → L
128(f). **attr**_max_iL: U → UI → L
128(g). **attr**_cur_oL: U → UI → L
128(h). **attr**_max_oL: U → UI → L
128(i). **attr**_cur_L: U → L
128(j). **attr**_max_L: U → L

- The maximum flow attributes are static attributes and are typically provided by the manufacturer as indicators of flows below which laminar flow can be expected.
- The current flow attributes as dynamic attributes.

130. Properties of pipeline materials may additionally include

  (a) kind of material[18],
  (b) paraffins,
  (c) naphtenes,
  (d) aromatics,
  (e) asphatics,
  (f) viscosity,
  (g) etcetera.

- We leave it to the student to provide the formalisations.

[18] For example **Brent Blend** Crude Oil

6.1.5. Material Laws of Flows and Leaks

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

Example: 46 Pipelines: Intra Unit Flow and Leak Law. We continue our line of Pipeline System examples (cf. the opening line of Example 45 on Slide 291).

131. For every unit of a pipeline system, except the well and the sink units, the following law apply.
132. The flows into a unit equal
    (a) the leak at the inputs
    (b) plus the leak within the unit
    (c) plus the flows out of the unit
    (d) plus the leaks at the outputs.

axiom

131. $\forall$ pls:PLS,b:B\We\Si,u:U ·
131. b $\in$ **obs_**Bs(pls)$\wedge$u=**obs_**U(b)$\Rightarrow$
131. **let** (iuis,ouis) = **mereo_**U(u) **in**
132. sum_cur_iF(iuis)(u) =
132(a). sum_cur_iL(iuis)(u)
132(b). $\oplus$ **attr_**cur_L(u)
132(c). $\oplus$ sum_cur_oF(ouis)(u)
132(d). $\oplus$ sum_cur_oL(ouis)(u)
131. **end**

133. The sum_cur_iF (cf. Item 132) sums current input flows over all input connectors.
134. The sum_cur_iL (cf. Item 132(a)) sums current input leaks over all input connectors.

135. The sum_cur_oF (cf. Item 132(c)) sums current output flows over all output connectors.

136. The sum_cur_oL (cf. Item 132(d)) sums current output leaks over all output connectors.

133. sum_cur_iF: UI-**set** → U → F
133. sum_cur_iF(iuis)(u) ≡ ⊕ ⟨**attr_**cur_iF(ui)(u)|ui:UI•ui ∈ iuis⟩
134. sum_cur_iL: UI-**set** → U → L
134. sum_cur_iL(iuis)(u) ≡ ⊕ ⟨**attr_**cur_iL(ui)(u)|ui:UI•ui ∈ iuis⟩
135. sum_cur_oF: UI-**set** → U → F
135. sum_cur_oF(ouis)(u) ≡ ⊕ ⟨**attr_**cur_iF(ui)(u)|ui:UI•ui ∈ ouis⟩
136. sum_cur_oL: UI-**set** → U → L
136. sum_cur_oL(ouis)(u) ≡ ⊕ ⟨**attr_**cur_iL(ui)(u)|ui:UI•ui ∈ ouis⟩
⊕: (F×F)|F* → F | (L×L)|L* → L

- where ⊕ is both an infix and a distributed-fix function which adds flows and or leaks.

Example: 47 Pipelines: Inter Unit Flow and Leak Law.

137. For every pair of connected units of a pipeline system the following law apply:
    (a) the flow out of a unit directed at another unit minus the leak at that output connector
    (b) equals the flow into that other unit at the connector from the given unit plus the leak at that connector.

137. ∀ pls:PLS,b,b′:B,u,u′:U·
137. {b,b′}⊆**obs**_Bs(pls)∧b≠b′∧u′=**obs**_U(b′)
137. ∧ **let** (iuis,ouis)=**mereo**_U(u),(iuis′,ouis′)=**mereo**_U(u′),
137. ui=**uid**_U(u),ui′=**uid**_U(u′) **in**
137. ui ∈ iuis ∧ ui′ ∈ ouis′ ⇒
137(a). **attr**_cur_oF(us′)(ui′) ⊖ **attr**_leak_oF(us′)(ui′)
137(b). = **attr**_cur_iF(us)(ui) ⊕ **attr**_leak_iF(us)(ui)
137. **end**
137. **comment:** b′ precedes b

- From the above two laws one can prove the **theorem:**
  - what is pumped from the wells equals
  - what is leaked from the systems plus what is output to the sinks.
- We need formalising the flow and leak summation functions.

6.2. Continuous Behaviours

- This section is still under research and development.
- The aim of this section is to relate
  - discrete behaviour domain models of some fragments of a domain
  - to continuous behaviour domain models of other fragments of that domain.
- By a continuous behaviour model$_\delta$ we mean
  - *a domain description that emphasises*
  - *the behaviour of materials, that is,*
  - *how they flow through parts, and related matters.*

6.2.1. Fluid Dynamics

- Continuous behaviour domain models classically express
  - the **fluid dynamics**$_\delta$
    - of **flows of fluids**,
    - that is, the **natural science** of
    - **liquids** and **gasses**.

- The natural science of fluids
  - (from `Wikipedia:`)
    - *“are based on foundational axioms of fluid dynamics*
    - *which are the conservation laws,*
    - *specifically, conservation of mass,*
    - *conservation of linear momentum*
    - *(also known as Newton’s Second Law of Motion),*
    - *and conservation of energy*
    - *(also known as First Law of Thermodynamics).*
    - *These are based on classical mechanics.*
    - *They are expressed using the Reynolds Transport Theorem.”*

6.2.1.1 Descriptions of Continuous Domain Behaviours

- We are not going to exemplify such descriptive natural science models.
- Their mathematics, besides being elegant and beautiful,
  - includes familiarity with
  - Bernoulli Equations,
  - Navier Stokes Equations, etc.
- For continuous behaviour domain models
  - we shall refer to such mathematical models
  - of the natural science of fluids.

6.2.1.2 Prescriptions of Required Continuous Domain Behaviours

- By a prescriptive domain model$_\delta$ we mean
  - *a desirable behaviour specification*
  - *as in, for example, a requirements prescription*
  - of a continuous time dynamic system.
- We are also not going to illustrate prescriptive domain models.
  - Their mathematics, besides also being elegant and beautiful,
    - is based on the descriptive natural science models;
    - but are now part of the engineering realm of *Control Theory*.
    - It includes such disciplines as
      - fuzzy control [Michel-etal-2010],
      - stochastic control [Karlin+Taylor1998] and
      - adaptive control [aastroem89], etc.

Example: 48 Pipelines: Fluid Dynamics and Automatic Control.

- We refer to Example 49 on Slide 307.
- In that example, next, we expect domain models
  - for the fluid dynamics of individual pipeline units: wells, pumps, pipes, valves, forks, joins and sinks,
  - as well as models (one or more) for sequences of such units,
  - extending, preferably to entire nets: from wells to sinks.
- And we expect **requirements description models**
  - again for each of some of the individual units:
    - pumps and valves in particular:
    - when they need and how they are **controlled**:
    - regulating pumps and valves and
    - which unit **attributes** need be **monitored**. ■

6.2.2. A Pipeline System Behaviour

- We shall model the behaviours of a composite pipeline system.
  - We shall be using basically the same form of the description as first illustrated in Sects. 2.8.2–2.8.7 (Slides 94–105) of Example 4.
  - That system, Sects. 2.8.2–2.8.7, can be interpreted as illustrating the central monitoring of vehicles spread over a wide geographical area.
  - The system to be illustrated in Example 49 can likewise be interpreted as illustrating the central monitoring of pipeline units (and their oil) spread over a wide geographical area.

Example: 49 A Pipeline System Behaviour.

- We consider (cf. Examples 30 on Slide 209 and 31 on Slide 211) the pipeline system units to represent also the following behaviours:
  - pls:PLS, Item 126(a) on Slide 287, to also represent the system process, pipeline_system, and for each kind of unit, cf. Example 30, there are the unit processes:
    - unit,
    - well (Item 98(c) on Slide 209),
    - pipe (Item 98(a)),
    - pump (Item 98(a)),
    - valve (Item 98(a)),
    - fork (Item 98(b)),
    - join (Item 98(b)) and
    - sink (Item 98(d) on Slide 209).


```
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) ≡
98(c).    is_We(u) → well(uid_U(u))(u),
98(a).    is_Pu(u) → pump(uid_U(u))(u),
98(a).    is_Pi(u) → pipe(uid_U(u))(u),
98(a).    is_Va(u) → valve(uid_U(u))(u),
98(b).    is_Fo(u) → fork(uid_U(u))(u),
98(b).    is_Jo(u) → join(uid_U(u))(u),
98(d).    is_Si(u) → sink(uid_U(u))(u)
```

- We illustrate essentials of just one of these behaviours.

98(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**
98(b). fork(ui)(u) ≡
98(b). **let** u′ = core_fork_behaviour(ui)(u) **in**
98(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_· · · _behaviour[s](ui)(u) also communicate with the pipeline_system behaviour.
  - What we have in mind here is to model a traditional supervisory control and data acquisition, SCADA system.

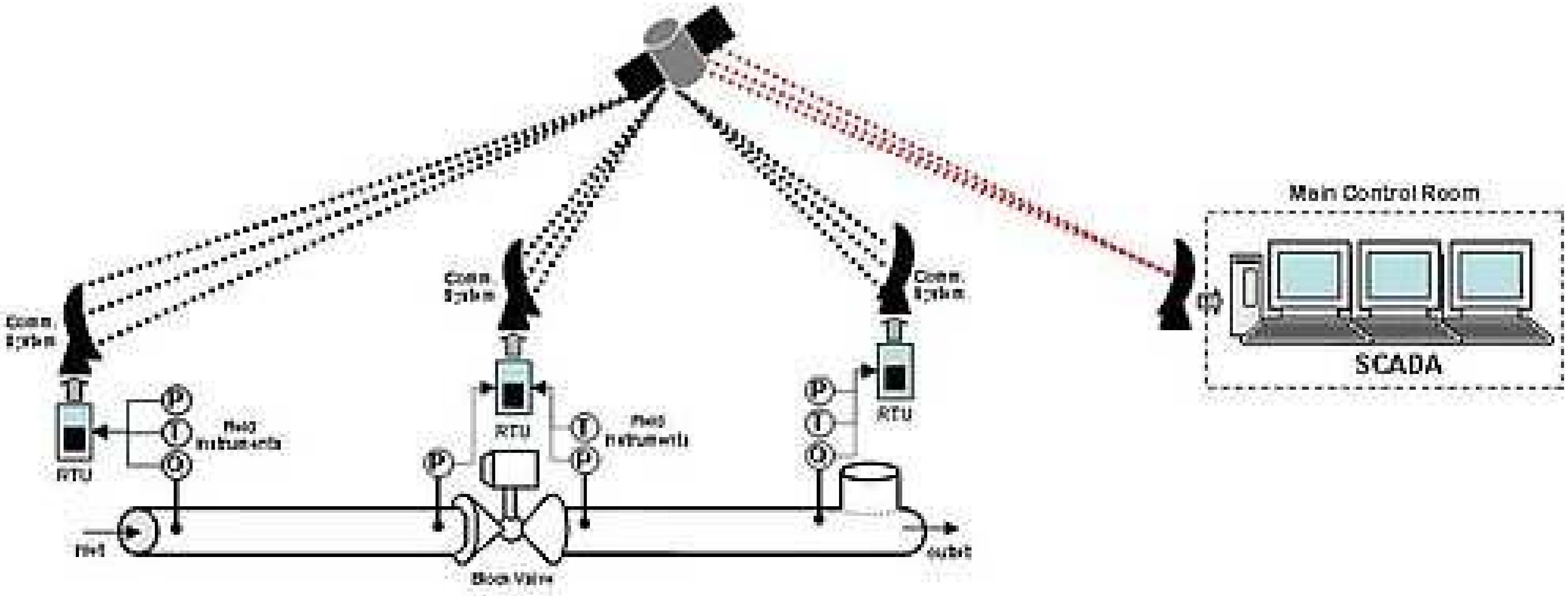

Figure 2: A supervisory control and data acquisition system

138. SCADA is then part of the scada_pipeline_system behaviour.

138. scada_pipeline_system: PLS →
138. **in**,**out** { pls_u_ch[ui]:ui:UI·i ∈ UIs(pls) } **Unit**
138. scada_pipeline_system(pls) ≡
138. scada(props(pls)) ∥ pipeline_system(pls)

- props was defined on Slide 204.

- We refer to Example 48 on Slide 305:
  - for all the core_· · · _behaviours
    - we expect the scada monitor
    - to be expressed in terms of a prescriptive domain model
    - which prescribes some optimal form of control of the pipeline net.

139. 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

139. Props

value

139. scada: Props $\rightarrow$ **in**,**out** { pls_ui_ch[ui] | ui:UI•ui $\in$ $\in$ uis } **Unit**

139. scada(props) $\equiv$

139(a). scada(scada_own_work(props))

139(b). $\lceil\rceil$ scada(scada_data_acqui_work(props))

139(c). $\lceil\rceil$ scada(scada_control_work(props))

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

140. The scada_data_acqui_work

 (a) non-deterministically, external choice, [], offers to accept data,

 (b) and scada_input_updates the scada state —

 (c) from any of the pipeline units.

value

```
140.    scada_data_acqui_work: Props → in,out { pls_ui_ch[ui] | ui:UI•ui ∈ ∈
140.    scada_data_acqui_work(props) ≡
140(a).      [] { let (ui,data) = pls_ui_ch[ui] ? in
140(b).           scada_input_update(ui,data)(props) end
140(c).           | ui:UI · ui ∈ uis }

140(b).  scada_input_update: UI × Data → Props → Props
```

type

```
140(a).  Data
```

141. 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.

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

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

See You in 30 Minutes — Thanks !