DOMAIN SCIENCE & ENGINEERING

The TU Wien Lectures, Fall 2022

—

Dines Bjørner

Technical University of Denmark

The Triptych Dogma

In order to *specify* **software**,
we must understand its requirements.

In order to *prescribe* **requirements**
we must understand the **domain**.

So we must **study, analyse** and **describe** domains.

- Day # 1 von Neumann Mon.24 Oct. 2022 • Seminar & Example • 10:15–11:00,11:15–12:00

- Day # 2 von Neumann Tue. 25 Oct. 2022 • Endurants • 8:15–9:00, 9:15–10:00

- Day # 3 von Neumann Thu. 27 Oct. 2022 • Endurants • 8:15–9:00, 9:15–10:00

- Day # 4 von Neumann Fri. 28 Oct. 2022 • Endurants • 8:15–9:00, 9:15–10:00

- Day # 5 von Neumann Mon. 31 Oct. 2022 • Example • 8:15–9:00, 9:15–10:00

- Day # 6 von Neumann Thu. 3 Nov. 2022 • Perdurants • 8:15–9:00, 9:15–10:00

- Day # 7 Gödel Fri. 4 Nov. 2022 • Perdurants • 8:15–9:00, 9:15–10:00

Day #5: Example: Pipelines

Appendix B. Pipelines

B.1 Endurants: External Qualities

We follow the ontology of Fig. B.1, the lefthand dashed box labelled *External Qualities*.

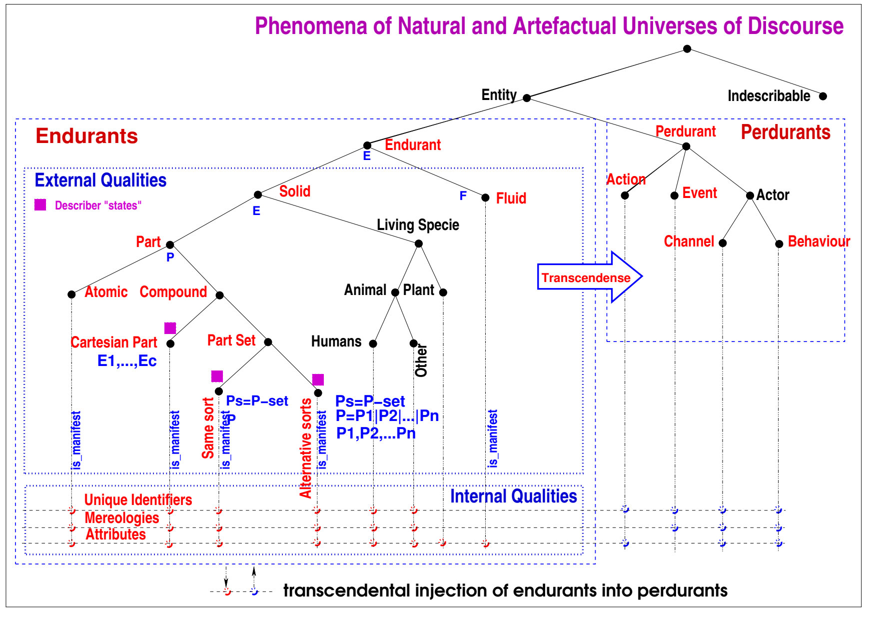

Figure B.1: Upper Ontology

B.1.1 Parts

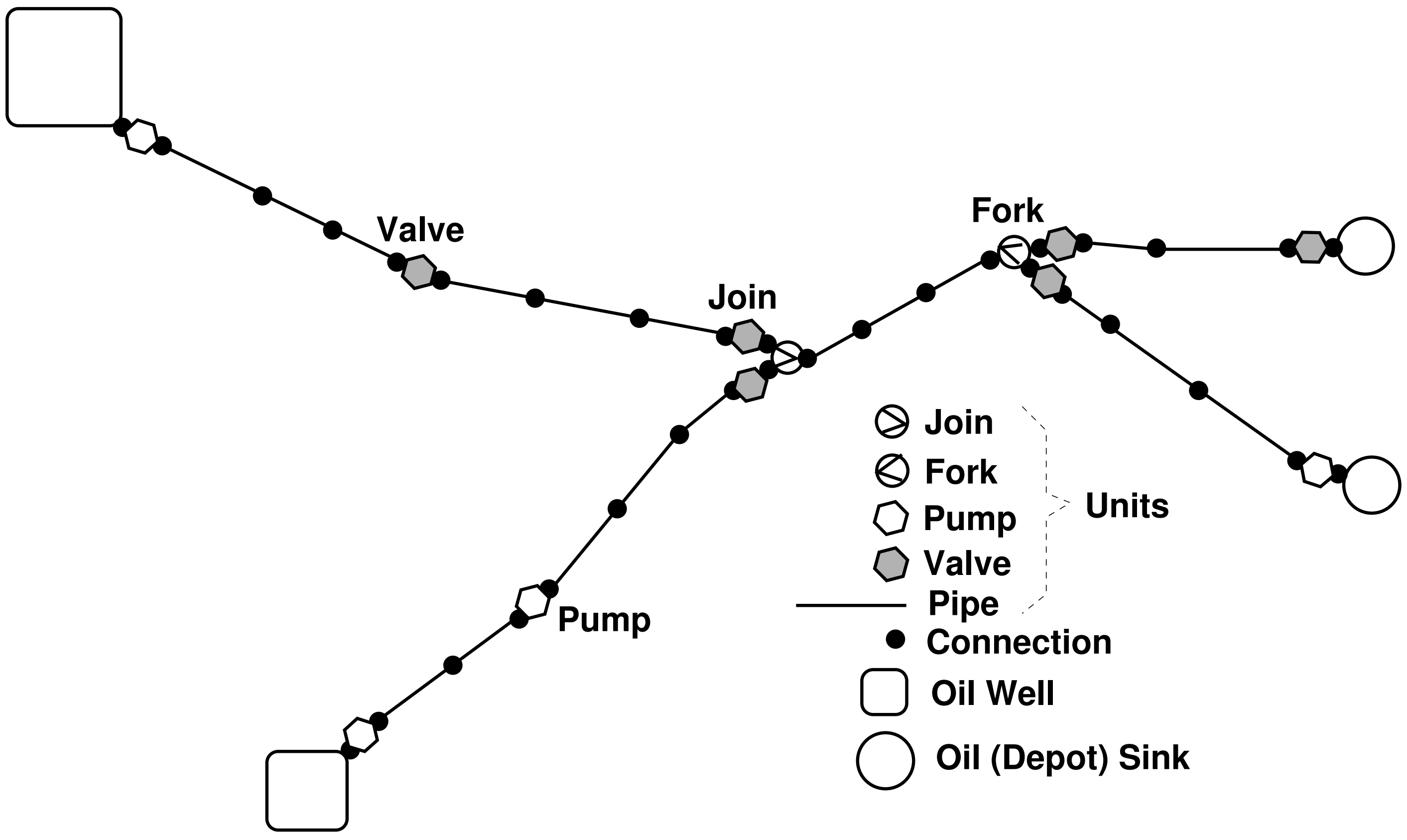

Figure B.2: An example pipeline system

280. A pipeline system contains a set of pipeline units and a pipeline system monitor.
281. The well-formedness of a pipeline system depends on its mereology and the routing of its pipes.
282. A pipeline unit is either a well, a pipe, a pump, a valve, a fork, a join, a plate[1], or a sink unit.
283. We consider all these units to be distinguishable, i.e., the set of wells, the set pipe, etc., the set of sinks, to be disjoint.

[1] A *plate* unit is a usually circular, flat steel plate used to "begin" or "end" a pipe segment.

type
280. PLS′, U, M
281. PLS = {| pls:PLS′•wf_PLS(pls) |}
value
281. wf_PLS: PLS → **Bool**
281. wf_PLS(pls) ≡
281. wf_Mereology(pls)∧wf_Routes(pls)∧wf_Metrics(pls)[2]
280. obs_Us: PLS → U-**set**
280. obs_M: PLS → M
type
282. U = We | Pi | Pu | Va | Fo | Jo | Pl | Si
283. We :: Well
283. Pi :: Pipe
283. Pu :: Pump
283. Va :: Valv
283. Fo :: Fork
283. Jo :: Join
283. Pl :: Plate
283. Si :: Sink

[2] wf_Mereology, wf_Routes and wf_Metrics will be explained in Sects. B.2.2.2 on Slide 545, B.2.3.2 on Slide 551, and B.2.4.3 on Slide 567.

B.1.2 An Endurant State

284. For a given pipeline system
285. we exemplify an endurant state σ
286. composed of the given pipeline system and all its manifest units, i.e., without plates.

value
284. pls:PLS
variable
285. σ := collect_state(pls)
value
286. collect_state: PLS
286. collect_state(pls) ≡ {pls} ∪ obs_Us(pls) \ Pl

B.2 Endurants: Internal Qualities

We follow the ontology of Fig. 4.1 on Slide 64, the lefthand vertical and horisontal lines.

B.2.1 Unique Identification

287. The pipeline system, as such,
288. has a unique identifier, distinct (different) from its pipeline unit identifiers.
289. Each pipeline unit is uniquely distinguished by its unit identifier.
290. There is a state of all unique identifiers.

type

288. PLSI

289. UI

value

287. pls:PLS

288. uid_PLS: PLS → PLSI

289. uid_U: U → UI

variable

290. σ_{uid} := { uid_PLS(pls) } ∪ xtr_UIs(pls)

axiom

289. ∀ u,u′:U·{u,u′}⊆obs_Us(pls)⇒u≠u′⇒uid_UI(u)≠uid_UI(u′)

289. ∧ uid_PLS(pls) ∉ {uid_UI(u)|u:U·u ∈ obs_Us(pls)}

291. From a pipeline system one can observe the set of all unique unit identifiers.

value

291. xtr_UIs: PLS → UI-**set**
291. xtr_UIs(pls) ≡ {uid_UI(u)|u:U·u ∈ obs_Us(pls)}

292. We can prove that the number of unique unit identifiers of a pipeline system equals that of the units of that system.

theorem:

292. ∀ pls:PLS·**card** obs_Us(pl)=**card** xtr_UIs(pls)

B.2.2 Mereology

B.2.2.1 PLS Mereology

293. The mereology of a pipeline system is the set of unique identifiers of all the units of that system.

type
293. PLS_Mer = UI-**set**iptyPLS_Merpls-mer-00
value
293. mereo_PLS: PLS → PLS_Meripobmereo_PLSpls-mer-00
axiomiptyWellformed Mereologiespls-mer-00
293. ∀ uis:PLS_Mer · uis = **card** xtr_UIs(pls)

B.2.2.2 Unit Mereologies

294. Each unit is connected to zero, one or two other existing input units and zero, one or two other existing output units as follows:
    (a) A well unit is connected to exactly one output unit (and, hence, has no “input”).
    (b) A pipe unit is connected to exactly one input unit and one output unit.
    (c) A pump unit is connected to exactly one input unit and one output unit.
    (d) A valve is connected to exactly one input unit and one output unit.
    (e) A fork is connected to exactly one input unit and two distinct output units.
    (f) A join is connected to exactly two distinct input units and one output unit.
    (g) A plate is connected to exactly one unit.
    (h) A sink is connected to exactly one input unit (and, hence, has no “output”).

```
   type
294.    MER = UI-set × UI-set
   value
294.    mereo_U: U → MER
   axiom
294.     wf_Mereology: PLS → Bool
294.     wf_Mereology(pls) ≡
294.       ∀ u:U•u ∈ obs_Us(pls)⇒
294.         let (iuis,ouis) = mereo_U(u) in iuis ∪ ouis ⊆ xtr_UIs(pls) ∧
294.           case (u,(card uius,card ouis)) of
294a.              (mk_We(we),(0,1)) → true,
294b.              (mk_Pi(pi),(1,1)) → true,
294c.              (mk_Pu(pu),(1,1)) → true,
294d.              (mk_Va(va),(1,1)) → true,
294e.              (mk_Fo(fo),(1,1)) → true,
294f.             (mk_Jo(jo),(1,1)) → true,
294f.             (mk_Pl(pl),(0,1)) → true, "begin"
294f.             (mk_Pl(pl),(1,0)) → true, "end"
294h.              (mk_Si(si),(1,1)) → true,
294.            _ → false end end
```

B.2.3 Pipeline Concepts, I

B.2.3.1 Pipe Routes

295. A route (of a pipeline system) is a sequence of connected units (of the pipeline system).
296. A route descriptor is a sequence of unit identifiers and the connected units of a route (of a pipeline system).

type

295. $\mathsf{R}' = \mathsf{U}^\omega$

295. R = {| r:Route′·wf_Route(r) |}

296. $\mathsf{RD} = \mathsf{UI}^\omega$

axiom

296. ∀ rd:RD · ∃ r:R·rd=descriptor(r)

value

296. descriptor: R → RD

296. descriptor(r) ≡ ⟨uid_UI(r[i])|i:**Nat**·1≤i≤**len** r⟩

297. Two units are adjacent if the output unit identifiers of one shares a unique unit identifier with the input identifiers of the other.

value

297. adjacent: U × U → **Bool**

297. adjacent(u,u′) ≡ **let** (,ouis)=mereo_U(u),(iuis,)=mereo_U(u′) **in** ouis ∩ iuis ≠ {} **en**

298. Given a pipeline system, *pls*, one can identify the (possibly infinite) set of (possibly infinite) routes of that pipeline system.
   (a) The empty sequence, $\langle\rangle$, is a route of *pls*.
   (b) Let u, u' be any units of *pls*, such that an output unit identifier of u is the same as an input unit identifier of u' then $\langle u, u'\rangle$ is a route of *pls*.
   (c) If r and r' are routes of *pls* such that the last element of r is the same as the first element of r', then $r\frown\mathbf{tl}\,r'$ is a route of *pls*.
   (d) No sequence of units is a route unless it follows from a finite (or an infinite) number of applications of the basis and induction clauses of Items 298a–298c.

value

```
298.  Routes: PLS → RD-infset
298.  Routes(pls) ≡
298a.     let rs = ⟨⟩ ∪
298b.         {⟨uid_UI(u),uid_UI(u′)⟩|u,u′:U·{u,u′}⊆obs_Us(pls) ∧ adjacent(u,u′)}
298c.       ∪ {r⁀tl r′|r,r′:R·{r,r′}⊆rs}
298d.     in rs end
```

B.2.3.2 Well-formed Routes

299. A route is acyclic if no two route positions reveal the same unique unit identifier.

value

299. is_acyclic_Route: R → **Bool**
299. is_acyclic_Route(r) ≡ ∼∃ i,j:**Nat**·{i,j}⊆**inds** r ∧ i≠j ∧ r[i]=r[j]

300. A pipeline system is well-formed if none of its routes are circular (and all of its routes embedded in well-to-sink routes).

value

300. wf_Routes: PLS → **Bool**
300. wf_Routes(pls) ≡
300. non_circular(pls) ∧ are_embedded_Routes(pls)

300. is_non_circular_PLS: PLS → **Bool**
300. is_non_circular_PLS(pls) ≡
300. ∀ r:R·r ∈ routes(p)∧acyclic_Route(r)

301. We define well-formedness in terms of well-to-sink routes, i.e., routes which start with a well unit and end with a sink unit.

value

```
301. well_to_sink_Routes: PLS → R-set
301. well_to_sink_Routes(pls) ≡
301.   let rs = Routes(pls) in
301.   {r|r:R·r ∈ rs ∧ is_We(r[1]) ∧ is_Si(r[len r])} end
```

302. A pipeline system is well-formed if all of its routes are embedded in well-to-sink routes.

```
302.  are_embedded_Routes: PLS → Bool
302.  are_embedded_Routes(pls) ≡
302.     let wsrs = well_to_sink_Routes(pls) in
302.     ∀ r:R · r ∈ Routes(pls) ⇒
302.        ∃ r′:R,i,j:Nat ·
302.          r′ ∈ wsrs
302.         ∧ {i,j}⊆inds r′∧i≤j
302.         ∧ r = ⟨r′[ k ]|k:Nat·i≤k≤j⟩ end
```

B.2.3.3 Embedded Routes

303. For every route we can define the set of all its embedded routes.

value

303. embedded_Routes: R → R-**set**

303. embedded_Routes(r) ≡ {⟨r[k]|k:**Nat**·i≤k≤j⟩ | i,j:**Nat**· i {i,j}⊆**inds**(r) ∧ i≤j}

B.2.3.4 A Theorem

304. The following theorem is conjectured:

(a) the set of all routes (of the pipeline system)

(b) is the set of all well-to-sink routes (of a pipeline system) and

(c) all their embedded routes

theorem:

304. ∀ pls:PLS ·
304. **let** rs = Routes(pls),
304. wsrs = well_to_sink_Routes(pls) **in**
304a. rs =
304b. wsrs ∪
304c. ∪ {{r′|r′:R · r′ ∈ is_embedded_Routes(r″)} | r″:R · r″ ∈ wsrs}
303. **end**

B.2.3.5 Fluids

305. The only fluid of concern to pipelines is the gas[3] or liquid[4] which the pipes transport[5].

type

305. GoL [= M]

value

305. obs_GoL: U → GoL

[3] Gaseous materials include: air, gas, etc.
[4] Liquid materials include water, oil, etc.
[5] The description of this document is relevant only to gas or oil pipelines.

B.2.4 Attributes

B.2.4.1 Unit Flow Attributes

306. A number of attribute types characterise units:

(a) estimated current well capacity (barrels of oil, etc.),

(b) pump height (a static attribute),

(c) current pump status (not pumping, pumping; a programmable attribute),

(d) current valve status (closed, open; a programmable attribute) and

(e) flow (barrels/second, a biddable attribute).

type

306a. WellCap
306b. Pump_Height
306c. Pump_State == {|**not_pumping**,**pumping**|}
306d. Valve_State == {|**closed**,**open**|}
306e. Flow

307. Flows can be added and subtracted,
308. added distributively and
309. flows can be compared.

value

307. $\oplus,\ominus$: Flow×Flow → Flow
308. $\oplus$: Flow-**set** → Flow
309. $<,\leq,=,\neq,\geq,>$: Flow × Flow → **Bool**

310. Properties of pipeline units include

   (a) estimated current well capacity (barrels of oil, etc.) [a biddable attribute],
   (b) pipe length [a static attribute],
   (c) current pump height [a biddable attribute],
   (d) current valve open/close status [a programmable attribute],
   (e) current [$\mathcal{L}$aminar] in-flow at unit input [a monitorable attribute],
   (f) current $\mathcal{L}$aminar] in-flow leak at unit input [a monitorable attribute],
   (g) maximum [$\mathcal{L}$aminar] guaranteed in-flow leak at unit input [a static attribute],

(h) current [$\mathcal{L}$aminar] leak unit interior [a monitorable attribute],
(i) current [$\mathcal{L}$aminar] flow in unit interior [a monitorable attribute],
(j) maximum $\mathcal{L}$aminar] guaranteed flow in unit interior [a monitorable attribute],
(k) current [$\mathcal{L}$aminar] out-flow at unit output [a monitorable attribute],
(l) current [$\mathcal{L}$aminar] out-flow leak at unit output [a monitorable attribute] and
(m) maximum guaranteed $\mathcal{L}$aminar out-flow leak at unit output [a static attribute.

type
310e In_Flow = Flow
310f In_Leak = Flow
310g Max_In_Leak = Flow
310h Body_Flow = Flow
310i Body_Leak = Flow
310j Max_Flow = Flow
310k Out_Flow = Flow
310l Out_Leak = Flow
310m Max_Out_Leak = Flow
value
310a attr_WellCap: We → WellCap
310b attr_LEN: Pi → LEN
310c attr_Height: Pu → Height
310d attr_ValSta: Va → VaSta
310e attr_In_Flow: U → UI → Flow
310f attr_In_Leak: U → UI → Flow
310g attr_Max_In_Leak: U → UI → Flow
310h attr_Body_Flow: U → Flow
310i attr_Body_Leak: U → Flow
310j attr_Max_Flow: U → Flow
310k attr_Out_Flow: U → UI → Flow
310l attr_Out_Leak: U → UI → Flow
310m attr_Max_Out_Leak: U → UI → Flow

311. Summarising we can define a two notions of flow:

   (a) static and
   (b) monitorable.

type

311a Sta_Flows = Max_In_Leak×In_Max_Flow>Max_Out_Leak

311b Mon_Flows = In_Flow×In_Leak×Body_Flow×Body_Leak×Out_Flow×Out_Leak

B.2.4.2 Unit Metrics

- Pipelines are laid out in the terrain.
  - Units have length and diameters.
  - Units are positioned in space: have altitude, longitude and latitude positions of its one, two or three connection PoinTs[6].

312. length (a static attribute),

313. diameter (a static attribute) and

314. position (a static attribute).

[6] 1 for *wells*, *plates* and *sinks*; 2 for *pipes*, *pumps* and *valves*; 1+2 for *forks*, 2+1 for *joins*.

type

312. LEN
313. ◯
314. POS == mk_One(pt:PT) | mk_Two(ipt:PT,opt:PT)
314. | mk_OneTwo(ipt:PT,opts:(lpt:PT,rpt:PT))
314. | mk_TwoOne(ipts:(lpt:PT,rpt:PT),opt:PT)
314. PT = Alt × Lon × Lat
314. Alt, Lon, Lat = ...

value

312. attr_LEN: U → LEN
313. attr_◯: U → ◯
314. attr_POS: U → POS

- We can summarise the metric attributes:

315. Units are subject to either of four (mutually exclusive) metrics:
    (a) Length, diameter and a one point position.
    (b) Length, diameter and a two points position.
    (c) Length, diameter and a one+two points position.
    (d) Length, diameter and a two+one points position.

type

315. Unit_Sta = Sta1_Metric | Sta2_Metric | Sta12_Metric | Sta21_Metric
315a Sta1_Metric = LEN × Ø × mk_One(pt:PT)
315b Sta2_Metric = LEN × Ø × mk_Two(ipt:PT,opt:PT)
315c Sta12_Metric = LEN × Ø × mk_OneTwo(ipt:PT,opts:(lpt:PT,rpt:PT))
315d Sta21_Metric = LEN × Ø × mk_TwpOne(ipts:(lpt:PT,rpt:PT),opt:PT)

B.2.4.3 Wellformed Unit Metrics

- The points positions of neighbouring units must "fit" one-another.

316. Without going into details we can define a predicate, wf_Metrics, that applies to a pipeline system and yields **true** iff neighbouring units must "fit" one-another.

value

316. wf_Metrics: PLS → **Bool**
316. wf_Metrics(pls) ≡ ...

B.2.4.4 Summary

- We summarise the static, monitorable and programmable attributes for each manifest part of the pipeline system:

type

PLS_Sta = PLS_net×...
PLS_Mon = ...
PLS_Prg = PLS_Σ×...
Well_Sta = Sta1_Metric×Sta_Flows×Orig_Cap×...
Well_Mon = Mon_Flows×Well_Cap×...
Well_Prg = ...
Pipe_Sta = Sta2_Metric×Sta_Flows×LEN×...
Pipe_Mon = Mon_Flows×In_Temp×Out_Temp×...
Pipe_Prg = ...
Pump_Sta = Sta2_Metric×Sta_Flows×Pump_Height×...
Pump_Mon = Mon_Flows×...
Pump_Prg = Pump_State×...

Valve_Sta = Sta2_Metric×Sta_Flows×...
Valve_Mon = Mon_Flows×In_Temp×Out_Temp×...
Valve_Prg = Valve_State×...
Fork_Sta = Sta12_Metric×Sta_Flows×...
Fork_Mon = Mon_Flows×In_Temp×Out_Temp×...
Fork_Prg = ...
Join_Sta = Sta21_Metric×Sta_Flows×...
Join_Mon = Mon_Flows×In_Temp×Out_Temp×...
Join_Prg = ...
Sink_Sta = Sta1_Metric×Sta_Flows×Max_Vol×...
Sink_Mon = Mon_Flows×Curr_Vol×In_Temp×Out_Temp×...
Sink_Prg = ...

317. Corresponding to the above three attribute categories we can define “collective” attribute observers:

value

317. sta_A_We: We → Sta1_Metric×Sta_Flows×Orig_Cap×...
317. mon_A_We: We → ηMon_Flows×ηWell_Cap×ηIn_Temp×ηOut_Temp×...
317. prg_A_We: We → ...
317. sta_A_Pi: Pi → Sta2_Metric×Sta_Flows×LEN×...
317. mon_A_Pi: Pi → $\mathcal{N}$Mon_Flows×ηIn_Temp×ηOut_Temp×...
317. prg_A_Pi: Pi → ...
317. sta_A_Pu: Pu → Sta2_Metric×Sta_Flows×LEN×...
317. mon_A_Pu: Pu → $\mathcal{N}$Mon_Flows×ηIn_Temp×ηOut_Temp×...
317. prg_A_Pu: Pu → Pump_State×...
317. sta_A_Va: Va → Sta2_Metric×Sta_Flows×LEN×...
317. mon_A_Va: Va → $\mathcal{N}$Mon_Flows×ηIn_Temp×ηOut_Temp×...
317. prg_A_Va: Va → Valve_State×...
317. sta_A_Fo: Fo → Sta12_Metric×Sta_Flows×...

317. mon_A_Fo: Fo → $\mathcal{N}$Mon_Flows×ηIn_Temp×ηOut_Temp×...
317. prg_A_Fo: Fo → ...
317. sta_A_Jo: Jo → Sta21_Metric×Sta_Flows×...
317. mon_A_Jo: Jo → Mon_Flows×ηIn_Temp×ηOut_Temp×...
317. prg_A_Jo: Jo → ...
317. sta_A_Si: Si → Sta1_Metric×Sta_Flows×Max_Vol×...
317. mon_A_Si: Si → $\mathcal{N}$Mon_Flows×ηIn_Temp×ηOut_Temp×...
317. prg_A_Si: Si → ...

317. $\mathcal{N}$Mon_Flows ≡ (ηIn_Flow,ηIn_Leak,ηBody_Flow,ηBody_Leak,ηOut_Flow,ηOut_Leak

- Monitored flow attributes
  - are [to be] passed as arguments to behaviours *by reference*
  - so that their monitorable attribute values can be sampled.

B.2.4.5 Fluid Attributes

- Fluids, we here assume, oil, as it appears in the pipeline units
  - have no unique identity,
  - have not mereology,
  - but does have attributes: hydrocarbons consisting predominantly of
    - aliphatic,
    - alicyclic and
    - aromatic hydrocarbons.
  - It may also contain small amounts of
    - nitrogen,
    - oxygen, and
    - sulfur

    compounds

318. We shall simplify, just for illustration, crude oil fluid of units to have these attributes:

(a) volume,
(b) viscosity,
(c) temperature,
(d) paraffin content (%age),
(e) naphtenes content (%age),

type		**value**	
318.	Oil	318b.	obs_Oil: U → Oil
318a.	Vol	318a.	attr_Vol: Oil → Vol
318b.	Visc	318b.	attr_Visc: Oil → Visc
318c.	Temp	318c.	attr_Temp: Oil → Temp
318d.	Paraffin	318d.	attr_Paraffin: Oil → Paraffin
318e.	Naphtene	318e.	attr_Naphtene: Oil → Naphtene

B.2.4.6 Pipeline System Attributes

- The “root” pipeline system is a compound.
- In its transcendentally deduced behavioral form
  - it is, amongst other “tasks”, entrusted with the monitoring and control of all its units.
    - ⋆ To do so it must, as a basically static attribute
    - ⋆ possess awareness, say in the form of a net diagram
      - · of how these units are interconnected,
      - · together with all their internal qualities,
      - · by type and by value.
  - Next we shall give a very simplified account of the possible pipeline system attribute.

319. We shall make use, in this example, of just a simple pipeline state, pls_ω.

– The pipeline state, pls_ω, embodies all the information that is relevant to the monitoring and control of an entire pipeline system, whether static or dynamic.

type
319. PLS_Ω

B.2.5 Pipeline Concepts, II: Flow Laws

320. "What flows in, flows out !". For $\mathcal{L}$aminar flows: for any non-well and non-sink unit the sums of input leaks and in-flows equals the sums of unit and output leaks and out-flows.

Law:

320. $\forall$ u:U\We\Si ·
320. sum_in_leaks(u) $\oplus$ sum_in_flows(u) =
320. attr_body_Leak$_{\mathcal{L}}$(u) $\oplus$
320. sum_out_leaks(u) $\oplus$ sum_out_flows(u)

value

sum_in_leaks: U → Flow
sum_in_leaks(u) ≡ **let** (iuis,) = mereo_U(u) **in** ⊕ {attr_In_Leak$_{\mathcal{L}}$(u)(ui)|ui:UI•ui ∈ iuis} **end**
sum_in_flows: U → Flow
sum_in_flows(u) ≡ **let** (iuis,) = mereo_U(u) **in** ⊕ {attr_In_Flow$_{\mathcal{L}}$(u)(ui)|ui:UI•ui ∈ iuis} **end**
sum_out_leaks: U → Flow
sum_out_leaks(u) ≡ **let** (,ouis) = mereo_U(u) **in** ⊕ {attr_Out_Leak$_{\mathcal{L}}$(u)(ui)|ui:UI•ui ∈ ouis} **end**
sum_out_flows: U → Flow
sum_out_flows(u) ≡ **let** (,ouis) = mereo_U(u) **in** ⊕ {attr_Out_Leak$_{\mathcal{L}}$(u)(ui)|ui:UI•ui ∈ ouis} **end**

321. "What flows out, flows in !". For $\mathcal{L}$aminar flows: for any adjacent pairs of units the output flow at one unit connection equals the sum of adjacent unit leak and in-flow at that connection.

Law:

321. ∀ u,u′:U·adjacent(u,u′) ⇒
321. **let** (,ouis)=mereo_U(u), (iuis′,)=mereo_U(u′) **in**
321. **assert:** uid_U(u′) ∈ ouis ∧ uid_U(u) ∈ iuis ′
321. attr_Out_Flow$_{\mathcal{L}}$(u)(uid_U(u′)) =
321. attr_In_Leak$_{\mathcal{L}}$(u)(uid_U(u))⊕attr_In_Flow$_{\mathcal{L}}$(u′)(uid_U(u)) **end**

- These "laws" should hold for a pipeline system without plates.

B.3 Perdurants

We follow the ontology of Fig. 4.1 on Slide 64, the right-hand dashed box labeled *Perdurants* and the right-hand vertical and horisontal lines.

B.3.1 State

- We introduce concepts of *manifest* and *structure* endurants.
  - The former are such compound endurants (Cartesians of sets) to which we ascribe internal qualities;
  - the latter are such compound endurants (Cartesians of sets) to which we **do not** ascribe internal qualities.
- The distinction is pragmatic.

322. For any given pipeline system we suggest the state to consist of the manifest endurants of all its non-plate units.

value

322. σ = obs_Us(pls)

B.3.2 Channel

323. There is a [global] array channel
indexed by a "set pair" of distinct manifest endurant part identifiers – signifying the possibility of the syncharonisation and communication between any pair of pipeline units and between these and the pipeline system, cf. last, i.e., bottom-most diagram of Fig. B.12 on Slide 610.

channel

323. { ch[{i,j}] | {i,j}:(PLSI|UI) · {i,j}$\subseteq\sigma_{id}$ }

B.3.3 Actions

- These are, informally, some of the actions of a pipeline system:

324. **start pumping**: from a state of not pumping to a state of pumping "at full blast !".[7]
325. **stop pumping**: from a state of (full) pumping to a state of no pumping at all.
326. **open valve**: from a state of a fully closed valve to a state of fully open valve.[8]
327. **close valve**: from a state of a fully opened valve to a state of fully closed valve.

- We shall not define these actions in this paper.
- But they will be referred to in the *pipeline_system* (Items 346a, 346b, 346c), the *pump* (Items 349a, 349b) and the *valve* (Items 352a, 352b) behaviours.

[7] – that is, we simplify, just for the sake of illustration, and do not consider "intermediate" states of pumping.
[8] – cf. Footnote 7.

B.3.4 Behaviours

B.3.4.1 Behaviour Kinds

- There are eight kinds of behaviours:

328. the pipeline system behaviour;[9]
329. the [generic] well behaviour,
330. the [generic] pipe behaviour,
331. the [generic] pump behaviour,
332. the [generic] valve behaviour,
333. the [generic] fork behaviour,
334. the [generic] join behaviour,
335. the [generic] sink behaviour.

[9]This "PLS" behaviour summarises the either global, i.e., *SCADA*[10]-like behaviour, or the fully distributed, for example, manual, human-operated behaviour of the monitoring and control of the entire pipeline system.

[10]Supervisory Control And Data Acquisition

B.3.4.2 Behaviour Signatures

336. The *pipeline_system* behaviour, *pls*,
337. The *well* behaviour signature lists the unique well identifier, the well mereology, the static well attributes, the monitorable well attributes, the programmable well attributes and the channels over which the well [may] interact with the pipeline system and a pipeline unit.
338. The *pipe* behaviour signature lists the unique pipe identifier, the pipe mereology, the static pipe attributes, the monitorable pipe attributes, the programmable pipe attributes and the channels over which the pipe [may] interact with the pipeline system and its two neighbouring pipeline units.

339. The *pump* behaviour signature lists the unique pump identifier, the pump mereology, the static pump attributes, the monitorable pump attributes, the programmable pump attributes and the channels over which the pump [may] interact with the pipeline system and its two neighbouring pipeline units.
340. The *valve* behaviour signature lists the unique valve identifier, the valve mereology, the static valve attributes, the monitorable valve attributes, the programmable valve attributes and the channels over which the valve [may] interact with the pipeline system and its two neighbouring pipeline units.

341. The *fork* behaviour signature lists the unique fork identifier, the fork mereology, the static fork attributes, the monitorable fork attributes, the programmable fork attributes and the channels over which the fork [may] interact with the pipeline system and its three neighbouring pipeline units.
342. The *join* behaviour signature lists the unique join identifier, the join mereology, the static join attributes, the monitorable join attributes, the programmable join attributes and the channels over which the join [may] interact with the pipeline system and its three neighbouring pipeline units.

343. The *sink* behaviour signature lists the unique sink identifier, the sink mereology, the static sing attributes, the monitorable sing attributes, the programmable sink attributes and the channels over which the sink [may] interact with the pipeline system and its one or more pipeline units.

value

336. pls: plso:PLSI → pls_mer:PLS_Mer → PLS_Sta → PLS_Mon →
336. PLS_Prg → { ch[{plsi,ui}] | ui:UI · ui ∈ σ_{ui} } **Unit**
337. well: wid:WI → well_mer:MER → Well_Sta → Well_mon →
337. Well_Prgr → { ch[{plsi,ui}] | wi:WI · ui ∈ σ_{ui} } **Unit**
338. πipe: UI → pipe_mer:MER → Pipe_Sta → Pipe_mon →
338. Pipe_Prgr → { ch[{plsi,ui}] | ui:UI · ui ∈ σ_{ui} } **Unit**
339. pump: pi:UI → pump_mer:MER → Pump_Sta → Pump_Mon →
339. Pump_Prgr → { ch[{plsi,ui}] | ui:UI · ui ∈ σ_{ui} } **Unit**
340. valve: vi:UI → valve_mer:MER → Valve_Sta → Valve_Mon →
340. Valve_Prgr → { ch[{plsi,ui}] | ui:UI · ui ∈ σ_{ui} } **Unit**
341. fork: fi:FI → fork_mer:MER → Fork_Sta → Fork_Mon →
341. Fork_Prgr → { ch[{plsi,ui}] | ui:UI · ui ∈ σ_{ui} } **Unit**
342. join: ji:JI → join_mer:MER → Join_Sta → Join_Mon →
342. Join_Prgr → { ch[{plsi,ui}] | ui:UI · ui ∈ σ_{ui} } **Unit**
343. sink: si:SI → sink_mer:MER → Sink_Sta → Sink_Mon →
343. Sink_Prgr → { ch[{plsi,ui}] | ui:UI · ui ∈ σ_{ui} } **Unit**

B.3.4.2.1 Behaviour Definitions

- We show the definition of only three behaviours:
- the pipe_line_system behaviour,
- the pump behaviour and
- the valve behaviour.

B.3.4.2.2 The Pipeline System Behaviour

344. The pipeline system behaviour
345. calculates, based on its programmable state, its next move;
346. if that move is [to be] an action on a named
    (a) pump, whether to start or stop pumping, then the named pump is so informed, whereupon the pipeline system behaviour resumes in the new pipeline state; or
    (b) valve, whether to open or close the valve, then the named valve is so informed, whereupon the pipeline system behaviour resumes in the new pipeline state; or
    (c) unit, to collect its monitorable attribute values for monitoring, whereupon the pipeline system behaviour resumes in the further updated pipeline state;
    (d) et cetera;

value

```
344.  pls(plsi)(uis)(pls_msta)(pls_mon)(pls_ω) ≡
345.    let (to_do,pls_ω′) = calculate_next_move(plsi,pls_mer,pls_msta,pls_mon,pls_prgr
346.    case to_do of
346a      mk_Pump(pi,α) →
346a          ch[ {plsi,pi} ] ! α assert: α ∈ {stop_pumping,pump};
346a          pls(plsi)(pls_mer)(pls_msta)(pls_mon)(pls_ω′),
346b      mk_Valve(vi,α) →
346b          ch[ {plsi,vi} ] ! α assert: α ∈ {open_valve,close_valve};
346b          pls(plsi)(pls_mer)(pls_msta)(pls_mon)(pls_ω′),
346c      mk_Unit(ui,monitor) →
346c          ch[ {plsi,ui} ] ! monitor;
346c          pls(plsi)(pls_mer)(pls_msta)(pls_mon)(update_pls_ω(ch[ {plsi,ui} ] ?,ui)(pls_ω
346d      ... end
344     end
```

- We leave it to the reader to define the calculate_next_move function !

B.3.4.2.3 The Pump Behaviours

347. The [generic] pump behaviour internal non-deterministically alternates between
348. doing own work (...), or
349. accepting pump directives from the pipeline behaviour.
    (a) If the directive is either to start or stop pumping, then that is what happens – whereupon the pump behaviour resumes in the new pumping state.
    (b) If the directive requests the values of all monitorable attributes, then these are *gathered*, communicated to the pipeline system behaviour – whereupon the pump behaviour resumes in the "old" state.

value

```
347.   pump(π)(pump_mer)(pump_sta)(pump_mon)(pump_prgr) ≡
348.     ...
349.    ⨅ let α = ch[{plsi,π}] ? in
349.       case α of
349a.         stop_pumping ∨ pump
349a.            → pump(π)(pump_mer)(pump_sta)(pump_mon)(α)[11] end,
349b.         monitor
349b.            → let mvs = gather_monitorable_values(π,pump_mon) in
349b.              ch[{plsi,π}] ! mvs;
349b.              pump(π)(pump_mer)(pump_sta)(pump_mon)(pump_prgr) end
349.       end
```

- We leave it to the reader to defined the gather_monitorable_values function.

[11] Updating the programmable pump state to either **stop_pumping** or **pump** shall here be understood to mean that the pump is set to not pump, respectively to pump.

B.3.4.2.4 The Valve Behaviours

350. The [generic] valve behaviour internal non-deterministically alternates between
351. doing own work (...), or
352. accepting valve directives from the pipeline system.
    (a) If the directive is either to open or close the valve, then that is what happens – whereupon the pump behaviour resumes in the new valve state.
    (b) If the directive requests the values of all monitorable attributes, then these are *gathered*, communicated to the pipeline system behaviour – whereupon the valve behaviour resumes in the "old" state.

value

```
350.   valve(vi)(valv_mer)(valv_sta)(valv_mon)(valv_prgr) ≡
351.     ...
352.    ⨅ let α = ch[ {plsi,π} ] ? in
352.       case α of
352a.          open_valve ∨ close_valve
352a.              → valve(vi)(val_mer)(val_sta)(val_mon)(α)[12] end,
352b.          monitor
352b.              → let mvs = gather_monitorable_values(vi,val_mon) in
352b.                 ch[ {plsi,π} ] ! (vi,mvs);
352b.                 valve(vi)(val_mer)(val_sta)(val_mon)(val_prgr) end
352.       end
```

[12] Updating the programmable valve state to either **open_valve** or **close_valve** shall here be understood to mean that the valve is set to open, respectively to closed position.

B.3.4.3 Sampling Monitorable Attribute Values

- Static and programmable attributes are, as we have seen, *passed by value* to behaviours.
- Monitorable attributes "surreptitiously" change their values so, as a technical point, these are *passed by reference* –
- by *passing attribute type names*.

353. From the name, ηA, of a monitorable attribute and the unique identifier, u_i, of the part having the named monitorable attribute
 one can then, "dynamically", "on-the-fly",
 as the part behaviour "moves-on", retrieve the value of the monitorable attribute. This can be illustrated as follows:
354. The unique identifier u_i is used in order to retrieve, from the global parts state, σ, that identified part, p.
355. Then attr_A is applied to p.

value

353. retr_U: UI → Σ → U
353. retr_U(ui)(σ) ≡ **let** u:U · u ∈ σ ∧uid_U(u)=ui **in** u **end**
354. retr_AttrVal: UI × ηA → Σ → A
355. retr_AttrVal(ui)(ηA)(σ) ≡ attr_A(retr_U(ui)(σ))

- retr_AttrVal(...)(...)(...) can now be applied in the body of the behaviour definitions, for example in gather_monitorable_values.

B.3.4.4 System Initialisation

- System initialisation means to "morph" all manifest parts
  - into their respective behaviours,
  - initialising them with their respective attribute values.

356. The *pipeline system* behaviour is initialised and "put" in parallel with the parallel compositions of
357. all initialised *well*,
358. all initialised *pipe*,
359. all initialised *pump*,
360. all initialised *valve*,
361. all initialised *fork*,
362. all initialised *join* and
363. all initialised *sink* behaviours.[13]

[13] Plates are treated as are structures, i.e., not "behaviourised"!

value

356. pls(uid_PLS(pls))(mereo_PLS(pls))((pls))((pls))((pls))
357. || || { well(uid_U(we))(mereo_U(we))(sta_A_We(we))(mon_A_We(we))(prg_A_We(we)) | we:Well · w ∈ σ }
358. || || { pipe(uid_U(pi))(mereo_U(pi))(sta_A_Pi(pi))(mon_A_Pi(pi))(prg_A_Pi(pi)) | pi:Pi · pi ∈ σ }
359. || || { pump(uid_U(pu))(mereo_U(pu))(sta_A_Pu(pu))(mon_A_Pu(pu))(prg_A_Pu(pu)) | pu:Pump · pu ∈ σ }
360. || || { valv(uid_U(va))(mereo_U(va))(sta_A_Va(va))(mon_A_Va(va))(prg_A_Va(va)) | va:Well · va ∈ σ }
361. || || { fork(uid_U(fo))(mereo_U(fo))(sta_A_Fo(fo))(mon_A_Fo(fo))(prg_A_Fo(fo)) | fo:Fork · fo ∈ σ }
362. || || { join(uid_U(jo))(mereo_U(jo))(sta_A_Jo(jo))(mon_A_J(jo))(prg_A_J(jo)) | jo:Join · jo ∈ σ }
363. || || { sink(uid_U(si))(mereo_U(si))(sta_A_Si(si))(mon_A_Si(si))(prg_A_Si(si)) | si:Sink · si ∈ σ }

- The sta_..., mon_..., and prg_A... functions are defined in Items 317 on Slide 570.
- Note: || { f(u)(...) | u:U · u ∈ {} } ≡ ().

B.4 Index

B.5 Illustrations of Pipeline Phenomena

Figure B.3: The Planned Nabucco Pipeline: http://en.wikipedia.org/wiki/Nabucco_Pipeline

Figure B.4: Pipeline Construction

Figure B.5: Pipe Segments

Figure B.6: Valves

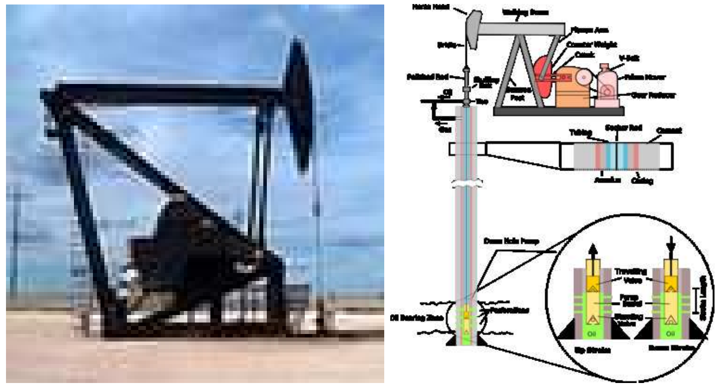

Figure B.7: Oil Pumps

Figure B.8: Gas Compressors

Figure B.9: New and Old Pigs

Figure B.10: Pig Launcher, Receiver

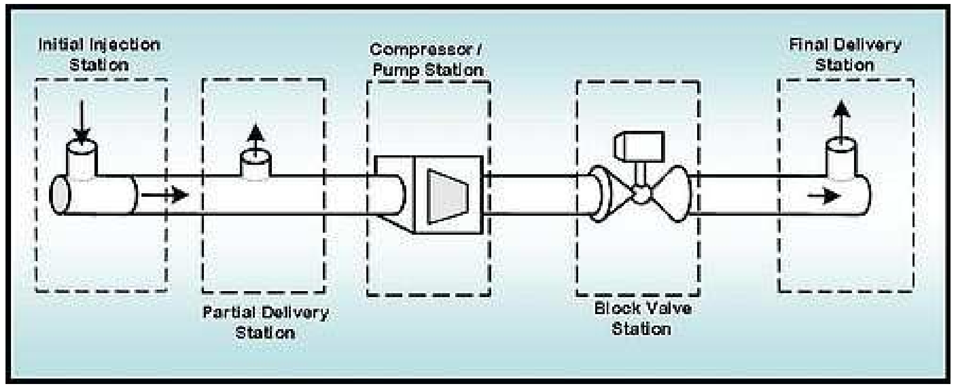

Figure B.11: Leftmost: A Well. 2nd from left: a Fork. Rightmost: a Sink

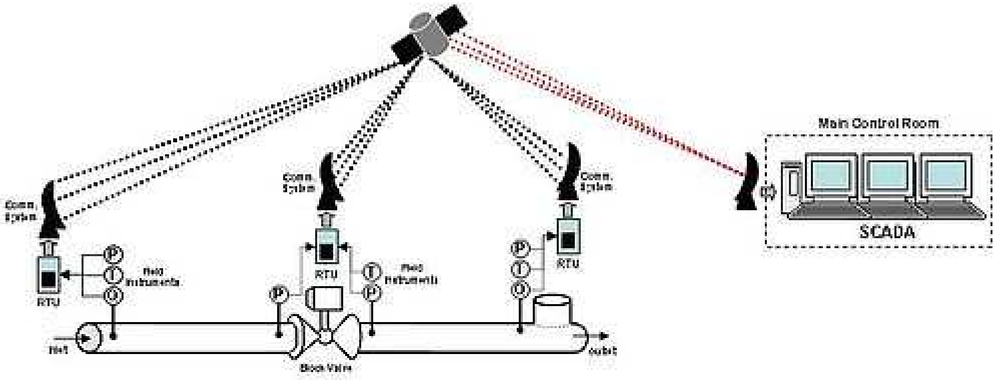

Figure B.12: A SCADA [Supervisory Control And Data Acquisition] Diagram