Domain Engineering
A Basis for Safety Critical Software

Dines Bjørner

Fredsvej 11, DK-2840 Holte, Danmark
DTU, DK-2800 Kgs. Lyngby, Denmark
E–Mail: bjorner@gmail.com, URL: www.imm.dtu.dk/˜dibj

1

2

Abstract. Before software can be designed we must have a reasonable
grasp of the requirements that the software is supposed to fulfil. And be-
fore requirements can be prescribed we must have a reasonable grasp of
the “underlying” application domain. Domain engineering now becomes 3
a software engineering development phase in which a precise descrip-
tion, 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 4
the first element, $\mathcal{D}$, of this triptych $(\mathcal{D}, \mathcal{R}, \mathcal{S})$ by an example, Sect. 2, in
which we show a description of a pipeline domain where, for example, the
operations of pumps and valves are safety critical. We then, Sects. 3–5,
summarise the methodological stages and steps of domain engineering.
We finally weave considerations of *system safety criticality* into a section
(Sect. 5) on domain facets. We believe this aspect of safety criticality is 5
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 en-
gineering [8], and to the related software design. That is, three distinct
“layers” of safety engineering.

1 Introduction 6

Before software can be designed we must have a reasonable grasp of the re-
quirements 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 7
phase in which a precise description, desirably formal, of the domain within
which the target software is to be embedded. **Requirements engineering** then 8
becomes a phase of software engineering in which one systematically derives
requirements prescriptions from the domain description — carving out and ex-
tending, 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 9
engineering phase which results in code (and further documentation). 10

We shall, in Sect. 2, give a fairly large example, approximately 10 Pages, 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. 11

We shall then, in Sect. 3 on the background of this substantial example, outline the basical principles, techniques and tools for describing domains — focusig only on endurants. 12

In Sect. 4 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 **requirement**s. 13

And, finally, in Sect. 5, 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 refer to [9] for a more thorough coverage of the notion of domain facets. 14

We consider the following ideas new: the idea of describing domains before prescribing requirements (but see [6, Part IV, 2006], [7, 2007], [9, written in 2007, published in 2010], [8, 2008], [10, 11, 2010], and [16, 2014]), 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 15

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. 16

To guide your intuition with respect to what a pipeline system might be we suggest some diagrams and some pictures. See Figs. 1 on the facing page and 2 on the next page.

The description only covers a few aspects of endurants. 17

2.1 Parts 18

1. A pipeline system contains a set of pipeline units and a pipeline system monitor.

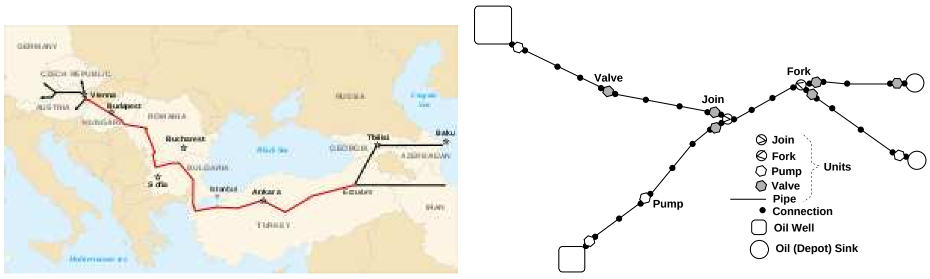

Fig. 1. Pipelines. Flow is right-to-left in left figure, but left-to-right in right figure.

Fig. 2. Some oil pipeline system units: pump, pipe, valve

2. The well-formedness of a pipeline system depends on its mereology (cf. Sect. 2.2) and the routing of its pipes (cf. Sect. 2.3).
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.

19

```
    type
1.    PLS′, US, U, M[1]
2.    PLS = {| pls:PLS′•wf_PLS(pls) |}[2]
    value
2.    wf_PLS: PLS → Bool[3]
2.    wf_PLS(pls) ≡ wf_Mereology(pls) ∧ wf_Routes(pls)[4]
1.    obs_Us: PS → U-set[5]
1.    obs_M: PLS → M[6]
    type
3.    U = We | Pi | Pu | Va | Fo | Jo | Si[7]
4.    We :: Well[8]
4.    Pi :: Pipe
4.    Pu :: Pump
4.    Va :: Valv
```

4. Fo :: Fork
4. Jo :: Join
4. Si :: Sink

2.2 Part Identification and Mereology 20

Unique Identification.

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

type
5. UI
value
5. $\text{uid\_UI: U} \to \text{UI}$[9]
axiom
5. $\forall\ \text{pls:PLS,u,u}':\text{U}\bullet\{\text{u,u}'\}\subseteq\text{obs\_Us(pls)}\Rightarrow\text{u}\neq\text{u}'\Rightarrow\text{uid\_UI(u)}\neq\text{uid\_UI(u}')$[10]

21 **Unique Identifiers.**

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

value
6. $\text{xtr\_UIs: PLS} \to \text{UI-}\textbf{set}$[11]
6. $\text{xtr\_UIs(pls)} \equiv \{\text{uid\_UI(u)}|\text{u:U}\bullet\text{u} \in \text{obs\_Us(pls)}\}$

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. $\forall\ \text{pls:PLS}\bullet\textbf{card}\ \text{obs\_Us(pl)}=\textbf{card}\ \text{xtr\_UIs(pls)}$

22 **Mereology.**

[1] PLS′, US, U and M are being defined as sorts: classes of endurant entities.
[2] PLS is the subtype of well-formed PLS entities.
[3] wf_PLS is the PLS well-formedness predicate whose signature is that of a function from PLS′ entities to truth values in **Bool**.
[4] wf_PLS(pls) is defined to be the conjunction of the well-formedness of the mereology of pls and pls defining only well-formed routes.
[5] obs_US is an observer function which maps plss into sets of units.
[6] obs_M is an observer function which maps plss into a monitor.
[7] U is defined to be the discriminated (::) union (|) of sorts We, Pi, Pu, Va, Fo, Jo and Si.
[8] We is discriminated from Pi, Pu, Va, Fo, Jo and Si by the constructor: :: mkWell, etcetera.
[9] uid_UI is the unique identifier observer function for parts u:U. It is total. uid_UI(u) yields the unique identifier of u.
[10] The axiom expresses that for all pipeline systems all two distinct units, u, u′ of such pipeline systems have distinct unique identifiers.
[11] xtr_UIs is a total function. It extracts all unique unit identifiers of a pipeline system.

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").

23

```
    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)[12] in iuis ∪ ouis ⊆ xtr_UIs(pls)[13]∧
8.               case (u,(card iuis,card ouis)) of[14]
8a                   (mk_We(we),(0,1)) → true,[15]
8b                   (mk_Pi(pi),(1,1)) → true,[16]
8c                   (mk_Pu(pu),(1,1)) → true,
8d                   (mk_Va(va),(1,1)) → true,
8e                   (mk_Fo(fo),(1,2)) → true,[17]
8f                   (mk_Jo(jo),(2,1)) → true,[18]
8g                   (mk_Si(si),(1,0)) → true,
8.                   _ → false end end
```

[12] The **let** clause names the pair resulting from mereo_U(u).

[13] The input and out unique identifiers are a subset of all pipe line unit unique identifiers.

[14] This **case**..**pattern**..**end** clause "sequentially matches" the pattern "against" the →.. clauses.

[15] Wells have 0 input and 1 output.

[16] Pipes, Pumps and Valves have 1 input and 1 out.

[17] Forks have 1 input and 2 outputs.

[18] Joins have 2 input and 1 output.

[19] Sinks have 1 input and 0 output.

2.3 Part Concepts 24

An aspect of domain analysis & description that was not covered in Sect. **??** 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.

25 **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′ = $U^{\omega}$[20]
9.      R = {| r:Route′•wf_Route(r) |}
10.     RD = $UI^{\omega}$
    axiom
10.     ∀ rd:RD • ∃ r:R•rd=descriptor(r)
    value
10.     descriptor: R → RD[21]
10.     descriptor(r) ≡ ⟨uid_UI(r[i])|i:Nat•1≤i≤len r⟩
```

26

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
```

27

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, ⟨⟩, 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*.

[20] U^{ω} denotes the class of finite and infinite length sequences of U elements.

[21] The descriptor function converts a finite or infinite length sequence of U elements to a "corresponding length" UI elements.

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[22]
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}[23]
12d.        in rs[24] end
```

Well-formed Routes. 28

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]
```

29

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.   wf_Routes: PLS → Bool
14.   wf_Routes(pls) ≡
14.       non_circular(pls) ∧ are_embedded_in_well_to_sink_Routes(pls)

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

30

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
```

31

[22] The Routes function generates the potentially infinite set of routes of a pipe line system.

[23] The **let** rs = ... clause is defined recursively and (cf. Footnote 24).

[24] rs is the smallest set which satisfies the **let** rs = ... equation..

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
```

32 **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.      {⟨r[k]|k:Nat•i≤k≤j⟩ | i,j:Nat• i {i,j}⊆inds(r) ∧ i≤j}
```

33 **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.   ∀ pls:PLS •
18.   let rs = Routes(pls),
18.       wsrs = well_to_sink_Routes(pls) in
18a.    rs =
18b.        wsrs ∪
18c.        ∪ {{r′|r′:R • r′ ∈ embedded_Routes(r″)} | r″:R • r″ ∈ wsrs}
17.   end
```

2.4 Materials 34

19. The only material of concern to pipelines is the gas[25] or liquid[26] which the pipes transport[27].

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

type
19. GoL
value
19. obs_GoL: U → GoL

2.5 Attributes 35

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).

36

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**

37

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.

38

```
    value
23a.    attr_WellCap: We → WellCap
23b.    attr_LEN: Pi → LEN
23c.    attr_Height: Pu → Height
23d.    attr_ValSta: Va → VaSta
23e.    attr_In_Flow_L: U → UI → Flow
23f.    attr_In_Leak_L: U → UI → Flow
23g.    attr_Max_In_Leak_L: U → UI → Flow
23h.    attr_body_Flow_L: U → Flow
23i.    attr_body_Leak_L: U → Flow
23j.    attr_Max_Flow_L: U → Flow
23k.    attr_Out_Flow_L: U → UI → Flow
23l.    attr_Out_Leak_L: U → UI → Flow
23m.    attr_Max_Out_Leak_L: U → UI → Flow
```

39 **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.    ∀ u:U\We\Si •
24.        sum_in_leaks(u) ⊕ sum_in_flows(u) =
24.        attr_body_Leak_L(u) ⊕
24.        sum_out_leaks(u) ⊕ sum_out_flows(u)
```

40

```
    value
        sum_in_leaks: U → Flow
        sum_in_leaks(u) ≡
            let (iuis,) = mereo_U(u) in
            ⊕ {attr_In_Leak_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_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_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_ℒ(u)(ui)|ui:UI•ui ∈ ouis} end
```

41

25. “What flows out, flows in !”. For ℒaminar 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.  ∀ u,u′:U•adjacent(u,u′) ⇒
25.      let (,ouis)=mereo_U(u), (iuis′,)=mereo_U(u′) in
25.      assert: uid_U(u′) ∈ ouis ∧ uid_U(u) ∈ iuis ′
25.      attr_Out_Flow_ℒ(u)(uid_U(u′)) =
25.      attr_In_Leak_ℒ(u)(uid_U(u))⊕attr_In_Flow_ℒ(u′)(uid_U(u)) end
```

Open Routes. 42

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 43

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. 44

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. 45

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. 46

3 Basic Domain Description 47

In this section and in Sect. 5 we shall survey basic principles of describing, re-
48 spectively, 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 •
49 **Example**: Pipeline systems, units and materials are entities (Page 2, Item 1.) ■

The method can thus be said to provide the *domain analysis prompt*: `is_en-
50 tity` 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** (Page 4, Item 5.), **mereology** (Page 5, Item 8.) and
the well capacity (Page 9, Item 20a.), pipe length (Page 9, Item 20b.), current pump
height (Page 9, Item 20c.), current valve open/close status (Page 9, Item 20d.) and
51 flow (Page 9, Item 20e.) **attributes** are qualities ■

By a **sort** (or **type** – which we take to be the same) 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*:
52 `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 en-
tity 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*:
53 `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
54 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)
55 holds if entity e is discrete.

By a **continuous endurant** we shall understand something which is pro-
longed without interruption, in an unbroken series or pattern • We use the
56 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.

57 **Endurant Entities.** We distinguish between endurant and perdurant entities.

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)≡ `is_endurant`(p)∧`is_discrete`(p) • We use the term **material** for
58 continuous endurants •

Discrete endurants are either atomic or are 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 ■ 59

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 ■ 60

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*[28]:

type P_1, P_2, ..., P_n;[29]
value **obs**_P_1: $P \rightarrow P_1$, **obs**_P_2: $P \rightarrow P_2$,...,**obs**_P_n: $P \rightarrow P_n$;[30]

61

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 (Page 2, Items 1.) illustrate the basic sorts (PLS′, US, U, M) and observers (obs_US, obs_M) of pipeline systems ■ 62

63 **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*, 64

* either
  type P1, P2, ..., PN, T = $\mathcal{E}$(P1,P2,...,PN)[31]
  value **obs**_T: P → T[32]

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[33];

[28] Throughout this paper the description texts are formulated in the `RAISE` [29] specification language `RSL` [28] – but other such model-oriented specification languages could be used, e.g., `Alloy` [32], `Event B` [1], `VDM` [21, 22, 27], or `Z` [41].

[29] This `RSL` **type** clause defines P_1, P_2, ..., P_n to be sorts.

[30] Thus `RSL` **value** clause defines n function values. All from type P into some type P_i.

[31] 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).

[32] **obs**_T is a function from any element of P to some element of T.

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

∗ or:

> **type**
> T = P1 | P2 | ... | PN[34]
> P_1, P_2, ..., P_n
> P1 :: mkP1(P_1), P2 :: mkP2(P_2), ..., PN :: mkPN(P_n) [35]
> **value**
> **obs**_T: P → T[36]

65 **Example**: **obs**_T: P → T is exemplified by obs_Us: PS → **U-set** (Page 2, Item 1.), T = P1 | P2 | ... | PN by We | Pu | Va | Fo | Jo | Si (Page 3, Item 3.) and P1 :: mkP1(P_1), P2 :: mkP2(P_2), ..., PN :: mkPN(P_n) by (Page 3, Item 4.) ■ 66

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 → M_1, **obs**_M_2: P → M_2, ..., **obs**_M_m: P → 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 Sect. 2.4 (Page 8, Item 19.) ■

67 **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). 68

Unique Part Identifiers: Without loss of generality we can assume that every part has a unique identifier[37]. 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, that is, their possible mereology and attributes are (also) 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;

69 **Example**: We refer to Page 4, Item 5. ■

[34] A|B is the union type of types A and B.
[35] Type definition A :: mkA(B) defines type A to be the set of elements mkA(b) where b is any element of type B
[36] **obs**_T is a function from any element of P to some element of T.
[37] That is, `has_unique_identifier(`p`)` for all parts p.

Part Mereology: By **mereology** [37] 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. 70

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. 71

Example: We refer to Items 8.–8g. Pages 5–5 ■ 72

Part and Material Attributes: Attributes are what really endows parts with qualities. The external properties[38] 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. 73

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*: 74

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 Sect. 2.5 Pages 9–10 ■

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. 75 76

[38] `is_discrete`,`is_continuous`,`is_atomic`,`is_compositehas_material`.

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. 77

There is the notion of **function signature**. A function signature, f: A $(\rightarrow|\overset{\sim}{\rightarrow})$ 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 $(\overset{\sim}{\rightarrow})$, and expresses a type, say T_R, of the result of the function. 78

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 $\rightarrow$ **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. 79

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. 80

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

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

have predicates $\mathcal{P}_f$ and $\mathcal{Q}_f$ delimit the value of f within that function space; $\mathcal{P}_f$ $\mathcal{Q}_f$ 81

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

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

have predicates $\mathcal{P}_e$ and $\mathcal{Q}_e$ delimit the value of e within the $\Sigma\times\Sigma\rightarrow$**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. 82

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 $\{\mathsf{pi}_1,\mathsf{pi}_2,...,\mathsf{pi}_n\}$, determine a number of channels $\{\mathsf{chs}[\,\mathsf{pi},\mathsf{pi}_j\,]|\mathsf{j}:\{1,2,...,n\}\}$ able to communicate messages of **type** M. Behaviour signatures: 83

$$b: pi{:}PI \times ATTR \rightarrow \textbf{in } in\_chs\ \textbf{out } out\_chs\ \ \textbf{Unit}$$

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

4 Interlude 84

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 [10, 11, 15]. In the next section, Sect. 5, 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 85

Some characterisations are:

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. 86

Failure: A domain *failure* occurs when the delivered service deviates from fulfilling the domain function, the latter being what the domain is aimed at [39]. 87

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 [39]. 88

Fault: The adjudged (i.e., the 'so-judged') or hypothesised cause of an error is a *fault* [39]. 89

Hazard: A **hazard** is any source of potential damage, harm or adverse health effects on something or someone under certain conditions at work. 90

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. 91

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 [36]: 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."*

4.2 System and Component Safety 92

There appears to be a number of safety concepts [36]: component safety, industrial safety, reliability, and system safety. We shall focus on component and system safety. 93

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. 94

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
95 parts. That is, "system-hood" presumes considerations of shared attributes.

System Safety: *"The primary concern of system safety [36] is the management of hazards: their identification, evaluation, elimination, and control through analysis,*
96 *design and management procedures."*

"System safety deals with systems as a whole rather than with subsystems or components [36]: 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."

97 The system interfaces are given by the mereology.

Component Safety: For a component, that is, an atomic part, we can, at most, speak of faults when considering safety.[39]

4.3 Stake-holder 98

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,
99 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 100

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

101 **Requirements.** By a **requirements** we understand (cf. IEEE Standard 610.12 [31]): *"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,

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

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. 102

We shall not cover requirements in this paper. We refer to [8]. 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 103

5.1 Introductory Notions

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 ● 104

We shall in this paper distinguish between the following facets: *intrinsics*, *support technologies*, *human behaviour*, *rules &*[40] *regulations* and *organisation & management*.

In the following we refer to respective subsections of [9] should the reader wish further elaborations of the facet concept.

Safety Criticality. *Safety critical systems are those systems whose failure may result* 105
in the loss of life, significant property damage or damage to the environment.[41] 106

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 paper we shall only consider the first step.

5.2 Intrinsics 107

By **domain intrinsics** [9, 1.4.1, 11–15][42] 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 ●

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

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

[42] [9, 1.4.1, 11–15] refers to publication [9], Sect. 1.4.1, Pages 11–15.

Example: The example of Sect. 2 focused on the intrinsics of pipeline systems
108 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
109 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
110 be described, also formally.[43]

5.3 Support Technologies 111

By domain **support technology** [9, 1.4.2, 15–17] 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
112 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;
113 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 &
114 control of support technology.

Describing causes of errors is not simple. With today's formal methods tools and techniques[43] quite a lot can be formalised — but not all!

[42] We refer to the example of Sect. 2. More specifically to the well-formedness of pipeline systems as expressed in wf_PLS (Page 3, Item 2.). We express hazards of the intrinsics of pipeline systems by named predicates over PLS′ and not PLS.

[43] These tools and techniques typically include two or more formal specification languages, for example: **VDM** [21, 22, 27], **DC** [42], **Event-B** [2], **RAISE/RSL** [29, 28, 4–6], **TLA+** [35] and **Alloy** [32]; one or more theorem proving tools, for example: **ACL** [34, 33], **Coq** [3], **Isabelle/HOL** [38], **STeP** [23], **PVS** [40] and **Z3** [24]; a model-checker, for example: **SMV** [26] and **SPIN/Promela** [30]; and other such tools and techniques; cf. [20].

5.4 Human Behaviour 115

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 ■ 116

By domain **human behaviour** [9, 1.4.6, 27–29] we shall understand any of a quality spectrum of humans[44] carrying out assigned work: from (i) *careful, diligent* and *accurate*, via (ii) *sloppy* dispatch, and (iii) *delinquent* work, to (iv) outright *criminal* pursuit ● 117

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 118

Rules and regulations [9, 1.4.4, 24–26] come in pairs $(\mathcal{R}_u, \mathcal{R}_e)$.

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 ■ 119

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 ■ 120

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: $\mathcal{P} : \Sigma \times \Sigma \rightarrow \textbf{Bool}$. 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.

[44] — in contrast to technology

121 **Regulations.** By a domain **regulation** we shall understand some text which
"prescribe" ("...", see below) the remedial actions that are to be taken when it
122 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 (i)
starts an administrative procedure which ensures that the rule is replaced; and
(ii) starts a series of actions which somehow brings the state of the pipeline into
123 one which poses no danger and then applies a non-hazard rule ■

Hazards: when stipulating "prescribe" regulations express requirements to emerg-
124 ing 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**.

125 **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 126

To [9, 1.4.3, 17–21] properly appreciate this section we need remind the reader of
concepts introduced earlier in this paper. 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
127 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.

128 **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[45] (say partition π_s), tacti-

[45] Strategic management, one can claim, deals with the management of the most generic and general, year-to-year company resources: invested capital, overall market, production and service goals, etc.

cal[46] (say partition π_t), respectively operational [47] (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 ● 129

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 pipeline 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 pipeline 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} ($\pi_{\ell_i} \in \pi_{\ell s}$) ■ 130

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. 131

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. 132

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, π:$\mathit{\Pi}$, of domain parts, p:P. For example:

```
type
    ORG = Str × Tac × Ope × ...
    Str, Tac, Ope = (Π-set)-set
value
    attr_ORG: P → ORG
axiom
    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. 133

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 "tow-holding" management becomes confused ■

[46] Tactical management, one can claim, deals with the management of the quarter/month-to-quarter/month resources "closest" to the implementation if strategic goals.

[47] Operational management, one can finally claim, deals with the management of day-to-day resources "closest" to the actual market, production and services.

134 **Management.** By domain **management** we shall understand such people who (such decisions which) (i) determine, formulate and thus set standards (cf. rules and regulations, above) concerning strategic, tactical and operational decisions; (ii) who ensure that these decisions are passed on to (lower) levels of management, and to floor staff; (iii) who make sure that such orders, as they were, are indeed carried out; (iv) who handle undesirable deviations in the carrying out of these orders cum decisions; and (v) who "backstops" complaints from lower
135 management levels and from floor staff •

Example: [Cf. examples on the previous page]. 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
136 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.
137 The "management" parts have their attributes form corresponding states (σ:Σ).

```
type
    $\Sigma_{STR}$, $\Sigma_{TAC}$, $\Sigma_{OPE}$,
```

An idealised rendition of management actions is:

```
value
    action$_{Strategic}$: $\Sigma_{STR}\to\Sigma_{TAC}\to\Sigma_{OPE}\to\Sigma_{STR}$
    action$_{Tactical}$: $\Sigma_{STR}\to\Sigma_{TAC}\to\Sigma_{OPE}\to\Sigma_{TAC}$
    action$_{Operational}$: $\Sigma_{STR}\to\Sigma_{TAC}\to\Sigma_{OPE}\to\Sigma_{OPE}$
```

138

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.

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.

action$_{Operational}$ expresses that tactical management considers the "global" state ($\Sigma_{STR}\times\Sigma_{TAC}\times\Sigma_{OPE}$) but potentially changes only the "operational"
139 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,
140 {p$_1$:P$_1$,p$_2$:P$_2$,...,p$_n$:P$_N$}: For example:

```
channel mgt:{{$\pi$,$\pi_j$}|$\pi_j$:PI$_j$•$\pi_j$ ∈ ...}:M
value
    p: $\pi$:$\Pi$ × pt:P → in,out {{$\pi$,$\pi_j$}|$\pi_j$:PI$_j$•$\pi_j$ ∈ ...} Unit
```

```
p(π,pt) ≡ ...
                                              [ management orders staff ]
    ⌈⌉ let (π_j,m) = query_boss(p) in
        m ! mgt[{π,π_j}]!m ;
        p(π,action_{down_s}(pt,m)) end
                                              [ management "listens" to staff ]
    ⌈⌉ let (π_j,m) = ⌈⌉ {mgt[{π,π_j}]? | ... } in
        p(π,action_{down_r}(pt,m)) end
                                              [ management reports to boss ]
    ⌈⌉ let (π_boss,m) = query_staff(pt) in
        m ! mgt[{π,π_boss}]!m ;
        p(π,action_{up_s}(pt,m)) end
                                              [ management "listens" to boss ]
    ⌈⌉ let (π_boss,m) = ⌈⌉ {mgt[{π,π_boss}]? | ... } in
        p(π,action_{up_s}(pt,m)) end ...
```

The boss communications express that process p serves a boss. All other communications express that process p interacts with staff (i.e., "subordinates and "others"). 141

Hazards: [Cf. faults and hazards on the previous page.] 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! 142

• • •

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 143

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 and encourage the reader to do likewise!.

6 Conclusion 144

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
145 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 de-
scriptions, in a sense, specified the language inherent in the described domain —
146 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 ? 147

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
148 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 149

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.

8 Bibliography 150

8.1 Notes

This conference contribution is part of a series of papers on the topic of domains. [7–14, 16–19]. In [8, 2008] we show how to "derive" requirements prescriptions from domain descriptions; [9, 2008] shows techniques for describing domain facets: intrinsics, support technologies, rules & regulations, management & organisation as well as human behaviour; [12, 2011] illuminates such concepts as simulation, demos, monitoring and control in the new light afforded by the domain viewpoint; [14, 2013] speculates on various issues of "computation for humanity" (!); and. [13, 2013] relates our modelling of mereology to the classical axiom systems for mereology. [25, 2014] provides a systematic introduction to principles, techniques and tools for the analysis and description of domain endurants. We hope to work out a paper ([15]) similar to [25] on domain perdurants.

8.2 References

1. J.-R. Abrial. The B Book: Assigning Programs to Meanings *and* Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge, England, 1996 and 2009.
2. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, Cambridge, England, 2009.
3. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. EATCS Series: Texts in Theoretical Computer Science. Springer, 2004.
4. D. Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. .
5. D. Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. Chapters 12–14 are primarily authored by Christian Krog Madsen.
6. D. Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
7. D. Bjørner. Domain Theory: Practice and Theories, Discussion of Possible Research Topics. In *ICTAC'2007*, volume 4701 of *Lecture Notes in Computer Science (eds. J.C.P. Woodcock et al.)*, pages 1–17, Heidelberg, September 2007. Springer.
8. D. Bjørner. From Domains to Requirements. In *Montanari Festschrift*, volume 5065 of *Lecture Notes in Computer Science (eds. Pierpaolo Degano, Rocco De Nicola and José Meseguer)*, pages 1–30, Heidelberg, May 2008. Springer.
9. D. Bjørner. Domain Engineering. In P. Boca and J. Bowen, editors, *Formal Methods: State of the Art and New Directions*, Eds. Paul Boca and Jonathan Bowen, pages 1–42, London, UK, 2010. Springer.
10. D. Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics, Part I of II: The Engineering Part*. *Kibernetika i sistemny analiz*, (4):100–116, May 2010.
11. D. Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics Part II of II: The Science Part*. *Kibernetika i sistemny analiz*, (2):100–120, May 2011.
12. D. Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. In *Rainbow of Computer Science, Festschrift for Hermann Maurer on the Occasion of His 70th Anniversary.*, Festschrift (eds. C. Calude, G. Rozenberg and A. Saloma), pages 167–183. Springer, Heidelberg, Germany, January 2011.
13. D. Bjørner. *A Rôle for Mereology in Domain Science and Engineering*. Synthese Library (eds. Claudio Calosi and Pierluigi Graziani). Springer, Amsterdam, The Netherlands, October 2013.

14. D. Bjørner. *Domain Science and Engineering as a Foundation for Computation for Humanity*, chapter 7, pages 159–177. Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC [Francis & Taylor], 2013. (eds.: Justyna Zander and Pieter J. Mosterman).
15. D. Bjørner. Domain Analysis & Description: Perdurants [Writing to begin Summer/Fall 2014]. Research Report, Fredsvej 11, DK-2840 Holte, Denmark, Summer/Fall 2014.
16. D. Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model. In S. Iida, J. Meseguer, and K. Ogata, editors, *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi*. Springer, May 2014.
17. D. Bjørner. *Domain Engineering: Technology Management, Research and Engineering*. A JAIST Press Research Monograph # 4, 536 pages, March 2009.
18. D. Bjørner. The Role of Domain Engineering in Software Development. Why Current Requirements Engineering Seems Flawed! In *Perspectives of Systems Informatics*, volume 5947 of *Lecture Notes in Computer Science*, pages 2–34, Heidelberg, Wednesday, January 27, 2010. Springer.
19. D. Bjørner and A. Eir. Compositionality: Ontology and Mereology of Domains. Some Clarifying Observations in the Context of Software Engineering in July 2008, eds. Martin Steffen, Dennis Dams and Ulrich Hannemann. In *Festschrift for Prof. Willem Paul de Roever Concurrency, Compositionality, and Correctness*, volume 5930 of *Lecture Notes in Computer Science*, pages 22–59, Heidelberg, July 2010. Springer.
20. D. Bjørner and K. Havelund. 40 Years of Formal Methods — 10 Obstacle and 3 Possibilities. In *FM 2014, Singapore, May 14-16, 2014*. Springer, 2014. Distinguished Lecture, .
21. D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.
22. D. Bjørner and C. B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.
23. N. Bjørner, A. Browne, M. Colon, B. Finkbeiner, Z. Manna, H. Sipma, and T. Uribe. Verifying Temporal Properties of Reactive Systems: A STeP Tutorial. *Formal Methods in System Design*, 16:227–270, 2000.
24. N. Bjørner, K. McMillan, and A. Rybalchenko. Higher-order Program Verification as Satisfiability Modulo Theories with Algebraic Data-types. In *Higher-Order Program Analysis*, June 2013. http://hopa.cs.rhul.ac.uk/files/proceedings.html.
25. D. Bjørner. Domain Types – Endurants. Submittted Paper, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, 2014.
26. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Five Cambridge Center, Cambridge, MA 02142-1493, USA, January 2000. ISBN 0-262-03270-8.
27. J. Fitzgerald and P. G. Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.
28. C. W. George, P. Haff, K. Havelund, A. E. Haxthausen, R. Milne, C. B. Nielsen, S. Prehn, and K. R. Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.
29. C. W. George, A. E. Haxthausen, S. Hughes, R. Milne, S. Prehn, and J. S. Pedersen. *The RAISE Development Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.
30. G. J. Holzmann. *The SPIN Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.

31. IEEE Computer Society. IEEE–STD 610.12-1990: Standard Glossary of Software Engineering Terminology. Technical report, IEEE, IEEE Headquarters Office, 1730 Massachusetts Avenue, N.W., Washington, DC 20036-1992, USA. Phone: +1-202-371-0101, FAX: +1-202-728-9614, 1990.
32. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.
33. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.
34. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000.
35. L. Lamport. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA, 2002.
36. N. G. Leveson. White Paper on Approaches to Safety Engineering. URL document: http://sunnyday.mit.edu/caib/concepts.pdf, MIT, April 2003. White paper.
37. E. Luschei. *The Logical Systems of Leśniewksi*. North Holland, Amsterdam, The Netherlands, 1962.
38. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL, A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
39. B. Randell. On Failures and Faults. In *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 18–39. Formal Methods Europe, Springer, 2003. Invited paper.
40. N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1999.
41. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.
42. C. C. Zhou and M. R. Hansen. *Duration Calculus: A Formal Approach to Real–time Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer–Verlag, 2004.