To be submitted for publication in Formal Aspects of Computing

From Domain Descriptions to Requirements Prescriptions:

A Different Approach to Requirements Engineering

Dines Bjørner

Fredsvej 11, DK-2840 Holte, Denmark.
DTU Compute, Technical University of Denmark, DK-2800 Kgs. Lyngby, Denmark.
E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~dibj

In [Bjø16b, *Manifest Domains: Analysis & Description*] we introduced a method for analysing and describing manifest domains. In this paper we show how to systematically, but, of course, not automatically, "derive" requirements prescriptions from domain descriptions. There are, as we see it, three kinds of requirements: (i) domain requirements — also referred to as the design assumptions, and the (ii) interface requirements and (iii) machine requirements — these two together referred to as the design requirements. The machine is the hardware and software to be developed from the requirements. (i) **Domain requirements** are those requirements which can be expressed sôlely using technical terms of the domain. (ii) **Interface requirements** are those requirements which can be expressed using technical terms of both the domain and the machine. (iii) **Machine requirements** are those requirements which can be expressed sôlely using technical terms of the machine. Since it is the machine we wish to design we refer to (ii-iii) as the design requirements and to (i) as the design assumptions. We show principles, techniques and tools for "deriving" domain requirements. The domain requirements development focus on (i.1) projection, (i.2) instantiation, (i.3) determination, (i.4) extension and (i.5) fitting. These domain-to-requirements operators can be described briefly: (i.1) projection removes such descriptions which are to be omitted for consideration in the requirements, (i.2) instantiation mandates specific mereologies, (i.3) determination specifies less non–determinism, (i.4) extension extends the evolving requirements prescription with further domain description aspects and (i.5) fitting resolves "loose ends" as they may have emerged during the domain-to-requirements operations. Domain requirements will also be referred to a design assumptions. We briefly review principles, techniques and tools for "deriving" interface requirements based on sharing domain (ii.1) endurants, (ii.2) actions, events and behaviours with their machine correspondants. Interface and machine requirements (next) will also be referred to as design requirements. Finally we review machine requirements while relating a number of machine requirements aspects to domain phenomena. The "twist" here is that we introduce the machine requirements concepts of (iii.1) derived machine requirements (or just derived requirements), (iii.2) technology requirements and (iii.3) development requirements. The reason for exploring machine requirements in some detail is to analyse requirements issues in the light of the domain concept. We think that there is some clarification to be gained. We claim that our approach contributes to a restructuring of the field of requirements engineering and its very many diverse concerns, a structuring that is logically motivated and is based on viewing software specifications as mathematical objects.

Keywords: Requirements engineering, domain description, design assumptions, design requirements

Correspondence and offprint requests to: Dines Bjørner, Fredsvej 11, DK-2840 Holte, Denmark. e-mail: bjorner@gmail.com

1. Introduction

In [Bjø16b, *Manifest Domains: Analysis & Description*] we introduced a method for analysing and describing manifest domains. In this paper we show how to systematically, but, of course, not automatically, "derive" requirements prescriptions from domain descriptions.

1.1. The Triptych Dogma of Software Development

We see software development progressing as follows: *Before one can design software one must have a firm grasp of the requirements. Before one can prescribe requirements one must have a reasonably firm grasp of the domain.* Software engineering, to us, therefore include these three phases: *domain engineering, requirements engineering* and *software design.*

1.2. Software As Mathematical Objects

Our base view is that computer programs are mathematical objects. That is, the text that makes up a computer program can be reasoned about. This view entails that computer program specifications can be reasoned about. And that the requirements prescriptions upon which these specifications are based can be reasoned about. This base view entails, therefore, that specifications, whether software design specifications, or requirements prescriptions, or domain descriptions, must [also] be formal specifications. This is in contrast to considering software design specifications being artifacts of sociological, or even of psychological "nature".

1.3. The Contribution of This Paper

We claim that the present paper content contributes to our understanding and practice of software engineering as follows: (i) it shows how the new phase of engineering, domain engineering, as introduced in [Bjø16b], forms a prerequisite for requirements engineering; (ii) it endows the "classical" form of requirements engineering with a structured set of development stages and steps: (a) first a domain requirements stage, (b) to be followed by an interface requirements stages, and (c) to be concluded by a machine requirements stage; (iii) it further structures and gives a reasonably precise contents to the stage of domain requirements: (1) first a projection and simplification step, (2) then an instantiation step, (3) then a determination step, (4) then an extension step, and (5) finally a fitting step — with these five steps possibly being iterated; (iv) it also structures and gives a reasonably precise contents to the stage of interface requirements based on a notion of shared entities; and (v) it finally structures and gives a reasonably precise contents to the stage of machine requirements: (α) soft requirements, (β) technology requirements and (γ) development requirements. Stages (a–c) and steps (1–5, α–γ), we claim, are new. They reflect a serious contribution, we claim, to a logical structuring of the field of requirements engineering and its very many otherwise seemingly diverse concerns.

1.4. Some Comments on the Paper Content

This paper is, perhaps, unusual in the following respects: (i) It is a methodology[1] paper, hence there are no "neat" theories, no succinctly expressed propositions, lemmas nor theorems, and hence no proofs. (ii) As a consequence the paper is borne by many, and by extensive examples. (iii) The examples of this paper are all focused on a generic road transport net. (iv) To reasonably fully exemplify the requirements approach, illustrating how our method copes with a seeming complexity of interrelated method aspects, the full example of this paper embodies very, very many description and prescription elements: hundreds of concepts (types, axioms, functions). (v) This methodology paper covers a "grand" area of software engineering: Many textbooks and papers are written on *Requirements Engineering*. We postulate, in contrast to all such books (and papers), that requirements engineering should be founded on domain engineering. Hence we must, somehow, show that our approach relates to major elements of what the *Requirements Engineering* books put forward. (vi) As a result this paper is long.

[1] By **methodology** we understand the study and knowledge of one or more methods ■ By a **method** understand the study and knowledge of the principle, techniques and tools for constructing some artifact, here (primarily) software ■

1.5. Structure of Paper

The structure of the paper is as follows: Section 2 provides a fair-sized, hence realistic example Sections 3–6 covers our approach to requirements development. Section 3 overviews the issue of 'requirements', relates our approach (Sects. 4–6) to *Systems*, *User and External Equipment* and *Functional Requirements*, and Sect. 3 also introduces the concepts of the *machine* to be requirements prescribed, the *domain*, the *interface* and the *machine requirements*. Section 4 covers the *domain requirements* stages of *projection* (Sect. 4.1), *instantiation* (Sect. 4.2), *determination* (Sect. 4.3), *extension* (Sect. 4.4) and *fitting* (Sect. 4.5). Section 5 covers key features of *interface requirements*: shared phenomena (Sect. 5.1), shared endurants (Sect. 5.2) and shared actions, shared eventsand shared behaviours (Sect. 5.3). Section 6 covers key features of *machine requirements*: soft requirements (Sect. 6.2), technology requirements (Sect. 6.3) and development requirements. (Sect. 6.4) Section 7 concludes the paper.

2. An Example Domain: Transport

In order to exemplify the various stages and steps of requirements development we first bring a domain description example. The example follows the steps of an idealised domain description. First we describe the endurants, then we describe the perdurants. Endurant description initially focus on the composite and atomic parts. Then on their "internal" qualities: unique identifications, mereologies, and attributes. The descriptions alternate between enumerated, i.e., labeled narrative sentences and correspondingly "numbered" formalisations. The narrative labels cum formula numbers will be referred to, frequently in the various steps of domain requirements development.

2.1. Endurants

Since we have chosen a manifest domain, that is, a domain whose endurants can be pointed at, seen, touched, we shall follow the analysis & description process as outlined in [Bjø16b] and formalised in [Bjø14b]. That is, we first identify, analyse and describe (manifest) parts, composite and atomic, abstract (Sect. 2.1.1) or concrete (Sect. 2.1.2). Then we identify, analyse and describe their unique identifiers (Sect. 2.1.3), mereologies (Sect. 2.1.4), and attributes (Sects. 2.1.5–2.1.5).

2.1.1. Domain, Net, Fleet and Monitor

Applying `observe_part_sorts` [Bjø16b, Sect. 3.1.6] to to a transport domain δ:Δ yields the following.

The root domain, Δ, is that of a composite traffic system (1.a..) with a road net, (1.b..) with a fleet of vehicles and (1.c..) of whose individual position on the road net we can speak, that is, monitor.[2]

1. We analyse the traffic system into
   a. a composite road net,
   b. a composite fleet (of vehicles), and
   c. an atomic monitor.

type
1. Δ
1.a. N
1.b. F
1.c. M

value
1.a. **obs_part_**N: $\Delta \to$ N
1.b. **obs_part_**F: $\Delta \to$ F
1.c. **obs_part_**M: $\Delta \to$ M

Applying `observe_part_sorts` [Bjø16b, Sect. 3.1.6] to a net, n:N, yields the following.

2. The road net consists of two composite parts,
   a. an aggregation of hubs and
   b. an aggregation of links.

type
2.a. HA
2.b. LA

value
2.a. **obs_part_**HA: N $\to$ HA
2.b. **obs_part_**LA: N $\to$ LA

[2] The monitor can be thought of, i.e., conceptualised. It is not necessarily a physically manifest phenomenon.

2.1.2. Hubs and Links

Applying `observe_part_types` [Bjø16b, Sect. 3.1.7] to hub and link aggregates yields the following.

3. Hub aggregates are sets of hubs.
4. Link aggregates are sets of links.
5. Fleets are set of vehicles.

type
3. H, HS = H-**set**
4. L, LS = L-**set**
5. V, VS = V-**set**

value
3. **obs_part**_HS: HA → HS
4. **obs_part**_LS: LA → LS
5. **obs_part**_VS: F → VS

6. We introduce some auxiliary functions.
   a. links extracts the links of a network.
   b. hubs extracts the hubs of a network.

value
6.a. links: Δ → L-**set**
6.a. links(δ) ≡ **obs_part**_LS(**obs_part**_LA(**obs_part**_N(δ)))
6.b. hubs: Δ → H-**set**
6.b. hubs(δ) ≡ **obs_part**_HS(**obs_part**_HA(**obs_part**_N(δ)))

2.1.3. Unique Identifiers

Applying `observe_unique_identifier` [Bjø16b, Sect. 3.2] to the observed parts yields the following.

7. Nets, hub and link aggregates, hubs and links, fleets, vehicles and the monitor all
   a. have unique identifiers
   b. such that all such are distinct, and
   c. with corresponding observers.

type
7.a. NI, HAI, LAI, HI, LI, FI, VI, MI
value
7.c. **uid**_NI: N → NI
7.c. **uid**_HAI: HA → HAI
7.c. **uid**_LAI: LA → LAI
7.c. **uid**_HI: H → HI
7.c. **uid**_LI: L → LI
7.c. **uid**_FI: F → FI
7.c. **uid**_VI: V → VI
7.c. **uid**_MI: M → MI
axiom
7.b. NI⋂HAI=Ø, NI⋂LAI=Ø, NI⋂HI=Ø, etc.

where axiom 7.b.. is expressed semi-formally, in mathematics. We introduce some auxiliary functions:

8. xtr_lis extracts all link identifiers of a traffic system.
9. xtr_his extracts all hub identifiers of a traffic system.
10. Given an appropriate link identifier and a net get_link 'retrieves' the designated link.
11. Given an appropriate hub identifier and a net get_hub 'retrieves' the designated hub.

value
8. xtr_lis: Δ → LI-**set**
8. xtr_lis(δ) ≡
8. **let** ls = links(δ) **in** {**uid**_LI(l)|l:L•l ∈ ls} **end**
9. xtr_his: Δ → HI-**set**
9. xtr_his(δ) ≡
9. **let** hs = hubs(δ) **in** {**uid**_HI(h)|h:H•k ∈ hs} **end**
10. get_link: LI → Δ ~→ L
10. get_link(li)(δ) ≡
10. **let** ls = links(δ) **in**
10. **let** l:L • l ∈ ls ∧ li=**uid**_LI(l) **in** l **end end**
10. **pre**: li ∈ xtr_lis(δ)
11. get_hub: HI → Δ ~→ H
11. get_hub(hi)(δ) ≡
11. **let** hs = hubs(δ) **in**
11. **let** h:H • h ∈ hs ∧ hi=**uid**_HI(h) **in** h **end end**
11. **pre**: hi ∈ xtr_his(δ)

2.1.4. Mereology

We cover the mereologies of all part sorts introduced so far. We decide that nets, hub aggregates, link aggregates and fleets have no mereologies of interest.Applying `observe_mereology` [Bjø16b, Sect. 3.3.2] to hubs, links, vehicles and the monitor yields the following.

12. Hub mereologies reflect that they are connected to zero, one or more links.
13. Link mereologies reflect that they are connected to exactly two distinct hubs.
14. Vehicle mereologies reflect that they are connected to the monitor.
15. The monitor mereology reflects that it is connected to all vehicles.
16. For all hubs of any net it must be the case that their mereology designates links of that net.
17. For all links of any net it must be the case that their mereologies designates hubs of that net.
18. For all transport domains it must be the case that
    a. the mereology of vehicles of that system designates the monitor of that system, and that
    b. the mereology of the monitor of that system designates vehicles of that system.

value
12. **obs_mereo_**H: H → LI-**set**
13. **obs_mereo_**L: L → HI-**set**

axiom
13. ∀ l:L•**card obs_mereo_**L(l)=2

value
14. **obs_mereo_**V: V → MI
15. **obs_mereo_**M: M → VI-**set**

axiom
16. ∀ δ:Δ, hs:HS•hs=hubs(δ), ls:LS•ls=links(δ) •
16. ∀ h:H•h ∈ hs•**obs_mereo_**H(h)⊆xtr_lis(δ) ∧
17. ∀ l:L•l ∈ ls•**obs_mereo_**L(l)⊆xtr_his(δ) ∧
18.a. **let** f:F•f=**obs_part_**F(δ) ⇒
18.a. **let** m:M•m=**obs_part_**M(δ),
18.a. vs:VS•vs=**obs_part_**VS(f) **in**
18.a. ∀ v:V•v ∈ vs⇒**uid_**V(v) ∈ **obs_mereo_**M(m)
18.b. ∧ **obs_mereo_**M(m) = {**uid_**V(v)|v:V•v ∈ vs}
18.b. **end end**

2.1.5. Attributes, I

We may not have shown all of the attributes mentioned below — so consider them informally introduced !

- **Hubs:** *location*s[3] are considered static, *hub state*s and *hub state space*s are considered programmable;
- **Links:** *length*s and *location*s are considered static, *link state*s and *link state space*s are considered programmable;
- **Vehicles:** *manufacturer name*, *engine type* (whether diesel, gasoline or electric) and *engine power* (kW/horse power) are considered static; *velocity* and *acceleration* may be considered reactive (i.e., a function of gas pedal position, etc.), *global position* (informed via a `GNSS: Global Navigation Satellite System`) and *local position* (calculated from a global position) are considered biddable ■

Applying `observe_attributes` [Bjø16b, Sect. 3.4.3] to hubs, links, vehicles and the monitor yields the following.

First hubs.

19. Hubs
    a. have geodetic locations, GeoH,
    b. have *hub states* which are sets of pairs of identifiers of links connected to the hub[4],
    c. and have *hub state spaces* which are sets of hub states[5].
20. For every net,
    a. link identifiers of a hub state must designate links of that net.
    b. Every hub state of a net must be in the hub state space of that hub.
21. We introduce an auxiliary function: xtr_lis extracts all link identifiers of a hub state.

type
19.a. GeoH
19.b. HΣ = (LI×LI)-**set**
19.c. HΩ = HΣ-**set**

value
19.a. **attr_**GeoH: H → GeoH
19.b. **attr_**HΣ: H → HΣ
19.c. **attr_**HΩ: H → HΩ

axiom
20. ∀ δ:Δ,
20. **let** hs = hubs(δ) **in**
20. ∀ h:H • h ∈ hs •
20.a. xtr_lis(h)⊆xtr_lis(δ)
20.b. ∧ **attr_**Σ(h) ∈ **attr_**Ω(h)
20. **end**

value
21. xtr_lis: H → LI-**set**
21. xtr_lis(h) ≡
21. {li | li:LI,(li′,li″):LI×LI •
21. (li′,li″) ∈ **attr_**HΣ(h) ∧ li ∈ {li′,li″}}

Then links.

22. Links have lengths.
23. Links have geodetic location.
24. Links have states and state spaces:
    a. States modeled here as pairs, (hi', hi''), of identifiers the hubs with which the links are connected and indicating directions (from hub h' to hub h''.) A link state can thus have 0, 1, 2, 3 or 4 such pairs.
    b. State spaces are the set of all the link states that a link may enjoy.

type
22. LEN
23. GeoL

[3] By location we mean a geodetic position.
[4] A hub state "signals" which input-to-output link connections are open for traffic.
[5] A hub state space indicates which hub states a hub may attain over time.

24.a. $L\Sigma = (HI \times HI)$-**set**
24.b. $L\Omega = L\Sigma$-**set**
value
22. **attr**_LEN: L → LEN
23. **attr**_GeoL: L → GeoL
24.a. **attr**_LΣ: L → LΣ
24.b. **attr**_LΩ: L → LΩ
axiom
24. ∀ n:N •
24. **let** ls = xtr−links(n), hs = xtr_hubs(n) **in**
24. ∀ l:L•l ∈ ls ⇒
24.a. **let** lσ = **attr**_LΣ(l) **in**
24.a. 0≤**card** lσ≤4
24.a. ∧ ∀ (hi′,hi″):(HI×HI)•(hi′,hi″) ∈ lσ
24.a. ⇒ {hi′,hi″}=**obs_mereo**_L(l)
24.b. ∧ **attr**_LΣ(l) ∈ **attr**_LΩ(l)
24. **end end**

Then vehicles.

25. Every vehicle of a traffic system has a position which is either 'on a link' or 'at a hub'.
    a. An 'on a link' position has four elements: a unique link identifier which must designate a link of that traffic system and a pair of unique hub identifiers which must be those of the mereology of that link.
    b. The 'on a link' position real is the fraction, thus properly between 0 (zero) and 1 (one) of the length from the first identified hub "down the link" to the second identifier hub.
    c. An 'at a hub' position has three elements: a unique hub identifier and a pair of unique link identifiers — which must be in the hub state.

type
25. VPos = onL | atH
25.a. onL :: LI HI HI R
25.b. R = **Real** **axiom** ∀ r:R • 0≤r≤1
25.c. atH :: HI LI LI
value
25. **attr**_VPos: V → VPos
axiom
25.a. ∀ n:N, onL(li,fhi,thi,r):VPos •
25.a. ∃ l:L•l ∈**obs_part**_LS(**obs_part**_N(n))
25.a. ⇒ li=**uid**_L(l)∧{fhi,thi}=**obs_mereo**_L(l),
25.c. ∀ n:N, atH(hi,fli,tli):VPos •
25.c. ∃ h:H•h ∈**obs_part**_HS(**obs_part**_N(n))
25.c. ⇒ hi=**uid**_H(h)∧(fli,tli) ∈ **attr**_LΣ(h)

26. We introduce an auxiliary function `distribute`.
    a. `distribute` takes a net and a set of vehicles and
    b. generates a map from vehicles to distinct vehicle positions on the net.
    c. We sketch a "formal" `distribute` function, but, for simplicity we omit the technical details that secures distinctness — and leave that to an axiom !

type
26.b. MAP = VI $\overrightarrow{m}$ VPos
axiom
26.b. ∀ map:MAP • **card dom** map = **card rng** map
value
26. distribute: VS → N → MAP
26. distribute(vs)(n) ≡
26.a. **let** (hs,ls) = (xtr_hubs(n),xtr_links(n)) **in**
26.a. **let** vps = {onL(**uid**_(l),fhi,thi,r) |
26.a. l:L•l ∈ls∧{fhi,thi}
26.a. ⊆**obs_mereo**_L(l)∧0≤r≤1}
26.a. ∪ {atH(**uid**_H(h),fli,tli)|
26.a. h:H•h ∈hs∧{fli,tli}
26.a. ⊆**obs_mereo**_H(h)} **in**
26.b. [**uid**_V(v)↦vp|v:V,vp:VPos•v ∈vs∧vp∈vps]
26. **end end**

27. We define two auxiliary functions:
    a. xtr_links extracts all links of a net and
    b. xtr_hub extracts all hubs of a net.

27.a. xtr_links: N → L-**set**
27.a. xtr_links(n)≡
27.a. **obs_part**_LS(**obs_part**_LA(n))
27.b. xtr_hubs: N → H-**set**
27.a. xtr_hubs(n)≡
27.a. **obs_part**_H(**obs_part**_HA_Δ(n))

And finally monitors. We consider only one monitor attribute.

28. The monitor has a vehicle traffic attribute.
    a. For every vehicle of the road transport system the vehicle traffic attribute records a possibly empty list of time marked vehicle positions.
    b. These vehicle positions are alternate sequences of 'on link' and 'at hub' positions
        i such that any sub-sequence of 'on link' positions record the same link identifier, the same pair of ''to' and 'from' hub identifiers and increasing fractions,
        ii such that any sub-segment of 'at hub' positions are identical,
        iii such that vehicle transition from a link to a hub is commensurate with the link and hub mereologies, and
        iv such that vehicle transition from a hub to a link is commensurate with the hub and link mereologies.

type
28. Traffic = VI $\overrightarrow{m}$ (T × VPos)*
value
28. **attr**_Traffic: M → Traffic

```
axiom
28.b.  ∀ δ:Δ •
28.b.    let m = obs_part_M(δ) in
28.b.    let tf = attr_Traffic(m) in
28.b.    dom tf ⊆ xtr_vis(δ) ∧
28.b.    ∀ vi:VI • vi ∈ dom tf •
28.b.      let tr = tf(vi) in
28.b.      ∀ i,i+1:Nat • {i,i+1}⊆dom tr •
28.b.        let (t,vp)=tr(i),(t′,vp′)=tr(i+1) in
28.b.        t<t′
28.b.i       ∧ case (vp,vp′) of
28.b.i           (onL(li,fhi,thi,r),onL(li′,fhi′,thi′,r′))
28.b.i             → li=li′∧fhi=fhi′∧thi=thi′∧r≤r′ ∧ li ∈ xtr_lis(δ) ∧ {fhi,thi} = obs_mereo_L(get_link(li)(δ)),
28.b.ii          (atH(hi,fli,tli),atH(hi′,fli′,tli′))
28.b.ii            → hi=hi′∧fli=fli′∧tli=tli′ ∧ hi ∈ xtr_his(δ) ∧ (fli,tli) ∈ obs_mereo_H(get_hub(hi)(δ)),
28.b.iii         (onL(li,fhi,thi,1),atH(hi,fli,tli))
28.b.iii           → li=fli∧thi=hi ∧ {li,tli} ⊆ xtr_lis(δ) ∧ {fhi,thi}=obs_mereo_L(get_link(li)(δ))
28.b.iii            ∧ hi ∈ xtr_his(δ) ∧ (fli,tli) ∈ obs_mereo_H(get_hub(hi)(δ)),
28.b.iv          (atH(hi,fli,tli),onL(li′,fhi′,thi′,0))
28.b.iv            → etcetera,
28.b.          _ → false
28.b.    end end end end end
```

2.2. Perdurants

Our presentation of example perdurants is not as systematic as that of example endurants. Give the simple basis of endurants covered above there is now a huge variety of perdurants, so we just select one example from each of the three classes of perdurants (as outline in [Bjø16b]): a simple hub insertion *action* (Sect. 2.2.1), a simple link disappearance *event* (Sect. 2.2.2) and a not quite so simple *behaviour*, that of road traffic (Sect. 2.2.3).

2.2.1. Hub Insertion Action

29. Initially inserted hubs, h, are characterised
    a. by their unique identifier which not one of any hub in the net, n, into which the hub is being inserted,
    b. by a mereology, {}, of zero link identifiers, and
    c. by — whatever — attributes, *attrs*, are needed.
30. The result of such a hub insertion is a net, n',
    a. whose links are those of n, and
    b. whose hubs are those of n augmented with h.

```
value
29.  insert_hub: H → N → N
30.  insert_hub(h)(n) as n′
29.a.    pre: uid_H(h) ∉ xtr_his(n)
29.b.     ∧ obs_mereo_H= {}
29.c.     ∧ ...
30.a.    post: obs_part_Ls(n) = obs_part_Ls(n′)
30.b.        ∧ obs_part_Hs(n) ∪ {h} = obs_part_Hs(n′)
```

2.2.2. Link Disappearance Event

We formalise aspects of the link disappearance event:

31. The result net, n':N', is not well-formed.
32. For a link to disappear there must be at least one link in the net;
33. and such a link may disappear such that
34. it together with the resulting net makes up for the "original" net.

```
value
31.  link_diss_event: N × N′ × Bool
31.  link_diss_event(n,n′) as tf
32.    pre: obs_part_Ls(obs_part_LS(n))≠{}
33.    post: ∃ l:L•l ∈ obs_part_Ls(obs_part_LS(n)) ⇒
34.          l ∉ obs_part_Ls(obs_part_LS(n′))
34.          ∧ n′ ∪ {l} = obs_part_Ls(obs_part_LS(n))
```

2.2.3. Road Traffic

The analysis & description of the road traffic behaviour is composed (i) from the description of the global values of nets, links and hubs, vehicles, monitor, a clock, and an initial distribution, *map*, of vehicles, "across" the net; (ii) from the description of channels between vehicles and the monitor; (iii) from the description of behaviour signatures, that is, those of the overall road traffic system, the vehicles, and the monitor; and (iv) from the description of the individual behaviours, that is, the overall road traffic system, *rts*, the individual vehicles, *veh*, and the monitor, *mon*.

Global Values: There is given some globally observable parts.

35. besides the domain, δ:Δ,
36. a net, n:N,
37. a set of vehicles, vs:V-**set**,
38. a monitor, m:M, and
39. a clock, clock, behaviour.
40. From the net and vehicles we generate an initial distribution of positions of vehicles.

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

value
35. δ:Δ
36. n:N = **obs_part**_N(δ),
36. ls:L-**set**=links(δ),hs:H-**set**=hubs(δ),
36. lis:LI-**set**=xtr_lis(δ),his:HI-**set**=xtr_his(δ)
37. va:VS=**obs_part**_VS(**obs_part**_F(δ)),
37. vs:Vs-**set**=**obs_part**_Vs(va),
37. vis:VI-**set** = {**uid**_VI(v)|v:V•v ∈ vs},
38. m:**obs_part**_M(δ),
38. mi=**uid**_MI(m),
38. ma:**attributes**(m)
39. clock: $\mathbb{T}$ → **out** {clk_ch[vi|vi:VI•vi ∈ vis]} **Unit**
40. vm:MAP•vpos_map = distribute(vs)(n);

Channels:

41. We additionally declare a set of vehicle-to-monitor-channels indexed
    a. by the unique identifiers of vehicles
    b. and the (single) monitor identifier.[6]

    and communicating vehicle positions.

channel
41. {v_m_ch[vi,mi]|vi:VI•vi ∈ vis}:VPos

Behaviour Signatures:

42. The road traffic system behaviour, rts, takes no arguments (hence the first **Unit**)[7]; and "behaves", that is, continues forever (hence the last **Unit**).
43. The vehicle behaviour
    a. is indexed by the unique identifier, uid_V(v):VI,
    b. the vehicle mereology, in this case the single monitor identifier mi:MI,
    c. the vehicle attributes, obs_attribs(v)
    d. and — factoring out one of the vehicle attributes — the current vehicle position.
    e. The vehicle behaviour offers communication to the monitor behaviour (on channel vm_ch[vi]); and behaves "forever".
44. The monitor behaviour takes
    a. the monitor identifier,
    b. the monitor mereology,
    c. the monitor attributes,
    d. and — factoring out one of the vehicle attributes — the discrete road traffic, drtf:dRTF, being repeatedly "updated" as the result of **in**put communications from (all) vehicles;
    e. the behaviour otherwise behaves forever.

value
42. rts: **Unit** → **Unit**
43. veh: vi:VI × mi:MI → vp:VPos → **out** vm_ch[vi,mi] **Unit**
44. mon: m:M × vis:VI-**set** → RTF → **in** {v_m_ch[vi,mi]|vi:VI•vi ∈ vis},clk_ch **Unit**

The Road Traffic System Behaviour:

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

value
45. rts() =
45.a. ‖ {veh(**uid**_VI(v),mi)(vm(**uid**_VI(v)))|v:V•v ∈ vs}
45.b. ‖ mon(mi,vis)([vi↦⟨⟩|vi:VI•vi ∈ vis])

where, wrt, the monitor, we dispense with the mereology and the attribute state arguments and instead just have a monitor traffic argument which records the discrete road traffic, MAP, initially set to "empty" traces (⟨⟩, of so far "no road traffic"!).

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

46. We describe here an abstraction of the vehicle behaviour **at** a Hub (hi).
    a. Either the vehicle remains at that hub informing the monitor of its position,
    b. or, internally non-deterministically,
        i moves onto a link, tli, whose "next" hub, identified by thi, is obtained from the mereology of the link identified by tli;

[6] Technically speaking: we could omit the monitor identifier.
[7] The **Unit** designator is an RSL technicality.

ii informs the monitor, on channel vm[vi,mi], that it is now at the very beginning (0) of the link identified by tli, whereupon the vehicle resumes the vehicle behaviour positioned at the very beginning of that link,

c. or, again internally non-deterministically, the vehicle "disappears — off the radar" !

47. We describe here an abstraction of the vehicle behaviour **on** a Link (ii). Either
    a. the vehicle remains at that link position informing the monitor of its position,
    b. or, internally non-deterministically, if the vehicle's position on the link has not yet reached the hub,
        i then the vehicle moves an arbitrary increment ℓ_ε (less than or equal to the distance to the hub) along the link informing the monitor of this, or
        ii else,
            A while obtaining a "next link" from the mereology of the hub (where that next link could very well be the same as the link the vehicle is about to leave),
            B the vehicle informs the monitor that it is now at the hub identified by thi, whereupon the vehicle resumes the vehicle behaviour positioned at that hub.
    c. or, internally non-deterministically, the vehicle "disappears — off the radar" !

```
46.  veh(vi,mi)(vp:atH(hi,fli,tli)) ≡
46.a.    v_m_ch[vi,mi]!vp ; veh(vi,mi)(vp)
46.b.  ⌈⌉
46.b.i   let {hi′,thi}=obs_mereo_L(get_link(tli)(n)) in
46.b.i                 assert: hi′=hi
46.b.ii  v_m_ch[vi,mi]!onL(tli,hi,thi,0) ;
46.b.ii  veh(vi,mi)(onL(tli,hi,thi,0)) end
46.c.  ⌈⌉ stop
```

```
47.  veh(vi,mi)(vp:onL(li,fhi,thi,r)) ≡
47.a.    v_m_ch[vi,mi]!vp ; veh(vi,mi,va)(vp)
47.b.  ⌈⌉ if r + ℓε≤1
47.b.i     then
47.b.i       v_m_ch[vi,mi]!onL(li,fhi,thi,r+ℓε) ;
47.b.i       veh(vi,mi)(onL(li,fhi,thi,r+ℓε))
47.b.ii    else
47.b.iiA     let li′:LI•li′ ∈ obs_mereo_H(get_hub(thi)(n)) in
47.b.iiB       v_m_ch[vi,mi]!atH(li,thi,li′);
47.b.iiB       veh(vi,mi)(atH(li,thi,li′)) end end
47.c.  ⌈⌉ stop
```

The Monitor Behaviour

48. The monitor behaviour evolves around
    a. the monitor identifier,
    b. the monitor mereology,
    c. and the attributes, ma:ATTR
    d. — where we have factored out as a separate arguments — a table of traces of time-stamped vehicle positions,
    e. while accepting messages
        i about time
        ii and about vehicle positions
    f. and otherwise progressing "in[de]finitely".

49. Either the monitor "does own work"
50. or, internally non-deterministically accepts messages from vehicles.
    a. A vehicle position message, vp, may arrive from the vehicle identified by vi.
    b. That message is appended to that vehicle's movement trace – prefixed by time (obtained from the time channel),
    c. whereupon the monitor resumes its behaviour —
    d. where the communicating vehicles range over all identified vehicles.

```
48.  mon(mi,vis)(trf) ≡
49.      mon(mi,vis)(trf)
50.  ⌈⌉
50.a.    []{let tvp = (clk_ch?,v_m_ch[vi,mi]?) in
50.b.       let trf′ = trf † [vi ↦ trf(vi)^<tvp>] in
50.c.       mon(mi,vis)(trf′)
50.d.       end end | vi:VI • vi ∈ vis}
```

We are about to complete a long, i.e., a 7.5 page example (!). We can now comment on the full example: The domain, $\delta : \Delta$ is a manifest part. The road net, $n : N$ is also a manifest part. The fleet, $f : F$, of vehicles, $vs : VS$, likewise, is a manifest part. But the monitor, $m : M$, is a concept. One does not have to think of it as a manifest "observer". The vehicles are on — or off — the road (i.e., links and hubs). We know that from a few observations and generalise to all vehicles. They either move or stand still. We also, similarly, know that. Vehicles move. Yes, we know that. Based on all these repeated observations and generalisations we introduce the concept of vehicle traffic. Unless positioned high above a road net — and with good binoculars — a single person cannot really observe the traffic. There are simply too many links, hubs, vehicles, vehicle positions and times. Thus we conclude that, even in a richly manifest domain, we can also "speak of", that is, describe concepts over manifest phenomena, including time !

2.3. Domain Facets

The example of this section, i.e., Sect. 2, does not reflect the concepts of domain facets such as (i) domain intrinsics, (ii) domain support technologies, (iii) domain rules, regulations & scripts, (iv) organisation & management, and (v) human behaviour. These facets are covered in [Bjø10a, 2008].

3. Requirements

This and the next three sections, Sects. 4–6, are the main sections of this paper. Section 4 is the most detailed and systematic section. It covers the *domain requirements* operations of projection, instantiation, determination, extension and, less detailed, fitting. Section 5 surveys the *interface requirements* issues of *shared phenomena*: shared endurants, shared actions, shared events and shared behaviour, and "completes" the exemplification of the detailed *domain extension* of our requirements into a *road pricing system*. Section 6 relates some machine requirements issues to the overall design of the *road pricing system*: derived requirements, technology requirements and development requirements. This the, initial section captures main concepts and principles of requirements.

• • •

Definition 1. Requirements (I): By a **requirements** we understand (cf., [IEE90, IEEE Standard 610.12]): *"A condition or capability needed by a user to solve a problem or achieve an objective"* ▪

The objective of requirements engineering is to create a requirements prescription: A **requirements prescription** specifies observable properties of endurants and perdurants of ***the machine*** such as the requirements stake-holders wish them to be ▪ The **machine** is what is required: that is, the *hardware* and *software* that is to be designed and which are to satisfy the requirements ▪ A requirements prescription thus (putatively) expresses what there should be. A requirements prescription expresses nothing about the design of the possibly desired (required) software. But as the requirements prescription is presented in the form of a model, one can base the design on that model. We shall show how a major part of a requirements prescription can be "derived" from "its" prerequisite domain description. Note that requirements is about *systems*.

Rule 1. The "Golden Rule" of Requirements Engineering: Prescribe only those requirements that can be objectively shown to hold for the designed software ▪ "Objectively shown" means that the designed software can either be tested, or be model checked, or be proved (verified), to satisfy the requirements. **Caveat**: Since we do not illustrate formal tests, model checking nor theorem proving, we shall, alas, not illustrate adherence to this rule.

Rule 2. An "Ideal Rule" of Requirements Engineering: When prescribing (including formalising) requirements, also formulate tests and properties for model checking and theorems whose proof should show adherence to the requirements ▪ The rule is labelled "ideal" since such precautions will not be shown in this paper. The rule is clear. It is a question for proper management to see that it is adhered to. See the "Caveat" above !

Rule 3. Requirements Adequacy: Make sure that requirements cover what users expect ▪ That is, do not express a requirement for which you have no users, but make sure that all users' requirements are represented or somehow accommodated. In other words: the requirements gathering process needs to be like an extremely "fine-meshed net": One must make sure that all possible stake-holders have been involved in the requirements acquisition process, and that possible conflicts and other inconsistencies have been obviated.

Rule 4. Requirements Implementability: Make sure that requirements are implementable ▪ That is, do not express a requirement for which you have no assurance that it can be implemented. In other words, although the requirements phase is not a design phase, one must tacitly assume, perhaps even indicate, somehow, that an implementation is possible. But the requirements in and by themselves, may stay short of expressing such designs. **Caveat:** The domain and requirements specifications are, in our approach, model-oriented. That helps expressing 'implementability'.

Definition 2: **Requirements (II):** By **requirements** we shall understand a document which prescribes desired properties of a machine: what endurants the machine shall "maintain", and what the machine shall (must; not should) offer of functions and of behaviours while also expressing which events the machine shall "handle" ▪ By a machine that "maintains" endurants we shall mean: a machine which, "between" users' use of that machine, "keeps" the data that represents these entities. From earlier we repeat:

Definition 3: **Machine:** By *machine* we shall understand a, or the, combination of hardware and software that is the target for, or result of the required computing systems development ▪ So this, then, is a main objective of requirements development: to start towards the design of the hardware + software for the computing system.

Definition 4: **Requirements (III):** To specify the machine ▪ When we express requirements and wish to "convert" such requirements to a realisation, i.e., an implementation, then we find that some requirements (parts) imply certain properties to hold of the hardware on which the software to be developed is to "run", and, obviously, that remaining — probably the larger parts of the — requirements imply certain properties to hold of that software.

3.1. Four Requirements Facets

We shall unravel requirements in two stages — (i) the first stage is sketchy (and thus informal) (ii) while the last stage is systematic and both informal and formal. The sketchy stage consists of (i.1) a narrative problem/objective sketch, (i.2) a narrative system requirements sketch, and (i.3) a narrative user & external equipment requirements sketch, (ii) The narrative and formal stage consists of: design assumptions prescription and design requirements prescription, It is systematic, and mandates both strict narrative and formal prescriptions. And it is "derivable" from the domain description.

3.1.1. Problem, Solution and Objective Sketch

Definition 5. Problem, Solution and Objective Sketch: By a problem, solution and objective sketch we understand a narrative which emphasises what the problem to be solved is, outlines a possible solution and sketches an objective of the solution ■

Example 1. The Problem/Objective Requirements: A Sketch: The *problem* is that of traffic congestion. The chosen *solution* is to [build and] operate a toll-road system integrated into a road net and charge toll-road users a usage fee. The *objective* is therefore to create a **road-pricing product.** By a road-pricing product we shall understand an information technology-based system containing computers and communications equipment and software that enables the recording of *vehicle* movements within the *toll-road* and thus enables the *owner* of the road net to charge the *owner* of the vehicles *fees* for the usage of that toll-road ■

3.1.2. Systems Requirements

Definition 6. System Requirements: By a **system requirements narrative** we understand a narrative which emphasises the overall assumed and/or required hardware and software system equipment ■

Example 2. The Road-pricing System Requirements: A Narrative: The requirements are based on the following constellation of system equipment: (i) there is assumed a GNSS: a GLOBAL NAVIGATION SATELLITE SYSTEM; (ii) there are *vehicles* equipped with GNSS receivers; (iii) there is a well-delineated road net called a *toll-road* net with specially equipped *toll-gate*s with *vehicle identification sensor*s, *barrier*s which afford (only specially equipped) vehicles to enter into and exit from the toll-road net; and (iv) there is a *road-pricing calculator*. **The system to be designed (from the requirements) is the *road-pricing calculator*.** These four system elements are required to behave and interact as follows: (a) The `GNSS` is assumed to continuously offer vehicles information about their global position; (b) *vehicles* shall contain a `GNSS receiver` which based on the global position information shall regularly calculate their timed local position and offer this to the *calculator* — while otherwise cruising the general road net as well as the toll-road net, the latter while carefully moving through toll-gates; (c) *toll-gates* shall register the identity of vehicles entering and exiting the toll-road and offer this information to the calculator; and (d) the *calculator* shall accept all messages from vehicles and gates and use this information to record the movements of vehicles and bill these whenever they exit the toll-road. The requirements are therefore to include **assumptions about** [1] the *GNSS* satellite and telecommunications equipment, [2] the vehicle *GNSS receiver* equipment, [3] the vehicle handling of *GNSS* input and forwarding, to the road pricing system, of its interpretation of *GNSS* input, [4] the toll-gate sensor equipment, [5] the toll-gate barrier equipment, [6] the toll-gate handling of entry, vehicle identification and exit sensors and the forwarding of vehicle identification to the road pricing calculator, and [7] the communications between toll-gates and vehicles, on "one side", and the road pricing calculator, on the "other side". It is in this sense that the requirements are for an information technology-based system of both software and hardware — not just hard computer and communications equipment, but also movement sensors and electro-mechanical "gear" ■

3.1.3. User and External Equipment Requirements

Definition 7: **User and External Equipment Requirements:** By a **user and external equipment requirements narrative** we understand a narrative which emphasises assumptions about the human user and external equipment interfaces to the system components ■ The user and external equipment requirements detail, and thus make explicit the assumptions listed in Example 2.

Example 3. The Road-pricing User and External Equipment Requirements: Narrative: The human users of the road-pricing system are, (a–c): (a) *vehicle drivers,* (b) toll-gate sensor, actuator and barrier *service staff,* and (c) the road-pricing calculator *service staff.* The external equipment are, (1–3): (1) firstly, the *GNSS* satellites and the telecommunications equipment which enables *communication* between, (i–iii), (i) the *GNSS* satellites and vehicles, (ii) vehicles and the road-pricing calculator and (iii) toll-gates and the road-pricing calculator. Moreover, the external *equipment* are (2) the toll-gates with their sensors: entry, vehicle identity, and exit, and the barrier actuator. The external *equipment* are, finally, (3), the vehicles ! ■

That is, although we do indeed exemplify domain and requirements aspects of users and external equipment, we do not expect to machine, i.e., to hardware or software design these elements; *they are assumed already implemented !*

3.1.4. Design Requirements

Definition 8. **Assumption and Design Requirements:** By **assumption and design requirements** we understand precise prescriptions of the endurants and perdurants of the (to be designed) system components and the assumptions which that design must rely upon ■

The design will be done, extensively, in the examples of Sects. 5–6. The assumptions upon which the design can be relied upon, that is, shall be verified ("against") are illustrated in Sect. 4

3.2. The Three Phases of Requirements Engineering

There are, as we see it, three kinds of design assumptions and requirements: (i) domain requirements (the primary design assumptions) , (ii) interface requirements and (iii) machine requirements. The last two being the primary design requirements. (i) The **domain requirements** are those requirements which can be expressed sôlely using technical terms of the domain ■ (ii) The **interface requirements** are those requirements which can be expressed only using technical terms of both the domain and the machine ■ (iii) The **machine requirements** are those requirements which, in principle, can be expressed sôlely using technical terms of the machine ■

Definition 9. **Verification Paradigm:** Some preliminary designations: let $\mathscr{D}$ designate the the domain requirements; let $\mathscr{R}$ designate the interface and machine requirements and let $\mathscr{S}$ designate the system design. Now $\mathscr{D}, \mathscr{S} \models \mathscr{R}$ shall be read: it must be verified that the $\mathscr{S}$ystem design satisfies the interface and machine $\mathscr{R}$equirements in the context of the $\mathscr{D}$omain requirements ■

The "in the context of $\mathscr{D}$..." term means that proofs of $\mathscr{S}$oftware design correctness with respect to $\mathscr{R}$equirements will often have to refer to $\mathscr{D}$omain requirements assumptions. We refer to [GGJZ00, Gunter, Jackson, Zave, 2000] for an analysis of a varieties of forms in which $\models$ relate to variants of $\mathscr{D}$, $\mathscr{R}$ and $\mathscr{S}$.

3.3. Order of Presentation of Requirements Prescriptions

The domain requirements development stage — as we shall see — can be sub-staged into: projection, instantiation, determination, extension and fitting. The interface requirements development stage — can be sub-staged into *shared: endurant, action, event and behaviour* developments, where "sharedness" pertains to phenomena shared between, i.e., "present" in, both the domain (concretely, manifestly) and the machine (abstractly, conceptually). These development stages need not be pursued in the order of the three stages and their sub-stages. We emphasize that one thing is the stages and steps of development, as for example these: projection, instantiation, determination, extension, fitting, shared endurants, shared actions, shared events, shared behaviours, etcetera, another thing is the requirements prescription that results from these development stages and steps. The further software development, after and on the basis of the requirements prescription starts only when all stages and steps of the requirements prescription have been fully developed. The domain engineer is now free to rearrange the final prescription, irrespective of the order in which the various sections were developed, in such a way as to give a most pleasing, pedagogic and cohesive reading (i.e., presentation). From such a requirements prescription one can therefore not necessarily see in which order the various sections of the prescription were developed.

3.4. Design Requirements and Design Assumptions

A crucial distinction is between design requirements and design assumptions. The **design requirements** are those requirements for which the system designer **has to** implement hardware or software in order satisfy system user expectations ■ The **design assumptions** are those requirements for which the system designer **does not** have to implement hardware or software, but whose properties the designed hardware, respectively software relies on for proper functioning ■

Example 4. **Road Pricing System — Design Requirements**: The design requirements for the road pricing calculator of this paper are for the design of: (ii) for that part of the vehicle software which interfaces the GNSS receiver and the road pricing

calculator (cf. Items 128.–131.), (iii) for that part of the toll-gate software which interfaces the toll-gate and the road pricing calculator (cf. Items 136.–138.) and (i) the road pricing calculator (cf. Items 167.–180.) ■

Example 5. Road Pricing System — Design Assumptions: The design assumptions for the road pricing calculator include: (i) that *vehicles* behave as prescribed in Items 127.–131., (ii) that the GNSS regularly offers vehicles correct information as to their global position (cf. Item 128.), (iii) that *toll-gates* behave as prescribed in Items 133.–138., and (iv) that the *road net* is formed and well-formed as defined in Examples 10–12 ■

Example 6. Toll-Gate System — Design Requirements: The design requirements for the toll-gate system of this paper are for the design of software for the toll-gate and its interfaces to the road pricing system, i.e., Items 132.–133. ■

Example 7. Toll-Gate System — Design Assumptions: The design assumptions for the toll-gate system include (i) that the vehicles behave as per Items 127.–131., and (ii) that the road pricing calculator behave as per Items 167.–180. ■

4. Domain Requirements

Domain requirements express the assumptions that a design must rely upon in order that that design can be verified.

Definition 10. Domain Requirements Prescription: A **domain requirements prescription** is that subset of the requirements prescription which can be expressed sôlely using terms from the domain description ■

To determine a relevant subset all we need is collaboration with requirements, cum domain stake-holders. Experimental evidence, in the form of example developments of requirements prescriptions from domain descriptions, appears to show that one can formulate techniques for such developments around a few domain-description-to-requirements-prescription operations. We suggest these: projection, instantiation, determination, extension and fitting. In Sect. 3.3 we mentioned that the order in which one performs these domain-description-to-domain-requirements-prescription operations is not necessarily the order in which we have listed them here, but, with notable exceptions, one is well-served in starting out requirements development by following this order.

4.1. Domain Projection & Simplification

Definition 11. Domain Projection: By a **domain projection & simplification** we mean *a subset of the domain description, one which projects out all those endurants: parts, materials and components, as well as perdurants: actions, events and behaviours that the stake-holders do not wish represented or relied upon by the machine. Simplification means that we simplify (refine) some internal qualities, like mereologies and/or attributes* ■

The resulting document is a partial domain requirements prescription. In determining an appropriate subset the requirements engineer must secure that the final "projection prescription" is complete and consistent — that is, that there are no "dangling references", i.e., that all entities and their internal properties that are referred to are all properly defined.

4.1.1. Domain Projection — Narrative

We now start on a series of examples that illustrate domain requirements development.

Example 8. Domain Requirements. Projection: A Narrative Sketch: We require that the road pricing system shall [at most] relate to the following domain entities – and only to these[8]: the net, its links and hubs, and their properties (unique identifiers, mereologies and some attributes), the vehicles, as endurants, and the general vehicle behaviours, as perdurants. We treat projection together with a concept of simplification. The example simplifications are vehicle positions and, related to the simpler vehicle position, vehicle behaviours. To prescribe and formalise this we copy the domain description. From that domain description we remove all mention of the hub insertion action, the link disappearance event, and the monitor ■

As a result we obtain $\Delta_{\not\mathscr{P}}$, the projected version of the domain requirements prescription[9].

[8] By 'relate to ... these' we mean that the required system does not rely on domain phenomena that have been "projected away".
[9] Restrictions of the net to the toll road nets, hinted at earlier, will follow in the next domain requirements steps.

4.1.2. ***Domain Projection — Formalisation***

The requirements prescription hinges, crucially, not only on a systematic narrative of all the projected, instantiated, determinated, extended and fitted specifications, but also on their formalisation. In the formal domain projection example we, regretfully, omit the narrative texts. In bringing the formal texts we keep the item numbering from Sect. 2, where you can find the associated narrative texts.

Example 9. **Domain Requirements — Projection**:

Main Sorts

type
1. $\Delta_{\mathscr{D}}$
1.a. $N_{\mathscr{D}}$
1.b. $F_{\mathscr{D}}$
value
1.a. **obs_part_**$N_{\mathscr{D}}$: $\Delta_{\mathscr{D}} \to N_{\mathscr{D}}$
1.b. **obs_part_**$F_{\mathscr{D}}$: $\Delta_{\mathscr{D}} \to F_{\mathscr{D}}$

type
2.a. $HA_{\mathscr{D}}$
2.b. $LA_{\mathscr{D}}$
value
2.a. **obs_part_**HA: $N_{\mathscr{D}} \to$ HA
2.b. **obs_part_**LA: $N_{\mathscr{D}} \to$ LA ■

Concrete Types

type
3. $H_{\mathscr{D}}$, $HS_{\mathscr{D}} = H_{\mathscr{D}}$**-set**
4. $L_{\mathscr{D}}$, $LS_{\mathscr{D}} = L_{\mathscr{D}}$**-set**
5. $V_{\mathscr{D}}$, $VS_{\mathscr{D}} = V_{\mathscr{D}}$**-set**
value
3. **obs_part_**$HS_{\mathscr{D}}$: $HA_{\mathscr{D}} \to HS_{\mathscr{D}}$

4. **obs_part_**$LS_{\mathscr{D}}$: $LA_{\mathscr{D}} \to LS_{\mathscr{D}}$
5. **obs_part_**$VS_{\mathscr{D}}$: $F_{\mathscr{D}} \to VS_{\mathscr{D}}$
6.a. links: $\Delta_{\mathscr{D}} \to$ **L-set**
6.a. links($\delta_{\mathscr{D}}$) ≡ **obs_part_**$LS_{\mathscr{R}}$(**obs_part_**$LA_{\mathscr{R}}$($\delta_{\mathscr{R}}$))
6.b. hubs: $\Delta_{\mathscr{D}} \to$ **H-set**
6.b. hubs($\delta_{\mathscr{D}}$) ≡ **obs_part_**$HS_{\mathscr{D}}$(**obs_part_**$HA_{\mathscr{D}}$($\delta_{\mathscr{D}}$)) ■

Unique Identifiers

type
7.a. HI, LI, VI, MI
value
7.c. **uid_**HI: $H_{\mathscr{D}} \to$ HI
7.c. **uid_**LI: $L_{\mathscr{D}} \to$ LI

7.c. **uid_**VI: $V_{\mathscr{D}} \to$ VI
7.c. **uid_**MI: $M_{\mathscr{D}} \to$ MI
axiom
7.b. HI⋂LI=Ø, HI⋂VI=Ø, HI⋂MI=Ø,
7.b. LI⋂VI=Ø, LI⋂MI=Ø, VI⋂MI=Ø ■

Mereology

value
12. **obs_mereo_**$H_{\mathscr{D}}$: $H_{\mathscr{D}} \to$ LI**-set**
13. **obs_mereo_**$L_{\mathscr{D}}$: $L_{\mathscr{D}} \to$ HI**-set**
13. **axiom** ∀ l:$L_{\mathscr{D}}$ • **card obs_mereo_**$L_{\mathscr{D}}$(l)=2
14. **obs_mereo_**$V_{\mathscr{D}}$: $V_{\mathscr{D}} \to$ MI
15. **obs_mereo_**$M_{\mathscr{D}}$: $M_{\mathscr{D}} \to$ VI**-set**
axiom
16. ∀ $\delta_{\mathscr{D}}$:$\Delta_{\mathscr{D}}$, hs:HS•hs=hubs(δ), ls:LS•ls=links($\delta_{\mathscr{D}}$) ⇒
16. ∀ h:$H_{\mathscr{D}}$•h ∈ hs ⇒
16. **obs_mereo_**$H_{\mathscr{D}}$(h)⊆xtr_his($\delta_{\mathscr{D}}$) ∧

17. ∀ l:$L_{\mathscr{D}}$•l ∈ ls •
16. **obs_mereo_**$L_{\mathscr{D}}$(l)⊆xtr_lis($\delta_{\mathscr{D}}$) ∧
18.a. **let** f:$F_{\mathscr{D}}$•f=**obs_part_**$F_{\mathscr{D}}$($\delta_{\mathscr{D}}$) ⇒
18.a. vs:$VS_{\mathscr{D}}$•vs=**obs_part_**$VS_{\mathscr{D}}$(f) **in**
18.a. ∀ v:$V_{\mathscr{D}}$•v ∈ vs ⇒
18.a. **uid_**$V_{\mathscr{D}}$(v) ∈ **obs_mereo_**$M_{\mathscr{D}}$(m) ∧
18.b. **obs_mereo_**$M_{\mathscr{D}}$(m)
18.b. = {**uid_**$V_{\mathscr{D}}$(v)|v:V•v ∈ vs}
18.b. **end** ■

Attributes: We project attributes of hubs, links and vehicles.
First **hubs:**

type
19.a. GeoH
19.b. $H\Sigma_{\mathscr{D}}$ = (LI×LI)**-set**
19.c. $H\Omega_{\mathscr{D}} = H\Sigma_{\mathscr{D}}$**-set**
value
19.b. **attr_**$H\Sigma_{\mathscr{D}}$: $H_{\mathscr{D}} \to H\Sigma_{\mathscr{D}}$
19.c. **attr_**$H\Omega_{\mathscr{D}}$: $H_{\mathscr{D}} \to H\Omega_{\mathscr{D}}$

axiom
20. ∀ $\delta_{\mathscr{D}}$:$\Delta_{\mathscr{D}}$,
20. **let** hs = hubs($\delta_{\mathscr{D}}$) **in**
20. ∀ h:$H_{\mathscr{D}}$ • h ∈ hs •
20.a. xtr_lis(h)⊆xtr_lis($\delta_{\mathscr{D}}$)
20.b. ∧ **attr_**$\Sigma_{\mathscr{D}}$(h) ∈ **attr_**$\Omega_{\mathscr{D}}$(h)
20. **end** ■

Then **links:**

type
23. GeoL
24.a. $L\Sigma_{\mathscr{D}} = (HI \times HI)$-**set**
24.b. $L\Omega_{\mathscr{D}} = L\Sigma_{\mathscr{D}}$-**set**

value
23. **attr_**GeoL: L → GeoL
24.a. **attr_**$L\Sigma_{\mathscr{D}}$: $L_{\mathscr{D}} \to L\Sigma_{\mathscr{D}}$
24.b. **attr_**$L\Omega_{\mathscr{D}}$: $L_{\mathscr{D}} \to L\Omega_{\mathscr{D}}$
axiom
24.a.– 24.b. on Page 5.

Finally **vehicles:** For 'road pricing' we need vehicle positions. But, for "technical reasons", we must abstain from the detailed description given in Items 25.–25.c. The 'technical reasons' are that we assume that the *GNSS* cannot provide us with direction of vehicle movement and therefore we cannot, using only the *GNSS* provide the details of 'offset' along a link (*onL*) nor the "from/to link" at a hub (*atH*). We therefore simplify vehicle positions.

51. A simplified vehicle position designates
    a. either a link
    b. or a hub,

type
51. SVPos = SonL | SatH

51.a. SonL :: LI
51.b. SatH :: HI
axiom
25.a.' ∀ n:N, SonL(li):SVPos •
25.a.' ∃ l:L•l ∈**obs_part_**LS(**obs_part_**N(n)) ⇒ li=**uid_**L(l)
25.c.' ∀ n:N, SatH(hi):SVPos •
25.c.' ∃ h:H•h ∈**obs_part_**HS(**obs_part_**N(n)) ⇒ hi=**uid_**H(h)

Global Values

value
35. $\delta_{\mathscr{D}}$:$\Delta_{\mathscr{D}}$,
36. n:$N_{\mathscr{D}}$ = **obs_part_**$N_{\mathscr{D}}(\delta_{\mathscr{D}})$,
36. ls:$L_{\mathscr{D}}$-**set** = links($\delta_{\mathscr{D}}$),

36. hs:$H_{\mathscr{D}}$-**set** = hubs($\delta_{\mathscr{D}}$),
36. lis:LI-**set** = xtr_lis($\delta_{\mathscr{D}}$),
36. his:HI-**set** = xtr_his($\delta_{\mathscr{D}}$)

Behaviour Signatures: We omit the monitor behaviour.

52. We leave the vehicle behaviours' attribute argument undefined.

type

52. ATTR
value
42. $trs_{\mathscr{D}}$: **Unit** → **Unit**
43. $veh_{\mathscr{D}}$: VI×MI×ATTR → ... **Unit**

The System Behaviour: We omit the monitor behaviour.

value
45.a. $trs_{\mathscr{D}}()$=‖{$veh_{\mathscr{D}}$(**uid_**VI(v),**obs_mereo_**V(v),_) | v:$V_{\mathscr{D}}$•v ∈ vs}

The Vehicle Behaviour: Given the simplification of vehicle positions we *simplify* the vehicle behaviour given in Items 46.–47.

46.' veh(vi,mi)(vp:SatH(hi)) ≡
46.a.' v_m_ch[vi,mi]!SatH(hi) ; veh(vi,mi)(SatH(hi))
46.b.i' ⌈ **let** li:LI•li ∈ **obs_mereo_**H(get_hub(hi)(n)) **in**
46.b.ii' v_m_ch[vi,mi]!SonL(li) ; veh(vi,mi)(SonL(li)) **end**
46.c.' ⌈ **stop**

47.' veh(vi,mi)(vp:SonL(li)) ≡
47.a.' v_m_ch[vi,mi]!SonL(li) ; veh(vi,mi,va)(SonL(li))
47.b.iiA' ⌈ **let** hi:HI•hi ∈ **obs_mereo_**L(get_link(li)(n)) **in**
47.b.iiB' v_m_ch[vi,mi]!SatH(hi) ; veh(vi,mi)(atH(hi)) **end**
47.c.' ⌈ **stop**

We can simplify Items 46.′–47.c.′ further.

53. veh(vi,mi)(vp) ≡
54. v_m_ch[vi,mi]!vp ; veh(vi,mi,va)(vp)
55. ⌈ **case** vp **of**
55. SatH(hi) →
56. **let** li:LI•li ∈ **obs_mereo_**H(get_hub(hi)(n)) **in**
57. v_m_ch[vi,mi]!SonL(li) ; veh(vi,mi)(SonL(li)) **end**,
55. SonL(li) →
58. **let** hi:HI•hi ∈ **obs_mereo_**L(get_link(li)(n)) **in**
59. v_m_ch[vi,mi]!SatH(hi) ; veh(vi,mi)(atH(hi)) **end end**
60. ⌈ **stop**

53. This line coalesces Items 46.′ and 47.′.
54. Coalescing Items 46.a.′ and 47.′.
55. Captures the distinct parameters of Items 46.′ and 47.′.
56. Item 46.b.i′.
57. Item 46.b.ii′.
58. Item 47.b.iiA′.
59. Item 47.b.iiB′.
60. Coalescing Items 46.c.′ and 47.c.′.

The above vehicle behaviour definition will be transformed (i.e., further "refined") in Sect. 5.3's Example 18; cf. Items 127.– 131. on Page 23 ■

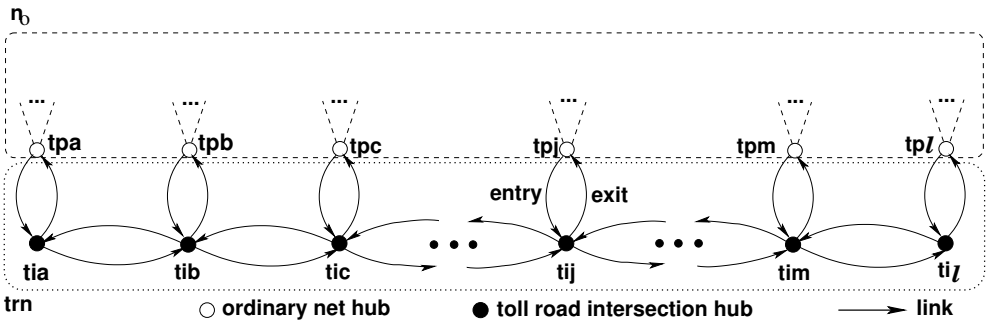

Fig. 1. A simple, linear toll-road net trn. tp_j: *t*oll *p*laza j, ti_j: *t*oll road *i*ntersection j.
Upper dashed sub-figure hint at an ordinary road net n_o.
Lower dotted sub-figure hint at a toll-road net trn.
Dash-dotted (- - -) "V"-images above tp_js hint at links to remaining "parts" of n_o.

4.2. Domain Instantiation

Definition 12. Instantiation: By **domain instantiation** we mean *a **refinement** of the partial domain requirements prescription (resulting from the projection step) in which the refinements aim at rendering the endurants: parts, materials and components, as well as the perdurants: actions, events and behaviours of the domain requirements prescription more concrete, more specific* ■ Instantiations usually render these concepts less general.

Properties that hold of the projected domain shall also hold of the (therefrom) instantiated domain.

Refinement of endurants can be expressed (i) either in the form of concrete types, (ii) or of further "delineating" axioms over sorts, (iii) or of a combination of concretisation and axioms. We shall exemplify the third possibility. Example 10 express requirements that the road net (on which the road-pricing system is to be based) must satisfy. Refinement of perdurants will not be illustrated (other than the simplification of the *vehicle* projected behaviour).

4.2.1. Domain Instantiation

Example 10. Domain Requirements. Instantiation Road Net: We now require that there is, as before, a road net, $n_{\mathscr{I}}$:$N_{\mathscr{I}}$, which can be understood as consisting of two, "connected sub-nets". A toll-road net, $trn_{\mathscr{I}}$:$TRN_{\mathscr{I}}$, cf. Fig. 1, and an ordinary road net, $n_{\mathscr{O}'}$. The two are connected as follows: The toll-road net, $trn_{\mathscr{I}}$, borders some toll-road plazas, in Fig. 1 shown by white filled circles (i.e., hubs). These toll-road plaza hubs are proper hubs of the 'ordinary' road net, $n'_{\mathscr{O}}$.

61. The instantiated domain, $\delta_{\mathscr{I}}$:$\Delta_{\mathscr{I}}$ has just the net, $n_{\mathscr{I}}$:$N_{\mathscr{I}}$ being instantiated.
62. The road net consists of two "sub-nets"
    a. an "ordinary" road net, n_o:$N_{\mathscr{O}'}$ and
    b. a toll-road net proper, trn:$TRN_{\mathscr{I}}$ —
    c. "connected" by an interface hil:HIL:
        i That interface consists of a number of toll-road plazas (i.e., hubs), modeled as a list of hub identifiers, hil:HI*.
        ii The toll-road plaza interface to the toll-road net, trn:$TRN_{\mathscr{I}}$[10], has each plaza, hil[i], connected to a pair of toll-road links: an entry and an exit link: $(l_e{:}L, l_x{:}L)$.
        iii The toll-road plaza interface to the 'ordinary' net, n_o:$N_{\mathscr{O}'}$, has each plaza, i.e., the hub designated by the hub identifier hil[i], connected to one or more ordinary net links, $\{l_{i_1}, l_{i_2}, \cdots, l_{i_k}\}$.

62.b. The toll-road net, trn:$TRN_{\mathscr{I}}$, consists of three collections (modeled as lists) of links and hubs:
    i a list of pairs of toll-road entry/exit links: $\langle(l_{e_1}, l_{x_1}), \cdots, (l_{e_\ell}, l_{x_\ell})\rangle$,
    ii a list of toll-road intersection hubs: $\langle h_{i_1}, h_{i_2}, \cdots, h_{i_\ell}\rangle$, and
    iii a list of pairs of main toll-road ("*u*p" and "*d*own") links: $\langle(ml_{i_{1u}}, ml_{i_{1d}}), (m_{i_{2u}}, m_{i_{2d}}), \cdots, (m_{i_{\ell u}}, m_{i_{\ell d}})\rangle$.
    d. The three lists have commensurate lengths (ℓ).

ℓ is the number of toll plazas, hence also the number of toll-road intersection hubs and therefore a number one larger than the number of pairs of main toll-road ("*u*p" and "*d*own") links ■

[10] We (sometimes) omit the subscript $\mathscr{I}$ when it should be clear from the context what we mean.

type
61. $\Delta_{\mathscr{I}}$
62. $N_{\mathscr{I}} = N_{\mathscr{P}'} \times HIL \times TRN$
62.a. $N_{\mathscr{P}'}$
62.b. $TRN_{\mathscr{I}} = (L\times L)^*\times H^*\times(L\times L)^*$
62.c. $HIL = HI^*$

axiom
62.d. $\forall\, n_{\mathscr{I}}{:}N_{\mathscr{I}}$ •
62.d. **let** $(n_\Delta,hil,(exll,hl,lll)) = n_{\mathscr{I}}$ **in**
62.d. **len** hil = **len** exll = **len** hl = **len** lll + 1
62.d. **end** ■

We have named the "ordinary" net sort (primed) $N_{\mathscr{P}'}$. It is "almost" like (unprimed) $N_{\mathscr{P}}$ — except that the interface hubs are also connected to the toll-road net entry and exit links.

The partial concretisation of the net sorts, $N_{\mathscr{P}}$, into $N_{\mathscr{I}}$ requires some additional well-formedness conditions to be satisfied.

63. The toll-road intersection hubs all[11] have distinct identifiers.

63. wf_dist_toll_road_isect_hub_ids: $H^*\to$**Bool**
63. wf_dist_toll_road_isect_hub_ids(hl) ≡
63. **len** hl = **card** xtr_his(hl)

64. The toll-road links all have distinct identifiers.

64. wf_dist_toll_road_u_d_link_ids: $(L\times L)^*\to$**Bool**
64. wf_dist_toll_road_u_d_link_ids(lll) ≡
64. 2 × **len** lll = **card** xtr_lis(lll)

65. The toll-road entry/exit links all have distinct identifiers.

65. wf_dist_e_x_link_ids: $(L\times L)^*\to$**Bool**
65. wf_dist_e_x_link_ids(exll) ≡
65. 2 × **len** exll = **card** xtr_lis(exll)

66. Proper net links must not designate toll-road intersection hubs.

66. wf_isoltd_toll_road_isect_hubs: $HI^*\times H^*\to N_{\mathscr{I}}\to$**Bool**
66. wf_isoltd_toll_road_isect_hubs(hil,hl)($n_{\mathscr{I}}$) ≡
66. **let** ls=xtr_links($n_{\mathscr{I}}$) **in**
66. **let** his = ∪ {**obs_mereo**_L(l)|l:L•l ∈ ls} **in**
66. his ∩ xtr_his(hl) = {} **end end**

67. The plaza hub identifiers must designate hubs of the 'ordinary' net.

67. wf_p_hubs_pt_of_ord_net: $HI^*\to N'_\Delta\to$**Bool**
67. wf_p_hubs_pt_of_ord_net(hil)(n'$_\Delta$) ≡
67. **elems** hil ⊆ xtr_his(n'_Δ)

68. The plaza hub mereologies must each,
    a. besides identifying at least one hub of the ordinary net,
    b. also identify the two entry/exit links with which they are supposed to be connected.

68. wf_p_hub_interf: $N'_\Delta\to$**Bool**
68. wf_p_hub_interf(n_o,hil,(exll,_,_)) ≡
68. ∀ i:**Nat** • i ∈ **inds** exll ⇒
68. **let** h = get_H(hil(i))(n'_Δ) **in**
68. **let** lis = **obs_mereo**_H(h) **in**
68. **let** lis′ = lis \ xtr_lis(n′) **in**
68. lis′ = xtr_lis(exll(i)) **end end end**

69. The mereology of each toll-road intersection hub must identify
    a. the entry/exit links
    b. and exactly the toll-road 'up' and 'down' links
    c. with which they are supposed to be connected.

69. wf_toll_road_isect_hub_iface: $N_{\mathscr{I}}\to$**Bool**
69. wf_toll_road_isect_hub_iface(_,_,(exll,hl,lll)) ≡
69. ∀ i:**Nat** • i ∈ **inds** hl ⇒
69. **obs_mereo**_H(hl(i)) =
69.a. xtr_lis(exll(i)) ∪
69. **case** i **of**
69.b. 1 → xtr_lis(lll(1)),
69.b. **len** hl → xtr_lis(lll(**len** hl−1))
69.b. _ → xtr_lis(lll(i)) ∪ xtr_lis(lll(i−1))
69. **end**

70. The mereology of the entry/exit links must identify exactly the
    a. interface hubs and the
    b. toll-road intersection hubs
    c. with which they are supposed to be connected.

70. wf_exll: $(L\times L)^*\times HI^*\times H^*\to$**Bool**
70. wf_exll(exll,hil,hl) ≡
70. ∀ i:**Nat** • i ∈ **len** exll
70. **let** (hi,(el,xl),h) = (hil(i),exll(i),hl(i)) **in**
70. **obs_mereo**_L(el) = **obs_mereo**_L(xl)
70. = {hi} ∪ {**uid**_H(h)} **end**
70. **pre**: **len** eell = **len** hil = **len** hl

71. The mereology of the toll-road 'up' and 'down' links must
    a. identify exactly the toll-road intersection hubs
    b. with which they are supposed to be connected.

71. wf_u_d_links: $(L\times L)^*\times H^*\to$**Bool**
71. wf_u_d_links(lll,hl) ≡
71. ∀ i:**Nat** • i ∈ **inds** lll ⇒
71. **let** (ul,dl) = lll(i) **in**
71. **obs_mereo**_L(ul) = **obs_mereo**_L(dl) =
71.a. **uid**_H(hl(i)) ∪ **uid**_H(hl(i+1)) **end**
71. **pre**: **len** lll = **len** hl+1

[11] A 'must' can be inserted in front of all 'all's,

We have used some additional auxiliary functions:

```
xtr_his: H*→HI-set
xtr_his(hl) ≡ {uid_HI(h)|h:H•h ∈ elems hl}
xtr_lis: (L×L)→LI-set
xtr_lis(l′,l″) ≡ {uid_LI(l′)}∪{uid_LI(l″)}
xtr_lis: (L×L)*− LI-set
xtr_lis(lll) ≡
 ∪{xtr_lis(l′,l″)|(l′,l″):(L×L)•(l′,l″)∈ elems lll}
```

72. The well-formedness of instantiated nets is now the conjunction of the individual well-formedness predicates above.

```
72. wf_instantiated_net: N𝒻 → Bool
72. wf_instantiated_net(n′Δ,hil,(exll,hl,lll))
63.     wf_dist_toll_road_isect_hub_ids(hl)
64.   ∧ wf_dist_toll_road_u_d_link_ids(lll)
65.   ∧ wf_dist_e_e_link_ids(exll)
66.   ∧ wf_isolated_toll_road_isect_hubs(hil,hl)(n′)
67.   ∧ wf_p_hubs_pt_of_ord_net(hil)(n′)
68.   ∧ wf_p_hub_interf(n′Δ,hil,(exll,_,_))
69.   ∧ wf_toll_road_isect_hub_iface(_,_,(exll,hl,lll))
70.   ∧ wf_exll(exll,hil,hl)
71.   ∧ wf_u_d_links(lll,hl)
```

4.2.2. Domain Instantiation — Abstraction

Example 11. Domain Requirements. Instantiation Road Net, Abstraction: Domain instantiation has refined an abstract definition of net sorts, n𝒹:N𝒹, into a partially concrete definition of nets, n𝒻:N𝒻. We need to show the refinement relation:

- abstraction(n𝒻) = n𝒹.

```
value
73.  abstraction: N𝒻 → N𝒹
74.  abstraction(n′Δ,hil,(exll,hl,lll)) ≡
75.    let n𝒹:N𝒹 •
75.      let hs = obs_part_HS𝒹(obs_part_HA𝒹(n′𝒹)),
75.          ls = obs_part_LS𝒹(obs_part_LA𝒹(n′𝒹)),
75.          ths = elems hl,
75.          eells = xtr_links(eell), llls = xtr_links(lll) in
76.          hs∪ths=obs_part_HS𝒹(obs_part_HA𝒹(n𝒹))
77.        ∧ ls∪eells∪llls=obs_part_LS𝒹(obs_part_LA𝒹(n𝒹))
78.    n𝒹 end end
```

73. The abstraction function takes a concrete net, n𝒻:N𝒻, and yields an abstract net, n𝒹:N𝒹.
74. The abstraction function doubly decomposes its argument into constituent lists and sub-lists.
75. There is postulated an abstract net, n𝒹:N𝒹, such that
76. the hubs of the concrete net and toll-road equals those of the abstract net, and
77. the links of the concrete net and toll-road equals those of the abstract net.
78. And that abstract net, n𝒹:N𝒹, is postulated to be an abstraction of the concrete net.

4.3. Domain Determination

Definition 13. Determination: By **domain determination** we mean *a refinement of the partial domain requirements prescription, resulting from the instantiation step, in which the refinements aim at rendering the endurants: parts, materials and components, as well as the perdurants: functions, events and behaviours of the partial domain requirements prescription less non-determinate, more determinate* ■

Determinations usually render these concepts less general. That is, the value space of endurants that are made more determinate is "smaller", contains fewer values, as compared to the endurants before determination has been "applied".

4.3.1. Domain Determination: Example

We show an example of 'domain determination'. It is expressed sôlely in terms of axioms over the concrete toll-road net type.

Example 12. Domain Requirements. Determination Toll-roads: We focus only on the toll-road net. We single out only two 'determinations':

All Toll-road Links are One-way Links

79. *The entry/exit and toll-road links*
    a. are always all one way links,
    b. as indicated by the arrows of Fig. 1 on Page 16,
    c. such that each pair allows traffic in opposite directions.

```
79. opposite_traffics: (L×L)* × (L×L)* → Bool
79. opposite_traffics(exll,lll) ≡
79.   ∀ (lt,lf):(L×L) • (lt,lf) ∈ elems exll^lll ⇒
79.a.     let (ltσ,lfσ) = (attr_LΣ(lt),attr_LΣ(lf)) in
79.a.′.    attr_LΩ(lt)={ltσ}∧attr_LΩ(ft)={ftσ}
79.a.″.  ∧ card ltσ = 1 = card lfσ
79.    ∧ let ({(hi,hi′)},{(hi″,hi‴)}) = (ltσ,lfσ) in
79.c.      hi=hi‴ ∧ hi′=hi″
```

79. **end end**

All Toll-road Hubs are Free-flow

80. *The hub state spaces* are singleton sets of the toll-road hub states which always allow exactly these (and only these) crossings:
    a. from *e*ntry links back to the paired e*x*it *l*ink*s*,
    b. from *e*ntry links to emanating *t*oll-road *l*ink*s*,
    c. from incident *t*oll-road links to e*x*it *l*ink*s*, and
    d. from incident *t*oll-road link to emanating *t*oll-road *l*ink*s*.

80.a.: from *e*ntry links back to the paired e*x*it *l*ink*s*:

```
80.a.  hσ_ex_ls: (L×L)→LΣ
80.a.  hσ_ex_ls(e,x) ≡ {(uid_LI(e),uid_LI(x))}
```

80.b.: from *e*ntry links to emanating *t*oll-road *l*ink*s*:

```
80.b.  hσ_et_ls: (L×L)×(Nat×(em:L×in:L)*)→LΣ
80.b.  hσ_et_ls((e,_),(i,ll)) ≡
80.b.    case i of
80.b.      2      → {(uid_LI(e),uid_LI(em(ll(1))))},
80.b.      len ll+1 → {(uid_LI(e),uid_LI(em(ll(len ll))))},
80.b.      _      → {(uid_LI(e),uid_LI(em(ll(i−1)))),
80.b.                 (uid_LI(e),uid_LI(em(ll(i))))}
80.b.    end
```

The *em* and *in* in the toll-road link list (em:L×in:L)* designate selectors for *em*anating, respectively *in*cident links. 80.c.: from incident *t*oll-road links to e*x*it *l*ink*s*:

```
80.  free_flow_toll_road_hubs: (L×L)*×(L×L)*→Bool
80.  free_flow_toll_road_hubs(exl,ll) ≡
80.    ∀ i:Nat•i ∈ inds hl ⇒
80.      attr_HΣ(hl(i)) =
80.a.        hσ_ex_ls(exl(i))
80.b.      ∪ hσ_et_ls(exl(i),(i,ll))
80.c.      ∪ hσ_tx_ls(exl(i),(i,ll))
80.d.      ∪ hσ_tt_ls(i,ll)
```

Predicates 79.a.′. and 79.a.″. express the same property.

```
80.c.  hσ_tx_ls: (L×L)×(Nat×(em:L×in:L)*)→LΣ
80.c.  hσ_tx_ls((_,x),(i,ll)) ≡
80.c.    case i of
80.c.      2      → {(uid_LI(in(ll(1))),uid_LI(x))},
80.c.      len ll+1 → {(uid_LI(in(ll(len ll))),uid_LI(x))},
80.c.      _      → {(uid_LI(in(ll(i−1))),uid_LI(x)),
80.c.                 (uid_LI(in(ll(i))),uid_LI(x))}
80.c.    end
```

80.d.: from incident *t*oll-road link to emanating *t*oll-road *l*ink*s*:

```
80.d.  hσ_tt_ls: Nat×(em:L×in:L)*→LΣ
80.d.  hσ_tt_ls(i,ll) ≡
80.d.    case i of
80.d.      2      → {(uid_LI(in(ll(1))),uid_LI(em(ll(1))))},
80.d.      len ll+1 → {(uid_LI(in(ll(len ll))),uid_LI(em(ll(len ll))))},
80.d.      _      → {(uid_LI(in(ll(i−1))),uid_LI(em(ll(i−1)))),
80.d.                 (uid_LI(in(ll(i))),uid_LI(em(ll(i))))}
80.d.    end ■
```

The example above illustrated 'domain determination' with respect to endurants. Typically "endurant determination" is expressed in terms of axioms that limit state spaces — where "endurant instantiation" typically "limited" the mereology of endurants: how parts are related to one another. We shall not exemplify domain determination with respect to perdurants. Typically perdurants are expressed in terms of expressions and statements. And, typically, perdurant nondeterminism is expressed in terms of the choice or parallelism operators: $C_{i_a} \sqcap C_{i_b}$ or $C_{x_a} \square C_{x_b}$ or $C_{p_a} \| C_{p_b}$. 'Perdurant determination' is then, typically, a matter of making the choice conditional (i,x) or of "sequentializing" parallelism (p).

(i) $C_{i_a} \sqcap C_{i_b} \Rightarrow$ **if** B_i **then** C_{i_a} **else** C_{i_b} **end**
(x) $C_{x_a} \square C_{x_b} \Rightarrow$ **if** B_x **then** C_{x_a} **else** C_{x_b} **end**
(p) $C_{p_a} \| C_{p_b} \Rightarrow C_{p_a}$; C_{p_b}

The above CSP "refinements" are just suggestive. For appropriate refinements we refer to [Hoa85, Ros97, Sch00].

4.4. Domain Extension

Definition 14. Extension: By **domain extension** we understand *the introduction of endurants and perdurants that were not feasible in the original domain, but for which, with computing and communication, and with new, emerging technologies, for example, sensors, actuators and satellites, there is the possibility of feasible implementations, hence the requirements, that what is introduced becomes part of the unfolding requirements prescription* ■

Usually extensions involving one of the main sorts entails extensions involving several of the main sorts. In our example we introduce (i.e., "extend") vehicles with GPSS-like sensors, and introduce toll-gates with entry sensors, vehicle identification sensors, gate actuators and exit sensors. Finally road pricing calculators are introduced.

4.4.1. The Requirements Example: Domain Extension

Example 13. Domain Requirements — Extension: We present the extensions in several steps. Some of them will be developed in this section. Development of the remaining will be deferred to Sect. 5.3. The reason for this deferment is that those last steps are examples of interface requirements. The initial extension-development steps are: [a] vehicle extension, [b] sort and unique identifiers of road price calculators, [c] vehicle to road pricing calculator channel, [d] sorts and dynamic attributes of toll-gates, [e] road pricing calculator attributes, [f] "total" system state, and [g] the overall system behaviour. This decomposition establishes system interfaces in "small, easy steps".

[a] Vehicle Extension:

81. There is a domain, $\delta_{\mathscr{E}}$:$\Delta_{\mathscr{E}}$, which contains
82. a fleet, $f_{\mathscr{E}}$:$F_{\mathscr{E}}$, that is,
83. a set, $vs_{\mathscr{E}}$:$VS_{\mathscr{E}}$, of
84. extended vehicles, $v_{\mathscr{E}}$:$V_{\mathscr{E}}$ — their extension amounting to
85. a dynamic reactive attribute, whose value, ti-gpos:TiGpos, at any time, reflects that vehicle's *time-stamped global position*.[12]
86. The vehicle's GNSS receiver calculates, *loc_pos*, its local position, lpos:LPos, based on these signals.
87. Vehicles access these external attributes via the external attribute channel, attr_TiGPos_ch.

type
81. $\Delta_{\mathscr{E}}$
82. $F_{\mathscr{E}}$
83. $VS_{\mathscr{E}} = V_{\mathscr{E}}$**-set**
84. $V_{\mathscr{E}}$
85. TiGPos = $\mathbb{T}$ × GPos
86. GPos, LPos

value
81. $\delta_{\mathscr{E}}$:$\Delta_{\mathscr{E}}$
82. **obs_part_**$F_{\mathscr{E}}$: $\Delta_{\mathscr{E}} \to F_{\mathscr{E}}$
82. f = **obs_part_**$F_{\mathscr{E}}$($\delta_{\mathscr{E}}$)
83. **obs_part_**$VS_{\mathscr{E}}$: $F_{\mathscr{E}} \to VS_{\mathscr{E}}$
83. vs = **obs_part_**$VS_{\mathscr{E}}$(f)
83. vis = xtr_vis(vs)
85. attr_TiGPos_ch[vi]?
86. loc_pos: GPos → LPos

channel
86. {attr_TiGPos_ch[vi]|vi:VI•vi ∈ vis}:TiGPos

We define two auxiliary functions,

88. xtr_vs, which given a domain, or a fleet, extracts its set of vehicles, and
89. xtr_vis which given a set of vehicles generates their unique identifiers.

value
88. xtr_vs: ($\Delta_{\mathscr{E}}$|$F_{\mathscr{E}}$|$VS_{\mathscr{E}}$) → $V_{\mathscr{E}}$**-set**
88. xtr_vs(arg) ≡
88. **is_**$\Delta_{\mathscr{E}}$(arg) → **obs_part_**$VS_{\mathscr{E}}$(**obs_part_**$F_{\mathscr{E}}$(arg)),
88. **is_**$F_{\mathscr{E}}$(arg) → **obs_part_**$VS_{\mathscr{E}}$(arg),
88. **is_**$VS_{\mathscr{E}}$(arg) → arg
89. xtr_vis: ($\Delta_{\mathscr{E}}$|$F_{\mathscr{E}}$|$VS_{\mathscr{E}}$) → VI**-set**
89. xtr_vis(arg) ≡ {**uid_**VI(v)|v ∈ xtr_vs(arg)}

[b] Road Pricing Calculator: Basic Sort and Unique Identifier:

90. The domain $\delta_{\mathscr{E}}$:$\Delta_{\mathscr{E}}$, also contains a pricing calculator, c:$C_{\delta_{\mathscr{E}}}$, with unique identifier ci:CI.

type
90. C, CI

value
90. **obs_part_**C: $\Delta_{\mathscr{E}}$ → C
90. **uid_**CI: C → CI
90. c = **obs_part_**C($\delta_{\mathscr{E}}$)
90. ci = **uid_**CI(c)

[c] Vehicle to Road Pricing Calculator Channel:

91. Vehicles can, on their own volition, offer the timed local position, viti-lpos:VITiLPos
92. to the pricing calculator, c:$C_{\mathscr{E}}$ along a vehicles-to-calculator channel, v_c_ch.

type
91. VITiLPos = VI × ($\mathbb{T}$ × LPos)

channel
92. {v_c_ch[vi,ci]|vi:VI,ci:CI•vi∈vis∧ci=**uid_**C(c)}:VITiLPos

[d] Toll-gate Sorts and Dynamic Types: We extend the domain with toll-gates for vehicles entering and exiting the toll-road entry and exit links. Figure 2 illustrates the idea of gates. Figure 2 on the next page is intended to illustrate a vehicle entering (or exiting) a toll-road entry link. The toll-gate is equipped with three sensors: an entry sensor, a vehicle identification sensor and an exit sensor. The entry sensor serves to prepare the vehicle identification sensor. The exit sensor serves to prepare the gate for closing when a vehicle has passed. The vehicle identify sensor identifies the vehicle and "delivers" a pair: the current time and the vehicle identifier. Once the vehicle identification sensor has identified a vehicle the gate opens and a message is sent to the road pricing calculator as to the passing vehicle's identity and the identity of the link associated with the toll-gate (see Items 108.- 109. on the facing page).

[12] We refer to literature on GNSS, *global navigation satellite systems*. The simple vehicle position, vp:SVPos, is determined from three to four time-stamped signals received from a like number of GNSS satellites [ESA].

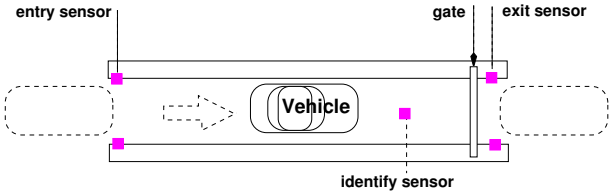

Fig. 2. A toll plaza gate

93. The domain contains the extended net, n:$N_{\mathscr{E}}$,
94. with the net extension amounting to the toll-road net, $TRN_{\mathscr{E}}$, that is, the instantiated toll-road net, trn:$TRN_{\mathscr{J}}$, is extended, into trn:$TRN_{\mathscr{E}}$, with entry, eg:EG, and exit, xg:XG, toll-gates.

From entry- and exit-gates we can observe

95. their unique identifier and
96. their mereology: pairs of entry-, respectively exit link and calculator unique identifiers; further
97. a pair of gate entry and exit sensors modeled as external attribute channels, (ges:ES,gls:XS), and
98. a time-stamped vehicle identity sensor modeled as external attribute channels.

type
93. $N_{\mathscr{E}}$
94. $TRN_{\mathscr{E}} = (EG \times XG)^* \times TRN_{\mathscr{J}}$
95. GI

value
93. **obs_part_**$N_{\mathscr{E}}$: $\Delta_{\mathscr{E}} \to N_{\mathscr{E}}$
94. **obs_part_**$TRN_{\mathscr{E}}$: $N_{\mathscr{E}} \to TRN_{\mathscr{E}}$
95. **uid_**G: (EG|XG) → GI
96. **obs_mereo_**G: (EG|XG) → (LI×CI)
94. trn:$TRN_{\mathscr{E}}$ = **obs_part_**$TRN_{\mathscr{E}}(\delta_{\mathscr{E}})$

channel
97. {attr_entry_ch[gi]|gi:GI•xtr_eGIds(trn)} "enter"
97. {attr_exit_ch[gi]|gi:GI•xtr_xGIds(trn)} "exit"
98. {attr_identity_ch[gi]|gi:GI•xtr_GIds(trn)} TIVI

type
98. TIVI = $\mathbb{T}$ × VI

We define some **auxiliary functions** over toll-road nets, trn:$TRN_{\mathscr{E}}$:

99. xtr_eGℓ extracts the ℓist of entry gates,
100. xtr_xGℓ extracts the ℓist of exit gates,
101. xtr_eGIds$s$ extracts the $s$et of entry gate identifiers,
102. xtr_xGIds$s$ extracts the $s$et of exit gate identifiers,
103. xtr_Gs extracts the $s$et of all gates, and
104. xtr_GIds$s$ extracts the $s$et of all gate identifiers.

value
99. xtr_eGℓ: $TRN_{\mathscr{E}} \to EG^*$
99. xtr_eGℓ(pgl,_) ≡ {eg|(eg,xg):(EG,XG)•(eg,xg)∈ **elems** pgl}
100. xtr_xGℓ: $TRN_{\mathscr{E}} \to XG^*$
100. xtr_xGℓ(pgl,_) ≡ {xg|(eg,xg):(EG,XG)•(eg,xg)∈ **elems** pgl}
101. xtr_eGIds: $TRN_{\mathscr{E}}$ → GI-**set**
101. xtr_eGIds(pgl,_) ≡ {**uid_**GI(g)|g:EG•g ∈ xtr_eGs(pgl,_)}
102. xtr_xGIds: $TRN_{\mathscr{E}}$ → GI-**set**
102. xtr_xGIds(pgl,_) ≡ {**uid_**GI(g)|g:EG•g ∈ xtr_xGs(pgl,_)}
103. xtr_Gs: $TRN_{\mathscr{E}}$ → G-**set**
103. xtr_Gs(pgl,_) ≡ xtr_eGs(pgl,_) ∪ xtr_xGs(pgl,_)
104. xtr_GIds: $TRN_{\mathscr{E}}$ → GI-**set**
104. xtr_GIds(pgl,_) ≡ xtr_eGIds(pgl,_) ∪ xtr_xGIds(pgl,_)

105. A **well-formedness condition** expresses
    a. that there are as many entry end exit gate pairs as there are toll-plazas,
    b. that all gates are uniquely identified, and
    c. that each entry [exit] gate is paired with an entry [exit] link and has that link's unique identifier as one element of its mereology, the other elements being the calculator identifier and the vehicle identifiers.

The well-formedness relies on awareness of

106. the unique identifier, ci:CI, of the road pricing calculator, c:C, and
107. the unique identifiers, vis:VI-**set**, of the fleet vehicles.

axiom
105. ∀ n:$N_{\mathscr{R}_3}$, trn:$TRN_{\mathscr{R}_3}$ •
105. **let** (exgl,(exl,hl,lll)) = **obs_part_**$TRN_{\mathscr{R}_3}$(n) **in**
105.a. **len** exgl = **len** exl = **len** hl = **len** lll + 1
105.b. ∧ **card** xtr_GIds(exgl) = 2 ∗ **len** exgl
105.c. ∧ ∀ i:**Nat**•i ∈ **inds** exgl•
105.c. **let** ((eg,xg),(el,xl)) = (exgl(i),exl(i)) **in**
105.c. **obs_mereo_**G(eg) = (**uid_**U(el),ci,vis)
105.c. ∧ **obs_mereo_**G(xg) = (**uid_**U(xl),ci,vis)
105. **end end**

[e] Toll-gate to Calculator Channels:

108. Toll-gate entry and exit gates offer passing a pair: whether it is an entry or an exit gates, and pair of the vehicle's identity and the time-stamped identity of the link associated with the toll-gate
109. to the road pricing calculator via channel.

type
108. EEVITiLI = ("Entry"|"Exit")×(VI×(T×SonL))

channel
109. {g_c_ch[gi,ci]|gi:GI•gi ∈ gis}:EEVITiLI

[f] Road Pricing Calculator Attributes:

110. The road pricing attributes include a programmable traffic map, trm:TRM, which, for each vehicle inside the toll-road net, records a chronologically ordered list of each vehicle's timed position, (τ,lpos), and
111. a static (total) road location function, vplf:VPLF. The *v*ehicle *p*osition *l*ocation *f*unction, vplf:VPLF, which, given a local position, lpos:LPos, yields *either* the simple vehicle position, svpos:SVPos, designated by the GNSS-provided position, *or* yields the response that the provided position is off the toll-road net The vplf:VPLF function is constructed, construct_vplf,
112. from awareness, of a geodetic road map, GRM, of the topology of the extended net, $n_{\mathscr{E}}$:$N_{\mathscr{E}}$, including the mereology and the geodetic attributes of links and hubs.

type
110. TRM = VI $\overrightarrow{m}$ ($\mathbb{T}$ × SVPos)*
111. VPLF = GRM → LPos → (SVPos | ″off_N″)
112. GRM

value
110. **attr**_TRM: $C_{\mathscr{E}}$ → TRM
111. **attr**_VPLF: $C_{\mathscr{E}}$ → VPLF

The geodetic road map maps geodetic locations into hub and link identifiers.

23. Geodetic link locations represent the set of point locations of a link.
19.a. Geodetic hub locations represent the set of point locations of a hub.
113. A geodetic road map maps geodetic link locations into link identifiers and geodetic hub locations into hub identifiers.
114. We sketch the construction, *geo_GRM*, of geodetic road maps.

type
113. GRM = (GeoL $\overrightarrow{m}$ LI) ⋃ (GeoH $\overrightarrow{m}$ HI)
value
114. geo_GRM: N → GRM
114. geo_GRM(n) ≡
114. **let** ls = xtr_links(n), hs = xtr_hubs(n) **in**
114. [**attr**_GeoL(l)↦**uid**_LI(l)|l:L•l ∈ ls]
114. ∪
114. [**attr**_GeoH(h)↦**uid**_HI(h)|h:H•h ∈ hs] **end**

115. The vplf:VPLF function obtains a simple vehicle position, svpos, from a geodetic road map, grm:GRM, and a local position , lpos:

value
115. obtain_SVPos: GRM → LPos → SVPos
115. obtain_SVPos(grm)(lpos) **as** svpos
115. **post**: **case** svpos **of**
115. SatH(hi) → within(lpos,grm(hi)),
115. SonL(li) → within(lpos,grm(li)),
115. ″off_N″ → **true end**

where *within* is a predicate which holds if its first argument, a local position calculated from a GNSS-generated global position, falls within the point set representation of the geodetic locations of a link or a hub. The design of the *obtain_SVPos* represents an interesting challenge.

[g] "Total" System State: Global values:

116. There is a given domain, $\delta_{\mathscr{E}}$:$\Delta_{\mathscr{E}}$;
117. there is the net, $n_{\mathscr{E}}$:$N_{\mathscr{E}}$, of that domain;
118. there is toll-road net, $trn_{\mathscr{E}}$:$TRN_{\mathscr{E}}$, of that net;
119. there is a set, $egs_{\mathscr{E}}$:$EG_{\mathscr{E}}$**-set**, of entry gates;
120. there is a set, $xgs_{\mathscr{E}}$:$XG_{\mathscr{E}}$**-set**, of exit gates;
121. there is a set, $gis_{\mathscr{E}}$:$GI_{\mathscr{E}}$**-set**, ofgate identifiers;
122. there is a set, $vs_{\mathscr{E}}$:$V_{\mathscr{E}}$**-set**, of vehicles;
123. there is a set, $vis_{\mathscr{E}}$:$VI_{\mathscr{E}}$**-set**, of vehicle identifiers;
124. there is the road-pricing calculator, $c_{\mathscr{E}}$:$C_{\mathscr{E}}$ and
125. there is its unique identifier, $ci_{\mathscr{E}}$:CI.

value
116. $\delta_{\mathscr{E}}$:$\Delta_{\mathscr{E}}$
117. $n_{\mathscr{E}}$:$N_{\mathscr{E}}$ = **obs_part**_$N_{\mathscr{E}}$($\delta_{\mathscr{E}}$)
118. $trn_{\mathscr{E}}$:$TRN_{\mathscr{E}}$ = **obs_part**_$TRN_{\mathscr{E}}$($n_{\mathscr{E}}$)
119. $egs_{\mathscr{E}}$:EG**-set** = xtr_egs($trn_{\mathscr{E}}$)
120. $xgs_{\mathscr{E}}$:XG**-set** = xtr_xgs($trn_{\mathscr{E}}$)
121. $gis_{\mathscr{E}}$:XG**-set** = xtr_gis($trn_{\mathscr{E}}$)
122. $vs_{\mathscr{E}}$:$V_{\mathscr{E}}$**-set** = **obs_part**_VS(**obs_part**_$F_{\mathscr{E}}$($\delta_{\mathscr{E}}$))
123. $vis_{\mathscr{E}}$:VI**-set** = {**uid**_VI($v_{\mathscr{E}}$)|$v_{\mathscr{E}}$:$V_{\mathscr{E}}$•$v_{\mathscr{E}}$ ∈ $vs_{\mathscr{E}}$}
124. $c_{\mathscr{E}}$:$C_{\mathscr{E}}$ = **obs_part**_$C_{\mathscr{E}}$($\delta_{\mathscr{E}}$)
125. $ci_{\mathscr{E}}$:CI$_{\mathscr{E}}$ = **uid**_CI($c_{\mathscr{E}}$)

[h] "Total" System Behaviour:

The signature and definition of the system behaviour is sketched as are the signatures of the vehicle, toll-gate and road pricing calculator. We shall model the behaviour of the road pricing system as follows: we shall not model behaviours nets, nhubs and links; thus we shall model only the behaviour of vehicles, veh, the behaviour of toll-gates, gate, and the behaviour of the road-pricing calculator, calc, The behaviours of vehicles and toll-gates are presented here. But the behaviour of the road-pricing calculator is "deferred" till Sect. 5.3 since it reflects an interface requirements.

126. The road pricing system behaviour, sys, is expressed as
    a. the parallel, ||, (distributed) composition of the behaviours of all vehicles, with the parallel composition of
    b. the parallel (likewise distributed) composition of the behaviours of all entry gates, with the parallel composition of
    c. the parallel (likewise distributed) composition of the behaviours of all exit gates, with the parallel composition of
    d. the behaviour of the road-pricing calculator,

```
value
126.  sys: Unit → Unit
126.  sys() ≡
126.a.      ‖ {veh(uid_V(v),(ci,gis),attr_TiGPos_ch)|v:V•v ∈ vs_ℰ}
126.b.    ‖ ‖ {gate("Entry")(gi,obs_mereo_G(eg),(attr_entry_ch[gi],attr_identify_ch[gi],attr_exit_ch[gi]))|eg:EG•eg ∈ egs_ℰ∧gi=uid_EG(eg)}
126.c.    ‖ ‖ {gate("Exit")(gi,obs_mereo_G(xg),(attr_entry_ch[gi],attr_identify_ch[gi],attr_exit_ch[gi]))|xg:XG•xg ∈ xgs_ℰ∧gi=uid_XG(xg)}
126.d.      ‖ calc(ci_ℰ,(vis_ℰ,gis_ℰ))(rlf)(trm)

127.  veh: vi:VI×(ci:CI×gis:GI-set)×𝕌TiGPos → out v_c_ch[vi,ci] Unit
133.  gate: ee:EE×gi:GI×(ci:CI×VI-set×LI)×(𝕌entry×𝕌identify×𝕌exit) → out g_c_ch[gi,ci] Unit
167.  calc: ci:CI×(vis:VI-set×gis:GI-set)×VPLF→TRM→ in {v_c_ch[vi,ci]|vi:VI•vi ∈ vis},{g_c_ch[gi,ci]|gi:GI•gi ∈ gis} Unit ■
```

Vehicle Behaviour: We refer to the vehicle behaviour, in the domain, described in Sect. 2's The Road Traffic System Behaviour Items 46. and Items 47., Page 9 and, projected, Page 15.

127. Instead of moving around by explicitly expressed internal non-determinism[13] vehicles move around by unstated internal non-determinism and instead receive their current position from the global positioning subsystem.
128. At each moment the vehicle receives its time-stamped global position, (τ,gpos):TiGPos,
129. from which it calculates the local position, lpos:VPos
130. which it then communicates, with its vehicle identification, (vi,(τ,lpos)), to the road pricing subsystem —
131. whereupon it resumes its vehicle behaviour.

```
value
127.  veh: vi:VI×(ci:CI×gis:GI-set)×𝕌TiGPos →
127.     out v_c_ch[vi,ci] Unit
127.  veh(vi,(ci,gis),attr_TiGPos_ch[vi]) ≡
128.    let (τ,gpos) = attr_TiGPos_ch[vi]? in
129.    let lpos = loc_pos(gpos) in
130.    v_c_ch[vi,ci] ! (vi,(τ,lpos)) ;
131.    veh(vi,(ci,gis),attr_TiGPos_ch[vi]) end end
127.    pre vi ∈ vis_ℰ ∧ ci = ci_ℰ ∧ gis = gis_ℰ
```

The above behaviour represents an assumption about the behaviour of vehicles. If we were to design software for the monitoring and control of vehicles then the above vehicle behaviour would have to be refined in order to serve as a proper interface requirements. The refinement would include handling concerns about the drivers' behaviour when entering, passing and exiting toll-gates, about the proper function of the GNSS equipment, and about the safe communication with the road price calculator. The above concerns would already have been addressed in a model of *domain facets* such as *human behaviour*, *technology support*, proper tele-communications *scripts*, etcetera. We refer to [Bjø10a].

Gate Behaviour: The entry and the exit gates have "vehicle enter", "vehicle exit" and "timed vehicle identification" sensors. The following assumption can now be made: during the time interval between a gate's vehicle "entry" sensor having first sensed a vehicle entering that gate and that gate's "exit" sensor having last sensed that vehicle leaving that gate that gate's vehicle time and "identify" sensor registers the time when the vehicle is entering the gate and that vehicle's unique identification. We sketch the toll-gate behaviour:

132. We parameterise the toll-gate behaviour as either an entry or an exit gate.
133. Toll-gates operate autonomously and cyclically.
134. The attr_enter_ch event "triggers" the behaviour specified in formula line Item 135.–137..
135. The time-of-passing and the identity of the passing vehicle is sensed by attr_passing_ch channel events.
136. Then the road pricing calculator is informed of time-of-passing and of the vehicle identity vi and the link li associated with the gate.
137. And finally, after that vehicle has left the entry or exit gate
138. that toll-gate's behaviour is resumed.

```
type
132.  EE = "Enter" | "Exit"
value
133.  gate: ee:EE×gi:GI×(ci:CI×VI-set×LI)×
133.        (𝕌enter×𝕌passing×𝕌leave) →
133.          out g_c_ch[gi,ci] Unit
133.  gate(ee,gi,(ci,vis,li),
133.       ea:(attr_enter_ch[gi],attr_passing_ch[gi],attr_leave_ch[gi])) ≡
134.    attr_enter_ch[gi] ? ;
135.    let (τ,vi) = attr_passing_ch[gi] ? in assert vi ∈ vis
136.    g_c_ch[gi,ci] ! (ee,(vi,(τ,SonL(li)))) ;
137.    attr_leave_ch[gi] ? ;
138.    gate(ee,gi,(ci,vis,li),ea)
133.    end
133.    pre ci = ci_ℰ ∧ vis = vis_ℰ ∧ li ∈ lis_ℰ
```

The above behaviour represents an assumption about the behaviour of toll-gates. If we were to design software for the monitoring and control of toll-gates then the above gate behaviour would have to be refined in order to serve as a proper interface requirements. The refinement would include handling concerns about the drivers' behaviour when entering, passing and exiting toll-gates, about the proper function of the entry, passing and exit sensors, about the proper function of the gate barrier (opening and closing), and about the safe communication with the road price calculator. The above concerns would already have been addressed in a model of *domain facets* such as *human behaviour*, *technology support*, proper tele-communications *scripts*, etcetera. We refer to [Bjø10a] ■

We shall define the *calc*ulator behaviour in Sect. 5.3 on Page 27. The reason for this deferral is that it exemplifies interface requirements.

[13] We refer to Items 46.b., 46.c. on Page 9 and 47.b., 47.b.ii, 47.c. on Page 9

4.5. Requirements Fitting

Often a domain being described "fits" onto, is "adjacent" to, "interacts" in some areas with, another domain: *transportation* with *logistics*, *health-care* with *insurance*, *banking* with *securities trading* and/or *insurance*, and so on. The issue of requirements fitting arises when two or more software development projects are based on what appears to be the same domain. The problem then is to harmonise the two or more software development projects by harmonising, if not too late, their requirements developments.

We thus assume that there are n domain requirements developments, $d_{r_1}, d_{r_2}, \ldots, d_{r_n}$, being considered, and that these pertain to the same domain — and can hence be assumed covered by a same domain description.

Definition 15. Requirements Fitting: By **requirements fitting** we mean a *harmonisation* of $n > 1$ domain requirements that have overlapping (shared) not always consistent parts and which results in n partial domain requirements', $p_{d_{r_1}}, p_{d_{r_2}}, \ldots, p_{d_{r_n}}$, and m shared domain requirements, $s_{d_{r_1}}, s_{d_{r_2}}, \ldots, s_{d_{r_m}}$, that "fit into" two or more of the partial domain requirements ■ The above definition pertains to the result of 'fitting'. The next definition pertains to the act, or process, of 'fitting'.

Definition 16. Requirements Harmonisation: By **requirements harmonisation** we mean a number of alternative and/or co-ordinated prescription actions, one set for each of the domain requirements actions: *Projection, Instantiation, Determination* and *Extension*. They are – we assume n separate software product requirements: *Projection:* If the n product requirements do not have the same projections, then identify a common projection which they all share, and refer to it as the common projection. Then develop, for each of the n product requirements, if required, a specific projection of the common one. Let there be m such specific projections, $m \leq n$. *Instantiation:* First instantiate the common projection, if any instantiation is needed. Then for each of the m specific projections instantiate these, if required. *Determination:* Likewise, if required, "perform" "determination" of the possibly instantiated common projection, and, similarly, if required, "perform" "determination" of the up to m possibly instantiated projections. *Extension:* Finally "perform extension" likewise: First, if required, of the common projection (etc.), then, if required, on the up m specific projections (etc.). These harmonization developments may possibly interact and may need to be iterated ■

By a **partial domain requirement**s we mean a domain requirements which is short of (that is, is missing) some prescription parts: text and formula ■ By a **shared domain requirement**s we mean a domain requirements ■ By **requirements fitting** m *s*hared $d$omain $r$equirements texts, $sdrs$, into n partial domain requirements we mean that there is for each $p$artial $d$omain $r$equirements, pdr_i, an identified, non-empty $s$ubset of $sdrs$ (could be all of $sdrs$), $ssdrs_i$, such that textually conjoining $ssdrs_i$ to pdr_i, i.e., $ssdrs_i \oplus pdr_i$ can be claimed to yield the "original" d_{r_i}, that is, $\mathscr{M}(ssdrs_i \oplus pdr_i) \subseteq \mathscr{M}(d_{r_i})$, where $\mathscr{M}$ is a suitable meaning function over prescriptions ■

4.6. Discussion

Facet-oriented Fittings: An altogether different way of looking at domain requirements may be achieved when also considering domain facets — not covered in neither the example of Sect. 2 nor in this section (i.e., Sect. 4) nor in the following two sections. We refer to [Bjø10a].

Example 14. Domain Requirements — Fitting: Example 13 hints at three possible sets of interface requirements: (i) for a road pricing [sub-]system, as will be illustrated in Sect. 5.3; (ii) for a vehicle monitoring and control [sub-]system, and (iii) for a toll-gate monitoring and control [sub-]system. The vehicle monitoring and control [sub-]system would focus on implementing the vehicle behaviour, see Items 127.- 131. on the preceding page. The toll-gate monitoring and control [sub-]system would focus on implementing the calculator behaviour, see Items 133.- 138. on the previous page. The fitting amounts to (a) making precise the (narrative and formal) texts that are specific to each of of the three (i–iii) separate sub-system requirements are kept separate; (b) ensuring that (meaning-wise) shared texts that have different names for (meaning-wise) identical entities have these names renamed appropriately; (c) that these texts are subject to commensurate and ameliorated further requirements development; etcetera ■

5. Interface Requirements

We remind the reader that **interface requirements** can be expressed only using terms from both the domain and the machine ■ Users are not part of the machine. So no reference can be made to users, such as *"the system must be user*

friendly", and the like ! By an **interface requirements** we [also] mean *a requirements prescription which refines and extends the domain requirements by considering those requirements of the domain requirements whose endurants (parts, materials) and perdurants (actions, events and behaviours) are **"shared"** between the domain and the machine (being requirements prescribed)* ■ The two interface requirements definitions above go hand–in–hand, i.e., complement one-another.

5.1. Shared Phenomena

By **sharing** we mean (a) that *some or all properties* of an ***endurant*** is represented both in the domain and "inside" the machine, and that their machine representation must at suitable times reflect their state in the domain; and/or (b) that an ***action*** requires a sequence of several "on-line" interactions between the machine (being requirements prescribed) and the domain, usually a person or another machine; and/or (c) that an ***event*** arises either in the domain, that is, in the environment of the machine, or in the machine, and need be communicated to the machine, respectively to the environment; and/or (d) that a ***behaviour*** is manifested both by actions and events of the domain and by actions and events of the machine ■ So a systematic reading of the domain requirements shall result in an identification of all shared endurants, parts, materials and components; and perdurants actions, events and behaviours. Each such shared phenomenon shall then be individually dealt with: ***endurant sharing*** shall lead to interface requirements for data initialisation and refreshment as well as for access to endurant attributes; ***action sharing*** shall lead to interface requirements for interactive dialogues between the machine and its environment; ***event sharing*** shall lead to interface requirements for how such event are communicated between the environment of the machine and the machine; and ***behaviour sharing*** shall lead to interface requirements for action and event dialogues between the machine and its environment.

Environment–Machine Interface: Domain requirements extension, Sect. 4.4, usually introduce new endurants into (i.e., 'extend' the) domain. Some of these endurants may become elements of the domain requirements. Others are to be projected "away". Those that are let into the domain requirements either have their endurants represented, somehow, also in the machine, or have (some of) their properties, usually some attributes, accessed by the machine. Similarly for perdurants. Usually the machine representation of shared perdurants access (some of) their properties, usually some attributes. The interface requirements must spell out which domain extensions are shared. Thus domain extensions may necessitate a review of domain projection, instantiations and determination. In general, there may be several of the projection–eliminated parts (etc.) whose dynamic attributes need be accessed in the usual way, i.e., by means of attr_XYZ_ch channel communications (where XYZ is a projection–eliminated part attribute).

Example 15. Interface Requirements — Projected Extensions: We refer to Fig. **??** on Page ??.We do not represent the GNSS system in the machine: only its "effect": the ability to record global positions by accessing the GNSS attribute (channel):

channel
87. {attr_TiGPos_ch[vi]|vi:VI•vi ∈ xtr_VIs(vs)}: TiGPos

And we do not really represent the gate nor its sensors and actuator in the machine. But we do give an idealised description of the gate behaviour, see Items 133.–138. Instead we represent their dynamic gate attributes:

(97.) the vehicle entry sensors (leftmost ■s),
(97.) the vehicle identity sensor (center ■), and
(98.) the vehicle exit sensors (rightmost ■s)

by channels — we refer to Example 13 (Sect. 5.3, Page 21):

channel
97. {attr_entry_ch[gi]|gi:GI•xtr_eGIds(trn)}″enter″
97. {attr_exit_ch[gi]|gi:GI•xtr_xGIds(trn)}″exit″
98. {attr_identity_ch[gi]|gi:GI•xtr_GIds(trn)} TIVI ■

5.2. Shared Endurants

Example 16. Interface Requirements. Shared Endurants: The main shared endurants are the vehicles, the net (hubs, links, toll-gates) and the price calculator. As domain endurants hubs and links undergo changes, all the time, with respect to the values of several attributes: *length, geodetic information, names, wear and tear* (where-ever applicable), *last/next scheduled maintenance* (where-ever applicable), *state* and *state space*, and many others. Similarly for vehicles: their position, velocity and

acceleration, and many other attributes. We then come up with something like hubs and links are to be represented as tuples of relations; each net will be represented by a pair of relations a hubs relation and a links relation; each hub and each link may or will be represented by several tuples; etcetera. In this database modeling effort it must be secured that "standard" operations on nets, hubs and links can be supported by the chosen relational database system ■

5.2.1. Data Initialisation

In general, one must prescribe data initialisation, that is provision for an interactive user interface dialogue with a set of proper display screens, one for establishing net, hub or link attributes names and their types, and, for example, two for the input of hub and link attribute values. Interaction prompts may be prescribed: next input, on-line vetting and display of evolving net, etc. These and many other aspects may therefore need prescriptions.

Example 17. **Interface Requirements. Shared Endurant Initialisation**: The domain is that of the road net, n:N. By 'shared road net initialisation' we mean the "ab initio" establishment, "from scratch", of a data base recording the properties of all links, l:L, and hubs, h:H, their unique identifications, **uid**_L(l) and **uid**_H(h), their mereologies, **obs_mereo**_L(l) and **obs_mereo**_H(h), the initial values of all their static and programmable attributes and the access values, that is, channel designations for all other attribute categories.

139. There are r_l and r_h "recorders" recording link, respectively hub properties – with each recorder having a unique identity.
140. Each recorder is charged with the recording of a set of links or a set of hubs according to some partitioning of all such.
141. The recorders inform a central data base, net_db, of their recordings (ri,hol,(u_j,m_j,$attrs_j$)) where
142. ri is the identity of the recorder,
143. hol is either a `hub` or a `link` literal,
144. u_j = **uid**_L(l) or **uid**_H(h) for some link or hub,
145. m_j = **obs_mereo**_L(l) or **obs_mereo**_H(h) for that link or hub and
146. $attrs_j$ are *attributes* for that link or hub — where *attributes* is a function which "records" all respective static and dynamic attributes (left undefined).

```
type
  RI
value
  rl,rh:NAT axiom rl>0 ∧ rh>0
type
  M = RI×"link"×LNK | RI×"hub"×HUB
  LNK = LI × HI-set × LATTRS
  HUB = HI × LI-set × HATTRS
```

```
value
  partitioning: L-set → Nat → (L-set)* | H-set → Nat → (H-set)*
  partitioning(s)(r) as sl
  post: len sl = r ∧ ∪ elems sl = s
        ∧ ∀ si,sj:(L-set|H-set) •
            si≠{}∧sj≠{}∧{si,sj}⊆elems ss⇒si ∩ sj={}
```

147. The $r_l + r_h$ recorder behaviours interact with the one net_db behaviour

```
channel
  r_db: RI×(LNK|HUB)
```

```
value
  link_rec: RI → L-set → out r_db Unit
  hub_rec: RI → H-set → out r_db Unit
  net_db: Unit → in r_db Unit
```

148. The data base behaviour, net_db, offers to receive messages from the link and hub recorders.
149. The data base behaviour, net_db, deposits these messages in respective variables.
150. Initially there is a net, $n : N$,
151. from which is observed its links and hubs.
152. These sets are partitioned into r_l, respectively r_h length lists of non-empty links and hubs.
153. The ab-initio data initialisation behaviour, ab_initio_data, is then the parallel composition of link recorder, hub recorder and data base behaviours with link and hub recorder being allotted appropriate link, respectively hub sets.
154. We construct, for technical reasons, as the reader will soon see, disjoint lists of link, respectively hub recorder identities.

```
value
  net_db:
variable
  lnk_db: (RI×LNK)-set
  hub_db: (RI×HUB)-set
value
  n:N
  ls:L-set = obs_Ls(obs_LS(n))
  hs:H-set = obs_Hs(obs_HS(n))
```

```
  lsl:(L-set)* = partitioning(ls)(rl)
  lhl:(H-set)* = partitioning(hs)(rh)
  rill:RI* axiom len rill = rl = card elems rill
  rihl:RI* axiom len rihl = rh = card elems rihl
  ab_initio_data: Unit → Unit
  ab_initio_data() ≡
    ‖ {lnk_rec(rill[i])(lsl[i])|i:Nat• 1≤i≤rl} ‖
    ‖ {hub_rec(rihl[i])(lhl[i])|i:Nat• 1≤i≤rh}
    ‖ net_db()
```

155. The link and the hub recorders are near-identical behaviours.
156. They both revolve around an imperatively stated **for all ... do ... end**. The selected link (or hub) is inspected and the "data" for the data base is prepared from
157. the unique identifier,
158. the mereology, and
159. the attributes.
160. These "data" are sent, as a message, prefixed the senders identity, to the data base behaviour.
161. We presently leave the ... unexplained.

```
value
147. link_rec: RI → L-set → Unit
155. link_rec(ri,ls) ≡
156.     for ∀ l:L•l ∈ ls do uid_L(l)
157.         let lnk = (uid_L(l),
158.                   obs_mereo_L(l),
159.                   attributes(l)) in
160.         rdb ! (ri,"link",lnk);
161.         ... end
156.     end
```

```
147. hub_rec: RI × H-set → Unit
155. hub_rec(ri,hs) ≡
156.     for ∀ h:H•h ∈ hs do uid_H(h)
157.         let hub = (uid_L(h),
158.                   obs_mereo_H(h),
159.                   attributes(h)) in
160.         rdb ! (ri,"hub",hub);
161.         ... end
156.     end
```

162. The net_db data base behaviour revolves around a seemingly "never-ending" cyclic process.
163. Each cycle "starts" with acceptance of some,
164. either link or hub data.
165. If link data then it is deposited in the link data base,
166. if hub data then it is deposited in the hub data base.

```
value
162. net_db() ≡
163.     let (ri,hol,data) = r_db ? in
164.     case hol of
165.         "link" → ... ; lnk_db := lnk_db ∪ (ri,data),
166.         "hub" → ... ; hub_db := hub_db ∪ (ri,data)
164.     end end ;
162.′    ... ;
162.     net_db()
```

The above model is an idealisation. It assumes that the link and hub data represent a well-formed net. Included in this well-formedness are the following issues: (a) that all link or hub identifiers are communicated exactly once, (b) that all mereologies refer to defined parts, and (c) that all attribute values lie within an appropriate value range. If we were to cope with possible recording errors then we could, for example, extend the model as follows: (i) when a link or a hub recorder has completed its recording then it increments an initially zero counter (say at formula Item 161.); (ii) before the net data base recycles it tests whether all recording sessions has ended and then proceeds to check the data base for well-formedness issues (a–b–c) (say at formula Item 162.′) ■

The above example illustrates the 'interface' phenomenon: In the formulas, for example, we show both manifest domain entities, viz., n, l, h etc., and abstract (required) software objects, viz., $(ui, me, attrs)$.

5.2.2. Data Refreshment

One must also prescribe data refreshment: an interactive user interface dialogue with a set of proper display screens one for selecting the updating of net, of hub or of link attribute names and their types and, for example, two for the respective update of hub and link attribute values. Interaction-prompts may be prescribed: next update, on-line vetting and display of revised net, etc. These and many other aspects may therefore need prescriptions.

5.3. Shared Actions, Events and Behaviours

We now illustrate the concept of shared perdurants via the domain requirements extension example of Sect. 4.4, i.e. Example 13 Pages 20–23.

Example 18. Interface Requirements — Shared Behaviours: Road Pricing Calculator Behaviour:

167. The road-pricing calculator alternates between offering to accept communication from
168. either any vehicle
169. or any toll-gate.

```
167. calc: ci:CI×(vis:VI-set×gis:GI-set)→RLF→TRM→
168.     in {v_c_ch[ci,vi]|vi:VI•vi ∈ vis},
169.        {g_c_ch[ci,gi]|gi:GI•gi ∈ gis} Unit
167. calc(ci,(vis,gis))(rlf)(trm) ≡
168.     react_to_vehicles(ci,(vis,gis))(rlf)(trm)
167. []
169.     react_to_gates(ci,(vis,gis))(rlf)(trm)
167.     pre ci = ci𝒶 ∧ vis = vis𝒶 ∧ gis = gis𝒶
```

170. If the communication is from a vehicle inside the toll-road net
171. then its toll-road net position, vp, is found from the road location function, rlf,
172. and the calculator resumes its work with the traffic map, trm, suitably updated,
173. otherwise the calculator resumes its work with no changes.

174. If the communication is from a gate,
175. then that gate is either an entry gate or an exit gate;
176. if it is an entry gate
177. then the calculator resumes its work with the vehicle (that passed the entry gate) now recorded, afresh, in the traffic map, trm.
178. Else it is an exit gate and
179. the calculator concludes that the vehicle has ended its to-be-paid-for journey inside the toll-road net, and hence to be billed;
180. then the calculator resumes its work with the vehicle now removed from the traffic map, trm.

```
168. react_to_vehicles(ci,(vis,gis),vplf)(trm) ≡
168.   let (vi,(τ,lpos)) = ⨅{v_c_ch[ci,vi]|vi:VI•vi∈ vis} in
170.     if vi ∈ dom trm
171.       then let vp = vplf(lpos) in
172.            calc(ci,(vis,gis),vplf)(trm†[vi↦trm⌢⟨(τ,vp)⟩]) end
173.       else  calc(ci,(vis,gis),vplf)(trm) end end

169. react_to_gates(ci,(vis,gis),vplf)(trm) ≡
169.   let (ee,(τ,(vi,li))) =
169.     ⨅{g_c_ch[ci,gi]|gi:GI•gi∈ gis} in
175.     case ee of
176.       "Enter" →
177.         calc(ci,(vis,gis),vplf)(trm∪[vi↦⟨(τ,SonL(li))⟩]),
178.       "Exit" →
179.         billing(vi,trm(vi)⌢⟨(τ,SonL(li))⟩);
180.         calc(ci,(vis,gis),vplf)(trm\{vi}) end end
```

The above behaviour is the one for which we are to design software ■

5.4. Discussion

TO BE WRITTEN

6. Machine Requirements

Definition 17. Machine Requirements: By machine requirements we shall understand such requirements which can be expressed "sôlely" using terms from, or of **the machine** ■

Definition 18. The Machine: By the machine we shall understand the hardware and software to be built from the requirements ■

The expression *which can be expressed "sôlely" using terms from, or of the machine* shall be understood with "a grain of salt". Let us explain. The machine requirements statements may contain references to domain entities but these are meant to be generic references, that is, references to certain classes of entities in general. We shall illustrate this "genericitiy" in some of the examples below.

6.1. Varieties of Machine Requirements

Analysis of different kinds of requirements, such as exemplified but not so classified in seminal textbooks [Lau02, van09] suggests the following categories of machine requirements: (i) derived requirements, Sect. 6.2, (ii) technology requirements, Sect. 6.3 and (iii) development requirements, Sect. 6.4.

6.2. Derived Requirements

Definition 19. Derived Requirements: By **derived requirements** we shall understand such machine requirements which focus on exploiting facilities of the software or hardware of the machine ■

We use the term 'derived' for the following reason: "exploiting facilities of the software", to us, means that design requirements, have resulted in requirements that imply the presence, in the machine, of concepts (i.e., hardware and/or software), and that it is these concepts that the **derived requirements** "rely" on. We illustrate two forms of derived requirements: actions and events. Derived behaviours could be illustrated but we rely on the interface requirements Example 18: *The Road Pricing Calculator* shared behaviour. There are other kinds of derived requirements. Some are query-like. We leave that for the reader to ponder about.

6.2.1. Derived Actions

Definition 20. Derived Action: By a **derived action** we shall understand (a) a conceptual action (b) that calculates a property or some non-Boolean value (c) from a machine behaviour state (d) as instigated by some actor ■

Example 19. Machine Requirements. Derived Action: Tracing Vehicles: The example is based on the *Road Pricing Calculator Behaviour* of Example 18 on Page 27. The "external" actor, i.e., a user of the *Road Pricing Calculator* system wishes to trace specific vehicles "cruising" the toll-road. That user (a *Road Pricing Calculator* staff), issues a command to the *Road Pricing Calculator* system, with the identity of a vehicle not already being traced. As a result the *Road Pricing Calculator* system augments a possibly void trace of the timed toll-road positions of vehicles. We augment the definition of the *calc*ulator definition Items 167.–180., Pages 27–28.

181. Traces are modeled by a pair of dynamic attributes:
    a. as a programmable attribute, *tra:TRA*, of the set of identifiers of vehicles being traced, and
    b. as a reactive attribute, *vdu:VDU*[14], that maps vehicle identifiers into time-stamped sequences of simple vehicle positions, i.e., as a subset of the *trm:TRM* programmable attribute.
182. The actor-to-calculator *begin* or *end* trace command, *cmd:Cmd*, is modeled as an autonomous dynamic attribute of the *calc*ulator.
183. The *calc*ulator signature is furthermore augmented with the three attributes mentioned above.
184. The occurrence and handling of an actor trace command is modeled as a non-deterministic external choice and a *react_to_trace_cmd* behaviour.
185. The reactive attribute value (`attr_vdu_ch ?`) is that subset of the traffic map (*trm*) which records just the time-stamped sequences of simple vehicle positions being traced (*tra*).

```
type
181.a. TRA = VI-set
181.b. VDU = TRM
182.   Cmd = BTr | ETr
182.   BTr :: VI
182.   ETr :: VI

value
183. calc: ci:CI×(vis:VI-set×gis:GI-set)×(ℂCmd,ℂTrace,ℂTrace)→RLF→TRM→TRA
168.,169.   in {v_c_ch[ci,vi]|vi:VI•vi ∈ vis}, {g_c_ch[ci,gi]|gi:GI•gi ∈ gis}  Unit
167. calc(ci,(vis,gis))(rlf)(trm) ≡
168.   react_to_vehicles(ci,(vis,gis),(attr_cmd_ch,attr_vdu_ch))(rlf)(trm)(tra)
167. []
169.   react_to_gates(ci,(vis,gis),(attr_cmd_ch,attr_vdu_ch))(rlf)(trm)(tra)
167. []
184.   react_to_trace_cmd(ci,(vis,gis),(attr_cmd_ch,attr_vdu_ch))(rlf)(trm)(tra)
167.   pre ci = ci𝒮 ∧ vis = vis𝒮 ∧ gis = gis𝒮
185.   axiom □ attr_vdu_ch[ci]? = trm|tra

168. react_to_vehicles(ci,(vis,gis),(attr_cmd_ch,attr_vdu_ch))(vplf)(trm)(tra) ≡
168.   let (vi,(τ,lpos)) = []{v_c_ch[ci,vi]|vi:VI•vi∈ vis} in
170.   if vi ∈ dom trm
171.     then
171.       let vp = vplf(lpos) in
171.       calc(ci,(vis,gis),(attr_cmd_ch,attr_vdu_ch))(vplf)(trm†[vi↦trm⌢⟨(τ,vp)⟩])(tra) end
173.     else  calc(ci,(vis,gis),(attr_cmd_ch,attr_vdu_ch))(vplf)(trm)(tra) end end

169. react_to_gates(ci,(vis,gis),(attr_cmd_ch,attr_vdu_ch))(vplf)(trm)(tra) ≡
169.   let (ee,(τ,(vi,li))) = []{g_c_ch[ci,gi]|gi:GI•gi∈ gis} in
175.   case ee of
176.     "Enter" →
176.       calc(ci,(vis,gis),(attr_cmd_ch,attr_vdu_ch))(vplf)(trm∪[vi↦⟨(τ,SonL(li))⟩])(tra),
178.     "Exit" →
178.       billing(vi,trm(vi)⌢⟨(τ,SonL(li))⟩);
178.       calc(ci,(vis,gis),(attr_cmd_ch,attr_vdu_ch))(vplf)(trm\{vi})(tra)
169.   end end
```

186. The *react_to_trace_cmd* alternative behaviour is either a "Begin" or an "End" request which identifies the affected vehicle.

[14] VDU: visual display unit

187. If it is a "Begin" request and the identified vehicle is already being traced then we do not prescribe what to do !
188. Else we resume the calculator behaviour, now recording that vehicle as being traced.
189. If it is an "End" request and the identified vehicle is already being traced then we do not prescribe what to do !
190. Else we resume the calculator behaviour, now recording that vehicle as no longer being traced.

```
186. react_to_trace_cmd(ci,(vis,gis),(attr_cmd_ch,attr_vdu_ch))(vplf)(trm)(tra) ≡
186.   let cmd = attr_cmd_ch[ci]? in
186.   case cmd of
186.     mkBTr(vi) →
187.       if vi ∈ tra then chaos
188.       else calc(ci,(vis,gis),(attr_cmd_ch,attr_vdu_ch))(vplf)(trm)(tra ∪ {vi})
186.     mkETr(vi) →
189.       if vi ∉ tra then chaos
190.       else calc(ci,(vis,gis),(attr_cmd_ch,attr_vdu_ch))(vplf)(trm)(tra\{vi})
186.   end end
```

The above behaviour, Items 167.–190., is the one for which we are to design software ■

Example 19 exemplifies a derived action requirement as per definition 20: (a) the action is conceptual, it has no physical counterpart in the domain; (b) it calculates (185.) a visual display (vdu); (c) the vdu value is based on a conceptual notion of traffic road maps (trm), an element of the calculator state; (d) the calculation is triggered by an actor (attr_cmd_ch).

6.2.2. Derived Events

Definition 21. Derived Event: By a **derived event** we shall understand (a) a conceptual event, (b) that calculates a property or some non-Boolean value (c) from a machine behaviour state change ■

Example 20. Machine Requirements. Derived Event: Current Maximum Flow: The example is based on the *Road Pricing Calculator Behaviour* of Examples 18 and 19. By "the current maximum flow" we understand a time-stamped natural number, the number representing the highest number of vehicles which at the time-stamped moment cruised or now cruises around the toll-road net. We augment the definition of the *calc*ulator definition Items 167.–190., Pages 27–30.

191. We augment *calc*ulator signature with
192. a time-stamped natural number valued dynamic programmable attribute, *(t:𝕋,max:Max)*.
193. Whenever a vehicle enters the toll-road net, through one of its gates,
    a. it is checked whether the resulting number of vehicles recorded in the *road traffic map* is higher than the hitherto *max*imum recorded number.
    b. If so, that programmable attribute has its number element "upped" by one.
    c. Otherwise not.

```
type
192.  MAX = 𝕋 × NAT
value
183.,191.  calc: ci:CI×(vis:VI-set×gis:GI-set)×(𝕌Cmd,𝕌Trace,𝕌Trace)→RLF→TRM→TRA→MAX
168.,169.      in {v_c_ch[ci,vi]|vi:VI•vi ∈ vis},{g_c_ch[ci,gi]|gi:GI•gi ∈ gis}  Unit
...
169.  react_to_gates(ci,(vis,gis),(attr_cmd_ch,attr_vdu_ch))(vplf)(trm)(tra)(t,m) ≡
169.    let (ee,(τ,(vi,li))) = □{g_c_ch[ci,gi]|gi:GI•gi∈ gis} in
175.    case ee of
176.      "Enter" →
193.a.          if card dom trm = m
176.,193.b.           then calc(ci,(vis,gis),(attr_cmd_ch,attr_vdu_ch))(vplf)(trm∪[vi↦⟨(τ,SonL(li))⟩])(tra)(τ,m+1),
176.,193.c.           else calc(ci,(vis,gis),(attr_cmd_ch,attr_vdu_ch))(vplf)(trm∪[vi↦⟨(τ,SonL(li))⟩])(tra)(t,m) end end
          ...
      end
```

The above behaviour, Items 167. on Page 27 through 193.c., is the one for which we are to design software ■

Example 20 exemplifies a derived event requirement as per definition 21: (a) the event is conceptual, it has no physical counterpart in the domain; (b) it calculates (193.b.) the max value based on a conceptual notion of traffic road maps (trm), (c) an element of the calculator state.

6.3. Technology Requirements

Definition 22. Technology Requirements: By **technology requirements** we shall understand such machine requirements which primarily focus on alleviating physical deficiencies of the hardware or inefficiencies of the software of the machine — cf. Items (i–ii), i.e., Sects. 6.3.1–6.3.2 below.

We shall, in particular, consider the following kinds of technology requirements: (i) performance requirements and (ii) dependability requirements with dependability requirements being concerned with either (a) accessibility, (b) availability, (c) integrity, (d) reliability, (e) safety, (f) security and/or (g) robustness.

6.3.1. Performance Requirements

Definition 23. Performance Requirements: By *performance requirements* we mean machine requirements that prescribe storage consumption, (execution, access, etc.) time consumption, as well as consumption of any other machine resource: number of CPU units (incl. their quantitative characteristics such as cost, etc.), number of printers, displays, etc., terminals (incl. their quantitative characteristics), number of "other", ancillary software packages (incl. their quantitative characteristics), of data communication bandwidth, etcetera ■

Example 21. Machine Requirements. Technology: Performance: (i) The road pricing system shall be able (i.1) to keep records of up to 50.000 vehicles at any time, (i.2) to record up to 10.000 vehicle positions per second, and (i.3) to bill up to 1000 (distinct) vehicles per second. (ii) A vehicle is assumed to access the road pricing calculator with a mean time between accesses of 5 seconds. (iii) A toll-gate is assumed to access the road pricing calculator with a mean time between accesses of 5 seconds ■

6.3.2. Dependability Requirements

Dependability is a complex notion.

Failures, Errors and Faults To properly define the concept of *dependability* we need first introduce and define the concepts of *failure, error,* and *fault.*

Definition 24. Failure: A machine *failure* occurs when the delivered service deviates from fulfilling the machine function, the latter being what the machine is aimed at [Ran03] ■

Definition 25. Error: An *error* is that part of a machine state which is liable to lead to subsequent failure. An error affecting the service is an indication that a failure occurs or has occurred [Ran03] ■

Definition 26. Fault: The adjudged (i.e., the 'so-judged') or hypothesised cause of an error is a *fault* [Ran03] ■

The term hazard is here taken to mean the same as the term fault. One should read the phrase: "adjudged or hypothesised cause" carefully: In order to avoid an unending trace backward as to the cause,[15] we stop at *the cause which is intended to be prevented or tolerated.*

Definition 27. Machine Service: The service delivered by a machine is its *behaviour* as it is perceptible by its user(s), where a user is a human, another machine or a(nother) system which *interacts* with it [Ran03] ■

Definition 28. Dependability: *Dependability* is defined as the property of a machine such that reliance can justifiably be placed on the service it delivers [Ran03] ■

We continue, less formally, by characterising the above defined concepts [Ran03]. "A given machine, operating in some particular environment (a wider system), may fail in the sense that some other machine (or system) makes, or could in principle have made, a *judgement* that the activity or inactivity of the given machine constitutes a *failure*". The concept of *dependability* can be simply defined as "the quality or the characteristic of being dependable", where the adjective 'dependable' is attributed to a machine whose failures are judged sufficiently rare or insignificant. *Impairments* to dependability are the unavoidably expectable circumstances causing or resulting from "undependability": faults, errors and failures. *Means* for dependability are the techniques enabling one to provide the ability to deliver a service on

[15] An example: "The reason the computer went down was the current supply did not deliver sufficient voltage, and the reason for the drop in voltage was that a transformer station was overheated, and the reason for the overheating was a short circuit in a plant nearby, and the reason for the short circuit in the plant was that . . . , etc."

which reliance can be placed, and to reach confidence in this ability. *Attributes* of dependability enable the properties which are expected from the system to be expressed, and allow the machine quality resulting from the impairments and the means opposing them to be assessed. Having already discussed the "threats" aspect, we shall therefore discuss the "means" aspect of the *dependability tree*.

- Attributes:
  - ◇ Accessibility
  - ◇ Availability
  - ◇ Integrity
  - ◇ Reliability
  - ◇ Safety
  - ◇ Security
- Means:
  - ◇ Procurement
    - ○ Fault prevention
    - ○ Fault tolerance
  - ◇ Validation
    - ○ Fault removal
    - ○ Fault forecasting
- Threats:
  - ◇ Faults
  - ◇ Errors
  - ◇ Failures

Despite all the principles, techniques and tools aimed at *fault prevention, faults* are created. Hence the need for *fault removal. Fault removal* is itself imperfect. Hence the need for *fault forecasting*. Our increasing dependence on computing systems in the end brings in the need for *fault tolerance*. We refer to special texts [Lap92] on the above four topics.

Definition 29. Dependability Attribute: By a *dependability attribute* we shall mean either one of the following: *accessibility, availability, integrity, reliability, robustness, safety* and *security*. That is, a machine is dependable if it satisfies some degree of "mixture" of being accessible, available, having integrity, and being reliable, safe and secure

The crucial term above is "satisfies". The issue is: To what "degree"? As we shall see — in a later section — to cope properly with dependability requirements and their resolution requires that we deploy mathematical formulation techniques, including analysis and simulation, from statistics (stochastics, etc.). In the next seven subsections we shall characterise the dependability attributes further. In doing so we have found it useful to consult [Lap92].

Accessibility Usually a desired, i.e., the required, computing system, i.e., the machine, will be used by many users — over "near-identical" time intervals. Their being granted access to computing time is usually specified, at an abstract level, as being determined by some internal nondeterministic choice, that is: essentially by *"tossing a coin"!* If such internal nondeterminism was carried over, into an implementation, some *"coin tossers"* might not get access to the machine "for a long- long time".

Definition 30. Accessibility: A system being *accessible* — in the context of a machine being dependable — means that some form of *"fairness"* is achieved in guaranteeing users "equal" access to machine resources, notably computing time (and what derives from that).

Example 22. Machine Requirements. Technology: Accessibility: No vehicle access to the road pricing calculator shall wait more than 2 seconds. No toll-gate access to the road pricing calculator shall wait more than 2 seconds.

Availability Usually a desired, i.e., the required, computing system, i.e., the machine, will be used by many users — over "near-identical" time intervals. Once a user has been granted access to machine resources, usually computing time, that user's computation may effectively make the machine unavailable to other users — by "going on and on and on"!

Definition 31. Availability: By *availability* — in the context of a machine being dependable — we mean its readiness for usage. That is, that some form of *"guaranteed percentage of computing time"* per time interval (or percentage of some other computing resource consumption) is achieved, for example, in the form of *"time slicing"* ■

Example 23. Machine Requirements. Technology: Availability: We simplify the availability requirements due to the apparent simplicity of the vehicle movement records and billings. The complete handling of the recording or billing of a vehicle movement shall be done without interference from other recordings or billings ■

Integrity

Definition 32. Integrity: A system has *integrity* — in the context of a machine being dependable — if it is and remains unimpaired, i.e., has no faults, errors and failures, and remains so, without these, even in the situations where the environment of the machine has faults, errors and failures ■

Integrity seems to be a highest form of dependability, i.e., a machine having integrity is 100% **dependable**! The machine is **sound** and is **incorruptible**.

Example 24. **Machine Requirements. Technology: Integrity**: We do not require an explicit formulation of integrity. We instead refer to the reliability, safety, security and robustness measures (below) ■

Reliability

Definition 33. **Reliability:** A system being *reliable* — in the context of a machine being dependable — means some measure of continuous correct service, that is, measure of (mean) time to failure (MTTF)

Example 25. **Machine Requirements. Technology: Reliability**: A road pricing calculator shall have a MTTF of least 10^8 seconds or approx. 40 months.

Safety

Definition 34. **Safety:** By *safety* — in the context of a machine 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 ■

Example 26. **Machine Requirements. Technology: Safety**: The road pricing system, now including the vehicle global position system and the toll-gate sensors and barrier actuator shall have a mean time to catastrophic failure equal to the MTTF, 10^8 seconds ■

Security We shall take a rather limited view of security. We are not including any consideration of security against brute-force terrorist attacks. We consider that an issue properly outside the realm of software engineering. Security, then, in our limited view, requires a notion of *authorised user*, with authorised users being fine-grained authorised to access only a well-defined subset of system resources (data, functions, etc.). An *unauthorised user* (for a resource) is anyone who is not authorised access to that resource.

Definition 35. **Security:** A system being *secure* — in the context of a machine being dependable — means that an *unauthorised user*, after believing that he or she has had access to a requested system resource: (i) cannot find out what the system resource is doing, (ii) cannot find out how the system resource is working and (iii) does not know that he/she does not know! That is, prevention of unauthorised access to computing and/or handling of information (i.e., data) ■

Example 27. **Machine Requirements. Technology: Security**: We omit exemplifying road pricing system security ■

Robustness

Definition 36. **Robustness:** A system is *robust* — in the context of dependability — if it retains its attributes after failure, and after maintenance ■

Thus a robust system is "stable" across failures and "across" possibly intervening "repairs" and "across" other forms of maintenance.

Example 28. **Machine Requirements. Technology: Robustness**: We restrict ourselves to consider only the software of the road pricing system. For every instance of restart after failure it shall be verified that all attributes have retained their appropriate values; and for every instance of software maintenance, see Sect. 6.4.2, the whole system shall be verified, i.e., tested, model checked and proven correct, to the same and full extent that the original system delivery was verified ■

Discussion:

TO BE WRITTEN

6.4. Development Requirements

Definition 37. Development Requirement: By **development requirements** we shall understand (i) process requirements (Sect. 6.4.1), (ii) maintenance requirements (Sect. 6.4.2), (iii) platform requirements (Sect. 6.4.3), (iv) management requirements (Sect. 6.4.4) and (v) documentation requirements (Sect. 6.4.5) ■

6.4.1. Process Requirements

Definition 38. Process Requirement: By a **development process requirements** we shall understand requirements which are concerned with the development process to be followed by the development engineers: whether pursuing formal methods, and to which degree, and (compatibly)/or whether pursuing best practices, and possible details thereof, and (compatibly)/or whether adhering otherwise to established, e.g., IEEE standards, etcetera ■

Example 29. Machine Requirements. Development: Road Pricing System: The road pricing system is to be developed according to the triptych approach; based on, or developing itself, a generic transport domain description, as per the approach outlined in [Bjø16b], expressing suitable predicates about that description, and testing, model checking and proving satisfaction of these verifications; accurately detailing the requirements prescriptions as per the approach outlined in [Bjø16a, this paper (!)]; etcetera; finally testing, model checking and proving satisfaction of $\mathscr{D},\mathscr{S} \models \mathscr{R}$.

6.4.2. Maintenance Requirements

Definition 39. Maintenance Requirements: By *maintenance requirements* we understand a combination of requirements with respect to: (i) *adaptive maintenance*, (iii) *corrective maintenance*, (ii) *perfective maintenance*, (iv) *preventive maintenance* and (v) *extensional maintenance* ■

Maintenance of building, mechanical, electrotechnical and electronic artifacts — i.e., of artifacts based on the natural sciences — is based both on documents and on the presence of the physical artifacts. Maintenance of software is based just on software, that is, on all the documents (including tests) entailed by software — see Definition 52 on Page 36.

Adaptive Maintenance

Definition 40. Adaptive Maintenance: By *adaptive maintenance* we understand such maintenance that changes a part of that software so as to also, or instead, fit to some other software, or some other hardware equipment (i.e., other software or hardware which provides new, respectively replacement, functions) ■

Example 30. Machine Requirements. Development: Adaptive Maintenance: Road pricing system adaptive maintenance shall conclude with a full set of successful formal software tests, model checks, and correctness proofs ■

Corrective Maintenance

Definition 41. Corrective Maintenance: By *corrective maintenance* we understand such maintenance which corrects a software error ■

Example 31. Machine Requirements. Development: Corrective Maintenance: Road pricing system corrective maintenance shall conclude with a full set of successful formal software tests, model checks, and correctness proofs ■

Perfective Maintenance

Definition 42. Perfective Maintenance: By *perfective maintenance* we understand such maintenance which helps improve (i.e., lower) the need for hardware storage, time and (hard) equipment ■

Example 32. Machine Requirements. Development: Perfective Maintenance: Road pricing system perfective maintenance shall conclude with a full set of successful formal software tests, model checks, and correctness proofs ■

Preventive Maintenance

Definition 43. Preventive Maintenance: By *preventive maintenance* we understand such maintenance which helps detect, i.e., forestall, future occurrence of software or hardware failures ■

Example 33. Machine Requirements. Development: Preventive Maintenance: Road pricing system preventive maintenance shall conclude with a full set of successful formal software tests, model checks, and correctness proofs ■

Extensional Maintenance

Definition 44. Extensional Maintenance: By *extensional maintenance* we understand such maintenance which adds new functionalities to the software, i.e., which implements additional requirements ■

Example 34. Machine Requirements. Development: Extensional Maintenance: Road pricing system extensional maintenance shall conclude with a full set of successful formal software tests, model checks, and correctness proofs ■

6.4.3. Platform Requirements

Delineation and Facets of Platform Requirements

Definition 45. Platform: By a [computing] *platform* is here understood a combination of hardware and systems software so equipped as to be able to develop and execute software, in one form or another ■

What the "in one form or another" is transpires from the next characterisation.

Definition 46. Platform Requirements: By *platform requirements* we mean a combination of the following: (i) *execution platform requirements*, (ii) *demonstration platform requirements*, (iii) *development platform requirements* and (iv) *maintenance platform requirements*

Execution Platform

Definition 47. Execution Platform Requirements: By *execution platform requirements* we shall understand such machine requirements which detail the specific (other) software and hardware for the platform on which the software is to be executed ■

Demonstration Platform

Definition 48. Demonstration Platform Requirements: By *demonstration platform requirements* we shall understand such machine requirements which detail the specific (other) software and hardware for the platform on which the software is to be demonstrated to the customer — say for acceptance tests, or for management demos, or for user training ■

Development Platform

Definition 49. Development Platform Requirements: By *development platform requirements* we shall understand such machine requirements which detail the specific software and hardware for the platform on which the software is to be developed ■

Maintenance Platform

Definition 50. Maintenance Platform Requirements: By *maintenance platform requirements* we shall understand such machine requirements which detail the specific (other) software and hardware for the platform on which the software is to be maintained ■

• • •

Example 35. Machine Requirements. Development: Platform Requirements: The road pricing system platform requirements are: the system shall executed and demonstrated on to be detailed and developed and maintained on to be detailed ■

6.4.4. Management Requirements

Definition 51. Management Requirements: By **management requirements** we shall understand requirements that express [Bjø11a, Believable Software Management]

Example 36. Machine Requirements. Development: Management:

TO BE WRITTEN

6.4.5. ***Documentation Requirements***

Definition 52. **Software:** By **software** we shall understand (i) not only ***code*** that may be the basis for executions by a computer, (ii) but also its full ***development documentation***: (ii.1) the stages and steps of ***application domain description,*** (ii.2) the stages and steps of ***requirements prescription,*** and (ii.3) the stages and steps of ***software design*** prior to code, with all of the above including all *validation* and *verification* (incl., *formal test* [test model, test suite, test result, etc.], *model-checking* and *proof*) *documents.* (iii) In addition, as part of our wider concept of software, we also include a comprehensive collection of *supporting documents:* (iii.1) ***training manuals,*** (iii.2) ***installation manuals,*** (iii.3) ***user manuals,*** (iii.4) ***maintenance manuals,*** and (iii.5–6) ***development and maintenance logbooks.*** ■

Definition 53. **Documentation Requirements:** By *documentation requirements* we mean requirements of any of the software documents that together make up software and hardware[16] ■

Example 37. **Machine Requirements. Development: Documentation**: The road pricing system documentation requirements shall include all of the software documents implied by Definition 52 above ■

6.5. Discussion

TO BE TYPED

7. Conclusion

We conclude by reviewing what has been achieved, present shortcomings, a few words on relations to "classical requirements engineering", and possible research challenges.

7.1. What has been Achieved ?

We have put forward a "new approach" to requirements engineering. The "newness" comes from its reliance on there being a reasonably "complete" domain description already at hand. We refer to the introductory section, Sect. 1.3 for a "repeat" of what we think is our contribution. We will, in this section examine some issues of requirements engineering.

(i) The field of software requirements specification has yet to find a stable form around which different contributors (book, paper and Web page authors) structure and within which they express their contributions. It seems, to this author, that there is a bewildering "culture" of different "schools". The various elements of a prevailing such "school" are named: (a) user requirements, (b) system requirements, (c) functional requirements and (d) non-functional requirements. But, to this author, it is hard to see the relations between any pair of these four "classes". Our decomposition into domain, machine and interface requirements is clearly related to either the domain, or the machine or both.

(ii) We have shown how requirements engineering structured into:

- *Problem, Solution and Objective Sketch*
- *System Requirements*
- *User and External Equipment Requirements*
- *Domain Requirements*
  - ◇ *Projection & Simplification*
  - ◇ *Instantiation*
  - ◇ *Determination*
  - ◇ *Extension*
  - ◇ *Fitting*
- *Interface Requirements*
  - ◇ *Shared Endurants*
    - ∞ *Intialisation*
    - ∞ *Refreshment*
  - ◇ *Shared Actions*
  - ◇ *Shared Events*
  - ◇ *Shared Behaviours*
- *Machine Requirements*

[16] — we omit a definition of what we mean by hardware such as the one we gave for software, cf. Definition 52.

- ◇◇ *Derived Requirements*
- ◇◇ *Technology Requirements*
  - ∞ *Performance*
  - ∞ *Dependability*[17]
- ◇◇ *Development Requirements*
- ∞ *Process Reqs.*
- ∞ *Maintenance Reqs.*[18]
- ∞ *Platform Reqs.*[19]
- ∞ *Management Reqs.*
- ∞ *Documentation Reqs.*

(iii) The above-hinted structuring, ((ii)), is logically motivated and is, we claim, a definite contribution to the field of requirements engineering, providing it, we claim, with a stable form.

(iv) The rôle of domain engineering in software engineering, together with domain requirements and interface requirements, is to ensure that the software meets client expectations. By avoiding requirements that express machine concepts the software does not exhibit such properties that make it "user-unfriendly". 'User-friendly' software reflects only concepts that relate to domain phenomena.

(v)

(vi)

7.2. Present Shortcomings

7.3. Comparison to "Classical" Requirements Engineering

[van09] [Lau02]

7.4. Future Work: Research Challenges

We have outlined three major stages of requirements development, and, within these, a number of steps. They can be used, we claim, to advantage, already now — as they have indeed been used over the years in projects with which we have been associated. But more experimental research and path-finder projects has to be absolved. Section **??** of the *Manifest Domains* "chapter" of these lectures covered 'open problems' with respect to domain science & engineering. So we shall only mention 'open problems' with respect to requirements engineering. A number of research issues need be studied. We list a few. (i) **Domain Facets:** Here we are interested in how various domain facets may give rise to special domain-to-requirements "derivation" principles, techniques and tools. We refer to [Bjø10a, 2008]. (ii) **Formal Aspects of Domain Requirements Operations:** (iii) (iv) (v) (vi) (vii) (viii)

8. Thanks

Thanks

8.1. Acknowledgments

9. Bibliography

9.1. Bibliographical Notes

I have thought about domain engineering for more than 20 years. But serious, focused writing only started to appear since [Bjø06, Part IV] — with [Bjø03, Bjø97] being exceptions: [Bjø07] suggests a number of domain science and engineering research topics; [Bjø10a] covers the concept of domain facets; [BE10] explores compositionality and Galois connections. [Bjø08, Bjø10c] show how to systematically, but, of course, not automatically, "derive" requirements prescriptions from domain descriptions; [Bjø11a] takes the triptych software development as a basis for outlining

[17] Accessibility, Availability, Integrity, Reliability, Safety, Security, Robustness

[18] Adaptive, Corrective, Perfective,Preventive, Extensional

[19] Execution, Development, Demonstration, Development, Maintenance

principles for believable software management; [Bjø09, Bjø14a] presents a model for Stanisław Leśniewski's [CV99] concept of mereology; [Bjø10b, Bjø11b] present an extensive example and is otherwise a precursor for the present paper; [Bjø11c] presents, based on the TripTych view of software development as ideally proceeding from domain description via requirements prescription to software design, concepts such as software demos and simulators; [Bjø13] analyses the TripTych, especially its domain engineering approach, with respect to [Mas43, Mas54, Maslow]'s and [PS04, Peterson's and Seligman's]'s notions of humanity: how can computing relate to notions of humanity; the first part of [Bjø14b] is a precursor for [Bjø16b] with the second part of [Bjø14b] presenting a first formal model of the elicitation process of analysis and description based on the prompts more definitively presented in the current paper; and with [Bjø14c] focus on domain safety criticality. The present paper, [Bjø16a], marks, for me, a high point, with [Bjø16b] now constituting the base introduction to domain science & engineering.

9.2. References

[BE10] Dines Bjørner and Asger 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.

[Bjø97] Dines Bjørner. Michael Jackson's Problem Frames: Domains, Requirements and Design. In Li ShaoYang and Michael Hinchley, editors, *ICFEM'97: International Conference on Formal Engineering Methods*, Los Alamitos, November 12–14 1997. IEEE Computer Society. Final Version.

[Bjø03] Dines Bjørner. Domain Engineering: A "Radical Innovation" for Systems and Software Engineering ? In *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, Heidelberg, October 7–11 2003. Springer–Verlag. The Zohar Manna International Conference, Taormina, Sicily 29 June – 4 July 2003. .

[Bjø06] Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.

[Bjø07] Dines 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.

[Bjø08] Dines 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.

[Bjø09] Dines Bjørner. On Mereologies in Computing Science. In *Festschrift: Reflections on the Work of C.A.R. Hoare*, History of Computing (eds. Cliff B. Jones, A.W. Roscoe and Kenneth R. Wood), pages 47–70, London, UK, 2009. Springer.

[Bjø10a] Dines Bjørner. Domain Engineering. In Paul Boca and Jonathan Bowen, editors, *Formal Methods: State of the Art and New Directions*, Eds. Paul Boca and Jonathan Bowen, pages 1–42, London, UK, 2010. Springer.

[Bjø10b] Dines 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.

[Bjø10c] Dines Bjørner. The Rôle 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.

[Bjø11a] Dines Bjørner. Believable Software Management. *Encyclopedia of Software Engineering*, 1(1):1–32, 2011.

[Bjø11b] Dines 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.

[Bjø11c] Dines 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.

[Bjø13] Dines 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).

[Bjø14a] Dines Bjørner. *A Rôle for Mereology in Domain Science and Engineering*. Synthese Library (eds. Claudio Calosi and Pierluigi Graziani). Springer, Amsterdam, The Netherlands, October 2014.

[Bjø14b] Dines Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model. In Shusaku Iida, José Meseguer, and Kazuhiro Ogata, editors, *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi*. Springer, May 2014.

[Bjø14c] Dines Bjørner. Domain Engineering – A Basis for Safety Critical Software. Invited Keynote, ASSC2014: Australian System Safety Conference, Melbourne, 26–28 May, December 2014.

[Bjø16a] Dines Bjørner. From Domain Descriptions to Requirements Prescriptions – A Different Approach to Requirements Engineering. *Submitted for consideration by Formal Aspects of Computing*, 2016.

[Bjø16b] Dines Bjørner. Manifest Domains: Analysis & Description. *Expected published by Formal Aspects of Computing*, 2016.

[CV99] R. Casati and A. Varzi. *Parts and Places: the structures of spatial representation*. MIT Press, 1999.

[ESA] ESA. Global Navigation Satellite Systems. Web, European Space Agency. There are several global navigation satellite systems (http://en.wikipedia.org/wiki/Satellite_navigation) either in operation or being developed: (1.) the US developed and operated GPS (NAVSTAR) system, http://en.wikipedia.org/wiki/Global_Positioning_System; (2.) the EU developed and (to be) operated Galileo system, http://en.wikipedia.org/wiki/Galileo_positioning_system; (3.) the Russian developed and (to be) operated GLONASS, http://en.wikipedia.org/wiki/GLONASS; and (4.) the Chinese Compass Navigation System, http://en.wikipedia.org/wiki/Compass_navigation_system.

[GGJZ00] Carl A. Gunter, Elsa L. Gunter, Michael A. Jackson, and Pamela Zave. A Reference Model for Requirements and Specifications. *IEEE Software*, 17(3):37–43, May–June 2000.

[Hoa85] Charles Anthony Richard Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985. Published electronically: http://www.usingcsp.com/cspbook.pdf (2004).

[IEE90] 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.

[Lap92] J.C. Laprie, editor. *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault–Tolerant Systems*. Springer–Verlag, Vienna, 1992. In English, French, German, Italian and Japanese.

[Lau02] Søren Lauesen. *Software Requirements - Styles and Techniques*. Addison-Wesley, UK, 2002.

[Mas43] Abraham Maslow. A Theory of Human Motivation. *Psychological Review*, 50(4):370–96, 1943. http://psychclassics.yorku.ca/-Maslow/motivation.htm.

[Mas54] Abraham Maslow. *Motivation and Personality*. Harper and Row Publishers, 3rd ed., 1954.

[PS04] Christopher Peterson and Martin E.P. Seligman. *Character strengths and virtues: A handbook and classification*. Oxford University Press, 2004.

[Ran03] Brian 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.

[Ros97] A. W. Roscoe. *Theory and Practice of Concurrency*. C.A.R. Hoare Series in Computer Science. Prentice-Hall, 1997. Now available on the net: http://www.comlab.ox.ac.uk/people/bill.roscoe/publications/68b.pdf.

[Sch00] Steve Schneider. *Concurrent and Real-time Systems — The CSP Approach*. Worldwide Series in Computer Science. John Wiley & Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, January 2000.

[van09] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

10. Indexes

10.1. Index of Definitions

10.2. Index of Examples

10.3. Index of Concepts

Contents