Domain Engineering

A Basis for Safety Critical Software

Dines Bjørner
Fredsvej 11, DK–2840 Holte, Denmark

March 18, 2014: 08:22

0. Summary

- **A Software Development Triptych:**
  - Before software can be designed
    - we must have a reasonable grasp
    - of the requirements
    - that the software is supposed to fulfil.
  - And before requirements can be prescribed
    - we must have a reasonable grasp
    - of the "underlying" application domain.

- **A Dogma:**
  - Domain engineering now becomes
    a software engineering development phase
    - in which a precise description,
    - desirably formal,
    - of the domain
    - within which the target software is to be embedded.
  - Requirements engineering then becomes a phase of software engineering
    - in which one systematically derives requirements prescriptions
    - from the domain description.
  - (Software design is then the software engineering phase
    which (also) results in code.)

- We illustrate the first element, $\mathcal{D}$, of this triptych $(\mathcal{D}, \mathcal{R}, \mathcal{S})$ by an example in which we show a description of a pipeline domain where, for example, the operations of pumps and valves are safety critical.
- We then summarise the methodological stages and steps of domain engineering.
- We finally weave considerations of ***system safety criticality*** into a section on domain facets.

- We believe this aspect of safety criticality is new:
  - We here connect safety criticality to domain engineering.
  - The study presented here need be deepened.
  - Similar connections need be made to
    - requirements engineering such as it can be "derived" from domain engineering, and to
    - the related software design.
  - That is, three distinct "layers" of safety engineering.

1. Introduction

- **A Software Development Triptych:**
  - Before software can be designed
    - we must have a reasonable grasp
    - of the requirements
    - that the software is supposed to fulfil.
  - And before requirements can be prescribed
    - we must have a reasonable grasp
    - of the "underlying" application domain.

- **Domain engineering** now becomes
  a software engineering development phase
  - in which a precise description,
  - desirably formal,
  - of the domain
  - within which the target software is to be embedded.

- **Requirements engineering** then becomes a phase of software engineering
  - ◈ in which one systematically derives
    - ⊚ requirements prescriptions
    - ⊚ from the domain description —
    - ⊚ carving out and extending, as it were, a subset of those
      - ∗ domain properties that are computable and
      - ∗ for which computing support is required.

- **Software design** is then
  - the software engineering phase
  - which results in code (and further documentation).

- We shall first
  - give a fairly large example, approximately 30 Slides,
  - of a postulated domain of (say, oil or gas) pipelines;
  - the focus will be on **endurant**s:
    - the observable **entities** that endure,
    - their **mereology**, that is, how they relate, and
    - their **attribute**s.
  - **Perdurant**s: **action**s, **event**s and **behaviour**s will be very briefly mentioned.

- We shall then
  - on the background of this substantial example,
  - outline the basical principles, techniques and tools
  - for describing domains —
  - focusig only on endurants.

- We shall review notions of **safety criticality**:
  - **safety**,
  - **failure**,
  - **error**,
  - **fault**,
  - **hazard** and
  - **risk**.
- Other notions will also be briefly characterised:
  - component and
  - system

  safety, and
  - **stake-holder**,
  - **machine** and
  - **requirements**.

- And, finally we shall detail the notion of **domain facet**s.
  - The various domain facets
    - somehow reflect domain views —
    - of logical or algebraic nature —
    - views that are shared across stake-holder groups,
    - but are otherwise clearly separable.
  - It is in connection with
    the summary explanation of respective domain facets
    that we identify respective **faults** and **hazards.**
  - The presentation is brief.

- We consider the following ideas new:
  - the idea of describing domains before prescribing requirements
  - and the idea of enumerating faults and hazards
    - as related to individual facets.
  - For the latter “discovery” we thank the organisers of ASSC 2014, notably Prof. Clive Victor Boughton.

2. An Example

- Our example is an abstraction of pipeline system endurants.
  - The presentation of the example
    - reflects a rigorous use of the domain analysis & description method outlined in Sect. 3,
    - but is relaxed with respect to not showing all – one could say intermediate – analysis steps and description texts,
      - but following stoichiometry ideas from chemistry
      - makes a few short-cuts here and there.
    - The use of the "stoichiometrical" reductions,
      - usually skipping intermediate endurant sorts,
      - ought properly be justified in each step —
      - and such is adviced in proper, industry-scale analyses & descriptions.

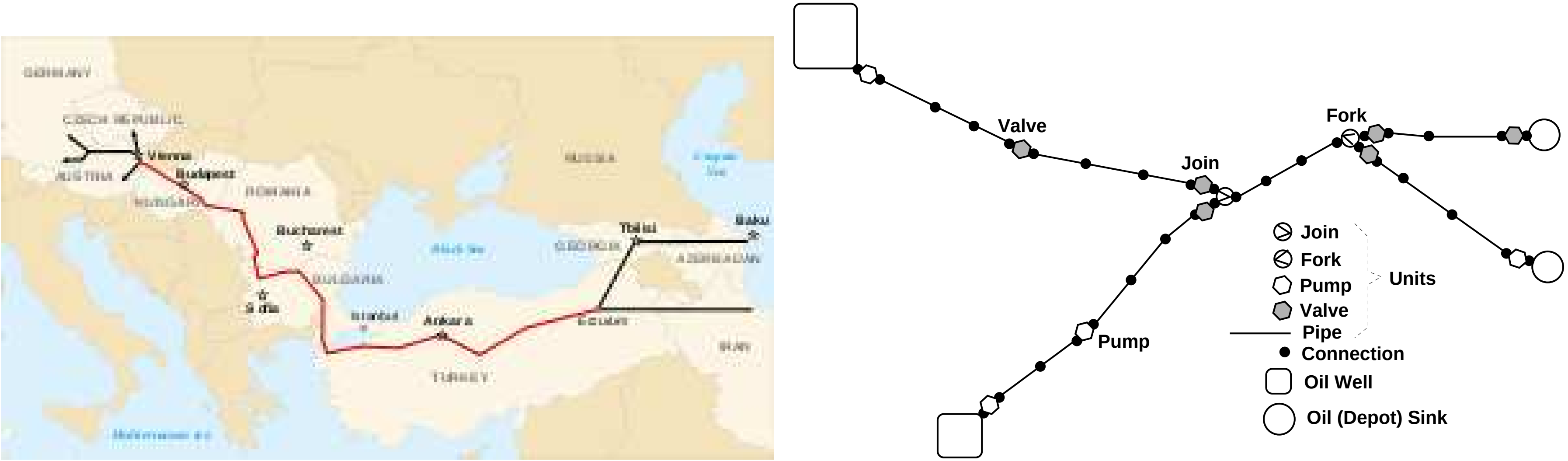

Figure 1: Pipelines. Flow is right-to-left in left figure, but left-to-right in right figure.

- The description only covers a few aspects of endurants.

Figure 2: Some oil pipeline system units: pump, pipe, valve

2.1. Parts

1. A pipeline system contains a set of pipeline units and a pipeline system monitor.
2. The well-formedness of a pipeline system depends on its mereology and the routing of its pipes.
3. A pipeline unit is either a well, a pipe, a pump, a valve, a fork, a join, or a sink unit.
4. 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.

type
1. PLS′, US, U, M
2. PLS = {| pls:PLS′•wf_PLS(pls) |}

value
2. wf_PLS: PLS → **Bool**
2. wf_PLS(pls) ≡ wf_Mereology(pls) ∧ wf_Routes(pls)
1. obs_Us: PS → U-**set**
1. obs_M: PLS → M

type
3. U = We | Pi | Pu | Va | Fo | Jo | Si
4. We :: Well
4. Pi :: Pipe
4. Pu :: Pump
4. Va :: Valv
4. Fo :: Fork
4. Jo :: Join
4. Si :: Sink

2.2. Part Identification and Mereology

2.2.1. Unique Identification

5. Each pipeline unit is uniquely distinguished by its unique unit identifier.

type
5. UI
value
5. uid_UI: U $\rightarrow$ UI
axiom
5. $\forall$ pls:PLS,u,u′:U•{u,u′}$\subseteq$obs_Us(pls)$\Rightarrow$u$\neq$u′$\Rightarrow$uid_UI(u)$\neq$uid_UI(u′)

[-1] uid_UI is the unique identifier observer function for parts u:U. It is total. uid_UI(u) yields the unique identifier of u.
[0] The axiom expresses that for all pipeline systems all two distinct units, u, u′ of such pipeline systems have distinct unique identifiers.

2.2.2. Unique Identifiers

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

value

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

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

theorem:

7. ∀ pls:PLS•**card** obs_Us(pl)=**card** xtr_UIs(pls)

[0]xtr_UIs is a total function. It extracts all unique unit identifiers of a pipeline system.

2.2.3. Mereology

8. Each unit is connected to zero, one or two other existing (formula line 8x.) input units and zero, one or two other existing (formula line 8x.) 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 sink is connected to exactly one input unit (and, hence, has no "output").

```
   type
8.     MER = UI-set × UI-set
   value
8.     mereo_U: U → MER
   axiom
8.      wf_Mereology: PLS → Bool
8.      wf_Mereology(pls) ≡
8.         ∀ u:U•u ∈ obs_Us(pls)⇒
8x.            let (iuis,ouis) = mereo_U(u) in iuis ∪ ouis ⊆ xtr_UIs(pls)∧
8.               case (u,(card iuis,card ouis)) of
8a.                  (mk_We(we),(0,1)) → true,
8b.                  (mk_Pi(pi),(1,1)) → true,
8c.                  (mk_Pu(pu),(1,1)) → true,
8d.                  (mk_Va(va),(1,1)) → true,
8e.                  (mk_Fo(fo),(1,2)) → true,
8f.                  (mk_Jo(jo),(2,1)) → true,
8g.                  (mk_Si(si),(1,0)) → true,
8.                 _ → false end end
```

2.3. Part Concepts

- An aspect of domain analysis & description that was not covered in Sect. 2 was that of derived concepts.
- Example pipeline concepts are
  - routes,
  - acyclic or cyclic,
  - circular,

  etcetera.
- In expressing well-formedness of pipeline systems
- one often has to develop subsidiary concepts such as these
- by means of which well-formedness is then expressed.

2.3.1. Pipe Routes

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

type

9. $R' = U^\omega$

9. $R = \{| \text{ r:Route}' \cdot \text{wf\_Route(r)} |\}$

10. $RD = UI^\omega$

axiom

10. $\forall$ rd:RD $\cdot$ $\exists$ r:R$\cdot$rd=descriptor(r)

value

10. descriptor: R $\rightarrow$ RD

10. descriptor(r) $\equiv$ $\langle$uid_UI(r[i])|i:**Nat**$\cdot$1$\leq$i$\leq$**len** r$\rangle$

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

value

11. adjacent: U × U → **Bool**
11. adjacent(u,u′) ≡
11. **let** (,ouis)=mereo_U(u),(iuis,)=mereo_U(u′) **in**
11. ouis ∩ iuis ≠ {} **end**

12. 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^**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 12a.–12c..

value

12. Routes: PLS → RD-**infset**
12. Routes(pls) ≡
12a.. **let** rs = ⟨⟩ ∪
12b.. {⟨uid_UI(u),uid_UI(u′)⟩|u,u′:U•{u,u′}⊆obs_Us(pls) ∧ adjacent(u,u′)}
12c.. ∪ {r^**tl** r′|r,r′:R•{r,r′}⊆rs}
12d.. **in** rs **end**

2.3.2. Well-formed Routes

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

value

13. acyclic_Route: R → **Bool**

13. acyclic_Route(r) ≡ ∼∃ i,j:**Nat**·{i,j}⊆**inds** r ∧ i≠j ∧ r[i]=r[j]

14. 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

14. $\quad$ wf_Routes: PLS → **Bool**
14. $\quad$ wf_Routes(pls) ≡
14. $\quad\quad$ non_circular(pls) ∧ are_embedded_in_well_to_sink_Routes(pls)

14. $\quad$ non_circular_PLS: PLS → **Bool**
14. $\quad$ non_circular_PLS(pls) ≡
14. $\quad\quad$ ∀ r:R•r ∈ routes(p)∧acyclic_Route(r)

15. 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
15.  well_to_sink_Routes: PLS → R-set
15.  well_to_sink_Routes(pls) ≡
15.     let rs = Routes(pls) in
15.     {r|r:R•r ∈ rs ∧ is_We(r[1]) ∧ is_Si(r[len r])} end
```

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

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

2.3.3. Embedded Routes

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

value

17. embedded_Routes: R → R-**set**

17. embedded_Routes(r) ≡

17. $\{\langle r[k] | k:\mathbf{Nat} \cdot i \leq k \leq j \rangle \mid i,j:\mathbf{Nat} \cdot i\ \{i,j\} \subseteq \mathbf{inds}(r) \wedge i \leq j\}$

2.3.4. A Theorem

18. 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:

18. $\forall$ pls:PLS $\cdot$
18. **let** rs = Routes(pls),
18. wsrs = well_to_sink_Routes(pls) **in**
18a.. rs =
18b.. wsrs $\cup$
18c.. $\cup$ $\{\{r'|r':R \cdot r' \in$ embedded_Routes$(r'')\} \mid r'':R \cdot r'' \in$ wsrs$\}$
17. **end**

2.4. Materials

19. The only material of concern to pipelines is the gas[1] or liquid[2] which the pipes transport[3].

type
19. GoL
value
19. obs_GoL: U $\rightarrow$ GoL

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

2.5. Attributes

2.5.1. Part Attributes

20. These are some attribute types:
    a. estimated current well capacity (barrels of oil, etc.),
    b. pipe length,
    c. current pump height,
    d. current valve open/close status and
    e. flow (e.g., volume/second).

type

20a.. WellCap

20b.. LEN

20c.. Height

20d.. ValSta == open | close

20e.. Flow

21. Flows can be added (also distributively) and subtracted, and
22. flows can be compared.

value

21. $\oplus, \ominus$: Flow×Flow → Flow

21. $\oplus$: Flow-**set** → Flow

22. $<, \leq, =, \neq, \geq, >$: Flow × Flow → **Bool**

23. Properties of pipeline units include
    a. estimated current well capacity (barrels of oil, etc.),
    b. pipe length,
    c. current pump height,
    d. current valve open/close status,
    e. current $\mathcal{L}$aminar in-flow at unit input,
    f. current $\mathcal{L}$aminar in-flow leak at unit input,
    g. maximum $\mathcal{L}$aminar guaranteed in-flow leak at unit input,
    h. current $\mathcal{L}$aminar leak unit interior,
    i. current $\mathcal{L}$aminar flow in unit interior,
    j. maximum $\mathcal{L}$aminar guaranteed flow in unit interior,
    k. current $\mathcal{L}$aminar out-flow at unit output,
    l. current $\mathcal{L}$aminar out-flow leak at unit output,
    m. maximum guaranteed $\mathcal{L}$aminar out-flow leak at unit output.

value

23a.. attr_WellCap: We $\rightarrow$ WellCap

23b.. attr_LEN: Pi $\rightarrow$ LEN

23c.. attr_Height: Pu $\rightarrow$ Height

23d.. attr_ValSta: Va $\rightarrow$ VaSta

23e.. attr_In_Flow$_{\mathcal{L}}$: U $\rightarrow$ UI $\rightarrow$ Flow

23f.. attr_In_Leak$_{\mathcal{L}}$: U $\rightarrow$ UI $\rightarrow$ Flow

23g.. attr_Max_In_Leak$_{\mathcal{L}}$: U $\rightarrow$ UI $\rightarrow$ Flow

23h.. attr_body_Flow$_{\mathcal{L}}$: U $\rightarrow$ Flow

23i.. attr_body_Leak$_{\mathcal{L}}$: U $\rightarrow$ Flow

23j.. attr_Max_Flow$_{\mathcal{L}}$: U $\rightarrow$ Flow

23k.. attr_Out_Flow$_{\mathcal{L}}$: U $\rightarrow$ UI $\rightarrow$ Flow

23l.. attr_Out_Leak$_{\mathcal{L}}$: U $\rightarrow$ UI $\rightarrow$ Flow

23m.. attr_Max_Out_Leak$_{\mathcal{L}}$: U $\rightarrow$ UI $\rightarrow$ Flow

2.5.2. Flow Laws

24. “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:

24. $\forall$ u:U\We\Si $\cdot$
24. sum_in_leaks(u) $\oplus$ sum_in_flows(u) =
24. attr_body_Leak$_{\mathcal{L}}$(u) $\oplus$
24. 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**

$\oplus$ {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**

$\oplus$ {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**

$\oplus$ {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**

$\oplus$ {attr_Out_Leak$_{\mathcal{L}}$(u)(ui)|ui:UI•ui ∈ ouis} **end**

25. "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:

25. $\forall$ u,u′:U·adjacent(u,u′) $\Rightarrow$
25. **let** (,ouis)=mereo_U(u), (iuis′,)=mereo_U(u′) **in**
25. **assert:** uid_U(u′) $\in$ ouis $\wedge$ uid_U(u) $\in$ iuis ′
25. attr_Out_Flow$_{\mathcal{L}}$(u)(uid_U(u′)) =
25. attr_In_Leak$_{\mathcal{L}}$(u)(uid_U(u))$\oplus$attr_In_Flow$_{\mathcal{L}}$(u′)(uid_U(u)) **end**

2.5.3. Open Routes

26. A route, r, is open

    a. if all valves, v, of the route are open and

    b. if all pumps, p, of the route are pumping.

value

26. is_open: R → **Bool**

26. is_open(r) ≡

26a.. ∀ mkPu(p):Pu · mkPu(p) ∈ **elems** r ⇒ is_pumping(p) ∧

26b.. ∀ mkVa(v):Va · mkVa(v) ∈ **elems** r ⇒ is_open(v)

2.6. Domain Perdurants

2.6.1. Actions

- We shall not formalise any specific actions.
- Informal examples of actions are:
  - opening and closing a well,
  - start and stop pumping,
  - open and close valves,
  - opening and closing a sink and
  - sense current unit flow.

2.6.2. Events

- We shall not formalise any specific events.
- Informal examples of events are:
  - empty well,
  - full sink,
  - start pumping signal to pump with no liquid material,
  - pump ignores start/stop pumping signal,
  - valve ignores opening/closing signal,
  - excessive to catastrophic unit leak, and
  - unit fire or explosion.

2.6.3. Behaviours

- We shall not formalise any specific behaviours.
- Informal examples of behaviours are:
  - start pumping and opening up valves across a pipeline system, and
  - stop pumping and closing down valves across a pipeline system.

3. Basic Domain Description

- In this section and the next we shall survey basic principles of describing, respectively,
  - domain intrinsics and other
  - domain facets.

- By an **entity** we shall understand a **phenomenon**
  - that can be **observed**, i.e., be
    - seen or
    - touched

    by humans,
  - or that can be **conceived**
    - as an **abstraction**
    - of an entity.
  - **Example**: Pipeline systems, units and materials are entities (Slide 18, Item 1.) ■

- The method can thus be said to provide the ***domain analysis prompt***:
  - is_entity
  - where is_entity(θ) holds if θ is an entity.

- A **domain** is characterised by its
  - observable, i.e., manifest *entities*
  - and their *qualities*.
- By a **quality** of an entity we shall understand
  - a property that can be given a *name* and
  - whose *value* can be
    - precisely measured by physical instruments
    - or otherwise identified.
- **Example**: **Unique identifiers** (Slide 20, Item 5.), **mereology** (Slide 22, Item 8.) and the well capacity (Slide 35, Item 20a..), pipe length (Slide 35, Item 20b..), current pump height (Slide 35, Item 20c..), current valve open/close status (Slide 35, Item 20d..) and flow (Slide 35, Item 20e..) **attributes** are qualities ■

- By a **sort** (or **type**) we shall understand
  - the largest set of entities
  - all of which have the same qualities.
- By an **endurant entity** (or just, an endurant) we shall understand
  - anything that can be observed or conceived,
  - as a "complete thing",
  - at no matter which given snapshot of time.
- Thus the method provides a *domain analysis prompt*:
  - `is_endurant` where
  - `is_endurant`(e) holds if entity e is an endurant.

- By a **perdurant entity** (or just, an perdurant) we shall understand
  - an entity
  - for which only a fragment exists
    if we look at or touch them
    at any given snapshot in time, that is,
  - were we to freeze time we would only see or touch
    a fragment of the perdurant.
- Thus the method provides a *domain analysis prompt*:
  - `is_perdurant` where
  - `is_perdurant`(e) holds if entity e is a perdurant.

- By a **discrete endurant** we shall understand something which is
  - separate or distinct in form or concept,
  - consisting of distinct or separate parts.

- Thus the method provides a ***domain analysis prompt***:
  - `is_discrete` where
  - `is_discrete`(e) holds if entity e is discrete.

- By a **continuous endurant**
  - we shall understand something which is
  - prolonged without interruption,
  - in an unbroken series or pattern.

We use the term **material** for continuous endurants.

- Thus the method provides a ***domain analysis prompt***:
  - `is_continuous` where
  - `is_continuous`(e) holds if entity e is a continuous entity.

3.0.1. Endurant Entities

We distinguish between endurant and perdurant entities.

3.0.1.1 Parts and Materials

- The manifest entities, i.e., the endurants, are called
  - parts, respectively
  - materials.
- We use the term **part** for discrete endurants,
  - that is: is_part(p)$\equiv$ is_endurant(p)$\wedge$is_discrete(p).
- We use the term **material** for continuous endurants.

- Discrete endurants are
  - either **atomic**
  - or **composite**.
- By an **atomic endurant** we shall understand
  - a discrete endurant which
  - in a given context,
  - is deemed to *not* consist of meaningful, separately observable proper **sub-parts**.
- The method can thus be said to provide the *domain analysis prompt*:
  - `is_atomic` where `is_atomic(`p`)` holds if p is an atomic part.
- **Example**: Pipeline units, U, and the monitor, M, are considered atomic ■

- By a **composite endurant** we shall understand
  - a discrete endurant which
  - in a given context,
  - is deemed to *indeed* consist of meaningful, separately observable proper sub-parts.
- The method can thus be said to provide the *domain analysis prompt*:
  - `is_composite` where `is_composite(`p`)` holds if p is an a composite part.
- **Example**: The pipeline system, PLS, and the set, Us, of pipeline units are considered composite entities ■

3.0.1.2 Part Observers

- From atomic parts we cannot observe any sub-parts.
- But from composite parts we can.
- For composite parts, p, the *domain description prompt*
  - ⟐ observe_part_sorts(p)
- yields some *formal description text* according to the following *schema*:

type P_1, P_2, ..., P_n;[4]
value **obs**_P_1: P→P_1, **obs**_P_2: P→P_2,...,**obs**_P_n: P→P_n;[5]

[4] This RSL **type** clause defines P_1, P_2, ..., P_n to be sorts.
[5] Thus RSL **value** clause defines n function values. All from type P into some type P_i.

- where
  - sort names P_1, P_2, ..., P_n
  - are chosen by the domain analyser,
  - must denote disjoint sorts, and
  - may have been defined already,
  - but not recursively
  - A **proof obligation** may need be **discharged** to secure **disjointness** of sorts.

- **Example**: Three formula lines (Slide 18, Items 1.) illustrate the basic sorts (PLS′, US, U, M) and observers (obs_US, obs_M) of pipeline systems ■

3.0.1.3 Sort Models

- A part sort is an **abstract type**.
  - Some part sorts, P, may have a concrete type model, T.
  - Here we consider only two such models:
    - one model is as sets of parts of sort A: T = A-**set**;
    - the other model has parts being of either of two or more alternative, disjoint sorts: T=P1|P2|...|PN.
- The *domain analysis prompt*:
  - `has_concrete_type`(p)
- holds if part p has a concrete type.
- In this case the *domain description prompt*
  - `observe_concrete_type`(p)

- yields some *formal description text* according to the following *schema*,
  - either

    type P1, P2, ..., PN, T = $\mathcal{E}$(P1,P2,...,PN)[6]
    value **obs**_T: P → T[7]

    where $\mathcal{E}$(...) is some type expression over part sorts and where P1,P2,...,PN are either (new) part sorts or are auxiliary (abstract or concrete) types[8];
  - or:

    type
    T = P1 | P2 | ... | PN[9]
    P_1, P_2, ..., P_n
    P1 :: mkP1(P_1), P2 :: mkP2(P_2), ..., PN :: mkPN(P_n) [10]
    value
    obs_T: P → T[11]

6 The concrete type definition T = $\mathcal{E}$(P1,P2,...,PN) define type T to be the set of elements of the type expressed by type expression $\mathcal{E}$(P1,P2,...,PN).

7 **obs**_T is a function from any element of P to some element of T.

8 The *domain analysis prompt*: `sorts_of`(t) yields a subset of {P1,P2,...,PN}.

- **Example**: **obs_T**: P → T
  is exemplified by obs_Us: PS → U-**set** (Slide 18, Item 1.),
- T = P1 | P2 | ... | PN
  by We | Pu | Va | Fo | Jo | Si (Slide 18, Item 3.) and
- P1 :: mkP1(P_1), P2 :: mkP2(P_2), ..., PN :: mkPN(P_n)
  by (Slide 18, Item 4.) ■

[9] A|B is the union type of types A and B.
[10] Type definition A :: mkA(B) defines type A to be the set of elements mkA(b) where b is any element of type B
[11] **obs**_T is a function from any element of P to some element of T.

3.0.1.4 Material Observers

- Some parts p of sort P may contain material.
  - The *domain analysis prompt*
    - `has_material(p)`
  - holds if composite part p contains one or more materials.
- The *domain description prompt* `observe_material_sorts(p)`
- yields some *formal description text* according to the following *schema*:

  type M_1, M_2, ..., M_m;
  value **obs**_M_1: P $\rightarrow$ M_1, **obs**_M_2: P $\rightarrow$ M_2, ..., **obs**_M_m: P $\rightarrow$ M_m;

  - where values, m_i, of type M_i satisfy `is_material`(m) for all i;
  - and where M_1, M_2, ..., M_m must be disjoint sorts.

- **Example**: We refer to Slide 34, Item 19. ■

3.0.2. Endurant Qualities

- We have already, above, treated the following properties of endurants:
  - is_discrete,
  - is_continuous,
  - is_atomic,
  - is_composite and
  - has_material.
- We may think of those properties as **external qualities**.
- In contrast we may consider the following **internal qualities**:
  - has_unique_identifier (parts),
  - has_mereology (parts) and
  - has_attributes (parts and materials).

3.0.2.1 Unique Part Identifiers

- Without loss of generality we can assume that every part has a unique identifier[12].
  - A **unique part identifier** (or just unique identifier) is a further undefined, abstract quantity.
  - If two parts are claimed to have the same unique identifier then they are identical.
- The *domain description prompt*: `observe_unique_identifier`(p)
- yields some *formal description text* according to the following *schema*:

  type PI;
  value **uid**_P: P → PI;

- **Example**: We refer to Slide 20, Item 5. ■

[12] That is, `has_unique_identifier`(p) for all parts p.

3.0.2.2 Part Mereology

- By **mereology** [Lesniewski1] we shall understand
  - the study, knowledge and practice of
  - parts,
  - their relations to other parts
  - and "the whole".
- Part relations are such as:
  - two or more parts being connected,
  - one part being embedded within another part, and
  - two or more parts sharing attributes.

- The *domain analysis prompt*:
  - **has_mereology**(p)
- holds if the part p is related to some others parts $(p_a, p_b, \ldots, p_c)$.
- The *domain description prompt*: **observe_mereology**(p) can then be invoked and
- yields some *formal description text* according to the following *schema*:

type MT = $\mathcal{E}$(PI$_A$,PI$_B$,...,PI$_C$);
value **mereo**_P: P → MT;

where $\mathcal{E}(...)$ is some type expression over unique identifier types of one or more part sorts.

- Mereologies are expressed in terms of structures of unique part identifiers.
- Usually mereologies are constrained. Constraints express
  - that a mereology's unique part identifiers must indeed reference existing parts, but also
  - that these mereology identifiers "define" a proper structuring of parts.
- **Example**: We refer to Items 8.–8g.. Slides 22–23 ■

3.0.2.3 Part and Material Attributes

- Attributes are what really endows parts with qualities.
  - The external properties[13]
  - are far from enough to distinguish one sort of parts from another.
  - Similarly with unique identifiers and the mereology of parts.
- We therefore assume, without loss of generality, that
  - every part, whether discrete or continuous,
  - whether, when discrete, atomic or composite,
  - has at least one attribute.

13

- is_discrete,
- is_continuous,
- is_atomic,
- is_composite
- has_material.

- By an **endurant attribute**, we shall understand
  - a property that is associated with an endurant e of sort E,
    - and if removed from endurant e,
    - that endurant would no longer be endurant e
    - (but may be an endurant of some other sort E'); and
  - where that property itself has no physical extent (i.e., volume),
    - as the endurant may have,
    - but may be measurable by physical means.

- The *domain description prompt* `observe_attributes`(p)
- yields some *formal description text* according to the following *schema*:

> **type** A_1, A_2, ..., A_n, ATTR;
> **value** **attr**_A_1:P→A_1, **attr**_A_2:P→A_2, ..., **attr**_A_n:P→A_n,
> **attr**_ATTR:P→ATTR;

- where **for** ∀ p:P, **attr**_A_i(**attr**_ATTR(p)) ≡ **attr**_A_i(p).
- **Example**: We refer to Slides 35–38 ■

3.0.3. Perdurant Entities

- We shall not cover the principles, tools and techniques
  - for "discovering", analysing and describing
  - domain actions, events and behaviours
  - to anywhere the detail with which
  - the "corresponding" principles, tools and techniques
  - were covered for endurants.

- But we shall summarise one essence for the description of perdurants.
- There is a notion of **state**.
  - Any composition of parts having dynamic qualities can form a state.
  - Dynamic qualities are qualities that may change.
  - Examples of such qualities are
    - the mereology of a part, and
    - part attributes whose value may change.

- There is the notion of **function signature**.
  - A function signature, f: A $(\rightarrow|\xrightarrow{\sim})$ R,
    - gives a name, say f, to a function,
    - expresses a type, say T_A, of the arguments of the function,
    - expresses whether the function is total $(\rightarrow)$ or partial $(\xrightarrow{\sim})$, and
    - expresses a type, say T_R, of the result of the function.

- There is the notion of **channel**s of synchronisation & communication between behaviours.
  - Channels have names, e.g., ch, ch_i, ch_o.
  - Channel names appear in the signature of behaviour functions: **value** b: A → **in** ch_i **out** ch_o R.
  - **in** ch_i indicates that behaviour b may express willingness to communicate an input message over channel ch_i; and
  - **out** ch_o indicates that behaviour b may express an offer to communicate an output message over channel ch_o.

- There is a notion of **function pre/post-conditions**.
  - A function pre-condition is a predicate over argument values.
    -
    -
  - A function post-condition is a predicate over argument and result values.
    -
    -
  -

- Action signatures
  - include states, Σ, in both
  - arguments, $\mathsf{A}\times\Sigma$, and results, Σ:
  - f: $\mathsf{A}\times\Sigma\rightarrow\Sigma$;
  - f denotes a function in the function space $\mathsf{A}\times\Sigma\rightarrow\Sigma$.
- Action pre/post-conditions:

$$\underline{\textbf{value}}\ \mathrm{f}(\mathrm{a},\sigma)\ \underline{\textbf{as}}\ \sigma';\ \underline{\textbf{pre}}:\ \mathcal{P}_f(\mathrm{a},\sigma);\ \underline{\textbf{post}}:\ \mathcal{Q}_f(\mathrm{a},\sigma,\sigma')$$

  - have predicates $\mathcal{P}_f$ and $\mathcal{Q}_f$
  - delimit the value of f within that function space;
  - $\mathcal{P}_f$
  - $\mathcal{Q}_f$

- Event signatures
  - are typically predicates from pairs of before and after states:
  - e: $\Sigma\times\Sigma\rightarrow\underline{\textbf{Bool}}$.
- Event pre/post-conditions

$$\underline{\textbf{value}}\ \text{e}: \Sigma\times\Sigma\rightarrow\underline{\textbf{Bool}};\ \text{e}(\sigma,\sigma') \equiv \mathcal{P}_e(\sigma) \wedge \mathcal{Q}_e(\sigma,\sigma')$$

  - have predicates $\mathcal{P}_e$ and $\mathcal{Q}_e$
  - delimit the value of e within the $\Sigma\times\Sigma\rightarrow\underline{\textbf{Bool}}$ function space;
  - $\mathcal{P}_e$ characterises states leading to event e;
  - $\mathcal{Q}_e$ characterises states, σ', resulting from the event caused by σ.

- In principle we can associate a behaviour with every part of a domain.
  - Parts, p, are characterised by their unique identifiers, pi:PI and a state, attrs:ATTRS.
  - We shall, with no loss of generality, assume part behaviours to be never-ending.
  - The unique part identifier, pi:PI, and its the part mereology, say {pi_1,pi_2,...,pi_n}, determine a number of channels
  - {chs[pi,pi_j]|j:{1,2,...,n}}
  - able to communicate messages of **type** M.

- Behaviour signatures:

  b: pi:PI × ATTR → **in** in_chs **out** out_chs **Unit**

  - then have
    - input channel expressions in_chs and
    - output channel expressions out_chs
    - be suitable predicates over
    - {chs[pi,pi_j]|j:{1,2,...,n}}.
  - **Unit** designate that b denote a never-ending process.
- We omit dealing with behaviour pre-conditions and invariants.

4. Interlude

- We have covered one aspect of the modelling of one set of domain entities, the intrinsic facets of endurants.
  - For the modelling of perdurants we refer to a forthcoming report.
- In the next section we shall survey the modelling of further domain facets.
- We shall accompany this survey to a survey of safety issues.
- To do so in a reasonably coherent way we need establish a few concepts:
  - the ***safety*** notions of
    - ***failure***,
    - ***error*** and
    - ***fault***;
  - the notion of ***stakeholder*** and
  - the notion of ***requirements***.

4.1. Safety-related Concepts

Some characterisations are:

4.1.0.1 Safety

By ***safety***, in the context of a domain being dependable, we mean

- some measure of continuous delivery of service of
  - either correct service,
  - or incorrect service after benign failure,
- that is: measure of time to catastrophic failure.

4.1.0.2 Failure

- A domain *failure* occurs
- when the delivered service
- deviates from fulfilling the domain function,
- the latter being what the domain is aimed at.

4.1.0.3 Error

- An *error*
- is that part of a domain state
- which is liable to lead to subsequent failure.
- An error affecting the service
- is an indication that a failure occurs or has occurred.

4.1.0.4 Fault

- The adjudged (i.e., the 'so-judged')
- or hypothesised cause of an error
- is a *fault*.

4.1.0.5 Hazard

- A **hazard** is
  - any source of potential damage, harm or adverse health effects
  - on something or someone under certain conditions at work.

4.1.0.6 Risk

- A **risk** is
  - the chance or probability that a person
  - will be harmed or experience an adverse health effect
  - if exposed to a hazard.
  - It may also apply to situations with property or equipment loss.

4.1.0.7 Faults and Hazards

- The concept of hazard is not the same as the concept of fault.
- *"System safety takes a larger view of hazards than just failures*[14]*:*
  - *Hazards are not always caused by failures, and all failures do not cause hazards.*
  - *Serious accidents have occurred*
    - *while system components*
    - *were all functioning exactly as specified,*
    - *that is, without failure.*
  - *If failures only are considered in a safety analysis, many potential accidents will be missed.*
  - *In addition, the engineering approaches to preventing failures*
    (increasing reliability)
    and preventing hazards
    (increasing safety)
    are different and sometimes conflict."

[14] Leveson: [White Paper on Approaches to Safety Engineering]

4.2. System and Component Safety

- There appears to be a number of safety concepts [15]:
  - component safety,
  - industrial safety,
  - reliability, and
  - system safety.
- We shall focus on component and system safety.

[15] Leveson: [White Paper on Approaches to Safety Engineering]

4.2.0.1 Component

- By a **component** we shall understand
  - ⟐ basically the same as an atomic part
  - ⟐ together with actions, events and behaviours
    - ⊚ whose state is anchored
    - ⊚ in one or more attributes of that part,
  - ⟐ such that these actions, etc.,
    do nor involve other component or [sub]system states.
  - ⟐ That is, "componentry"
    excludes considerations of shared attributes.

4.2.0.2 System

- By a **system** or **sub-system** we shall understand
  - basically the same as a composite part
  - together with actions, events and behaviours
    - whose state is anchored
    - in one or more attributes
      - of that part
      - as well as of one or more other parts.
  - That is, "system-hood"
    presumes considerations of shared attributes.

4.2.0.3 System Safety

- *"The primary concern of system safety* [16]
  - *is the management of hazards:*
    - *their identification,*
    - *evaluation,*
    - *elimination, and*
    - *control*

  through

    - *analysis,*
    - *design and*
    - *management procedures."*

[16] Leveson: [White Paper on Approaches to Safety Engineering]

- *"System safety deals with systems as a whole rather than with subsystems or components* [17]*:*
  - ⟐ *Safety is an emergent property of systems,*
  - ⟐ *not a component property.*
  - ⟐ *One of the principle responsibilities of system safety is*
  - ⟐ *to evaluate the interfaces between the system components*
  - ⟐ *and determine the effects of component interaction,*
  - ⟐ *where the set of components includes*
    - *humans,*
    - *machines, and*
    - *the environment."*
- The system interfaces are given by the mereology.

[17] Leveson: [White Paper on Approaches to Safety Engineering]

4.2.0.4 Component Safety

- For a component,
- that is, an atomic part,
- we can, at most, speak of faults
- when considering safety.[18]

[18] The borderline between hazards that are not faults and faults is too vague.

4.3. Stake-holder

- By a **domain stake-holder** we shall understand
  - a person, or a group of persons, “united” somehow in their common interest in, or dependency on the domain; or
  - an institution, an enterprise, or a group of such, (again) characterised (and, again, loosely) by their common interest in, or dependency on the domain.

- **Examples**: The following are examples of pipeline stake-holders:
  - the owners of the pipeline,
  - the oil or gas companies using the pipeline,
  - the pipeline managers and workers,
  - the owners and neighbours of the lands occupied by the pipeline,
  - the citizens possibly worried about gas- or oil pollution,
  - the state authorities regulating and overseeing pipelining,
  - etcetera ■

4.4. Machines and Requirements

4.4.1. Machine

- By the **machine** we shall understand
  - the combination of
  - hardware, say computers and communication, and
  - software.

4.4.2. Requirements

- By a **requirements** we understand (cf. IEEE Standard 610.12):
  - *"A condition or capability needed by a user to solve a problem or achieve an objective"* .
- We shall think only of requirements as requirements to a machine.
- We can now "repeat" the definitions
  - of safety, failure, error and fault given above,
  - but now with the term
    - 'domain' replaced by the term 'machine'
    - (sometimes with the term 'domain+machine').
- This then becomes the context in which most safety criticality is discussed.

- We shall not cover requirements in this talk.
- We refer to
  `[Bjørner: From Domains to Requirements, 2008]`.
  - That paper describes how to "derive"
  - systematically, but, of course, not automatically
  - major parts of requirements prescriptions from a domain descriptions.
- Thus we shall not cover the classical approach to safety analysis.
  - Instead we shall cover what we think is a novel approach to safety analysis.
  - One in which first get an as complete as possible overview of "all" safety aspects of a domain.

5. Domain Facets and Safety Criticality

5.1. Introductory Notions

5.1.1. Facet

- By a **domain facet** we shall understand
  - one amongst a finite set of generic ways
  - of analysing a domain:
  - a view of the domain,
  - such that the different facets cover conceptually different views,
  - and such that these views together cover the domain.

- We shall in this talk distinguish between the following facets:
  - *intrinsics*,
  - *support technologies*,
  - *human behaviour*,
  - *rules &*[19] *regulations* and
  - *organisation & management*.

[19] We use the ampersand '&' between terms A and B to emphasize that we mean to refer to one subject, the conjoint $A\&B$

5.1.2. Safety Criticality

- *Safety critical systems*
  - *are those systems whose failure may result in*
  - *the loss of life,*
  - *significant property damage or*
  - *damage to the environment.*[20]

[20] John C. Knight: Safety Critical Systems: Challenges and Directions http://www.-cs.virginia.edu/~jck/publications/knight.state.of.the.art.summary.pdf

- For each of the domain facet categories we shall look for a corresponding, domain-specific category of hazards.
  - That is, we shall view safety criticality in potentially three steps:
    - from the point of view of the domain in which a computing system is to be inserted, hence first developed,
    - from the point of view of the requirements prescribed for such a system, and
    - from the point of view of the machine (i.e., hardware + software) design of that system.
  - In this talk we shall only consider the first step.

5.2. Intrinsics

- By **domain intrinsics** we shall understand
  - those phenomena and concepts of a domain which are basic to any of the other facets (listed earlier and treated, in some detail, below),
  - with such domain intrinsics initially covering at least one specific, hence named, stake-holder view.
- **Example**: The introductory example focused on
  - the intrinsics of pipeline systems as well as
  - some derived concepts (routes etc.) ■

- **Hazards**: The following are examples of hazards based sôlely on the intrinsics of the domain:
  - environmental hazards:
    - destruction of one or more pipeline units due to
    - an earth quake, an explosion, a fire or something "similar"
    - occurring in the immediate neighbourhood of these units;
  - design faults:
    - the pipeline net is not acyclic;
  - etcetera ■

- Intrinsics hazards are such which violate the well-formedness of the domain.
  - A "domain description" is presented, but it is not a well-formed domain description.
  - One could claim that
    - whichever (event) falls outside the intrinsics domain description,
    - whether it violates well-formedness criteria for domain parts
    - or action, event or behaviour pre/post-conditions,
    - is a hazard.
  - In the context of system safety we shall take the position that
    - explicitly identified hazards
    - must be described,
    - also formally.[21]

[20]

* We refer to the main example.
* More specifically to the well-formedness of pipeline systems as expressed in wf_PLS (Slide 18, Item 2.).
* We express hazards of the intrinsics of pipeline systems by named predicates over PLS′ and not PLS.

5.3. Support Technologies

- By domain **support technology** we shall understand
  - technological ways and means of implementing
  - certain observed phenomena or
  - certain conceived concepts.
- The facet of support technology, as a concept,
  - is related to actions of specific parts;
  - that is, a part may give rise to one or more support technologies,
  - and we say that the support technologies 'reside' in those parts.

- **Examples**:
  - wells are, in the intrinsics facet description abstracted as atomic units but in real instances they are complicated (composite) entities of pumps, valves and pipes;
  - pumps are similarly, but perhaps not as complicated complex units;
  - valves likewise; and
  - sinks are, in a sense, the inverse of wells ■

- **Faults**:
  - a pump may fail to respond to a ***stop pump*** signal; and
  - a valve may fail to respond to an ***open valve*** signal ■
- I think it is fair to say that
  - most papers on the design of safety critical software are on
  - software for the monitoring & control of support technology.

- Describing causes of errors is not simple.
  - With today's formal methods tools and techniques[21]
  - quite a lot can be formalised — but not all!

[21] These tools and techniques typically include

- two or more formal specification languages, for example:
  - **VDM**,
  - **DC**,
  - **Event-B**,
  - **RAISE/RSL**,
  - **TLA+** and
  - **Alloy**;
- one or more theorem proving tools, for example:
  - **ACL**,
  - **Coq**,
  - **Isabelle/HOL**,
  - **STeP**,
  - **PVS** and
  - **Z3**;
- a model-checker, for example:
  - **SMV** and
  - **SPIN/Promela**;
- and other such tools and techniques.

5.4. Human Behaviour

- A proper domain description includes humans as both
  - (usually atomic) parts and
  - the behaviours that we (generally) “attach” to parts.
- **Examples**: The human operators that
  - operate wells, valves, pumps and sinks;
  - check on pipeline units;
  - decide on the flow of material in pipes,
  - etcetera ■

- By domain **human behaviour** we shall understand
  - any of a quality spectrum of humans[22] carrying out assigned work:
    - from (i) *careful*, *diligent* and *accurate*,

    via
    - (ii) *sloppy* dispatch, and
    - (iii) *delinquent* work,

    to
    - (iv) outright *criminal* pursuit.

[22] — in contrast to technology

- Typically human behaviour focus on actions and behaviours that are carried out by humans.
  - The intrinsics description of actions and behaviours
  - focus sôlely on intended, careful, diligent and accurate performance.
- **Hazards**: This leaves "all other behaviours" as hazards!
- Proper hazard analysis, however, usually
  - explicitly identifies failed human behaviours,
  - for example, as identified deviations from
  - described actions etc.
- Hazard descriptions thus follows from "their corresponding" intrinsics descriptions ■

5.5. Rules & Regulations

- Rules and regulations come in pairs $(\mathcal{R}_u, \mathcal{R}_e)$.

5.5.1. Rules

- By a domain **rule** we shall understand some text
  - which prescribes how people are, or equipment is,
  - "expected" (for "..." see below) to behave
  - when dispatching their duty,
  - respectively when performing their function.
- **Example**: There are rules for operating pumps. One is:
  - A pump, p, on some well-to-sink route $r = r'^\frown\langle p\rangle^\frown r''$,
  - may not be started
  - if there does not exist an open, embedded route r''' such that $\langle p\rangle^\frown r'''$
  - ends in an open sink ■

- **Hazards**: when stipulating "expected" (as above)
  - the rules more or less implicitly
  - express also the safety criticality:
  - that is, when people are, or equipment is,
  - behaving erroneously ■
- **Example**: A domain rule which states, for example,
  - that a pump, p, on some well-to-sink route $r = r'^\frown\langle p\rangle^\frown r''$,
  - may be started even
  - if there does not exist an open, embedded route r''' such that $\langle p\rangle^\frown r'''$
  - ends in an open sink

is a hazardous rule ■

- **Modelling Rules:**
  - We can model a rule by giving it
    - both a syntax
    - and a semantics.
  - And we can choose to model the semantics of a rule, $\mathbb{R}_u$,
    - as a predicate, $\mathcal{P}$, over pairs of states: $\mathcal{P} : \Sigma\times\Sigma\rightarrow$**Bool**.
    - That is, the meaning, $\mathcal{M}$, of $\mathbb{R}_u$ is $\mathcal{P}$.
      - An action or an event has changed a state σ into a state σ'.
      - If $\mathcal{P}(\sigma,\sigma')$ is **true**
        it shall mean that the rule as been obeyed.
      - If it is **false** it means that the rule has been violated.

5.5.2. Regulations

- By a domain **regulation** we shall understand
  - some text which “prescribe” (“. . . ”, see below)
  - the remedial actions that are to be taken
  - when it is decided
  - that a rule has not been followed
  - according to its intention .

- **Example**: There are regulations for operating pumps and valves:
  - Once it has been discovered that a rule is hazardous
    - there should be a regulation which
      - starts an administrative procedure
        which ensures that the rule is replaced; and
      - starts a series of actions which somehow brings
        the state of the pipeline into one which poses no danger
        and then applies a non-hazard rule ■

- **Hazards**: when stipulating "prescribe"
  - regulations express requirements
  - to emerging hardware and software ■

- **Modelling Regulations:**
  - We can model a regulation by giving it
    - both a syntax
    - and a semantics.
  - And we can choose to model the semantics of a regulation, $\mathbb{R}_e$,
    - as a state-transformer, $\mathcal{S}$, over pairs of states: $\mathcal{S} : \Sigma\times\Sigma\rightarrow\Sigma$.
    - That is, the meaning, $\mathcal{M}$, of $\mathbb{R}_e$ is $\mathcal{S}$.
    - A state-transformation $\mathcal{S}(\sigma,\sigma')$
    - for rule $\mathbb{R}_u$
    - results in a state σ'' where:
      - if $\mathcal{P}(\sigma,\sigma')$ is **true**
      - then $\sigma' = \sigma''$,
      - else σ'' is a corrected state such that $\mathcal{P}(\sigma,\sigma'')$ is **true**.

5.5.3. Discussion

- *Where do rules & regulations reside ?"* That is,
  - *"Who checks that rules are obeyed ?"* and
  - *"Who ensures that regulations are applied when rules fail ?"*
- Are some of these checks and follow-ups relegated
  - to humans (i.e., parts) or
  - to machines (i.e., "other" parts) ?
- that is, to the behaviour of part processes ?
- The next section will basically answer those questions.

5.6. Organisation & Management

- To properly appreciate this section we need remind the reader of concepts introduced earlier in this talk.
  - With parts we associate
    - mereologies,
    - attributes and
    - behaviours.
  - Support technology
    - is related to actions and
    - these again focused on parts.
  - Humans are often modelled
    - first as parts,
    - then as their associated behaviour.

- It is out of this seeming jigsaw puzzle of
  - parts,
  - mereologies,
  - attributes,
  - humans,
  - rules and
  - regulations

  that we shall now form and model the concepts of
  - organisation and
  - management.

5.6.1. Organisation

- By domain **organisation** we shall understand
  - one or more partitionings of resources
    - where resources are usually representable as parts and materials and
    - where usually a resource belongs to exactly one partition;
  - such that n such partitionings typically reflects
    - strategic (say partition π_s),
    - tactical (say partition π_t), respectively
    - operational (say partition π_o)

    concerns (say for $n = 3$),
  - and where "descending" partitions,
    - say π_s, π_t, π_o,
    - represents *coarse, medium* and *fine* partitions, respectively ∎

- **Examples**: This example only illustrates production aspects.
  - At the strategic level one may partition a pipeline system into
    - just one component:
      the entire collection of all pipeline units, π.
  - At the tactical level one may further partition the system into
    - the partition of all wells, π_{ws},
    - the partition of all sinks, π_{ss}, and
    - a partition of all pipeline routes, $\pi_{\ell s}$, that
      - $\pi_{\ell s}$, is the set of all routes of π
      - excluding wells and sinks.
  - At the organisational level may further partition the system into
    - the partitions of individual wells, π_{w_i} ($\pi_{w_i} \in \pi_{ws}$),
    - the partitions of individual sinks, π_{s_j} ($\pi_{s_i} \in \pi_{ws}$) and
    - the partitions of individual pipeline routes, π_{r_k} ($\pi_{\ell_i} \in \pi_{\ell s}$) ■

- A domain organisation serves
  - to structure management and non-management staff levels and
  - the allocation of
    - strategic,
    - tactical and
    - operational

    concerns across all staff levels;
  - and hence the "lines of command":
    - who does what, and
    - who reports to whom,
      - administratively and
      - functionally.

- Organisations are conceptual parts, that is,
  - partitions are concepts,
  - they are conceptual parts
  - in addition, i.e., adjoint to physical parts.
- They serve as “place-holders” for management.

- **Modelling Organisations:**
  - We can normally model an organisation as an attribute of some, usually composite, part.
    - Typically such a model would be in terms of the one or more partitionings of unique identifiers, $\pi:\Pi$, of domain parts, p:P.
    - For example:

type
  ORG = Str × Tac × Ope × ...
  Str, Tac, Ope = (Π-**set**)-**set**
value
  attr_ORG: P → ORG
axiom
  $\mathcal{P}$: ORG → ... → **Bool**

where we leave the details of the partitionings Str, Tac, Org, ... and the axiom governing the individual partitionings and their relations for further analysis.

- **Faults** and **Hazards**:
  - There are erroneous and there are risky organisations.
  - An **erroneous** organisation is, for example, one in which
    - one or more partitions are left isolated
    - with respect to there being no management "tow-holder".
  - A **hazardous** organisation is, for example, one
    - that consists of too many partitionings,
    - whereby related management becomes confused ■

5.6.2. Management

- By domain **management** we shall understand such people who (such decisions which)
  - determine, formulate and set standards concerning
    - strategic,
    - tactical and
    - operational

    decisions;
  - who ensure that these decisions are passed on to (lower) levels of management, and to floor staff;
  - who make sure that such orders, as they were, are indeed carried out;
  - who handle undesirable deviations in the carrying out of these orders cum decisions;
  - and who "backstops" complaints from lower management levels and from floor staff.

- **Example**: [Cf. examples on Slide 129].
  - At the strategic level there is the
    - overall management of the pipeline system.
  - At the tactical level there may be the management of
    - all wells;
    - all sinks;
    - specific (disjoint) routes.
  - At the operational there may then be the management of
    - individual wells,
    - individual sinks, and
    - individual groups of valves and pumps ■

- **Modelling Management:**
  - Some parts are associated with strategic management.
    - They will have their unique identifiers, $\pi : \Pi$, belong to some partition in an str:Str.
  - Other parts are associated with tactical management.
    - They will have their unique identifiers, $\pi : \Pi$, belong to some partition in a corresponding tac:Tac.
  - Yet other parts are associated with operational management.
    - They will have their unique identifiers, $\pi : \Pi$, belong to some partition in the corresponding ope:Ope.

- The "management" parts have their attributes form corresponding states (σ:Σ).

  type

  Σ_{STR}, Σ_{TAC}, Σ_{OPE},

- An idealised rendition of management actions is:

  value

  $\text{action}_{Strategic}$: $\Sigma_{STR} \rightarrow \Sigma_{TAC} \rightarrow \Sigma_{OPE} \rightarrow \Sigma_{STR}$
  $\text{action}_{Tactical}$: $\Sigma_{STR} \rightarrow \Sigma_{TAC} \rightarrow \Sigma_{OPE} \rightarrow \Sigma_{TAC}$
  $\text{action}_{Operational}$: $\Sigma_{STR} \rightarrow \Sigma_{TAC} \rightarrow \Sigma_{OPE} \rightarrow \Sigma_{OPE}$

- $\mathsf{action}_{Strategic}$ expresses that strategic management considers the "global" state $(\Sigma_{STR}\times\Sigma_{TAC}\times\Sigma_{OPE})$ but potentially changes only the "strategy" state.
- $\mathsf{action}_{Tactical}$ expresses that tactical management considers the "global" state $(\Sigma_{STR}\times\Sigma_{TAC}\times\Sigma_{OPE})$ but potentially changes only the "tactical" state.
- $\mathsf{action}_{Operational}$ expresses that tactical management considers the "global" state $(\Sigma_{STR}\times\Sigma_{TAC}\times\Sigma_{OPE})$ but potentially changes only the "operational" state.

- We can normally model management as part of the behavioural model of some, usually composite part.
  - Typically such a model would be in terms communication procedures between managers, p:P, and their immediate subordinates, $\{p_1{:}P_1, p_2{:}P_2, \ldots, p_n{:}P_N\}$:
  - For example:

channel mgt:$\{\{\pi,\pi_j\}|\pi_j$:$PI_j \cdot \pi_j \in \ldots\}$:M
value
p: π:Π × pt:P → **in**,**out** $\{\{\pi,\pi_j\}|\pi_j$:$PI_j \cdot \pi_j \in \ldots\}$ **Unit**
p(π,pt) ≡ ...

[management orders staff]

⌈⌉ **let** (π_j,m) = $\text{query}_{\text{boss}}$(p) **in**
m ! mgt[$\{\pi,\pi_j\}$]!m ;
p(π,action_{down_s}(pt,m)) **end**

[management "listens" to staff]

⌈⌉ **let** (π_j,m) = ⌈⌉ {mgt[$\{\pi,\pi_j\}$]? | ... } **in**
p(π,action_{down_r}(pt,m)) **end**

[management reports to boss]

⌈⌉ **let** (π_{boss},m) = $\text{query}_{\text{staff}}$(pt) **in**
m ! mgt[$\{\pi,\pi_{\text{boss}}\}$]!m ;
p(π,action_{ups_s}(pt,m)) **end**

[management "listens" to boss]

⌈⌉ **let** (π_{boss},m) = ⌈⌉ {mgt[$\{\pi,\pi_{\text{boss}}\}$]? | ... } **in**
p(π,action_{ups_s}(pt,m)) **end** ...

- **Hazards**: [Cf. faults and hazards, Slide 133.]
  - Faults and hazards of organisations & management come about also as the result of "mis-management":
    - Strategic management updates tactical and operational management states.
    - Tactical management updates strategic and operational management states.
    - Operational management updates strategic and tactical management states.
    - That is: these states are not clearly delineated,
    - Etcetera!

5.6.2.1 Discussion

- This section on organisation & management
  - is rather terse;
  - in fact it covers a whole, we should think,
  - novel and interesting theory
    of business organisation & management

5.7. Discussion

- There may be other facets
  - but our point has been made:
  - that an analysis of hazards (including faults)
  - can, we think, be beneficially structured
  - by being related to reasonably distinct facets.
- A mathematical explanation of the concept of facet is needed.
  - One that helps partition the domain phenomena and concepts
  - into disjoint descriptions.
  - We are thinking about it.

6. Conclusion

6.1. The Author's Scientific & Engineering Background

- The present author's research has since the early 1970s focused on
  - ⟐ programming methodology:
    - how to develop software
    - such that it was correct
    - with respect to some specification —
    - call it requirements.
  - ⟐ The emphasis was on
    - abstract software specifications and their
    - refinement or transformation into code.

- Programming language semantics
  - and the stage- and step-wise development of compilers,
  - in many, up to nine stages and steps,
  - became a highlight of the 1980s.
- The step from programming language semantics to domain descriptions followed:
  - Domain descriptions, in a sense, specified
  - the language inherent in the described domain —
  - that is: "spoken" by its actors, etc.

- Since the early 1990s I therefore additionally focused on domain descriptions.
  - Now an additional goal of software development might be achieved:
  - securing that the software met customers' expectations.
- With the observation that
  - requirements prescriptions can be systematically — but, of course, not automatically — "derived" from domain descriptions
  - a bridge was established: from domains via requirements to software.

6.2. What Have We Achieved ?

- When Dr Clive Victor Boughton, on November 4, 2013, approached me on the subject of
  - *"Software Safety: New Challenges and Solutions"*,
  - I therefore, naturally questioned:
    - can one stratify the issues of safety criticality into three phases:
    - searching for sources of faults and hazards in domains,
    - elaborating on these while "discovering" further sources during requirements engineering, and,
    - finally, during early stages of software design.
  - I believe we have answered that question partially
    - with there being good hopes for further stratification.

- Yes, I would indeed claim that
  - we have contributed to the "greater" issues of safety critical systems
  - by suggesting a discipline framework for of faults "discovery" and hazards:
  - investigate the domains, the requirements and the design.

6.3. Further Work

- But, clearly, that work has only begun.

7. Acknowledgements

- I thank Dr Clive Victor Boughton of aSSCa &c.
  - for having the courage to convince his colleagues to invite me,
  - for having inspired me to observe that
    faults and hazards can be "discovered"
    purely in the context of domain descriptions,
  - for his support in answering my many questions,
  - and for otherwise arranging my visit.

Thanks