Domain Engineering
A Basis for Safety Critical Software

Dines Bjørner

Fredsvej 11, DK-2840 Holte, Danmark
E–Mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~dibj

Abstract

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. (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, 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 *domain safety criticality*[1] into a section (Sect. 5) on domain facets. We believe this aspect of safety criticality is new: the connection of issues of 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 (Bjørner 2008), and to the related software design. That is, three distinct "layers" of safety engineering.

1 Introduction

1.1 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). .

1.2 Domain Description

To us a domain description is a set of pairs of narrative, that is, informal, and formal description texts. The narrative texts should go hand-in-hand with the formal texts; that is, the narrative should be "a reading" of the formalisation; and the formalisation should be an abstraction that emphasises properties of the domain, not some intricate, for example, "executable" model of the domain.[2] These "pairings" will be amply illustrated in Sect. 2. The meaning of a domain description is basically a heterogeneous algebra[3], that is: sets of typed entities and a set of typed operations over these. The formalisation language is here the `RAISE` (George et al. 1995) Specification Language `RSL` (George et al. 1992); but it could be any of, for example, `Alloy` (Jackson 2006), `Event B` (Abrial 2009*a*), `VDM-SL` (Bjørner and Jones 1978, 1982, Fitzgerald and Larsen 1998) or `Z` (Woodcock and Davies 1996). That is, the main structure of the description of the domain endurants, such as we shall advocate it, may, by some readers, be thought of as an ontology (Benjamin and Fensel 1998, Fox 2000). But our concept a domain description is a much wider concept of ontology than covered by (Benjamin and Fensel 1998); it is more in line with (Mellor and Oliver 1997, Fox 2000).

1.3 A Domain Description "Ontology"

We shall, in Sect. 2, give a fairly large example, approximately 3.5 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.

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

The mathematical structure that is built up when describing a domain hinges on the following elements: there are *entities;* entities are either *endurants* or *perdurants;* endurants are either *discrete* or *continuous;* discrete endurants are also called *parts;* continuous endurants are also called *materials;* parts are either

[1] – as opposed to 'system safety criticality'

[2] Domain descriptions are usually not deiscriptions of computable phenomena.

[3] This is just one of the ways in which a domain description differs from an ontology.

atomic or *composite;* parts have *unique identifiers, mereologies* and *attributes;* materials have attributes; so entities are what we see and unique identifiers, mereologies and attributes are entity qualities. A domain description is then composed from one or more part and material descriptions; descriptions of unique part identifiers, part mereologies and part attributes. This structure that, to some, may remind them of an *"upper ontology."* Different domain descriptions all basically have the same "upper ontology."

1.4 A Method: its Principles, Tools and Techniques

By a **method** we shall understand a set of **principles** of **selecting** and **applying** a number of **techniques** and **tools,** for **analysing** and **constructing** an artefact. By a **formal method** we shall understand a set of a method whose techniques and tools can be given a mathematical basis that enable formal reasoning.

The **principles** of our approach to domain analysis and description are embodies in the above "upper ontology". The **tools of analysis** are embodied in a number of **domain analysis prompts**. Analysis prompts form a comprehensive and small set of predicates mentally "executed" by the domain analyser. The **tools of description** are embodied in a number of **domain description prompts**. Description prompts form a comprehensive and small set of RSL-text generating functions mentally "executed" by the domain describer. The domain analyser and describer is usually one and the same person the domain engineer cum scientist. The analysis and description **techniques** are outlined in the texts of Sects. 3 and 5. We claim that this formulation of the concept of method and formal method, their endowment with prompt tools, and the underlying techniques is new.

1.5 Safety Criticality

In Sect. 4 we shall review notions of **safety criticality**: **safety**, **failure**, **error**, **fault**, **hazard** and **risk**. We emphasize that we are focusing sôlely on issues of **domain safety**. That is, we are not dealing with **system safety** where we understand the term 'system' to designate a composition of software and hardware that is being designed in order to solve problems, including such which may aris from issues of domain safety. 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 (Bjørner 2010*a*) for a more thorough coverage of the notion of domain facets.

1.6 Contribution

We consider the following ideas new: the idea of describing domains before prescribing requirements (but see (Bjørner 2006*c*, Part IV, 2006), (Bjørner 2007, 2007), (Bjørner 2010*a*, written in 2007, published in 2010), (Bjørner 2008, 2008), (Bjørner 2010*b*, 2011*a*, 2010), and (Bjørner 2014*b*, 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

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.

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

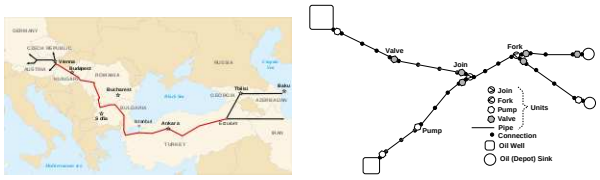

Figure 1: Pipelines. Flow is right-to-left in left figure, but left-to-right in right figure.

The description only covers a few aspects of endurants.

Figure 2: Pump, pipe and valve pipeline units

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 (cf. Sect. 2.2.3) and the routing of its pipes (cf. Sect. 2.3.2).
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′, U, M[4]
2.    PLS = {| pls:PLS′•wf_PLS(pls) |}[5]
    value
2.    wf_PLS: PLS′ → Bool[6]
2.    wf_PLS(pls) ≡
2.      wf_Mereology(pls) ∧ wf_Routes(pls)[7]
1.    obs_Us: PLS → U-set[8]
1.    obs_M: PLS → M[9]
    type
3.    U = We | Pi | Pu | Va | Fo | Jo | Si[10]
4.    We :: Well[11]
```

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 → UI[12]
axiom
5. ∀ pls:PLS,u,u′:U•
5. {u,u′}⊆obs_Us(pls)⇒
5. u≠u′⇒uid_UI(u)≠uid_UI(u′)[13]

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**[14]
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)

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)[15] **in**
8x. iuis ∪ ouis ⊆ xtr_UIs(pls)[16]∧
8. **case** (u,(**card** iuis,**card** ouis)) **of**[17]
8a (mk_We(we),(0,1)) → **true**,[18]
8b (mk_Pi(pi),(1,1)) → **true**,[19]
8c (mk_Pu(pu),(1,1)) → **true**,
8d (mk_Va(va),(1,1)) → **true**,
8e (mk_Fo(fo),(1,2)) → **true**,[20]
8f (mk_Jo(jo),(2,1)) → **true**,[21]
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 Sects. 2.1–2.2 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^ω[23]
9. R = {| r:Route′•wf_Route(r) |}
10. RD = UI^ω
axiom
10. ∀ rd:RD • ∃ r:R•rd=descriptor(r)
value
10. descriptor: R → RD[24]
10. descriptor(r) ≡ ⟨uid_UI(r[i])|i:**Nat**•1≤i≤**len** r⟩

[4] PLS′, US, U and M are being defined as sorts, i.e., sets of endurant entities.
[5] PLS is the subtype (i.e., subset) of well-formed PLS entities.
[6] wf_PLS is the PLS well-formedness predicate whose signature (i.e., type) is that of a function from PLS′ entities to truth values in **Bool**.
[7] 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.
[8] obs_US is an observer function which maps plss into sets of units.
[9] obs_M is an observer function which maps plss into a monitor.
[10] U is defined to be the discriminated (::) union (|) of sorts We, Pi, Pu, Va, Fo, Jo and Si.
[11] We is discriminated from Pi, Pu, Va, Fo, Jo and Si by the constructor: :: mkWell, etcetera.
[12] uid_UI is the unique identifier observer function for parts u:U. It is total. uid_UI(u) yields the unique identifier of u.
[13] The axiom expresses that for all pipeline systems all two distinct units, u, u′ of such pipeline systems have distinct unique identifiers.
[14] xtr_UIs is a total function. It extracts all unique unit identifiers of a pipeline system.
[15] The **let** clause names the pair resulting from mereo_U(u).
[16] The input and out unique identifiers are a subset of all pipe line unit unique identifiers.
[17] This **case**..**pattern**..**end** clause "sequentially matches" the pattern "against" the →.. clauses.
[18] Wells have 0 input and 1 output.
[19] Pipes, Pumps and Valves have 1 input and 1 out.
[20] Forks have 1 input and 2 outputs.
[21] Joins have 2 input and 1 output.
[22] Sinks have 1 input and 0 output.

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, ⟨⟩, is a route of *pls*.
    b Let u, u' be any units of *pls*, such that an output unit identifier of u is the same as an input unit identifier of u' then $\langle u, u' \rangle$ is a route of *pls*.
    c If r and r' are routes of *pls* such that the last element of r is the same as the first element of r', then $r^\frown\mathbf{tl}r'$ is a route of *pls*.
    d No sequence of units is a route unless it follows from a finite (or an infinite) number of applications of the basis and induction clauses of Items 12a–12c.

```
value
12.  Routes: PLS → RD-infset[25]
12.  Routes(pls) ≡
12a.    let rs = ⟨⟩ ∪
12b.          {⟨uid_UI(u),uid_UI(u′)⟩|u,u′:U•{u,u′}
12b.          ⊆ obs_Us(pls) ∧ adjacent(u,u′)}
12c.        ∪ {r^tl r′|r,r′:R•{r,r′}⊆rs}[26]
12d.    in rs[27] 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) ≡
13.    ∼∃ 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.  wf_Routes: PLS → Bool
14.  wf_Routes(pls) ≡
14.    non_circular(pls) ∧
14.    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)
```

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 ∧
15.    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.  embedded_in_well_to_sink_Routes: PLS → Bool
16.  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.    {⟨r[k]|k:Nat•i≤k≤j⟩
17.      | i,j:Nat• i {i,j}⊆inds(r) ∧ i≤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.  ∀ 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″)}
18c.        | r″:R • r″ ∈ wsrs}
17.  end
```

2.4 Materials

19. The only material of concern to pipelines is the gas[28] or liquid[29] which the pipes transport[30].

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

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

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

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

[26] The **let** rs = ... clause is defined recursively and (cf. Footnote 27).

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

[28] Gaseous materials include: air, gas, etc.

[29] Liquid materials include water, oil, etc.

[30] The description of this paper 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. ⊕,⊖: Flow×Flow → Flow
21. ⊕: Flow-set → Flow
22. <,≤,=,≠,≥,>: 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 → 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
```

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. ∀ 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)
```

```
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
```

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. ∀ u,u′:U•adjacent(u,u′) ⇒
25.   let (,ouis) = mereo_U(u),
25.       (iuis′,) = mereo_U(u′) in
25.   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))
25.   ⊕ attr_In_Flow_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
26a.     ⇒ is_pumping(p) ∧
26b.   ∀ mkVa(v):Va • mkVa(v) ∈ elems r
26b.     ⇒ 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 in Sect. 5 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 (Page 2, 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** (Page 3, Item 5.), **mereology** (Page 3, Item 8.)and the well capacity (Page 5, Item 20a.), pipe length (Page 5, Item 20b.), current pump height (Page 5, Item 20c.), current valve open/close status (Page 5, Item 20d.) and flow (Page 5, 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*: `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.

3.1 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 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 ■

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 a composite part.

Example: The pipeline system, PLS, and the set, Us, of pipeline units are considered composite entities ■

3.1.1 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,[31]
value **obs_**P_1: $\mathsf{P}\rightarrow\mathsf{P}_1$,
obs_P_2: $\mathsf{P}\rightarrow\mathsf{P}_2$,
...,
obs_P_n: $\mathsf{P}\rightarrow\mathsf{P}_n$,[32]

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 ■

3.1.2 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

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

[32] This RSL **value** clause defines n function values. All from type P into some type P_i.

type
P1, P2, ..., PN,
T = $\mathcal{E}$(P1,P2,...,PN)[33]
value
obs_T: P → T[34]

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[35];

* or:

type
T = P1 | P2 | ... | PN[36]
P_1, P_2, ..., P_n
P1 :: mkP1(P_1),
P2 :: mkP2(P_2),
...,
PN :: mkPN(P_n) [37]
value
obs_T: P → T[38]

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 2, Item 3.) and P1 :: mkP1(P_1), P2 :: mkP2(P_2), ..., PN :: mkPN(P_n) by (Page 2, Item 4.) ■

3.1.3 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 4, Item 19.) ■

3.2 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).

[33] 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).

[34] **obs_**T is a function from any element of P to some element of T.

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

[36] A|B is the union type of types A and B.

[37] Type definition A :: mkA(B) defines type A to be the set of elements mkA(b) where b is any element of type B

[38] **obs_**T is a function from any element of P to some element of T.

3.2.1 Unique Part Identifiers.

Without loss of generality we can assume that every part has a unique identifier[39]. 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;

Example: We refer to Page 3, Item 5. ■

3.2.2 Part Mereology.

By **mereology** (Luschei 1962) 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. Pages 3–3 ■

3.2.3 Part and Material Attributes.

Attributes are what really endows parts with qualities. The external properties `is_discrete`,`is_continuous`,`is_atomic`,`is_compositehas_materi` 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.

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;
value attr_A_1:P→A_1,
attr_A_2:P→A_2,
...,
attr_A_n:P→A_n;

Example: We refer to Sect. 2.5 Pages 5–5 ■

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

3.3 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, A×Σ, and results, Σ: f: A×Σ→Σ; f denotes a function in the function space A×Σ→Σ. Action pre/post-conditions:

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

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

Event signatures are typically predicates from pairs of before and after states: e: Σ×Σ→**Bool**. Event pre/post-conditions

value
  e: Σ×Σ→**Bool**;
  e(σ,σ') ≡
    $\mathcal{P}_e(\sigma) \wedge \mathcal{Q}_e(\sigma,\sigma')$

have predicates $\mathcal{P}_e$ and $\mathcal{Q}_e$ delimit the value of e within the Σ×Σ→**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 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:

  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 $\{\mathsf{chs}[\,\mathsf{pi},\mathsf{pi}_j\,]|\mathsf{j}:\{1,2,...,n\}\}$. **Unit** designate that b denote a never-ending process. We omit dealing with behaviour pre-conditions and invariants.

4 Interlude

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 (Bjørner 2010*b*, 2011*a*, 2014*a*). 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 *stake-holder* and the notion of *requirements*.

4.1 Safety-related Concepts

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.

Failure: A domain *failure* occurs when the delivered service deviates from fulfilling the domain function, the latter being what the domain is aimed at (Randell 2003).

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 (Randell 2003).

Fault: The adjudged (i.e., the 'so-judged') or hypothesised cause of an error is a *fault* (Randell 2003).

Hazard: A **hazard** is any source of potential damage, harm or adverse health effects on something or someone under certain conditions at work. Hazards are thus domain faults or are faults of the environment of the domain.

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.

4.2 Domain versus System Safety

We must reiterate that we are, in this paper, concerned only with issues of domain safety. Usually safety criticality is examined in the context of (new) systems design. When considering domain safety issues we are concerned with hazards of domain entities without any consideration of whether these hazards enter or do not enter into conceived systems.

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 ■

5 Domain Facets and Safety Criticality

5.1 Introductory Notions

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 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 (Bjørner 2010*a*) should the reader wish further elaborations of the facet concept.

5.2 Intrinsics

By **domain intrinsics** (Bjørner 2010*a*, 1.4.1, 11–15)[41] 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 example of Sect. 2 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.[42]

5.3 Support Technologies

By domain **support technology** (Bjørner 2010*a*, 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 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 ■

Hazards: 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[43] quite a lot can be formalised — but not all!

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** (Bjørner 2010*a*, 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 •

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 follow from "their corresponding" intrinsics descriptions ■

5.5 Rules & Regulations

Rules and regulations (Bjørner 2010*a*, 1.4.4, 24–26) 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

[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] (Bjørner 2010*a*, 1.4.1, 11–15) refers to publication (Bjørner 2010*a*), Sect. 1.4.1, Pages 11–15.

[42] We refer to the example of Sect. 2. More specifically to the well-formedness of pipeline systems as expressed in wf_PLS (Page 2, 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** (Bjørner and Jones 1978, 1982, Fitzgerald and Larsen 1998), **DC** (Zhou and Hansen 2004), **Event-B** (Abrial 2009*a*), **RAISE/RSL** (George et al. 1995, 1992, Bjørner 2006*a*,*b*,*c*), **TLA+** (Lamport 2002) and **Alloy** (Jackson 2006); one or more theorem proving tools, for example: **ACL** (Kaufmann et al. 2000*b*,*a*), **Coq** (Bertot and Castéran 2004), **Isabelle/HOL** (Nipkow et al. 2002), **STeP** (Bjørner et al. 2000), **PVS** (Shankar et al. 1999) and **Z3** (Bjørner et al. 2013); a model-checker, for example: **SMV** (Clarke et al. January 2000) and **SPIN/Promela** (Holzmann 2003); and other such tools and techniques; cf. (Bjørner and Havelund 2014).

[44] — in contrast to technology

$r = r'^\frown\langle p\rangle^\frown r''$, 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: $\mathcal{P} : \Sigma\times\Sigma\to\mathbf{Bool}$. 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 (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 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: $\mathcal{S} : \Sigma\times\Sigma\to\Sigma$. 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 (Bjørner 2010*a*, 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 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[45] (say partition π_s), tactical[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 •

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} ($\pi_{\ell_i} \in \pi_{\ell s}$) ■

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, π:Π, 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.

5.6.2 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 management levels and from floor staff •

Example: 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;

[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.

[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.

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, π : Π, belong to some partition in an str:Str. Other parts are associated with tactical management. They will have their unique identifiers, π : Π, belong to some partition in a corresponding tac:Tac. Yet other parts are associated with operational management. They will have their unique identifiers, π : Π, 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
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$_{Operational}$: $\Sigma_{STR}\to\Sigma_{TAC}\to\Sigma_{OPE}\to\Sigma_{OPE}$

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" 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$_2$,...,p$_n$:P$_N$}: For example:

```
channel mgt:{{π,π_j}|π_j:PI_j•π_j ∈ ...}:M
value
  p: π:Π × pt:P →
       in,out {{π,π_j}|π_j:PI_j•π_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").

Hazards: 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!

• • •

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 and encourage the reader to do likewise!

6 Conclusion

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 meet customers' expectations. With the observation (Bjørner 2008) 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.

6.1 Comparison to Other Work

(Bjørner 2014*b*) contains a large section, Sect. 4.1 (4+ pages), which compares our domain analysis and description approach to the domain analysis approaches of *Ontology and Knowledge Engineering*, *Database Analysis* (Bachmann Diagrams, Relational Data Models, Entity Set Relations, etc., *Prieto-Dĩaz's* work, *Domain Specific Languages*, *Feature-oriented Domain Analysis*, *Software Product Line Engineering*, Michael Jackson's *Problem Frames*, *Domain Specific Software Architectures*, *Domain Driven Design*, *Unified Modelling Language*, etcetera. We refer to (Bjørner 2014*b*) for its lengthy discussion and almost 30 citations. (Bjørner 2014*b*, Sect. 4.1) shows that our approach is significantly different from the *above-enumerated approaches.*

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 disciplined framework for faults "discovery"and hazards: investigate separately 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, ANU, &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. I also, with thanks, acknowledge comments and remarks by the ASSC program chair, Dr Anthony Cant and especially by hos colleague Dr Brendan Mahony. Their joint paper (Cant and Mahony 2012), alas, came only to my attention in the last days before the present paper had to be submitted.

8 Bibliography

8.1 Notes

This conference contribution is part of a series of papers on the topic of domains. (Bjørner 2007, 2008, 2010*a*,*b*, 2011*a*,*b*, 2013*a*,*b*, 2014*b*, 2009, 2010*c*, Bjørner and Eir 2010). In (Bjørner 2008) we show how to "derive" requirements prescriptions from domain descriptions; (Bjørner 2010*a*) shows techniques for describing domain facets: intrinsics, support technologies, rules & regulations, management & organisation as well as human behaviour; (Bjørner 2011*b*) illuminates such concepts as simulation, demos, monitoring and control in the new light afforded by the domain viewpoint; (Bjørner 2013*b*) speculates on various issues of "computation for humanity" (!); (Bjørner 2013*a*) relates our modelling of mereology to the classical axiom systems for mereology; and (Bjørner 2014*c*) provides a systematic introduction to principles, techniques and tools for the analysis and description of domain endurants.

8.2 References

Abrial, J.-R. (1996 and 2009*b*), The B Book: Assigning Programs to Meanings *and* Modeling in Event-B: System and Software Engineering, Cambridge University Press, Cambridge, England.

Abrial, J.-R. (2009*a*), *Modeling in Event-B: System and Software Engineering*, Cambridge University Press, Cambridge, England.

Benjamin, J. & Fensel, D. (1998), *The Ontological Engineering Initiative (KA)2.* Internet publication + Formal Ontology in Information Systems, University of Amsterdam, SWI, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands and University of Karlsruhe, AIFB, 76128 Karlsruhe, Germany, 1998.http://www.aifb.uni-karlsruhe.de/WBS/broker/KA2.htm.

Bertot, Y. & Castéran, P. (2004), *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, EATCS Series: Texts in Theoretical Computer Science, Springer.

Bjørner, D. (2006*a*), *Software Engineering, Vol. 1: Abstraction and Modelling*, Texts in Theoretical Computer Science, the EATCS Series, Springer.

Bjørner, D. (2006*b*), *Software Engineering, Vol. 2: Specification of Systems and Languages*, Texts in Theoretical Computer Science, the EATCS Series, Springer. Chapters 12–14 are primarily authored by Christian Krog Madsen.

Bjørner, D. (2006*c*), *Software Engineering, Vol. 3: Domains, Requirements and Software Design*, Texts in Theoretical Computer Science, the EATCS Series, Springer.

Bjørner, D. (2007), Domain Theory: Practice and Theories, Discussion of Possible Research Topics, *in* 'ICTAC'2007', Vol. 4701 of *Lecture Notes in Computer Science (eds. J.C.P. Woodcock et al.)*, Springer, Heidelberg, pp. 1–17.

Bjørner, D. (2008), From Domains to Requirements, *in* 'Montanari Festschrift', Vol. 5065 of *Lecture Notes in Computer Science (eds. Pierpaolo Degano, Rocco De Nicola and José Meseguer)*, Springer, Heidelberg, pp. 1–30.

Bjørner, D. (2009), *Domain Engineering: Technology Management, Research and Engineering*, A JAIST Press Research Monograph # 4, 536 pages.

Bjørner, D. (2010*a*), Domain Engineering, *in* P. Boca & J. Bowen, eds, 'Formal Methods: State of the Art and New Directions', Eds. Paul Boca and Jonathan Bowen, Springer, London, UK, pp. 1–42.

Bjørner, D. (2010*b*), '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.

Bjørner, D. (2010*c*), The Rôle of Domain Engineering in Software Development. Why Current Requirements Engineering Seems Flawed!, *in* 'Perspectives of Systems Informatics', Vol. 5947 of *Lecture Notes in Computer Science*, Springer, Heidelberg, pp. 2–34.

Bjørner, D. (2011*a*), '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.

Bjørner, D. (2011*b*), 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), Springer, Heidelberg, Germany, pp. 167–183.

Bjørner, D. (2013*a*), *A Rôle for Mereology in Domain Science and Engineering*, Synthese Library (eds. Claudio Calosi and Pierluigi Graziani), Springer, Amsterdam, The Netherlands.

Bjørner, D. (2013*b*), *Domain Science and Engineering as a Foundation for Computation for Humanity*, Computational Analysis, Synthesis, and Design of Dynamic Systems, CRC [Francis & Taylor], chapter 7, pp. 159–177. (eds.: Justyna Zander and Pieter J. Mosterman).

Bjørner, D. (2014*a*), Domain Analysis & Description: Perdurants [Writing to begin Summer/Fall 2014], Research Report, Fredsvej 11, DK-2840 Holte, Denmark.

Bjørner, D. (2014*b*), Domain Analysis: Endurants – An Analysis & Description Process Model, *in* J. Meseguer & K. Ogata, eds, 'Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi', pages 1–34, Springer.

Bjørner, D. (2014*c*), Domain Analysis, Fredsvej 11, DK-2840 Holte, Denmark. (**Note:** This is currently the "definitive" paper on domain description methodology: `www.imm.dtu.dk/~dibj/2014/domain-analysis.pdf`.)

Bjørner, D. & Eir, A. (2010), 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', Vol. 5930 of *Lecture Notes in Computer Science*, Springer, Heidelberg, pp. 22–59.

Bjørner, D. & Havelund, K. (2014), 40 Years of Formal Methods — 8 Obstacle and 3 Possibilities, *in* 'FM 2014, Singapore, May 14-16, 2014', Springer. Distinguished Lecture.

Bjørner, D. & Jones, C. B., eds (1978), *The Vienna Development Method: The Meta-Language*, Vol. 61 of *LNCS*, Springer.

Bjørner, D. & Jones, C. B., eds (1982), *Formal Specification and Software Development*, Prentice-Hall.

Bjørner, N., Browne, A., Colon, M., Finkbeiner, B., Manna, Z., Sipma, H. & Uribe, T. (2000), 'Verifying Temporal Properties of Reactive Systems: A STeP Tutorial', *Formal Methods in System Design* **16**, 227–270.

Bjørner, N., McMillan, K. & Rybalchenko, A. (2013), Higher-order Program Verification as Satisfiability Modulo Theories with Algebraic Data-types, *in* 'Higher-Order Program Analysis'. http://hopa.cs.rhul.ac.uk/files/proceedings.html.

Cant, A. & Mahony, B. (2012), Safety protocols: a new safety engineering paradigm, *in* Australian System Safety Conference (ASSC 2012).

Clarke, E. M., Grumberg, O. & Peled, D. A. (January 2000), *Model Checking*, The MIT Press, Five Cambridge Center, Cambridge, MA 02142-1493, USA. ISBN 0-262-03270-8.

Fitzgerald, J. & Larsen, P. G. (1998), *Modelling Systems – Practical Tools and Techniques in Software Development*, Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK. ISBN 0-521-62348-0.

George, C. W., Haff, P., Havelund, K., Haxthausen, A. E., Milne, R., Nielsen, C. B., Prehn, S. & Wagner, K. R. (1992), *The RAISE Specification Language*, The BCS Practitioner Series, Prentice-Hall, Hemel Hampstead, England.

George, C. W., Haxthausen, A. E., Hughes, S., Milne, R., Prehn, S. & Pedersen, J. S. (1995), *The RAISE Development Method*, The BCS Practitioner Series, Prentice-Hall, Hemel Hampstead, England.

Fox, C. (2000), *The Ontology of Language: Properties, Individuals and Discourse*. CSLI Publications, Center for the Study of Language and Information, Stanford University, California, ISA, 2000.

Holzmann, G. J. (2003), *The SPIN Model Checker, Primer and Reference Manual*, Addison-Wesley, Reading, Massachusetts.

IEEE Computer Society (1990), 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.

Jackson, D. (2006), *Software Abstractions: Logic, Language, and Analysis*, The MIT Press, Cambridge, Mass., USA. ISBN 0-262-10114-9.

Kaufmann, M., Manolios, P. & Moore, J. S. (2000*a*), *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Academic Publishers.

Kaufmann, M., Manolios, P. & Moore, J. S. (2000*b*), *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers.

Lamport, L. (2002), *Specifying Systems*, Addison–Wesley, Boston, Mass., USA.

Luschei, E. (1962), *The Logical Systems of Leśniewksi*, North Holland, Amsterdam, The Netherlands.

Mellor, D.H. & Oliver, A., editors (1997), *Properties*. Oxford Readings in Philosophy. Oxford Univ Press, May 1997. ISBN: 0198751761, 320 pages.

Nipkow, T., Paulson, L. C. & Wenzel, M. (2002), *Isabelle/HOL, A Proof Assistant for Higher-Order Logic*, Vol. 2283 of *Lecture Notes in Computer Science*, Springer-Verlag.

Randell, B. (2003), On Failures and Faults, *in* 'FME 2003: Formal Methods', Vol. 2805 of *Lecture Notes in Computer Science*, Formal Methods Europe, Springer, pp. 18–39. Invited paper.

Shankar, N., Owre, S., Rushby, J. M. & Stringer-Calvert, D. W. J. (1999), *PVS Prover Guide*, Computer Science Laboratory, SRI International, Menlo Park, CA.

Woodcock, J. C. P. & Davies, J. (1996), *Using Z: Specification, Proof and Refinement*, Prentice Hall International Series in Computer Science.
URL: *http://www.comlab.ox.ac.uk/usingz.html*

Zhou, C. C. & Hansen, M. R. (2004), *Duration Calculus: A Formal Approach to Real–time Systems*, Monographs in Theoretical Computer Science. An EATCS Series, Springer–Verlag.