A Rôle for Domain Engineering in Software Development
Why Current Requirements Engineering Seems Flawed !

Dines Bjørner
Fredsvej 11, DK-2840 Holte, Denmark
bjorner@gmail.com – www.imm.dtu.dk/~db

April 19, 2012: 14:00

Contents

1 Opening **2**

2 Domain Engineering **3**
2.1 Transport Simple Entities . . . 3
2.1.1 Transportation Nets . . . 4
2.1.2 Communities and People . . . 9
2.1.3 An Aside on Simple Entity Equality Modulo an Attribute . . . 10
2.1.4 Fleets and Vehicles . . . 11
2.1.5 Vehicles and People . . . 12
2.1.6 Community & Fleet States . . . 13
2.1.7 Time . . . 13
2.1.8 Timetables . . . 14
2.2 Transport Actions . . . 16
2.2.1 Transport Net Actions . . . 17
2.2.2 People and Vehicle Actions . . . 17
2.2.3 Time Table Actions . . . 19
2.3 Transport Events . . . 20
2.3.1 Transport Net Events . . . 20
2.3.2 People Events . . . 21

2.3.3 Vehicle Events . . . 21
2.3.4 Timetable Events . . . 22
2.4 Transport Behaviours . . . 22
2.4.1 Community and Person Behaviours . . . 22
2.4.2 Fleet and Vehicle Behaviours . . . 26
2.5 Discussion of Domain Engineering . . . 29

3 Requirements Engineering 29
3.1 Preliminaries . . . 29
3.1.1 The Machine = Hardware + Software . . . 29
3.1.2 Requirements Prescription . . . 29
3.1.3 A Suitable Decomposition of the Requirements Prescription . . . 29
3.1.4 An Aside on Our Example . . . 30
3.2 Business Process Re-engineering (BPR) . . . 30
3.3 Domain Requirements . . . 30
3.3.1 Projection . . . 31
3.3.2 Instantiation . . . 31
3.3.3 Determination . . . 32
3.3.4 Extension . . . 34
3.4 Interface and Machine Requirements . . . 38
3.5 Discussion of Requirements Engineering . . . 39

4 Software Design 39

5 Concluding Remarks 39
5.1 Domain Models as a Prerequisite for RE . . . 40
5.2 Oh Yes, Conventional RE Contains Elements of DE . . . 40
5.3 Domain Engineering as a Free-standing Activity . . . 41
5.4 Domain Theories . . . 41
5.5 Domain Science . . . 41
5.6 References . . . 41

1 Opening

Before we can design software, the how, we must understand its requirements, the what.
2 Before we can formulate requirements, we must understand the [application] domain.

Examples of domains are:

- air traffic,
- airports,
- container lines,
- banks,
- hospitals,
- pipelines,
- railways,
- stock exchanges,
- "the market",

etcetera. Thus we "divide" the process of developing software into three major phases: 3

- $\mathbb{D}$omain engineering,
- $\mathbb{R}$equirements engineering, and
- $\mathbb{S}$oftware design.

and pursue these phases such that $\mathbb{D}, \mathbb{S} \models \mathbb{R}$, that is, such that we can prove the correctness of the $\mathbb{S}$oftware design with respect to the $\mathbb{R}$equirements presecription in the context of the $\mathbb{D}$omain description, that is, under assumptions about the domain. 4

So let's take a look at what such a domain description might look like and how we might "derive" a [domain] requirements prescription from a domain description.

We shall not go into a methodology of constructing domain descriptions.

2 Domain Engineering 11

We choose as our example domain that of transportation systems, δ:Δ. From any such δ we can observe (obs_) a number of simple entities (Sect. 2.1, Pages 3–16), actions (Sect. 2.2, Pages 16–20), events (Sect. 2.3, Pages 20–22), and behaviours (Sect. 2.4, Pages 22–29). This section will therefore be structured accordingly.

Thus domains are composed from one or more simple entities, actions, events and behaviours; and it is the job of the domain analyser to "discover" these entities, their composition, use and other properties.

2.1 Transport Simple Entities 12

1. There are five classes of simple entities in our example:
   a transportation nets, cf. Sect. 2.1.1 Pages 4–9,
   b people, cf. Sect. 2.1.2 Pages 9–11,
   c vehicles, cf. Sect. 2.1.4 on page 11,
   d time, cf. Sect. 2.1.7 on page 13, and
   e timetables, cf. Sect. 2.1.8 on page 14.

type
1a. N
1b. C
1c. F
1d. T
1e. TT

value
1a. obs_N: $\Delta \to$ N
1b. obs_C: $\Delta \to$ C
1c. obs_F: $\Delta \to$ F
1d. obs_T: $\Delta \to$ T
1e. obs_TT: $\Delta \to$ TT

2.1.1 Transportation Nets 13

Nets, Hubs and Links

2. Nets are composite simple entities from which one can observe
   a sets: hs:HS, of zero, one or more hubs and
   b sets: ls:LS, of zero, one or more links.

type
2. H, L
value
2a. obs_HS: N → HS[1]
2a. obs_Hs: HS → H-**set**
2b. obs_LS: N → LS
2b. obs_Ls: LS → L-**set**

14

Hub and Link Identifiers

3. Hubs and links are uniquely identified.
4. Hub and link identifiers are all distinct.

type
3. HI, LI
value
3. mer_HI: H → HI
3. mer_LI: L → LI
axiom
4. ∀ n:N, h,h′:H, l,l′:L •
4. {h,h′}⊆obs_Hs(n) ∧ {l,l′}⊆obs_Ls(n) ⇒
4. h≠h′⇒mer_HI(h)≠mer_HI(h′) ∧
4. l≠l′⇒mer_LI(l)≠mer_LI(l′)

We say that hub and link identifiers are mereological attributes of hubs, respectively links.

15

5. From a net one can extract (χtr[2]) the hub identifiers of all its hubs.
6. From a net one can extract the link identifiers of all its links.

[1]The prefix obs_ can be pronounced: 'observe' (obs_erve).
[1]mer_HI reads: "the HI 'mereology' contribution from the argument (here H); that is, the prefix mer_ can be pronounced 'mereology' (mer_eology).
[2]The prefix χtr can be pronounced 'extract' (χtract).

value
5. χtrHIs: N → HI-**set**
5. χtrHIs(n) ≡ {mer_HI(h)|h:H•h ∈ obs_Hs(n)}
6. χtrLIs: N → LI-**set**
6. χtrLIs(n) ≡ {mer_LI(l)|l:L•l ∈ obs_Ls(n)}

16

7. Given a net and an identifier of a hub of the net one can get (γet[3]) that hub from the net.
8. Given a net and an identifier of a link of the net one can get that link from the net.

value
7. γetH: N → HI $\xrightarrow{\sim}$ H
7. γetH(n)(hi) ≡
7. **if** hi ∈ χtrHIs(n)
7. **then let** h:H • mer_HI(h)=hi **in** h **end**
7. **else chaos end**
8. γetL: N → LI $\xrightarrow{\sim}$ L
8. γetL(n)(li) ≡
8. **if** li ∈ χtrLIs(n)
8. **then let** l:L • mer_LI(l)=li **in** l **end**
8. **else chaos end**

17

Mereology

9. From a hub one can observe the identifiers of all the (zero or more) links incident upon (or emanating from), i.e., connected to the hub.
10. From a link one can observe the distinct identifiers of the two distinct hubs the link connects.
11. The link identifiers observable from a hub must be identifiers of links of the net.
12. The hub identifiers observable from a link must be identifiers of hubs of the net.

18

value
9. mer_LIs: H → LI-**set**
10. mer_HIs: L → HI-**set**
axiom
9. ∀ n:N,h:H,l:L•h ∈ obs_Hs(n)∧l ∈ obs_Ls(n) ⇒
10. **card** mer_HIs(l)=2
11. ∧ ∀ li:LI • li ∈ mer_LIs(h) ⇒ li ∈ χtrLIs(n)
12. ∧ ∀ hi:HI • hi ∈ mer_HIs(l) ⇒ hi ∈ χtrHIs(n)

19

[3]The prefix γet can be pronounced 'get'.

Maps

Maps, m:M, are abstractions of nets. We shall model maps as follows:

13. hub identifiers map into singleton maps from link identifiers to hub identifiers, such that
    a. if, in m, h_i
    b. maps into $[l_{ij} \mapsto h_j]$,
    c. then h_j maps into $[l_{ij} \mapsto h_i]$ in m, for all such h_i.

type
13. M = HI $\overrightarrow{m}$ (LI $\overrightarrow{m}$ HI)
axiom
13a. ∀ m:M,h_i:HI • h_i ∈ **dom** m ⇒
13b. **let** [l_ij↦h_j] = m(h_i) **in**
13c. h_j ∈ **dom** m ∧ m(h_j)=[l_ij↦h_i]
13a. **end**

14. From a net one can extract its map.

value
14. χtrM: N → M
14. χtrM(n) ≡
14. [hi↦[lij↦hj|
14. lij:LI • lij ∈ mer_LIs(γetH(n)(hi))
14. ∧ hj = γetL(n)(lij)\{hi}] |
14. hi:HI•hi ∈ χtrHIs(n)]

Routes

15. By a route of a net we shall here understand a non-zero sequence of alternative hub and link identifiers such that
    a. adjacent elements of the list are hub and link identifiers of hubs, respectively links of the net, and such that
    b. a link identifier identifies a link one of whose adjacent hubs are indeed identified by the "next" hub identifier of the route, respectively such that
    c. a hub identifier identifies a hub one of whose connected links are indeed identified by the "next" link identifier of the route.

type
93. R′ = (LI|HI)*
93. R = {|r:R′•∃ n:N•wf_R(r)(n)|}
value
93. wf_R: R′ → N → **Bool**
93. wf_R(r)(n) ≡ proper_adjacency(r) ∧ embedded_route(r)(n)

93. proper_adjacency: R′ → **Bool**
93. proper_adjacency(r) ≡
93. ∀ i:**Nat**•{i,i+1}⊆**inds** r⇒is_LI(r(i))∧is_HI(r(i+1))∨is_HI(r(i))∧is_LI(r(i+1))

93. embedded_route: R′ → N → **Bool**
93. embedded_route(r)(n) ≡
93. ∀ i:**Nat**•{i,i+1}⊆**inds** r ⇒
93. is_LI(r(i)) → r(i+1) ∈ mer_HIs(γetL(r(i))(n)),
93. is_HI(r(i)) → r(i+1) ∈ mer_LIs(γetL(r(i))(n))

23

16. Given a net one can calculate the possibly infinite set of all, possibly cyclic but finite length routes:
    a. if li is an identifier of a link of a net then ⟨li⟩ is a route of the net;
    b. if hi is an identifier of a hub of a net then ⟨hi⟩ is a route of the net;
    c. if r and r′ are routes of a net n and if the last identifier of r is the same as the first identifier of r′ then r⌢**tl**r′ is a route of the net.
    d. Only such routes which can be constructed by applying rules 96–16c a finite[4] number of times are proper routes of the net.
17. Similarly one can extract routes from maps.

24

value
94. χtrRs: N → R-**set**
94. χtrRs(n) ≡ **in**
16b. **let** rs={⟨li⟩|li:LI•li ∈ χtrLIs(n)}∪{⟨hi⟩|hi:HI•hi ∈ χtrHIs(n)}
16b. ∪ {⟨hi,li⟩ | hi:HI,li:LI • ⟨hi⟩ ∈ rs
16b. ∧ li ∈ χtrLIs(n)∧li ∈ mer_LIs(γetH(n)(hi))}
16b. ∪ {⟨li,hi⟩ | li:LI,hi:HI • ⟨li⟩ ∈ rs
16b. ∧ hi ∈ χtrHIs(n)∧hi ∈ mer_HIs(γetL(n)(li))}
16c. ∪ {r⌢**tl** r′|r,r′:R•{r,r′}⊆rs ∧ r(**len** rl)=**hd** r′} **in**

[3] is_LI and is_LI are specification language "built-in" functions, one for each type name. In general is_K(e), where K is a type name, expresses whether the simple entity e is of type K (or not).
[4] If applied infinitely many times we include infinite length routes.

94. rs **end**

17. χtrRs: M → R-**set**
17. χtrRs(m) **as** rs
17. **pre** ∃ n:N • m = χtrM(n)
17. **post** ∃ n:N • m = χtrM(n) ∧ rs = routes(n)

For later use we define a concept of a 'stuttered sampling' of a route r. The sequence ℓ is said to be a 'sampling' of a route r if zero or more elements of r are not in ℓ; and the sequence ℓ is said to be a 'stuttering' of a route r if zero or more elements of r are repeated in ℓ — while, in both cases ('sampling' an 'stuttering') the elements of r in ℓ follow their order in r.

18. A sequence, ℓ, of link and hub identifiers (in any order) is a 'stuttered sampling' of a route, r, of a net
    a. if there exists a mapping, mi, from indices of the former into ascending and distinct indices of the latter
    b. such that for all indexes, i, in ℓ, we have that ℓ(i) = r(mi(i)) ∧ i≤mi(i).

type
18a. IM′ = **Nat** $\xrightarrow[m]{}$ **Nat**
18a. IM = {|im:IM′•wf_IM(im)|}

value
18a. wf_IM: IM′ → **Bool**
18a. wf_IM(im) ≡
18a. **dom** im = {1..**max dom** im}
18a. ∧ ∀ i:**Nat** • {i,i+1}⊆**dom** im ⇒ im(i) ≤ im(i+1)

18. is_stuttered_sampling: (LI|HI)* × R → **Bool**
18. is_stuttered_sampling(ℓ,r) ≡
18a. ∃ im:IM • **dom** im = **inds** ℓ ∧ **rng** im⊆**inds** r ⇒
18b. ∀ i:**Nat** • i ∈ **dom** im ⇒ ℓ(i) = r(mi(i))

Hub and Link States A state of a hub (a link) indicates which are the permissible flows of traffic.

19. The state of a hub is a set of pairs of link identifiers where these are the identifiers of links connected to the hub.

20. The state of a link is a set of pairs of distinct hub identifiers where these are the identifiers of the two hubs connected to the link.
21. The state space of a hub is a set of hub states.
22. The state space of a link is a set of link states.

We say that states and state spaces are $\alpha\tau r$ibutes of hubs and links. 28

type
19. HΣ = (LI × LI)**-set**
20. LΣ = (HI × HI)**-set**
21. HΩ = HΣ**-set**
22. LΩ = LΣ**-set**

value
19. $\alpha\tau r$HΣ: H → HΣ
20. $\alpha\tau r$LΣ: L → LΣ
21. $\alpha\tau r$HΩ: H → HΩ
22. $\alpha\tau r$LΩ: L → LΩ

axiom
∀ n:N, h:H, l:L • h ∈ obs_Hs(n)∧l ∈ obs_Ls(n) ⇒
19. **let** hσ = $\alpha\tau r$HΣ(h),
20. lσ = $\alpha\tau r$LΣ(l) **in**
19. ∀ (li,li′):(LI×LI)•(li,li′)∈ hσ⇒{li,li′}⊆χtrLIs(n)
20. ∧ ∀ (hi,hi′):(HI×HI)•(hi,hi′)∈ lσ⇒{hi,hi′}⊆χtrHIs(n)
21. ∧ hσ ∈ $\alpha\tau r$HΩ(h)
22. ∧ lσ ∈ $\alpha\tau r$LΩ(l) **end**

2.1.2 Communities and People 29

23. A community is a community of people here considered an unordered set.
24. As simple entities we consider people (persons) to be uniquely identifier atomic dynamic inert entities.

    We shall later view such people as a main state component of people as behaviours.
25. No two persons have the same unique identifier.
26. Essential attributes of persons are:

    a name, c gender, e height,
    b ancestry, d age, f weight,

    and others. We omit expressing statistically determined relations between values of some of these attributes.

30 Additional attributes will be brought forward in the next section (Vehicles).

type
23. P
91. PI
value
23. obs_Ps: C → P-**set**
91. $\alpha\tau r$PI: P → PI
axiom
92. ∀ p,p′:P • p≠p′ ⇒ $\alpha\tau r$PI(p) ≠ $\alpha\tau r$PI(p′)
type
26. PNm, PAn, PGd, PAg, PHe, PWe, ...
value
26a. $\alpha\tau r$PNm: P → PNm
26b. $\alpha\tau r$PAn: P → PAn
26c. $\alpha\tau r$PGd: P → PGd
26d. $\alpha\tau r$PAg: P → PAg
26e. $\alpha\tau r$PHe: P → PHe
26f. $\alpha\tau r$PWe: P → PWe

31

27. From any set of persons one can extract its corresponding set of unique person identifiers.

value
27. χtrPIs: P-**set** → PI-**set**
27. χtrPIs(ps) ≡ {obs_PI(p)|p:P•p ∈ ps}
axiom
27. ∀ ps:P-**set** • **card** ps = **card** χtrPIs(ps)

2.1.3 An Aside on Simple Entity Equality Modulo an Attribute 32

Attributes have names and values. (Not just people, but also the simple entities of nets,, hubs and links, as well as of other simple entities to be introduced later.) Some attributes are dynamic, that is, their values may change. We wish to be able to express that a simple entity, p, some of whose attribute values may change, is "still, basically, that same" simple entity, that is, that $p = p'$ — where we assume that the only thing which does not change
33 is some notion of a unique simple entity identifier.

28. The attribute observers of people are those of observing names, ancestry, gender, age, height, weight, and others.

    Let SE$\alpha\tau r$set stand for the set of attribute functions of the simple entity whose class (type) is SE.

29. Then to express that a simple entity of type SE in invariant modulo some observer function $\alpha\tau r$A, specifically, in this case, that a person is invariant wrt. height, we write as is shown in formula 29. below, where p and p′ is the ("before", "after") person that is claimed to be "the same", i.e. invariant modulo $\alpha\tau r$A.

type
28. P$\alpha\tau r$set = {| $\alpha\tau r$PNm, $\alpha\tau r$An, $\alpha\tau r$Gd, $\alpha\tau r$Ag, $\alpha\tau r$He, $\alpha\tau r$We, ...|}
axiom
29. ∀ $\alpha\tau r\mathcal{F}$:P$\alpha\tau r$set• $\alpha\tau r\mathcal{F}$ ∈ P$\alpha\tau r$set\{$\alpha\tau r$H} ⇒ $\alpha\tau r\mathcal{F}$(p)=$\alpha\tau r\mathcal{F}$(p′)

Formula line 28. is not a definition in the specification language, but is a notational convention, that is, it is meta-linguistic and saves us a lot of trivial writing.

2.1.4 Fleets and Vehicles

34

30. A fleet is a composite simple entity.
31. From a fleet one can observe its atomic simple sub-entities of vehicle.
    a Vehicles, in addition to their unique vehicle identity,
    b may enjoy some static attributes: weight, size, etc., and dynamic attributes: directed velocity, directed acceleration,
    c position on the net:
    d at a hub or on a link, etc.

35

type
30. F
31. V
31a. VI
31b. We, Sz, ..., DV, DA, ...
value
31. obs_Vs: F → V-**set**
31a. obs_VI: V → VI
31b. $\alpha\tau r$We: V → We, ..., $\alpha\tau r$DV: V → DV, ...
31c. $\alpha\tau r$VP: V → VP
type
31d. VP == atH(hi) | onL(fhi,li,f:**Real**,thi) **axiom** 0<f≪1
axiom
31a. ∀ v,v′:V•v≠v′ ⇒ obs_VI(v)≠obs_VI(v′)

36

32. Buses are vehicles, but not all vehicles are buses.

33. Vehicles are either in the traffic (to be defined later) or are not.
34. From any set of vehicles one can extract its corresponding set of unique vehicle identifiers.

type
32. B ⊂ V

value
32. is_B: V → **Bool**
33. is_InTF: V → **Bool**
34. χtrVIs: V-**set** → VI-**set**
34. χtrVIs(vs) ≡ {obs_VI(v)|v:V•v ∈ vs}

axiom
34. ∀ vs:V-**set** • **card** vs = **card** χtrVIs(vs)

2.1.5 Vehicles and People

37

35. Vehicles in traffic have a driver who is a person, and distinct vehicles have distinct drivers.
36. Vehicles in traffic have zero, one or more passengers – who are persons different from the driver.
37. Vehicles have one owner (who is a person) and persons own zero or more vehicles.

35. ατrDriver: V $\xrightarrow{\sim}$ PI
35. **pre** ατrDriver(v): is_InTF(v)
36. ατrPass: V → PI-**set**
36. **pre** ατrPass(v): is_InTF(v) ⇒ ατrDriver(v)∉ατrPs(v)
37. ατrOwner: V → PI
37. ατrOwn: P → VI-**set** [..listed here, but not in Sect. 2.1.2..]

38

38. In the (domain state) context of the set of persons, ps, and the set of vehicles, vs, in the domain (δ:Δ), we have the following constraints:
    a the person, p, identified by pi, as the owner of a vehicle, v, in vs, is in ps; and
    b the vehicle, v, identified by vi, as being owned be a person, p, in ps, is in vs.

38. **axiom** ∀ δ:Δ,ps:P-**set**,vs:V-**set** • ps=obs_Ps(δ)∧vs=obs_Vs(δ) ⇒
38a. ∀ v:V • v ∈ vs ⇒ ατrOwner(v) ∈ χtrPIs(ps)
38b. ∧ ∀ p:P • p ∈ ps ⇒ ατrOwn(p) ⊆ χtrVIs(vs)

39

39. Given a set of persons one can extract the set of the unique person identifiers of these persons.
40. Given a set of persons one can extract the set of the unique vehicle identifiers of vehicles owned by these persons.
41. Given a set of persons and a unique person identifier (of one of these persons) one can get that person.
42. Given a set of vehicles one can extract the set of the unique vehicles identifiers of these vehicles.

40

value
39. χtrPIs: P-**set** $\rightarrow$ PI-**set**
39. χtrPIs(ps) $\equiv$ $\{\alpha\tau r$PI(p)|p:P•p $\in$ ps$\}$

40. γetP: P-**set** $\rightarrow$ PI $\xrightarrow{\sim}$ P
40. γetP(ps)(pi) $\equiv$ **let** p:P• p $\in$ ps $\wedge$ pi=$\alpha\tau r$PI(p) **in** p **end**
40. **pre** pi $\in$ χtrPIs(ps)

41. χtrVIs: V-**set** $\rightarrow$ VI-**set**
41. χtrVIs(vs) $\equiv$ $\{\alpha\tau r$VI(v)|v:V•v $\in$ vs$\}$

42. γetV: V-**set** $\rightarrow$ VI $\xrightarrow{\sim}$ V
42. γetV(vs)(vi) $\equiv$ **let** v:V• v $\in$ vs $\wedge$ vi=$\alpha\tau r$VI(v) **in** v **end**
42. **pre** vi $\in$ χtrVIs(vs)

2.1.6 Community & Fleet States

41

43. We shall later need to refer to a state consisting of pairs of communities and fleets.

43. CFΣ = C $\times$ F

2.1.7 Time

42

Time is an elusive "quantity", ripe, always, for philosophical discourses, for example: [4, J. M. E. McTaggart], [2, Wayne D. Blizard (1990)] and [7, Johan van Benthem (1991)]. Here we shall take a somewhat more mundane view of time.

43

44. Time is here considered a dense, enumerable set of points.
45. A time interval is the numerical distance between two such points.

46. There is a time starting point and thus we can speak of the time interval since then!
    a One can compare two times and one can compare two time intervals.
    b One can add a time and an interval to obtain a time.
    c One can subtract a time interval from a time to obtain, conditionally, a time.
    d One can subtract a time from a time to obtain, conditionally, a time interval.
    e One can multiply a time interval with a real to obtain a time interval.
    f One can divide one time interval by another to obtain a real.

44

type
44. T
45. TI
value
46. obs_TI: T → TI
46a. $<,\leq,=,>,\geq$: ((T×T)|(TI×TI)) → **Bool**
46b. +: T×TI → T
46c. −: T×TI $\xrightarrow{\sim}$ T **axiom** ∀ −(t,ti) • obs_TI(t)≥ti
46d. −: ((T×T)|(TI×TI)) $\xrightarrow{\sim}$ TI **axiom** ∀ −(τ,τ') • $\tau'\leq\tau$
46e. ∗: TI×**Real** → TI
46f. /: TI×TI → **Real**

2.1.8 Timetables

45

By a timetable we shall here understand a transport timetable: a listing of the times that public transport services, say a bus, arrive and depart specified locations. We shall model a concept of timetables in four "easy" steps by first defining bus stops, then bus schedules and finally timetables.

46

Bus Stops To properly define a timetable we thus need to introduce the notion of 'specified locations'.

47. By a bus location (that is, a bus stop), we shall understand a location
    a either *at a hub*
    b or *down a fraction of the distance between two hubs (a from and a to hub) along a link.*

48. The fraction is a real close to 0 and certainly much less than 1.

type
47. S = atH | onL
47a. atH == $\mu\alpha\kappa$AtH(hi:HI)
47b. onL == $\mu\alpha\kappa$OnL(fhi:HI,li:LI,f:Frac,thi:HI)
48. Frac = **Real** **axiom** ∀ f:F•0<f≪1

47

Bus Schedules

49. A bus stop visit is modelled as a triple: an arrival time, a bus stop location and a departure time — such that the latter is larger than (i.e., "after") the former.

50. A bus schedule is a pair: a route and a list of two or more "consecutive" bus stop visits where "consecutiveness" has two parts:

    a the projection of the list of bus stop visits onto just a list of its "at Hub" and "on Link" identifiers must form a stuttered sampling of the route,

    b departure times of the "former" bus stop visit must be "before" the arrival time of the latter, and

    c if two or more consecutive stops along the same link, then a former stop must be a fraction down the link less than a latter stop.

48

type
49. BV = T × S × T **axiom** ∀ (at,bs,dt):S • at<dt
50. BS′ = R × BVL, BVL = BV*
50. BS = {|bs•wf_BS(bs)|}
value
50. wf_BS(r,l) ≡
50b. is_stuttered_sampling(proj(l),r)
50b. ∧ ∀ i:**Nat**•{i,i+1}<**inds** l ⇒
50b. **case** (l(i),l(i+1)) **of**
50b. ((_,atH(hi),dt),(at,atH(hi′),_)) → dt<at,
50b. ((_,atH(hi),dt),(at,onL(fi,li,f,ti),_)) → dt<at,
50b. ((_,onL(fi,li,f,ti),dt),(at,atH(hi),_)) → dt<at,
50b. ((_,onL(fi,li,f,ti),_),(at,onL(fi′,li′,f′,ti′),_)) → dt<at
50c. ∧ fi=fi′∧li=li′∧ti=ti′ ⇒ f<f′ **end**
50a. proj: BV* → (HI|LI)*
50a. proj(bvl) ≡
50a. ⟨ **case** bs **of** atH(hi) → hi, onL(_,li,_,_) → li **end**
50a. | i:**Nat**,bv:BV: i ∈ **inds** bvl ∧ bv=bvl(i)=(_,bs,_) ⟩

49

Bus Transport Timetables

51. Bus schedules are grouped into bus lines
52. and bus schedules have distinct identifiers.
53. A timetable is now a pair of
    a. a transport map and
    b. a table which
        i. to each bus line associates a sub-timetable
            - which to each bus schedule identifier
            - associates a bus schedule,

such that

a. no bus schedule identifier appears twice in the timetable and
b. each bus schedule is commensurate with the transport map.

50

type
51. BLId
52. BSId
53. $TT' = M \times TBL$
53b. $TBL = BLid \xrightarrow{m} SUB\_TT$
53(b)i. $SUB\_TT = BSId \xrightarrow{m} BS$
53. $TT = \{|tt:TT' \bullet wf\_TT(tt)|\}$

value
53. wf_TT: $TT' \rightarrow$ **Bool**
53. wf_TT(m,tbl) ≡
53a. $\forall$ bsm,bsm′:(BSId $\xrightarrow{m}$ BS)•{bsm,bsm′}⊆**rng** tbl⇒**dom** bsm ∩ **dom** bsm′={}
53b. $\wedge \forall$ (r,bvl):BS • (r,bvl) ∈ **rng** bsm ⇒ r ∈ routes(m)

2.2 Transport Actions

51

We consider each of four of the these three kinds of transport simple entities as being "the center" of events: the net, people and vehicles and timetables.

2.2.1 Transport Net Actions

52

54. One can insert hubs into a net to obtain an updated net. The inserted hub has no 'connected link identifiers'.

55. One can remove a hub from a net to obtain an updated net. The removed hub must have no 'connected link identifiers'.

56. One can insert a link into a net to obtain an updated net. The inserted link must have two existing 'connecting hub identifiers' and their hubs (cannot have contained the link identifier of the inserted link) must now record that link identifier as the only change to their attributes.

57. One can remove a link from a net to obtain an updated net. The hubs identified by the removed links' 'connecting hubs' must have their 'connected link identifiers' no longer reflecting the removed link — as their only change.

53

```
value
54. insertH: H → N ~→ N
54. insertH(h)(n) as n′
54.   pre h∉obs_Hs(n)
54.   post obs_Hs(n)=obs_Hs(n′) ∪ {h} ∧
54.     obs_Ls(n)=obs_Ls(n′)

55. removeH: HI → N ~→ N
55. removeH(hi)(n) as n′
55.   pre hi ∈ χtrHIs(n)
55.   post obs_LIs(get_HI(hi)(n))={} ∧
55.     obs_Hs(n′)=obs_Hs(n)\{get_HI(hi)(n)}

56. insertL: L → N ~→ N
56. insertL(l)(n) as n′
56.   pre l∉obs_Ls(n)
56.   post obs_Ls(n′)=obs_Ls(n)∪{l}
56.     let {hi,hi′}=obs_HIs(l) in
56.     let (h,h′)=(γetH(hi)(n),γetH(hi′)(n)),
56.       (nh,nh′)=(γetH(hi)(n′),γetH(hi′)(n′)) in
56.     obs_LIs(nh)=obs_LIs(h)∪{obs_LI(l)},
56.     obs_LIs(nh′)=obs_LIs(h′)∪{obs_LI(l)} end end

57. removeL: LI → N ~→ N
57. removeL(li)(n) as n′
57.   pre li ∈ χtrLIs(n)
57.   post obs_Ls(n)=obs_Ls(n′)\{l}
57.     let {hi,hi′}=obs_HIs(get_L(li)(n)) in
57.     let (h,h′)=(get_H(hi)(n),get_H(hi′)(n)),
57.       (nh,nh′)=(get_H(hi)(n′),get_H(hi′)(n′)) in
57.     obs_LIs(nh)=obs_LIs(h)\{li},
57.     obs_LIs(nh′)=obs_LIs(h′)\{li} end end
```

2.2.2 People and Vehicle Actions

54

58. We shall only consider actions on people and vehicles in the (state) context of the community and fleet of a transport system, cf. Sect. 2.1.5, Item 38 (Page 12).

59. People can transfer (xfer) ownership of vehicles (being transferred vi,v,v′) one-at-a-time, from one person (fpi,fp – selling) to another person (tpi,tp buying).

55

```
value
58. xfer_V: PI×VI×PI → (C×F) → (C×F)
58. xfer_V(fpi,vi,tpi)(c,f) as (c′,f′)
```

58. **pre** ...
58. **post** xfer_V(fpi,vi,tpi)(obs_Ps(c),obs_Vs(f)) = (ps′,vs′)
58. ∧ ∀ $\mathcal{F}_C$:$\alpha\tau r$Cs(c)•$\mathcal{F}_C$(c)=$\mathcal{F}_C$(c′)
58. ∧ ∀ $\mathcal{F}_F$:$\alpha\tau r$Fs(f)•$\mathcal{F}_F$(f)=$\mathcal{F}_F$(f′)
58. xfer_V: PI×VI×PI → (P-**set**×V-**set**) → (P-**set**×V-**set**)
59. xfer_V(fpi,vi,tpi)(ps,vs) **as** (ps′,vs′)
60a. **pre** fpi≠tpi∧{fpi,tpi}⊆χtrPIs(ps)∧vi ∈ χtrVIs(vs)
60b. **post let** (fp,tp)=(γetP(fpi)(ps),γetP(tpi)(ps)),
60c. (fp′,tp′)=(γetP(fpi)(ps′),γetP(tpi)(ps′)),
60d. (v,v′)=(γetV(vi)(vs),γetP(vi)(vs′)) **in**
60e. ps\{fp,tp} = ps′\{fp′,tp′} ∧ vs\{v} = vs\{v′}
60f. ∧ fp′ = sell(fp,vi) ∧ tp′ = buy(tp,vi) ∧ v′ = xfer_Owner(vi,fp,tp) **end**

56 We define the three auxiliary functions: sell, buy and xfer_Owner below.

60. We explain the above pre/post conditions:
    a The from and to persons must be distinct and they and the identified vehicle must be in the current domain state.
    b We need to be able to refer to the from and to persons before
    c and after the transfer vehicle ownership action,
    d as well as to the vehicle changing ownership.
    e Except for the persons and vehicle involved in the transfer operation no changes occur to the persons and vehicles of the current domain state.
    f Simultaneously the from person sells the vehicle, the to person buys that same vehicle and the vehicle changes owner.

57

value
61. sell: P × VI → P
61. sell(p,vi) **as** p′
61a. obs_PI(p)=obs_PI(p′)
61b. ∧ vi ∈ $\alpha\tau r$Own(p) ∧ vi ∉ $\alpha\tau r$Own(p′)
61c. ∧ ∀ F:P$\alpha\tau r$set\{$\alpha\tau r$VI}•F(p)=F(p′)
62. buy: P × VI → P
62. buy(p,vi) **as** p′
62a. obs_PI(p)=obs_PI(p′)
62b. ∧ vi ∉ $\alpha\tau r$Own(p) ∧ vi ∈ $\alpha\tau r$Own(p′)
62c. ∧ ∀ F:P$\alpha\tau r$set\{$\alpha\tau r$VI}•F(p)=F(p′)
63. xfer_Owner: PI × V × PI → V
63. xfer_Owner(fpi,v,tpi) **as** v′
63a. obs_VI(v)=obs_VI(v′)
63b. ∧ fpi=$\alpha\tau r$Owner(v) ∧ tpi≠$\alpha\tau r$Owner(v)
63c. ∧ fpi≠$\alpha\tau r$Owner(v′) ∧ tpi=$\alpha\tau r$Owner(v′)
63d. ∧ ∀ F:P$\alpha\tau r$set\{$\alpha\tau r$VI}•F(p)=F(p′)

58

61. The buyer function:
    a The seller identity is unchanged.
    b The vehicle was owned by the seller before, but not after the transfer.
    c All other seller attributes are unchanged.
62. The seller function:
    a The buyer identity is unchanged.
    b The vehicle was not owned by the buyer before, but is owned by the buyer after the transfer.
    c All other buyer attributes are unchanged.
63. The vehicle ownership change function:
    a The vehicle identity is unchanged.
    b The seller identity is noted in the vehicle before the transfer but is not noted after the transfer.
    c The buyer identity is not noted in the vehicle before the transfer but is noted after the transfer.
    d All other vehicle attributes are unchanged.

2.2.3 Time Table Actions

59

Timetables are dynamic inert simple entities. They do not change their value by own volition. Their value is changed only by some external action upon them.

64. One can create an empty timetable.
65. One can inquire whether a timetable is empty.
66. One can inquire as to the set of bus line identifies of a timetable.
67. One can inquire as to the set of all bus lines' unique bus schedules identifiers.
68. For every bus line identity one can inquire as to the set of unique bus schedule identifiers.
69. One can insert a bus schedule with an appropriate new bus schedule identifier into a timetable.
70. One can delete an appropriately identified bus schedule from a non-empty timetable.

60

value
64. emptyTT: **Unit** → TT
64. emptyTT() **as** tt **axiom** is_empty(tt)
65. is_emptyTT: TT → **Bool**
65. is_emptyTT(_,tbl) ≡ **case** m **of** (_,[bli↦bsm]∪ tbl′)→**false**,_→**true end**
66. χtrBLIds: TT → BLId-**set**
66. χtrBLIds(_,tbl) ≡ **dom** tbl
67. χtrBSIds: TT → BSid-**set**
67. χtrBSIds(_,tbl) ≡ ∪{tbl(bli)|bli:BLid•bli ∈ **dom** tbl}
68. χtrBSIds: TT × BLid → BSid-**set**
68. χtrBSIds((_,tbl),bli) ≡ **dom** tbl(bli)
69. insert_BS: (BLid × (BSid × BS)) → TT $\xrightarrow{\sim}$ TT
69. insert_BS(bli,(bsi,bs))(m,tbl) **as** (m′,tbl′)
69. **pre** wf_TT(m,tbl) ∧ bsi ∉ χtrBSids(m,tbl)
69. **post** wf_TT(m′,tbl′) ∧ m=m′
69. ∧ bli∉ **dom** tbl ⇒ tbl′ = tbl ∪ [bli↦[bsi↦bs]]
69. ∧ bli ∈ **dom** tbl ⇒ tbl′ = tbl † [bli↦tbl(bli)∪[bsi↦bs]]
70. delete_BS: (BLid × (BSid × BS)) → TT $\xrightarrow{\sim}$ TT
70. delete_BS(bli,(bsi,bs))(m,tbl) **as** (m′,tbl′)
70. **pre** wf_TT(m,tbl) ∧ bli ∈ **dom** tbl ∧ bsi ∈ **dom**(tbl(bli))
70. **post** wf_TT(m′,tbl′) ∧ m=m′ ∧ tbl′ = tbl † [bli↦tbl(bli)\{bsi}]

2.3 Transport Events 61

2.3.1 Transport Net Events

Events are characterisable by a predicate over before/after state pairs and times. The event of a mudslide "removing" the linkage between two hubs can be modelled as follows: first the removal of the affected link (ℓ, connecting hubs h' and h''), then the insertion of two fresh hubs (h''' and h''''), and finally the insertion of new links (ℓ' and ℓ'' between h' and h''', respectively h'' and
62 h''''). With these "actions" as the only actions at or during the event we have that:

71. A link_disappearance predicate can be defined as follows:
    a. there exists h' and h'' in net n with these hubs becoming nh' and nh'' in net n', and
    b. there exists exactly and only h''' and h'''' in the new net n' which were not in the old net n,
    c. exactly one link, ℓ', has disappeared from net n (that is: was in n but is not in n'), and exactly two links, ℓ'', ℓ''', (which were not in n) have appeared in net n',
    d. the two new links, ℓ'' and ℓ''', are linking h' with h''', respectively h'' with h'''',
    e. hub h' (h'') is no longer connected to ℓ' (ℓ'), but includes ℓ'' (ℓ'''),
    f. hub h''' (h'''') connects to only ℓ'' (ℓ'''), and
    g. link ℓ' (ℓ'') connects $\{h', h'''\}$ ($\{h', h'''\}$).

63

The event predicate *link_disappearance* is between the nets before and after the event – and some arbitrary time.

type
T
value
71. link_disappearance: N × N → T → **Bool**
71. link_disappearance(n,n′)(t) ≡
71. **let** (hs,ls)=(obs_Hs,obs_Ls)(n), (hs′,ls′)=(obs_Hs,obs_Ls)(n′) **in**
71a. ∃ h′,h″:H•{h,h′}⊆hs ∩ hs′
71a. ∧ **let** (hi′,hi″)=(obs_HI(h′),obs_HI(h″)) **in**
71a. **let** (nh′,nh″)=(get_H(hi′)(n′),get_H(hi″)(n′)) **in**
71b. ∃ h‴,h⁗:H•{h‴,h⁗}=hs′\hs
71c. ∧ ∃ l′:L•{l′}=obs_Ls(n) ∩ obs_Ls(n′) ∧ ∃ l″,l‴:L•{l″,l‴}=obs_Ls(n′)\obs_Ls(n′)
71d. ∧ $\alpha\tau r$HIs(l″)={hi′,obs_HI(h‴)}∧$\alpha\tau r$HIs(l‴)={hi″,obs_HI(h⁗)}
71e. ∧ $\alpha\tau r$LIs(h′)=$\alpha\tau r$LIs(nh′)\{obs_LI(l′)}∪ obs_LI(l″) ∧ $\alpha\tau r$LIs(h″)=$\alpha\tau r$LIs(nh″)\{obs_LI(l′)}∪ obs_LI(l‴)
71f. ∧ $\alpha\tau r$HIs(l′)={obs_HI(nh′),obs_HI(h‴)}
71g. ∧ $\alpha\tau r$HIs(l″)={obs_HI(nh″),obs_HI(h⁗)}
end end end

2.3.2 People Events 64

72. People are born and people pass away.

value
72. birth: P-**set** × P-**set** → T → **Bool**
72. birth(ps,ps′)(t) ≡ ∃ p:P • p ∉ ps ∧ p ∈ ps′ ∧ ps′=ps ∪{p}
72. death: P-**set** × P-**set** → T → **Bool**
72. death(ps,ps′)(t) ≡ ∃ p:P • p ∈ ps ∧ p ∉ ps′ ∧ ps′=ps\{p}

2.3.3 Vehicle Events 65

73. Vehicles are manufactured and vehicles are scrapped.

74. Two or more vehicles end up in a mass collision.

value
73. mfgd: V-**set** × V-**set** → T → **Bool**
73. mfgd(vs,vs′)(t) ≡ ∃ v:V • v ∉ vs ∧ v ∈ vs′ ∧ vs′=vs ∪{v}
73. scrpd: V-**set** × V-**set** → T → **Bool**
73. scrpd(vs,vs′)(t) ≡ ∃ v:V • v ∈ vs ∧ v ∉ ps′ ∧ vs′=vs\{v}
74. coll: V-**set** × V-**set** → T → **Bool**
74. coll(vs,vs′)(t) ≡ χtrVIs(vs)=χtrVIs(vs′)

74. $\wedge$ $\exists$ vs″:V-**set** • **card** vs″$\geq$2 $\wedge$ vs″$\subset$vs′
74. $\wedge$ $\forall$ v,v′:V-**set**•v$\neq$b′ $\wedge$ {v,v′}$\subseteq$vs″ $\wedge$ samePos(v,v′)
74. samePos: V $\times$ V $\rightarrow$ T $\rightarrow$ **Bool**
74. samePos(v,v′)(t) $\equiv$
74. **case** ($\alpha\tau r$VP,$\alpha\tau r$VP) **of** (onL(fhi,li,f,thi),onL(fhi,li,f,thi)) $\rightarrow$ **true**, _ $\rightarrow$ **false end**

2.3.4 Timetable Events 66

Timetables are considered to be concepts. They may be recorded on paper, electronically or on billboards. Somehow they, i.e., the timetable for some specific form of vehicles and for some specific net, are all copies of one another. They somehow do not disappear. So we decide not to conjure an image, or images, of timetable events and then "model" it, or them.

2.4 Transport Behaviours 67

One thing is a simple entity, or a constellation of simple entities; another thing is a behaviour "centered around" that, or those, simple entities: a net, a person, a vehicle, or other such simple entities as behaviours. As we shall soon see, we model behaviours as processes with a notion of a state which significantly includes a simple net entity, a simple person entity, respectively a simple vehicle entity. Colloquially we can thus speak of some phenomenon, both by referring to it as a simple entity and by referring to it as a be-
68 haviour. The complexity of transport behaviours is such that we "stepwise" refine a sketch of transport behaviours; first we sketch some aspects of People Behaviours (Sect. 2.4.1), then similarly of Vehicle Behaviours (Sect. 2.4.2), of Timetable Behaviours before tackling the more composite Net Behaviours

2.4.1 Community and Person Behaviours 69

We make a distinction between describing the dynamically varying number of people of our domain, δ:Δ — modelled as the behaviour community — and the individual person, modelled as the behaviours nascent and person.

We need to model each individual person behaviour and do so as a CSP process [3]. We also need to model the dynamically varying number of person behaviours. But CSP cannot
70 model that "easily". So we use some technical tricks — of which we are not "proud".

The model, with one community and an indefinite number of nascent and person behaviours, is not really a proper model of the domain of people. The model of the birth of persons — reflected in the community and nascent/person behaviours — and the decease of persons — reflected in the same behaviours — is not a very good model. The problem is that we know of no formal specification language which handles the dynamic creation
71 and demise of processes.[5]

[5]The π-Calculus is a mathematical system (a notation etc.) for investigating mobile processes and

A Community System Behaviour

75. The concurrent constellation of one community and an indefinite number of pairs of nascent and person behaviours will be referred to as the people_system behaviour.

76. The people_system behaviour is refers to a global (constant) value pids: an indefinite set of the unique identifiers of *nascent* (as yet unborn) and persons.

77. Each individual of the indefinite number of nascent behaviours is initialised with its (future) unique person identity.

78. The community behaviour models the birth of persons and kicks off the identified nascent behaviour by communicating a person (i.e., a "baby") to the nascent behaviour.

79. The identity of a "deceased" person behaviour is communicated to the community behaviour. 72

80. The communications mentioned in Items 78–79 are modelled by CSP output/inputs over a set of unique person identified community_to_nascent channels, CtN(pi), and person_to_community channels, NtC(pi) channels.

81. Once a nascent behaviour "comes alive" (i.e., a person is alive), communication related to "death" notification concerning that person is from that person's behaviour to the community behaviour via the appropriate person_to_community, PtC(pi) channel.

value
76. pids:PI-**set**

75. people_system: **Unit** → **Unit**
75. people_system() ≡
76. community()
77. ‖ ‖{nascent(pi)|pi:PI•pi ∈ pids}

channel
80. {CtN(pi)|pi:PI•pi ∈ pids}: mkBirth(pi:PI,p:P)
81. {PtC(pi)|pi:PI•pi ∈ pids}: mkDeceased(pi:PI,"deceased")

73

for giving semantics to the kind of formal specification language which handles the dynamic creation and demise of processes.

A Community Behaviour

82. The community behaviour refers to a global (constant) value of the set of unique person identifiers — of unborn, living or "deceased" persons.

83. We distinguish between two distinct sets of events:

    a persons being born (a singleton event) and

    b persons passing away (a singleton event).

84. A birth gives rise to a person, p, being communicated to its identified (obs_PI(p)) nascent behaviour.

85. A person behaviour informs the community behaviour of the decease of that person.

74

```
variable
    lps:P-set := {} [ living persons ]
value
82. community: Unit →
82.     out {CtN[i]|i:PI•i ∈ pids}
82.     in {PtC[i]|i:PI•i ∈ pids}  Unit
82. community() ≡
84.     (let p:P•p ∉ lps ∧ obs_PI(p) ∈ pids in
84.      (lps := lps ∪ {p} ‖ CtN(obs_PI(p))!mkBirth(obs_PI(p),p)) end
84.      community())
82.     ⌈⌉
85.     (let m = ⌈⌉{PtC(pi)?|pi:PI•pi ∈ pids} in
85.      assert: ∃ pi:PI•m = mkDeceased("deceased",pi) ;
85.      let mkDeceased("deceased",pi) = m in
85.      let p:P • p ∈ lps ∧ obs_PI(p)=pi in
85.      lps := lps \ {p} end end end
85.      community())
```

75

A Nascent Behaviour

86. A nascent behaviour

87. awaits a "birth" notification (in the form of a person identifier and a person) from the community behaviour and

88. becomes an appropriate person behaviour.

```
value
86. nascent: pi:PI → in CtN(pi) out ... Unit
86. nascent(pi) ≡
87.     let m = CtN(pi) ? in
88.     if m=mkMfgd(pi,p)
88.         then let mkBirth(pi,p) = m in person(pi)(p) end
88.         else chaos end end
```

76

A Person Behaviour

89. The person behaviour has as state-component the atomic simple person entity.

90. We distinguish between four distinct sets of pairs of events and actions:

a death;
b buying and
c selling;
d driver on and
e driver off; and
f passenger on and
g passenger off.

77

```
type
90. PAoE == death|buy|sell|start|stop|enter|leave
value
89. person: pi:PI × P → in ... out PtPs(pi) ... Unit
90. person(pi)(p) ≡
90.     let a = death⌈|buy⌈|sell⌈|start⌈|stop⌈|enter⌈|leave in
90.     let p′ = case a of
90a.                 death     → ″deceased″,
90b.                 buy       → buy_act(p),      90c. sell      → sell_act(p),
90d.                 driv_on   → driv_on_act(p),  90e. driv_off  → driv_off_act(p),
90f.                 pass_on   → pass__act(p)     90g. pass_off  → pass_off_act(p)
89.                end in
89.     if p′=″deceased″
89.         then PtoPs(pi)!mkDeceased(″deceased″) ; stop
89.         else person(pi)(p′)
89.     assert: pi=obs_PI(p)=obs_PI(p′) end end end
```

2.4.2 Fleet and Vehicle Behaviours 78

We describe the concepts of a fleet of a dynamically varying number of vehicles and individual vehicles using identical modelling techniques as those used for the description of a community of persons.

We shall therefore restart the numbering of the narrative and formalised items below as from Item 75 on page 23. The reader can then "verify" that the two models, that of a community of persons and that of a fleet of vehicles have rather identical behavioural
79 structures.

A Vehicle System Behaviour

75. The concurrent constellation of one fleet (of vehicles) and an indefinite number of pairs of latent and vehicle behaviours will be referred to as the vehicle_system behaviour.

76. The fleet behaviour refers to a global constant value, vids: an indefinite set of the unique identifiers of *latent*, actual and `"scrapped"` vehicles.

77. Each individual of the indefinite number of latent behaviours is initialised with its (future) unique vehicle identity.

78. The fleet behaviour models the manufacturing of vehicles and kicks off the identified latent behaviour by communicating a properly identified vehicle to that latent behaviour.

79. The identity of of a "scrapped" vehicle behaviour is communicated to the fleet be-
80 haviour.

80. The communications mentioned in Items 78–79 are modelled by CSP output/inputs over a set of unique vehicle identified fleet_to_latent vehicle channels, FtL(vi).

81. Once a latent vehicle behaviour "comes alive" (i.e., a vehicle has been manufactured and is operating), communication related to `"scrap"` notification concerning that vehicle is from that vehicle's behaviour to the fleet behaviour via the appropriate vehicle_to_fleet, VtF(pi) channel.

value
76. vids:VI-**set**

75. vehicle_system: **Unit** → **Unit**
75. vehicle_system() ≡
76. fleet(vids)
77. ‖ ‖{latent(vi)|vi:VI•vi ∈ vids}

channel
80. {FtL(pi)|vi:VI•vi ∈ vids}: mkMfgd(vi:VI,v:V)
81. {VtF(pi)|vi:VI•vi ∈ vids}: mkScrapped(vi:VI,″scrapped″)

81

A Vehicle Fleet Behaviour

82. The fleet behaviour refers to a global (constant) value, vids. the set of unique vehicle identifiers — of yet to be manufactured, manufactured and scrapped vehicles.

83. We distinguish between two distinct sets of events:

    a vehicles being manufactured (a singleton event) and

    b vehicles being scrapped (a singleton event).

84. Vehicle manufacturing gives rise to a vehicle, v, being communicated to its identified (obs_VI(v)) latent behaviour.

85. A vehicle behaviour informs the fleet behaviour of the scrapping of that vehicle.

82

variable
avs:V-**set** := {} [active or scrapped vehicles]
value
82. fleet: **Unit** →
82. **out** {FtL[vi]|vi:VI•i ∈ vids}
82. **in** {CtF[vi]|vi:VI•i ∈ vids} **Unit**
82. fleet() ≡
84. (**let** v:V•v ∉ avs ∧ obs_VI(v) ∈ vids **in**
84. (avs := avs ∪ {v} ‖ FtL(obs_VI(v))!mkMfgd(obs_VI(v),v)) **end**
84. fleet())
82. ⌈⌉
85. (**let** m = ⌈⌉{VtF(vi)?|vi:VI•vi ∈ vids} **in**
85. **assert:**∃ vi:VI • m = mkScrapped(vi,″scrapped″) ;
85. **let** mkScrapped(vi,″scrapped″) = m **in**
85. **let** v:V • v ∈ avs ∧ obs_VI(v)=vi **in**
85. avs := avs \ {v} **end end end**
85. fleet())

83

A Latent Behaviour

86. A latent behaviour
87. awaits a manufactured notification (including a vehicle) from the fleet behaviour and
88. becomes an appropriate vehicle behaviour.

```
value
86. latent: vi:VI → in VtL(vi) out ... Unit
86. latent(vi) ≡
87.     let m = PstN(vi) ? in
88.     if m=mkMfgd("manufactured",v) assert: vi=obs_VI(v)
88.         then let mkMfgd(_,v) = m in vehicle(vi)(v) end
88.         else chaos end end
```

84

A Vehicle Behaviour

89. The vehicle behaviour has as state-component the atomic simple vehicle entity.
90. We distinguish between one event and four distinct sets of pairs or triples of actions:

a scrap (event);	f passenger on,
b buying	g and passenger off;
c and selling;	h and entering the net,
d driver on	i driving on the net,
e and driver off;	j and leaving the net.

85

```
type
90. VAoE == scrap|buy|sell|driv_on|driv_off|pass_on||pass_off|enter|drive|leave
value
89. vehicle: vi:VI → V → in ... out VtF(pi) ... Unit
90. vehicle(vi)(v) ≡
90.     let a = scrap⌈|buy⌈|sell⌈|driv_on⌈|driv_off⌈|pass_on⌈|pass_off⌈|enter⌈|drive⌈|leave in
90.     let v′ = case a of
90a.                scrap    → "scrapped",
90b.                buy      → buy_act(v),      90c.   sell      → sell_act(v),
90d.                driv_on  → driv_on_act(v),  90e.   driv_off  → driv_off_act(v),
90f.                pass_on  → pass_on_act(v),  90g.   pass_off  → pass_off_act(v),
90h.                enter    → enter_act(v),    90i.   drive     → drive_act(v),
90j.                leave    → leave_act(v),
```

89. **end in**
89. **if** v′=″scrapped″
89. **then** VtF(vi) ! mkScrapped(vi,″scrapped″) ; **stop**
89. **else** vehicle(vi)(v′)
89. **assert:** vi=obs_VI(v)=obs_VI(v′) **end end end**

2.5 Discussion of Domain Engineering 86

We have just touched a few issues of a methodology for domain engineering. Thus we have not dealt with principles and techniques of describing domain facets: intrinsics, support technologies, rules and regulations, scripts, management and organisation, and human behaviour. Each of these, and other methodological topics have an own set of principles and techniques and an emerging underlying theory. One will be touched upon in tomorrow's 10:30 am colloquium.

3 Requirements Engineering 87

3.1 Preliminaries

3.1.1 The Machine = Hardware + Software

By 'the machine' we shall understand the software to be developed and hardware (equipment + base software) to be configured for the domain application.

3.1.2 Requirements Prescription 88

The core part of the requirements engineering of a computing application is the requirements prescription. A requirements prescription tells us which parts of the domain are to be supported by 'the machine'. A requirements is to satisfy some goals. Usually the goals cannot be prescribed in such a manner that they can served directly as a basis for software design. Instead we derive the requirements from the domain descriptions and then argue (incl. prove) that the goals satisfy the requirements. In this paper we shall not show the latter but shall show the former.

3.1.3 A Suitable Decomposition of the Requirements Prescription 89

We consider three forms of requirements prescription: the domain requirements, the interface requirements and the machine requirements. Recall that the machine is the hardware and software (to be required). Domain requirements are those whose technical terms are from the domain only. Machine requirements are those whose technical terms are from the machine only. Interface requirements are those whose technical terms are from both.

3.1.4 An Aside on Our Example 90

We shall continue our "ongoing" example. Our requirements is for a toll-road system. The goals of having a toll-road system are: to decrease transport times between selected hubs of a general net; and to decrease traffic accidents and fatalities while moving on the
91 toll-road net as compared to comparable movements on the general net. The toll-road net, however, must be paid for by its users. Therefore toll-road net entries and exits occur at toll-road plazas with these plazas containing entry and exit toll-booths where tickets can be issued, respectively collected and travel paid for. We shall very briefly touch upon these toll-booths, in the Extension part (as from Page 34) of the next section, Sect. 3.3. So all the other parts of the next section (Sect. 3.3) serve to build up to the Extension part.

3.2 Business Process Re-engineering (BPR) 92

Before embarking on the detailed elaboration of requirements it is advised that a thorough, rough-sketching of the re-engineering of the business processes take place.

A toll-road system is a special net consisting of a linear sequence of toll-road links separated by toll-road hubs. Vehicles gain access to these hubs and links by entering (and leaving) the toll-road net at toll plazas, through entry (respectively exit) booths connected to the toll-road hubs by plaza to toll-road hub hubs. Vehicles collect tickets upon entering the toll-road net. Vehicles move around the toll-road hubs and links. And vehicles return tickets and pay for using the toll-road net upon leaving that net.

3.3 Domain Requirements 93

Domain requirements cover all those aspects of the domain — simple entities, actions,
94 events and behaviours — which are to be supported by 'the machine'. Thus domain requirements are developed by systematically "revising" cum "editing" the domain description: which parts are to be **projected:** left in or out; which general descriptions are to be **instantiated** into more specific ones; which non-deterministic properties are to be made more **determinate**; and which parts are to be **extended** with such computable domain
95 description parts which are not feasible without IT.

Projection, instantiation, determination and extension are the basic engineering tasks of domain requirements engineering. An example may best illustrate what is at stake. The example is that of a toll-way system — in contrast to the general nets covered by description Items 1a–22 (Pages 3–9). See Fig. 1.

The links of the general net of Fig. 1 are all two-way links, so are the plaza-to-toll-way links of the toll-way net of Fig. 1. The toll-way links are all one-way links. The hubs of the general net of Fig. 1 are assumed to all allow traffic to move in from any link and onto any link. The plaza hubs do not show links to "an outside" — but they are assumed. Vehicles enter the toll-way system from the outside and leave to the outside. The toll-way hubs allow traffic to move in from the plaza-to-toll-way link and back onto that or onto the one

or two toll-way links emanating from that hub, as well as from toll-way links incident upon that hub onto toll-way links emanating from that hub or onto the toll-way-to-plaza link. 96

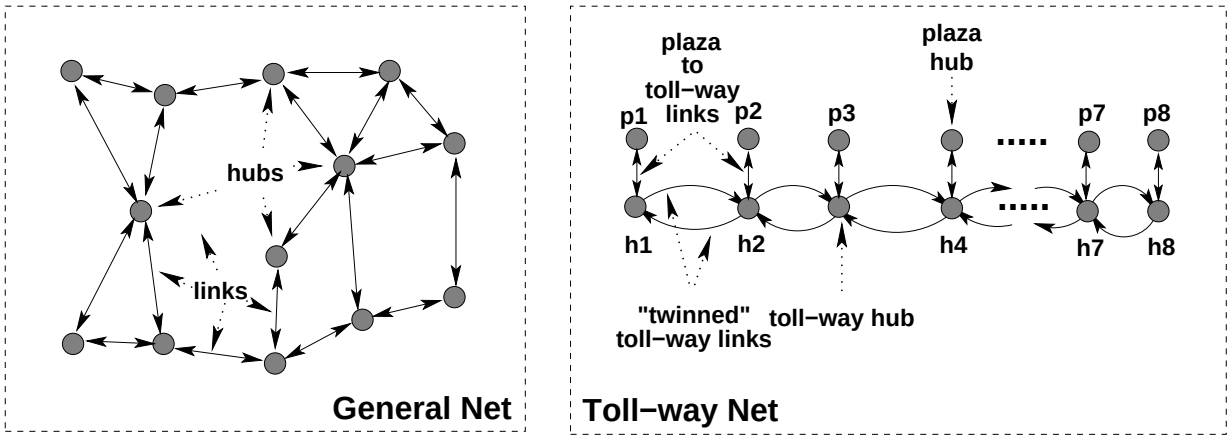

Figure 1: General and Toll-way Nets

3.3.1 Projection 97

We keep what is needed to prescribe the toll-road system and leave out the rest.

91. We keep the description, narrative and formalisation,
    a nets, hubs, links,
    b hub and link identifiers,
    c hub and link states,
92. as well as related observer functions.

type
91a. N, H, L
91b. HI, LI
91c. HΣ, LΣ
value
92. obs_Hs,obs_Ls,obs_HI,obs_LI,
92. obs_HIs,obs_LIs,obs_HΣ,obs_L Σ

3.3.2 Instantiation 98

99

From the general net model of earlier formalisations we instantiate the toll-way net model now described.

93. The net is now concretely modelled as a pair of sequences.
94. One sequence models the plaza hubs, their plaza-to-toll-way link and the connected toll-way hub.
95. The other sequence models the pairs of "twinned" toll-way links.
96. From plaza hubs one can observe their hubs and the identifiers of these hubs.
97. The former sequence is of m such plaza "complexes" where $m \geq 2$; the latter sequence is of $m-1$ "twinned" links.
98. From a toll-way net one can abstract a proper net.

99. One can show that the posited abstraction function yields well-formed nets, i.e., nets which satisfy previously stated axioms.

```
type
93. TWN = PC* × TL*
94. PC = PH × L × H
95. TL = L × L
value
94. obs_H: PH → H, obs_HI: PH → HI
axiom
97. ∀ (pcl,tll):TWN •
97.    2≤len pcl∧len pcl=len tll+1
value
98. abs_N: TWN → N
98. abs_N(pcl,tll) as n
98.   pre: wf_TWN(pcl,tll)
98.   post:
98.     obs_Hs(n) =
98.       {h,h′|(h,_,h′):PC•(h,_,h′)∈ elems pcl} ∧
98.     obs_Ls(n) =
98.       {l|(_,l,_):PC•(_,l,_)∈ elems pcl} ∪
98.       {l,l′|(l,l′):TL•(l,l′)∈ elems tll}

theorem:
99. ∀ twn:TWN • wf_TWN(twn) ⇒ wf_N(abs_N(twn))
```

Model Well-formedness wrt. Instantiation Instantiation restricts general nets to toll-way nets. Well-formedness deals with proper mereology: that observed identifier references are proper. The well-formedness of instantiation of the toll-way system model can be defined as follows:

100. The i'plaza complex, (p_i, l_i, h_i), is instantiation-well-formed if
    a link l_i identifies hubs p_i and h_i, and
    b hub p_i and hub h_i both identifies link l_i; and if
101. the i'th pair of twinned links, tl_i, tl'_i,
    a has these links identify the toll-way hubs of the i'th and i+1'st plaza complexes $((p_i, l_i, h_i)$ respectively $(p_{i+1}, l_{i+1}, h_{i_1}))$.

```
value
   Instantiation_wf_TWN: TWN → Bool
   Instantiation_wf_TWN(pcl,tll) ≡
100.   ∀ i:Nat • i ∈ inds pcl⇒
100.     let (pi,li,hi)=pcl(i) in
100a.      obs_LIs(li)={obs_HI(pi),obs_HI(hi)}
100b.    ∧ obs_LI(li)∈ obs_LIs(pi)∩ obs_LIs(hi)
101.   ∧ let (li′,li″) = tll(i) in
101.       i < len pcl ⇒
101.         let (pi′,li‴,hi′) = pcl(i+1) in
101a.          obs_HIs(li) = obs_HIs(li′) = {obs_HI(hi),obs_HI(hi′)}
       end end end
```

102

103

3.3.3 Determination

101 Determination, in this example, fixes states of hubs and links. The state sets contain only one set. Twinned toll-way links allow traffic only in opposite directions. Plaza to toll-way hubs allow traffic in both directions. Toll-way hubs allow traffic to flow freely from plaza to toll-way links and from incoming toll-way links to outgoing toll-way links and toll-way to plaza links.

We omit formalisation. The determination-well-formedness of the toll-way system
104 model can be defined as follows[6]:

[6] i ranges over the length of the sequences of twinned toll-way links, that is, one less than the length of the sequences of plaza complexes. This "discrepancy" is reflected in out having to basically repeat formalisation of both Items 103a and 103b.

Model Well-formedness wrt. Determination We need define well-formedness wrt. determination. Please study Fig. 2.

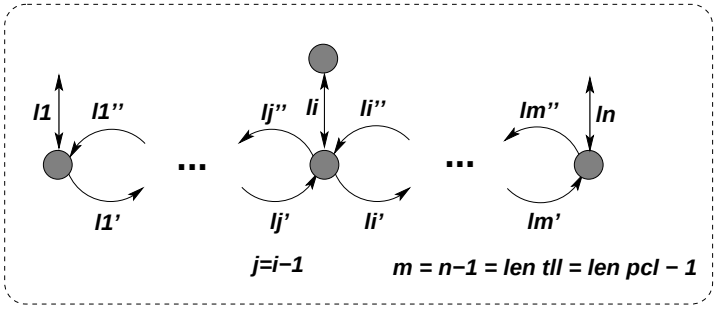

Figure 2: Hubs and Links

105

102. All hub and link state spaces contain just one hub, respectively link state.
103. The i'th plaza complex, pcl(i):(p_i, l_i, h_i) is determination-well-formed if
    a l_i is open for traffic in both directions and
    b p_i allows traffic from h_i to "revert"; and if
104. the i'th pair of twinned links (li', li'') (in the context of the i+1st plaza complex, pcl(i+1):$(p_{i+1}, l_{i+1}, h_{i+1})$) are determination-well-formed if
    a link l'_i is open only from h_i to h_{i+1} and
    b link l''_i is open only from h_{i+1} to h_i; and if
105. the jth toll-way hub, h_j (for $1 \leq j \leq \mathbf{len}$ pcl) is determination-well-formed if, depending on whether j is the first, or the last, or any "in-between" plaza complex positions,
    a [the first:] hub $i = 1$ allows traffic in from l_1 and l''_1, and onto l_1 and l'_1.
    b [the last:] hub $j = i + 1 = \mathbf{len}$ pcl allows traffic in from $l_{\mathbf{len}\,\mathsf{tll}}$ and $l''_{\mathbf{len}\,\mathsf{tll}-1}$, and onto $l_{\mathbf{len}\,\mathsf{tll}}$ and $l'_{\mathbf{len}\,\mathsf{tll}-1}$.
    c [in-between:] hub $j = i$ allows traffic in from l_i, l''_i and l'_i and onto l_i, l'_{i-1} and l''_i.

106

```
value
103.  Determination_wf_TWN: TWN → Bool
103.  Determination_wf_TWN(pcl,tll) ≡
103.    ∀ i:Nat• i ∈ inds tll ⇒
103.      let (pi,li,hi) = pcl(i),
103.          (npi,nli,nhi) = pcl(i+1), in
103.          (li′,li″) = tll(i) in
102.        obs_HΩ(pi)={obs_HΣ(pi)}∧obs_HΩ(hi)={obs_HΣ(hi)}
102.      ∧ obs_LΩ(li)={obs_LΣ(li)}∧obs_LΩ(li′)={obs_LΣ(li′)}
102.      ∧ obs_LΩ(li″)={obs_LΣ(li″)}
103a.     ∧ obs_LΣ(li)
103a.         = {(obs_HI(pi),obs_HI(hi)),(obs_HI(hi),obs_HI(pi))}
103a.     ∧ obs_LΣ(nli)
103a.         = {(obs_HI(npi),obs_HI(nhi)),(obs_HI(nhi),obs_HI(npi))}
103b.     ∧ {(obs_LI(li),obs_LI(li))}⊆obs_HΣ(pi)
103b.     ∧ {(obs_LI(nli),obs_LI(nli))}⊆obs_HΣ(npi)
104a.     ∧ obs_LΣ(li′)={(obs_HI(hi),obs_HI(nhi))}
104b.     ∧ obs_LΣ(li″)={(obs_HI(nhi),obs_HI(hi))}
105.      ∧ case i+1 of
105a.       2 → obs_HΣ(h_1)=
105a.             {(obs_LΣ(l_1),obs_LΣ(l_1)),
105a.             (obs_LΣ(l_1),obs_LΣ(l_1″)),
105a.             (obs_LΣ(l″_1),obs_LΣ(l_1)),
105a.             (obs_LΣ(l″_1),obs_LΣ(l′_1))},
105b.       len pcl → obs_HΣ(h_i+1)=
105b.             {(obs_LΣ(l_len pcl),obs_LΣ(l_len pcl)),
105b.             (obs_LΣ(l_len pcl),obs_LΣ(l′_len tll)),
105b.             (obs_LΣ(l″_len tll),obs_LΣ(l_len pcl)),
105b.             (obs_LΣ(l″_len tll),obs_LΣ(l′_len tll))},
105c.       _ → obs_HΣ(h_i)=
105c.             {(obs_LΣ(l_i),obs_LΣ(l_i)),
105c.             (obs_LΣ(l_i),obs_LΣ(l′_i)),
105c.             (obs_LΣ(l_i),obs_LΣ(l″_i−1)),
105c.             (obs_LΣ(l″_i),obs_LΣ(l′_i)),
105c.             (obs_LΣ(l″_i),obs_LΣ(l′_i−1)),
105c.             (obs_LΣ(l″_i),obs_LΣ(l′_i))}
103.  end end
```

3.3.4 Extension

107

For our example we choose to consider the toll plazas. A toll plaza, in addition to its hub,
108 also contains vehicle entry and exit booths. We refer to Fig. 3.

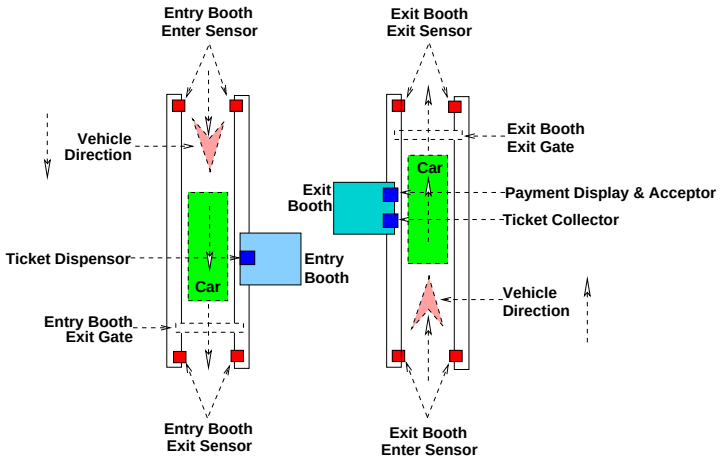

Figure 3: Entry and Exit Toll Boths

109

The following is a prolonged example. It contains three kinds of formalisations: a `RAISE/CSP` model, a `Duration Calculus` model [8, 5] and a `Timed Automata` model [1, 5].

110 **A `RAISE/CSP` Model:** Without much ado:

106. A toll plaza consists of a one pair of an entry booth and and entry gate and one pair of an exit booth and an exit gate.

107. Entry booths consist of an entry sensor, a ticket dispenser and an exit sensor.

108. Exit booths consist of an entry sensor, a ticket collector, a payment display and a payment component.

type
106. PZ = (EB×G) × (XB×G)
107. EB = ...
108. XB = ...

111 **Cars:** We summarize an earlier model of vehicles:

109. There are vehicles.

110. Vehicles have unique vehicle identifications.

type
109. V
110. VId
value
110. obs_VId: V → VId
axiom
110. ∀ v,v′:V • v≠v′ ⇒ obs_VId(v) ≠ obs_VId(v′)

Entry Booths: The description now given is an idealisation. It assumes that everything works: that the vehicles behave as expected and that the electro-mechanics of booths and gates do likewise. 112

111. An entry_sensor registers whether a car is entering the entry booth or not,
    a. that is, for the duration of the car passing the entry_sensor that sensor senses the car identification cid
    b. otherwise it senses "nothing".

113

112. A ticket_dispenser
    a. either holds a ticket or does not hold a ticket, i.e., no_ticket;
    b. normally it does not hold a ticket;
    c. the ticket_dispenser holds a ticket soon after a car has passed the entry_sensor;
    d. the passing car collects the ticket –
    e. after which the ticket_dispenser no longer holds a ticket.

113. An exit_sensor
    a. registers the identification of a car leaving the toll booth
    b. otherwise it senses "nothing".

— ***Gates:*** As part of entry/exit booths: 114

114. A gate
    a. is either closed or open;
    b. it is normally closed;
    c. if a car is entering it is secured set to close (as a security measure);
    d. once a car has collected a ticket it is set to open;
    e. and once a car has passed the exit_sensor it is again set to close.

— ***A Simple Formalisation:***

115

```
type
  C, CI
  G = open | close
  TK == Ticket | no_ticket
value
  obs_CI: (C|Ticket) → CI
channel
  entry_sensor:CI
  ticket_dispenser:Ticket
  exit_sensor:CI
  gate_ch:G
value
  vs:V-set
  eb:EB,xb:XB,eg,xg:G
```

116

```
value
  eg,xg:G, eb:EB, xb:XB, vs:V-set

  system: G×EV×V-set×XB×G → Unit
  system(eg,eb,vs,xb,xg) ≡
      entry_gate(eg)
      ‖ entry_booth(eb)
      ‖ ‖{car(obs_CId(c),c)|ci:C,v:C•c ∈ cs}
      ‖ exit_booth(xb)
      ‖ exit_gate(xg)
```

117

```
  car: CI × C → out entry_sensor,exit_sensor
                in ticket_dispenser  Unit
  car(ci,c) ≡
      entry_sensor ! ci ;
      let ticket = ticket_dispenser ? assert: ticket ≠ no_ticket in
      ticket_dispenser ! no_ticket ;
      exit_sensor ! ci ;
      car(add(ticket,c)) end
```

118

```
  entry_booth: Unit → in entry_sensor, exit_sensor
                      out ticket_dispenser
                      out gate_ch  Unit
```

```
entry_booth(b) ≡
    gate_ch ! close ;
    let ci = entry_sensor ? in
    gate_ch ! open ;
    ticket_dispenser ! make_ticket(cid) ;
    let res = ticket_dispenser ? assert: res = no_ticket ;
    let ci′ = exit_sensor ? assert: ci′ = ci ;
    gate_ch ! close ;
    entry_booth(add_ticket(ticket,b)) end end end
```

119

```
entry_gate: G → in gate  Unit
entry_gate(g) ≡
    case gate_ch ? of
        close → exit_gate(close) assert: g = open,
        open → exit_gate(open) assert: g = close
    end
```

120

```
add_ticket: Ticket × C ~→ C
    pre add_Ticket(t,c): ∼has_Ticket(c)
    post: add_Ticket(t,c): has_Ticket(c)
has_ticket: (C|B) → Bool
obs_ticket: (C|B) ~→ Ticket
    pre obs_ticket(cb): has_Ticket(cb)
rem_ticket: (C ~→ C) | (B ~→ B)
    pre rem_ticket(cb): has_Ticket(cb)
    post rem_ticket(cb): ∼has_Ticket(cb)
```

In the next section, "A `Duration Calculus` Model" we shall start refining the descriptions given above. We do so in order to handle failures of vehicles to behave as expected and of the electro-mechanics of booths and gates.

A `Duration Calculus` Model: For DC we refer to [8, 5]. 121
We abstract the channels of the `RAISE/CSP` model to now be Boolean-valued variables.

```
type
    ES = Bool [true=passing, false=not_passing]
    TD = Bool [true=ticket, false=no_ticket]
    G  = Bool [true=open, false=closing⌈⌉closed⌈⌉opening]
    XS = Bool [true=car_has_just_passed, false=car_passing⌈⌉no-one_passing]
variable
    entry_sensor:ES := false ;
```

ticket_dispenser:TD := **false** ;
gate:G := **false** ;
exit_sensor:XS := **false** ;

122

115. No matter its position, the gate must be closed within no more than δ_{eg} time units after the entry_sensor has registered that a car is entering the toll booth.

116. A ticket must be in the ticket_dispenser within δ_{et} time units after the entry_sensor has registered that a car is entering the toll booth.

117. The ticket is in the ticket_dispenser at most δ_{tdc} time units

118. The gate must be open within δ_{go} time units after a ticket has been collected.

119. The exit sensor is registering (i.e., is on) the identification of exiting cars and is not registering anything when no car is passing (i.e., is off).

115. $\sim(\lceil\text{entry\_sensor}\rceil\ ;\ (\ell = \delta_{eg} \wedge \lceil\text{gate}\rceil))$
116. $\sim(\lceil\text{entry\_sensor}\rceil\ ;\ (\ell = \delta_{et} \wedge \lceil\sim\text{ticket\_dispenser}\rceil))$
117. $\Box(\lceil\sim\text{ticket\_dispenser}\rceil \Rightarrow \ell < \delta_{tdc})$
118. $\sim(\lceil\text{ticket\_dispenser}\rceil\ ;\ (\lceil\sim\text{ticket\_dispenser} \wedge \sim\text{gate}\rceil \wedge \ell \geq \delta_{go}))$
119. $\Box(\lceil\text{gate=closing}\rceil \Rightarrow \lceil\sim\ \text{exit\_sensor}\rceil)$

123 **A `Timed Automata` Model:** A timed automaton [1, 5] for a configuration of an entry gate, its entry booth and a car is shown in Fig. 4 on the facing page. Figure 5 on page 40 shows the a car, an exit booth and its exit gate interactions. They are more-or-less "derived" from the example of Sect. 7.5 of [1, Alur & Dill, 1994] (Pages 42–45). The right half of the car timed automaton of Fig. 4 on the facing page is to be thought of as the same as the left half of the car timed automaton of Fig. 5 on page 40, cf. the vertical dotted (⋮) line.

124

3.4 Interface and Machine Requirements 125

Interface requirements take into consideration both the domain description and the machine: the hardware + base systems software upon which the software to be designed is to be implemented. So interface requirements are not exclusively "derived" from the narrated and formalised domain description. And the machine requirements make hardly any concrete reference to the domain description.

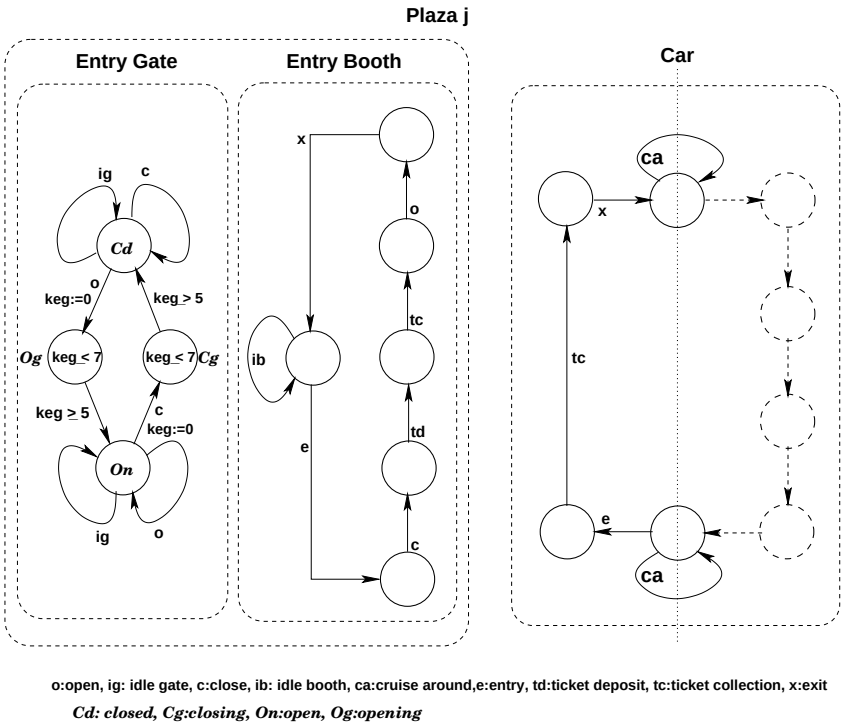

Figure 4: A `timed automata` model of gate, entry booth and car interactions

3.5 Discussion of Requirements Engineering 126

As was the case for our coverage of domain engineering, there is more to requirements engineering than shown in this talk !

4 Software Design 127

We shall likewise omit any serious coverage of the software design process — except for these remarks: (1) As the domain description serves as a model for the development of the requirements development, (2) so do the requirements prescription serve as a model for software design; that is, (3) our whole software development is model-oriented.

5 Concluding Remarks 128

We have over-viewed the TRIPTYCH approach to software development: core aspects of domain engineering and core aspects of requirements. The conclusions that one may be able to draw from the example — or at least a reasonable small number of such examples — are that domains can be described, informally as well as formally; that domain requirements can be systematically (but not automatically) "derived" from domain descriptions; and

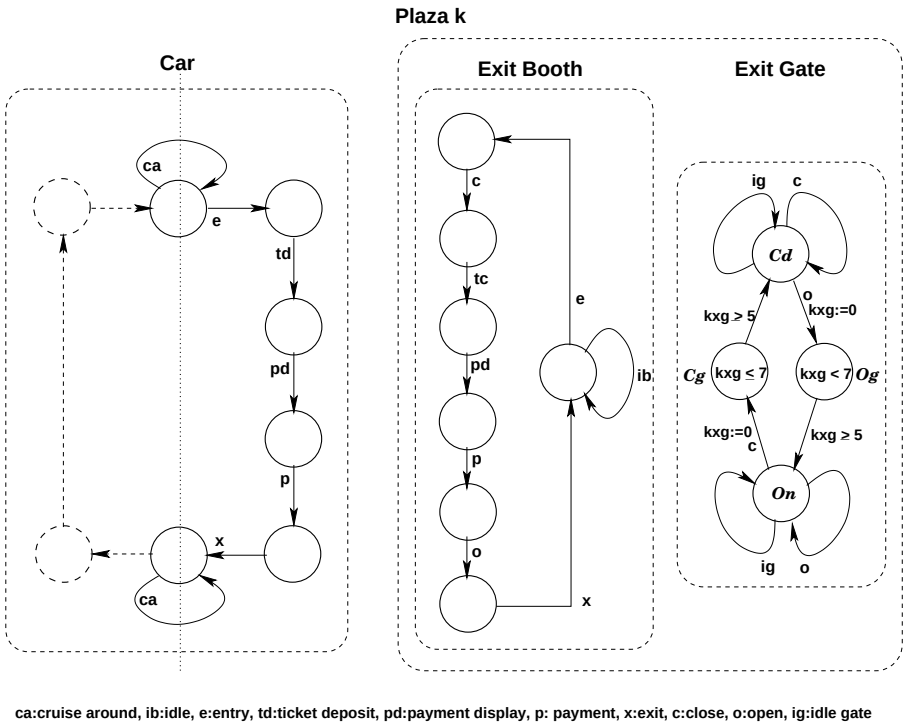

Figure 5: A `timed automata` model of car, exit booth and gate interactions

that **this approach puts requirements engineering in a rather new light.**

5.1 Domain Models as a Prerequisite for RE 129

Domains are seldomly computable, requirements must always be. It has been suggested — and strongly so — that requirements engineering be based on a domain description covering at least the "area of requirements interest". We have not covered 'Interface Requirements', those aspects of requirements which can be expressed using terms from both the domain and the machine — there shared entities, but, again, a concise domain model would help significantly we claim. And we have also not covered 'Machine Requirements', those aspects of requirements which can be expressed using terms from just the machine, so here domain models do not contribute much. So I suggest that we revise research into and practice of RE.

5.2 Oh Yes, Conventional RE Contains Elements of DE 130

Indeed, most RE texts contain repeated references to the necessity of considering "the domain". But these "necessity references" do not require that the requirements engineer separately model the domain, do not really expect the requirements engineer to go well beyond the scope and span of the requirements when considering the domain, and do

not formally relate domains and requirements. Here, we are strongly suggesting that domains be understood, be described (informally and formally) independent of requirements considerations.

5.3 Domain Engineering as a Free-standing Activity 131

Aim is to just understand a domain: *"What is a container line industry" "What is a railway system" "What is a hospital" "What is a financial service industry"* Just like a physicist try understand *"the big bang"*, an economist try understand *a country's national debt process*, a biologist try understand some *aspects of evolution*. To create a domain model, to study it and make it ready for general use make take 10-20 years. It took physicists many years before their theory of matter could be applied. But that is no reason for not doing domain engineering and science.

5.4 Domain Theories 132

By a domain theory we shall understand a theory about the model of the domain as described. A domain description is a foundation for a theory. The proof system of the formal specification language in which the domain description is expressed is another foundation. Theorems derived from these two foundations contribute to the theory. An examples of 133
a domain theorem for railways could be: Assuming that train traffic is on time wrt. a train timetable we can expect the following to hold: given that a train timetable is modulo some time interval, then the # of trains arriving at a station minus the # of trains ending their journey at that station plus the # of trains starting their journey at that station equals the # of trains leaving that station.

Domain models should be aimed at establishing domain theories.

5.5 Domain Science 134

By domain science we understand the theoretical foundation specific to the engineering of domain descriptions. Examples of issues of domain science are: (i) a theory of a calculus of domain description constructors such as illustrated in tomorrow's speculative talk; (ii) a theory of mereology models, cf. my April 2009 [TonyHoare75thBirthday] paper: for every $\mathcal{M}$ereology there is "a correspondning CSP$_\mathcal{M}$ expression", and for every CSP$_\mathcal{M}$ expression there is "a corresponding $\mathcal{M}$ereology". (iii) etcetera. 135

5.6 References

[1] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994. (Preliminary versions appeared in Proc. 17th ICALP, LNCS 443, 1990, and Real Time: Theory in Practice, LNCS 600, 1991).

[2] W. D. Blizard. A Formal Theory of Objects, Space and Time. *The Journal of Symbolic Logic*, 55(1):74–89, March 1990.

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

[4] J. M. E. McTaggart. The Unreality of Time. *Mind*, 18(68):457–84, October 1908. New Series. See also: [6].

[5] E.-R. Olderog and H. Dierks. *Real-Time Systems: Formal Specification and Automatic Verification*. Cambridge University Press, UK, 2008.

[6] R. L. Poidevin and M. MacBeath, editors. *The Philosophy of Time*. Oxford University Press, 1993.

[7] J. van Benthem. *The Logic of Time*, volume 156 of *Synthese Library: Studies in Epistemology, Logic, Methhodology, and Philosophy of Science (Editor: Jaakko Hintika)*. Kluwer Academic Publishers, P.O.Box 17, NL 3300 AA Dordrecht, The Netherlands, second edition, 1983, 1991.

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

