4. Lecture 4: Domain Descriptions — Perdurants

4.1. States

4.1.1. General

- The characterisation of the concept of perdurant
  - mentioned time,
  - but implied a concept that we shall call state.
- In this version of this seminar
  - we shall not cover the modelling of time phenomena —
  - but we shall model that some actions occur before others.

- By a **state** we shall understand a collection of parts
  - such that each of these parts have dynamic attributes.
- We can characterise the state
  - by giving it a type,
  - for example, Σ, where the **state type definition**
  - $\Sigma = \mathsf{S}_1 \times \mathsf{S}_2 \times \cdots \times \mathsf{S}_s$
  - assembles the types of the parts making up the state —
  - where we assume that types S_1, S_2, ..., S_s
    - are types of parts
    - such that no S_i is a sub-part (of a subpart, ...) of some S_j,
    - and such that each part has **dynamic attribute**s.

Example: 33 Net and Vessel States.

- We may consider a transport net, n:N, to represent a state (subject to the actions of maintaining a net: adding or removing a hub, adding or removing a link, etc.).
- We may also consider a hub, h:H, to represent a state (subject to the changing of a hub traffic signal: from red to green, etc., for specific directions through the hub).
- We may consider a container vessel to represent a state (subject to adding or removing containers from, respectively onto the top of stacks). ■

Thus the context determines how wide a scope the domain designer chooses for the state concept.

4.1.2. State Invariants

- States are subject to invariants.

Example: 34 State Invariants: Transport Nets.

- Net hubs and links may be inserted into and removed from nets.
- Thus is also introduced changes to the net mereology.
- Yet, the axioms, as illustrated in Example 26, must remain invariant.
- Likewise changes to dynamic attributes may well be subject to the holding of certain well-formedness constraints.
- We will illustrate this claim.

With each hub we associate a hub [link] state and a hub [link] state space.

79. A hub [link] state models the permissible routes from hub input links to (same) hub output links [respectively through a link].

80. A hub [link] state space models the possible set of hub [link] states that a hub [link] is intended to “occupy”.

type

79. $H\Sigma = (LI \times LI)$**-set**, $L\Sigma = HI$**-set**

80. $H\Omega = H\Sigma$**-set**, $L\Omega = L\Sigma$**-set**

value

79. attr_HΣ: $H \rightarrow H\Sigma$, attr_LΣ: $L \rightarrow L\Sigma$

80. attr_HΩ: $H \rightarrow H\Omega$, attr_LΩ: $L \rightarrow L\Omega$

81. For any given hub, h, with links, $l_1, l_2, ..., l_n$ incident upon (i.e., also emanating from) that hub, each hub state in the hub state space
82. must only contain such pairs of (not necessarily distinct) link identifiers that are identifiers of $l_1, l_2, ..., l_n$.

value

81. wf_HΩ: H → **Bool**
81. wf_HΩ(h) ≡ ∀ hσ:HΣ · hσ ∈ attr_HΩ(h) ⇒ wf_HΣ(h)

81. wf_HΣ: H → **Bool**
81. wf_HΣ(h) ≡
82. ∀ (li,li′):(LI×LI)·(li,li′)∈ attr_HΣ(h) ⇒ {li,li′} ⊆ mereo_H(h)

- This well-formedness criterion is part of the state invariant over nets.
  - We never write down the full state invariant for nets.
  - It is tacitly assume to be the collection of all the axioms and well-formedness predicates over net parts. ■

4.2. A Final Note on Endurant Properties

- The properties of parts and materials are fully captured by
  - (i) the unique part identifiers,
  - (ii) the part mereology and
  - (iii) the full set ofpart attributes and material attributes
- We therefore postulate a property function
  - when when applied to a part or a material
  - yield this triplet, (i–iii), of properties
  - in a suitable structure.

type

$\text{Props} = \{|\text{PI}|\textbf{nil}|\} \times \{|(\text{PI-}\textbf{set}\times\ldots\times\text{PI-}\textbf{set})|\textbf{nil}|\} \times \text{Attrs}$

value

$\text{props: Part}|\text{Material} \rightarrow \text{Props}$

- where
  - Part stands for a **part type**,
  - Material stands for a **material type**,
  - PI stand for **unique part identifiers** and
  - **PI-set**×...×**PI-set** for **part mereologies**.
- The {|...|} denotes a proper specification language sub-type and **nil** denotes the empty type.

5. Discrete Perdurants

5.1. General

- 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.*

- We shall consider **action**s and **event**s
  - to occur instantaneously,
  - that is, in time, but taking no time
- Therefore we shall consider **action**s and **event**s to be **perdurant**s.

5.2. Discrete Actions

- By a **function** we understand
  - a thing
  - which when **applied** to a **value**, called its **argument**,
  - **yields** a **value**, called its **result**.
- An **action** is
  - a **function**
  - **invoked** on a **state value**
  - and is one that potentially changes that value.

Example: 35 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. 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 optional Z.

Example: 36 Action Signatures: Nets and Vessels.

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

5.2.2. 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: 37 Transport Nets: Insert Hub Action.

83. The insert action applies to a net and a hub and conditionally yields an updated net.

   a The condition is that there must not be a hub in the "argument" net with the same unique hub identifier as that of the hub to be inserted and

   b the hub to be inserted does not initially designate links with which it is to be connected.

   c The updated net contains all the hubs of the initial net "plus" the new hub.

   d and the same links.

value

83. insert_H: N $\rightarrow$ H $\stackrel{\sim}{\rightarrow}$ N
83. insert_H(n)(h) **as** n′, **pre**: pre_insert_H(n)(h), **post**: post_insert_H(n)(h)

83a. pre_insert_H(n)(h) $\equiv$
83a. $\sim\exists$ h′:H $\cdot$ h′ $\in$ obs_Hs(n) $\wedge$ uid_HI(h)=uid_HI(h′)
83b. $\wedge$ mereo_H(h) = {}

83c. post_insert_H(n)(h)(n′) $\equiv$
83c. obs_Hs(n) $\cup$ {h} = obs_Hs(n′)
83d. $\wedge$ obs_Ls(n) = obs_Ls(n′)

- We refer to the notes accompanying these lectures.
- There you will find definitions of insert_link, remove_hub and remove_link action functions.

Modelling Actions, I/III

- 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

- First 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. Discrete Events

- By an **event** 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: 38 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. Event Signatures

- An event signature
  - is a predicate signature
  - having an event name,
  - 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**.

5.3.2. Event Definitions

- An event definition 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 (⇒), and
  - a post-condition expression over the argument(s).
- evt$(\sigma,\sigma') = \exists$ (ev:ET) • pre_evt(ev)$(\sigma) \Rightarrow$ post_evt(ev)(σ,σ').
- There may be variations to the above form.

Example: 39 Narrative of Link Event. The disappearance of a link in a net, for example due to a mud slide, or a bridge falling down, or a fire in a road tunnel, can, for example be described as follows:

84. Link disappearance is expressed as a predicate on the "before" and "after" states of the net. The predicate identifies the "missing" ℓink (!).

85. Before the disappearance of link ℓ in net n

   a the hubs h' and h'' connected to link ℓ

   b were connected to links identified by $\{l'_1, l'_2, \ldots, l'_p\}$ respectively $\{l''_1, l''_2, \ldots, l''_q\}$

   c where, for example, l'_i, l''_j are the same and equal to $\mathsf{uid\_}\Pi(\ell)$.

86. After link ℓ disappearance there are instead

a two separate links, ℓ_i and ℓ_j, "truncations" of ℓ

b and two new hubs h''' and h''''

c such that ℓ_i connects h' and h''' and

d ℓ_j connects h'' and h'''';

e Existing hubs h' and h'' now have mereology

i. $\{l'_1, l'_2, \ldots, l'_p\} \setminus \{\mathsf{uid\_}\Pi(\ell)\} \cup \{\mathsf{uid\_}\Pi(\ell_i)\}$ respectively

ii. $\{l''_1, l''_2, \ldots, l''_q\} \setminus \{\mathsf{uid\_}\Pi(\ell)\} \cup \{\mathsf{uid\_}\Pi(\ell_j)\}$

87. All other hubs and links of n are unaffected. ■

Example: 40 Formalisation of Link Event. Continuing Example 39 above:

84. link_disappearance: N × N → **Bool**
84. link_disappearance(n,n′) ≡
84. $\exists\ \ell$:L · pre_link_dis(n,ℓ) ⇒ post_link_dis(n,ℓ,n′)

85. pre_link_dis: N × L → **Bool**
85. pre_link_dis(n,ℓ) ≡ $\ell \in$ obs_Ls(n)

88. We shall "explain" *link disappearance* as the combined, instantaneous effect of

   a first a **remove link** "event" where the **removed link** connected **hubs** hi_j and hi_k;

   b then the **insertion** of two new, "fresh" **hubs**, h_α and h_β;

   c "followed" by the **insertion** of two new, "fresh" links $\mathsf{l}_{j\alpha}$ and $\mathsf{l}_{k\beta}$ such that

      i. $\mathsf{l}_{j\alpha}$ connects hi_j and h_α and

      ii. $\mathsf{l}_{k\beta}$ connects hi_k and $\mathsf{h}_{k\beta}$

value

88. $\text{post\_link\_dis}(n,\ell,n') \equiv$

88a. **let** n'' = $\text{remove\_L}(n)(\text{uid\_L}(\ell))$ **in**

88b. **let** h_α,h_β:H $\cdot$ $\{h_\alpha,h_\beta\} \cap \text{obs\_Hs}(n)=\{\}$ **in**

88b. **let** n''' = $\text{insert\_H}(n'')(h_\alpha)$ **in**

88b. **let** n'''' = $\text{insert\_H}(n''')(h_\beta)$ **in**

88c. **let** $l_{j\alpha},l_{k\beta}$:L $\cdot$ $\{l_{j\alpha},l_{k\beta}\} \cap \text{obs\_Ls}(n)=\{\}$ **in**

88(c)i. **let** n''''' = $\text{insert\_L}(n'''')(l_{j\alpha})$ **in**

88(c)ii. $n' = \text{insert\_L}(n''''')(l_{k\beta})$ **end end end end end end**

- We refer to the notes accompanying these lectures.
- There you will find definitions of insert_link, remove_hub and remove_link action functions. ■

Modelling Events I/II

- 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** (Sect. 12).
- Roughly **discrete behaviours**
  - proceed in discrete (time) steps —
  - where, in this seminar, we omit considerations of time.
  - Each step corresponds to an **action** or an **event** or a time interval between these.
  - **Action**s and **event**s 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 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** 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** ($\sqcap$) or as **external non-deterministic behaviours** ($[]$).

- 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.

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 will be done in Example **??**,
    - 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: 41 A Road Traffic System. We continue our long line of examples around transport nets. The present example interprets these as road nets.

5.4.2.1 Continuous Traffic

- For the road traffic system
  - perhaps the most significant example of a behaviour
  - is that of its traffic

89. the continuous time varying discrete positions of vehicles, vp:VP[19],
90. where time is taken as a dense set of points.

type

90. $c\mathbb{T}$

89. $cRTF = c\mathbb{T} \to (V \underset{m}{\to} VP)$

[19] For VP see Item 108a on Slide 243.

5.4.2.2 Discrete Traffic

- We shall model, not continuous time varying traffic, but

91. discrete time varying discrete positions of vehicles,
92. where time can be considered a set of linearly ordered points.

92. $\mathrm{d}\mathbb{T}$
91. $\mathrm{dRTF} = \mathrm{d}\mathbb{T} \underset{m}{\rightarrow} (\mathrm{V} \underset{m}{\rightarrow} \mathrm{VP})$

93. The road traffic that we shall model is, however, of vehicles referred to by their unique identifiers.

type
93. $\mathrm{RTF} = \mathrm{d}\mathbb{T} \underset{m}{\rightarrow} (\mathrm{VI} \underset{m}{\rightarrow} \mathrm{VP})$

5.4.2.3 Time: An Aside

- We shall take a rather simplistic view of time [wayne.d.blizard.90,mctaggart-t0,prior68,J.van.Benthem.Log

94. We consider d$\mathbb{T}$, or just $\mathbb{T}$, to stand for a totally ordered set of time points.
95. And we consider $\mathbb{TI}$ to stand for time intervals based on $\mathbb{T}$.
96. We postulate an infinitesimal small time interval δ.
97. $\mathbb{T}$, in our presentation, has lower and upper bounds.
98. We can compare times and we can compare time intervals.
99. And there are a number of "arithmetics-like" operations on times and time intervals.

type

94. $\mathbb{T}$

95. $\mathbb{TI}$

value

96. δ:$\mathbb{TI}$

97. $\mathbb{MIN}, \mathbb{MAX}$: $\mathbb{T} \to \mathbb{T}$

97. $<,\leq,=,\geq,>$: $(\mathbb{T}\times\mathbb{T})|(\mathbb{TI}\times\mathbb{TI}) \to$ **Bool**

98. $-$: $\mathbb{T}\times\mathbb{T} \to \mathbb{TI}$

99. $+$: $\mathbb{T}\times\mathbb{TI},\mathbb{TI}\times\mathbb{T} \to \mathbb{T}$

99. $-,+$: $\mathbb{TI}\times\mathbb{TI} \to \mathbb{TI}$

99. $*$: $\mathbb{TI}\times$**Real** $\to \mathbb{TI}$

99. $/$: $\mathbb{TI}\times\mathbb{TI} \to$ **Real**

100. We postulate a global clock behaviour which offers the current time.

101. We declare a channel clk_ch.

value
100. clock: $\mathbb{T} \rightarrow$ **out** clk_ch **Unit**
100. clock(t) $\equiv$ ... clk_ch!t ... clock(t $\sqcap$ t+δ)
channnel
101. clk_ch:$\mathbb{T}$

5.4.2.4 Road Traffic System Behaviours

102. Thus we shall consider our road traffic system, rts, as
    a the concurrent behaviour of a number of vehicles and, to “observe”, or, as we shall call it, to monitor their movements,
    b the monitor behaviour.

value
102. trs() =
102a. ‖ {veh(uid_V(v))(v)|v:V•v ∈ vs}
102b. ‖ mon(m)([])

- where the “extra” monitor argument ([])
  - records the discrete road traffic, RTF,
  - initially set to the empty map (of, “so far no road traffic”!).

5.4.2.5 Globally Observable Parts

- There is given

103. a net, **n:N**,

104. a set of vehicles, **vs:V-set**, and

105. a monitor, **m:M**.

- The **n:N**, **vs:V-set** and **m:M** are observable from the road traffic system domain.

value

103. n:N = obs_N(Δ)

103. ls:L-**set** = obs_Ls(obs_LS(n)), hs:H-**set** = obs_Hs(obs_HS(n)),

103. lis:LI-**set** = {uid_L(l)|l:L·l ∈ ls}, his:HI-**set** = {uid_H(h)|h:H·h ∈ hs}

104. vs:V-**set** = obs_Vs(obs_VS(obs_F(Δ))), vis:V-**set** = {uid_V(v)|v:V·v ∈

105. m:obs_M(Δ)

5.4.2.6 Channels

- In order for the monitor behaviour to assess the vehicle positions
  - these vehicles communicate their positions
  - to the monitor
  - via a vehicle to monitor channel.
- In order for the monitor to time-stamp these positions
  - it must be able to "read" a clock.

106. Thus we declare a set of channels indexed by the unique identifiers of vehicles and communicating vehicle positions; and
107. a single clock to monitor channel.

channel

106. $\{vm\_ch[vi]|vi:VI{\cdot}vi \in vis\}:VP$

107. clkm_ch:dT

5.4.2.7 An Aside: Attributes of Vehicles

108. Dynamic attributes of vehicles include

   a position

      i. at a hub (about to enter the hub — referred to by the link it is coming from, the hub it is at and the link it is going to, all referred to by their unique identifiers or

      ii. some fraction "down" a link (moving in the direction from a from hub to a to hub — referred to by their unique identifiers)

      iii. where we model fraction as a real between 0 and 1 included.

   b velocity, acceleration, etcetera.

type

108a.	VP = atH | onL
108(a)i.	atH :: fli:LI × hi:HI × tli:LI
108(a)ii.	onL :: fhi:HI × li:LI × frac:FRAC × thi:HI
108(a)iii.	FRAC = **Real**, **axiom** $\forall$ frac:FRAC $\cdot$ $0 \leq$ frac ≤ 1
108b.	Vel, Acc, ...

5.4.2.8 Behaviour Signatures

109. The road traffic system behaviour, rts, takes no arguments; and "behaves", that is, continues forever.
110. The vehicle behaviours are indexed by the unique identifier, uid_V(v):VI, the vehicle part, v:V and the vehicle position; offers communication to the monitor behaviour; and behaves "forever".
111. The monitor behaviour takes monitor part, m:M, as argument and also the discrete road traffic, drtf:dRTF; the behaviour otherwise runs forever.

value

109. rts: **Unit** → **Unit**

110. veh: vi:VI → v:V → VP → **out** vm_ch[vi] **Unit**

111. mon: m:M → RTF → **in** {vm_ch[vi]|vi:VI•vi ∈ vis},clkm_ch **Unit**

5.4.2.9 The Vehicle Behaviour

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

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

113. We describe here an abstraction of the vehicle behaviour **at** a **H**ub (**hi**).
   a Either the vehicle remains at that hub informing the monitor,
   b or, internally non-deterministically,
      i. moves onto a link, **tli**, whose "next" hub, identified by **thi**, is obtained from the mereology of the link identified by **tli**;
      ii. informs the monitor, on channel **vm[vi]**, that it is now on the link identified by **tli**,
      iii. whereupon the vehicle resumes the vehicle behaviour positioned at the very beginning (**0**) of that link,
   c or, again internally non-deterministically,
   d the vehicle "disappears — off the radar" !

113. veh(vi)(v)(vp:atH(fli,hi,tli)) ≡
113a. vm_ch[vi]!vp ; veh(vi)(v)(vp)
113b. ⊓
113(b)i. **let** {hi′,thi}=mereo_L(get_L(tli)(n)) **in assert:** hi′=hi
113(b)ii. vm_ch[vi]!onL(tli,hi,0,thi) ;
113(b)iii. veh(vi)(v)(onL(tli,hi,0,thi)) **end**
113c. ⊓
113d. **stop**

114. We describe here an abstraction of the vehicle behaviour on a Link (ii). Either
    a the vehicle remains at that link position informing the monitor,
    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, 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 that it is now at the hub identified by thi,
            B. whereupon the vehicle resumes the vehicle behaviour positioned at that hub.
115. or, internally non-deterministically,
116. the vehicle "disappears — off the radar" !

```
112.       veh(vi)(v)(vp:onL(fhi,li,f,thi)) ≡
114a.           vm_ch[ vi ]!vp ; veh(vi)(v)(vp)
114b.      ⌈⌉
114c.           if f + δ<1
114(c)i.            then vm_ch[ vi ]!onL(fhi,li,f+δ,thi) ;
114(c)i.                 veh(vi)(v)(onL(fhi,li,f+δ,thi))
114(c)ii.           else let li′:LI•li′ ∈ mereo_H(get_H(thi)(n)) in
114(c)iiA.               vm_ch[ vi ]!atH(li,thi,li′);
114(c)iiB.               veh(vi)(v)(atH(li,thi,li′)) end end
115.        ⌈⌉
116.            stop
```

5.4.2.10 The Monitor Behaviour

117. 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 "in[de]finitely".

118. Either the monitor "does own work"

119. or, internally non-deterministically accepts messages from vehicles.

    a A vehicle position message, vp, 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.

117. mon(m)(rtf) $\equiv$
118. mon(own_mon_work(m))(rtf)
119. $\sqcap$
119a. $\lbrack\rbrack$ { **let** ((vi,vp),t) = (vm_ch[vi]?,clkm_ch?), **in**
119b. **let** rtf′ = rtf † [t $\mapsto$ rtf(max **dom** rtf) † [vi $\mapsto$ vp]] **in**
119c. mon(m)(rtf′) **end**
119d. **end** | vi:VI · vi $\in$ vis }

118. own_mon_work: M $\rightarrow$ TBL $\rightarrow$ M

- We do not describe the clock behaviour by other than stating that it continually offers the current time on channel clkm_ch. ■

Example: 42 A Pipeline System Behaviour.

- We consider pipeline system units to represent also the following behaviours:
  - For each kind of unit, cf. Example 29 on Slide 179, there are the unit processes:
    - unit,
    - well (Item 78c on Slide 180),
    - pipe (Item 78a),
    - pump (Item 78a),
    - valve (Item 78a),
    - fork (Item 78b),
    - join (Item 78b) and
    - sink (Item 78d on Slide 180).

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) ≡
78c. is_We(u) → well(uid_U(u))(u),
78a. is_Pu(u) → pump(uid_U(u))(u),
78a. is_Pi(u) → pipe(uid_U(u))(u),
78a. is_Va(u) → valve(uid_U(u))(u),
78b. is_Fo(u) → fork(uid_U(u))(u),
78b. is_Jo(u) → join(uid_U(u))(u),
78d. is_Si(u) → sink(uid_U(u))(u)

- We illustrate essentials of just one of these behaviours.

78b. 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**
78b. fork(ui)(u) ≡
78b. **let** u′ = core_fork_behaviour(ui)(u) **in**
78b. 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.

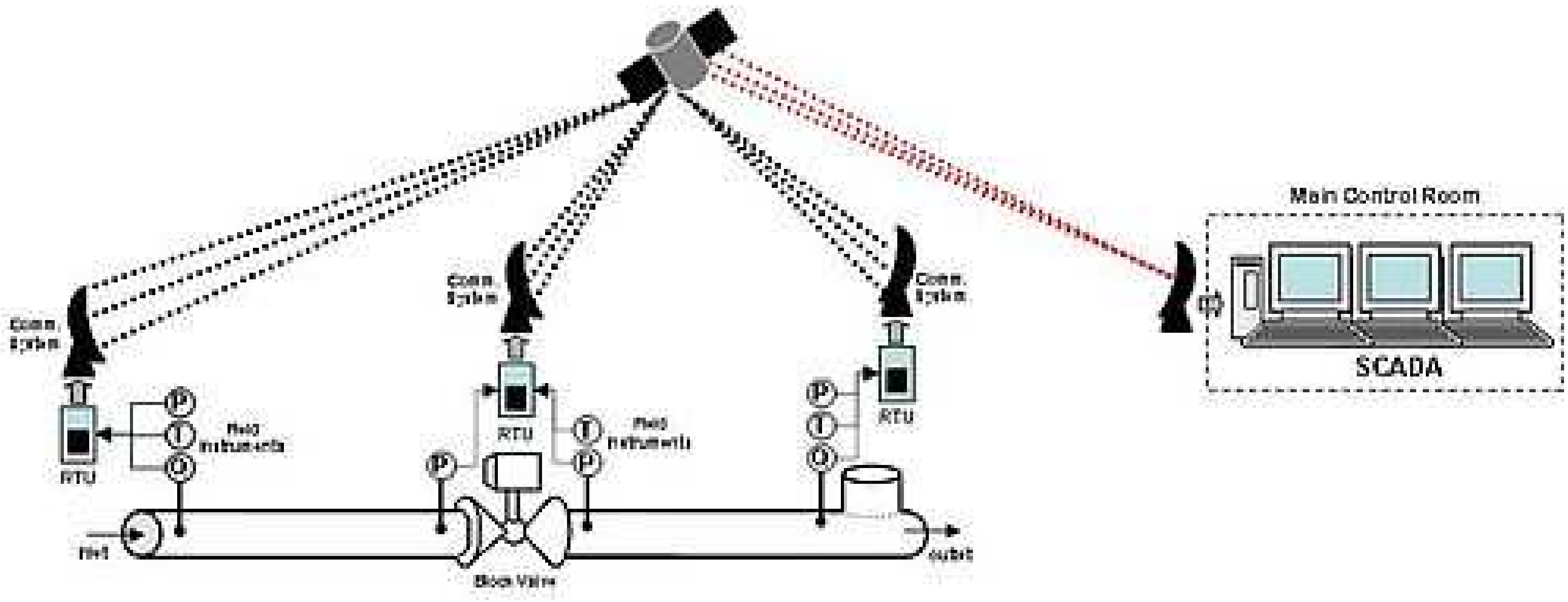

Figure 1: A supervisory control and data acquisition system

- SCADA is then part of the pipeline_system behaviour.

120.

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

- props was defined on Slide 205.

121. scada non-deterministically (internal choice, $\sqcap$), alternates between continually

a doing own work,

b acquiring data from pipeline units, and

c controlling selected such units.

type
121. Props
value
121. scada: Props → **in**,**out** { pls_ui_ch[ui] | ui:UI•ui ∈ ∈ uis } **Unit**
121. scada(props) ≡
121a. scada(scada_own_work(props))
121b. $\sqcap$ scada(scada_data_acqui_work(props))
121c. $\sqcap$ scada(scada_control_work(props))

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

122. 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
122. scada_data_acqui_work: Props → **in**,**out** { pls_ui_ch[ui] | ui:UI•ui ∈ ∈
122. scada_data_acqui_work(props) ≡
122a. ⌷ { **let** (ui,data) = pls_ui_ch[ui] ? **in**
122b. scada_input_update(ui,data)(props) **end**
122c. | ui:UI • ui ∈ uis }

122b. scada_input_update: UI × Data → Props → Props
type
122a. Data

123. 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_update**s the scada state.

123. scada_control_work: Props → **in**,**out** { pls_ui_ch[ui] | ui:UI•ui ∈ ∈ uis
123. scada_control_work(props) ≡
123a. **let** (ui,ctrl) = analyse_scada(ui,props) **in**
123b. pls_ui_ch[ui] ! ctrl ;
123c. scada_output_update(ui,ctrl)(props) **end**

123c. scada_output_update UI × Ctrl → Props → Props
type
123a. 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.

6. Seminar Conclusion
6.1. Other Work on Domain Analysis

- Our comparison hinges on basically the following two facets:
  - domain analysis and
  - domain description.
- We shall see that the former term, seen across the surveyed literature,
  - covers techniques that are claimed used in many steps of software engineering,
  - but that they seldom, if ever, involve formal concept analysis as we understand it.

6.1.1. An Enumeration

- Formal Concept Analysis: Ganter & Will **Mathematics**
- Miscellaneous Directions **Software Engineering**
  - Business Process [Re-]engineering, BPE, BPRE
  - Ontological Engineering
  - Knowledge and Knowledge Engineering, KE
  - Prieto-Dĩaz's Domain Analysis
  - Software Product Line Engineering
  - M.A. Jackson's Problem Frames
  - Domain Specific Software Architectures, DSS
  - Domain Driven Design, DDD
  - Feature-oriented Domain Analysis, FODA
  - Unified Modelling Language, UML

6.1.2. Summary of Comparisons

- It should now be clear from the above that there are basically two notions from above that relate to our notion of **domain analysis**.
  - (i) Prieto-Dĩaz's notion of *'Domain Analysis'*, and
  - (ii) Jackson's notion of *Problem Frames*.
- But it should also be clear that none of the surveyed literature,
  - except, of course, Ganter & Wille's `[GanterWille:ConceptualAnalysis1999]` *Formal Concept Analysis, Mathematical Foundations*,
  - covers our notion of **domain analysis**
  - as it hinges crucially on Ganter & Wille's **formal concept analysis**.

6.2. What Have We Achieved ?

- Identification and modelling of **domain entities**
  - **endurants**
    - **atomic parts** and **composite parts** (**obs_P**),
    - **part properties**
      - **unique identification** (**uid_P**),
      - **mereology** (**mereo_P**),
      - **attributes** (**attr_Q**),
  - and **perdurants**
    - **action signatures** and actions,
    - **event signaures** and events, and
    - **behaviour signatures** and behaviours.
- As ontological concepts the structuring and treatment
  - of the above is **possibly new** to you.

6.3. What Needs Further Research

- Endurants and Perdurants
- Mereology
- Formal Conceot Analysis of Perdurants
- Etcetera, etcetera !

7. Questions ?


Software Engineering 1
Software Engineering 2
Software Engineering 3
软件工程 1
抽象与建模
SOFTWARE ENGINEERING 1
Abstraction and Modeling
软件工程 2
系统与语言规约
SOFTWARE ENGINEERING 2
Specification of Systems and Languages
软件工程 3
领域、需求与软件设计
SOFTWARE ENGINEERING 3
Domains, Requirements, and Software Design