From Domains to Requirements*

Methodology Contributions to Domain Analysis and Requirements Engineering

Dines Bjørner
Fredsvej 11, DK-2840 Holte, Denmark
bjorner@gmail.com

July 15, 2010: 11:47

Abstract

In [9, Barry Boehm 81] two qualities of software were characterised (30+ years ago): *"the right software"* and *"software that is right"*. The former is software that offers its users exactly and only what they expect from that software. In this paper we shall take *"the right software"* to be software whose data and functions accurately reflect the are of their work (i.e., the application domain, or just *domain*). *"Software that is right"* is software which correctly implements its *requirements* (in the context of assumptions about the *domain*). Seminal works [20, 21, M.A. Jackson], [17, David Lorge Parnas] and [28, Axel van Lamsweerde] have stressed the importance of careful domain analysis in conjunction with similarly careful requirements analysis and prescription. In this paper we shall "isolate" domain description (incl. analysis) into a separate phase, which we shall call *domain engineering*[1], ideally preceding *requirements engineering*.. In the above cited works, where we especially acknowledge the influence, in our work, from [20, M.A. Jackson 1995], domain analysis appears to be tightly interwoven with requirements analysis. In this paper we shall separate the two and "pretend" that software can be developed in three "ideally" consecutive phases: *domain description development*[2] and *requirements prescription development*[3]. We shall not cover a third phase of software development: *software design*. We shall structure *domain modelling* into the composed modelling on *domain facets* such as *(a) intrinsics, (b) support technologies, (c) rules and regulations, (d) scripts (licenses and contracts), (e) management and organisation* and *(f) human behaviour*. We shall show that signaificant parts of requirements prescriptions can be systematically "derived" from a domain descruption, in particular the two parts of requirements that we shall call *domain requirements* and *interface requirements*. We shall structure *domain requirements modelling* into the staged modelling of such *domain requirements facets* as *(g) projection, (h) instantiation, (i) determination, (j) extension* and *(k) fitting*, as well as the staged modelling of such *interface requirements facets* as *(l) shared simple entities, (m) shared actions, (n) shared events* and *(o) shared behaviours*. We suggest, but can only point to empirical observations, that the systematic adherence to the items *(a)–(o)* contributes

*This is an evolving draft of an invited, to be refereed paper for a special JTCS issue in honour of NNN's KKKth Anniversary. I expect the draft to be completed by August 10, 2010. Currently I have started a complete rewite of the text. All examples will be renewed. **As of July 15, 2010 my rewite ends at the beginning of Sect. 1.4.**

[1]But domain engineering does not entail the construction, as in ordinary engineering, of a domain. It entails the construction of a domain description

[2]colloquially: domain engineering

[3]colloquially: requirements engineering

towards achieving *"the right software"*, while pursuing all phases using formal techniques (notably specification, verification) contributes towards achieving *"that the software is right"*.

Keywords:
Domain Engineering, Requirements Engineering, Formal Specification

Contents

1 Introduction **3**
1.1 What are Domains . . . 4
1.2 What is a Domain Description . . . 4
1.3 Description Languages . . . 5
1.4 Contributions of This Paper . . . 5
1.5 Relation to Other Engineering Disciplines . . . 5
1.6 Structure of Paper . . . 6
1.7 Examples and Formalisation . . . 6

2 A Specification Ontology **6**
2.1 Entities . . . 6
2.2 Actions . . . 8
2.3 Events . . . 9
2.4 Behaviours . . . 10

3 Domain Engineering **10**
3.1 Business Processes . . . 11
3.2 Intrinsics . . . 12
3.3 Support Technologies . . . 13
3.4 Rules and Regulations . . . 14
3.5 Scripts, Licenses and Contracts . . . 14
3.5.1 Scripts . . . 14
3.5.2 Licenses and Contracts . . . 15
3.6 Management and Organisation . . . 16
3.7 Human Behaviour . . . 17
3.8 Discussion . . . 18

4 Requirements Engineering **18**
4.1 Business Process Re-engineering . . . 18
4.2 Domain Requirements . . . 19
4.2.1 Projection . . . 20
4.2.2 Instantiation . . . 20
4.2.3 Determination . . . 20
4.2.4 Extension . . . 21
4.2.5 Fitting . . . 21
4.2.6 Discussion . . . 22
4.3 Interface Requirements . . . 22
4.3.1 Entity Interfaces . . . 22

4.3.2 Action Interfaces . . . 22
4.3.3 Event Interfaces . . . 22
4.3.4 Behaviour Interfaces . . . 23
4.3.5 Discussion . . . 23
4.4 Machine Requirements . . . 23

5 Discussion **23**
5.1 What Have We Achieved – and What Not . . . 23
5.2 What Have We Omitted . . . 24
5.3 Domain Engineering Can Be Pursued Just By Itself . . . 24
5.4 Domain Descriptions Are Not Normative . . . 24
5.5 "Requirements Always Change" . . . 24
5.6 What Can Be Described and Prescribed . . . 24
5.7 Relation to Other Works . . . 25
5.8 "Ideal" Versus Real Developments . . . 25
5.9 A Reference Model for Domains, Goals, Requirements and Software . . . 25
5.10 Domain Versus Ontology Engineering . . . 26

6 Conclusion **26**

7 Bibliographical Notes **26**

1 Introduction

Before we can design software we must have a robust understanding of its requirements. And before we can prescribe requirements we must have a robust understanding of the environment, or, as we shall call it, the domain in which the software is to serve – and as it is at the time such software is first being contemplated.

In consequence we suggest that software, "ideally"[4], be developed in three phases.

First a phase of **domain engineering.** In this phase a reasonably comprehensive description is constructed from an analysis of the domain. That description, as it evolves, is analysed with respect to inconsistencies, conflicts and completeness on one hand, and, on the other hand, in order to achieve pleasing concepts in terms of which to abstractly model the domain.

Then a phase of **requirements engineering.** This phase is strongly based, as we shall see (in Sect. 4), on an available, necessary and sufficient domain description. Guided by the domain and requirements engineers the *requirements stakeholders* point out which domain description parts are to be left (*projected*) out of the *domain requirements*, and of those left what forms of *instantiations, determinations* and *extensions* are required. Similarly the requirements stakeholders, guided by the domain and requirements engineers, inform as to which domain *entities, actions, events* and *behaviours* are *shared* between the domain and the *machine*, that is, the *hardware* and the *software* being required. In this paper we shall only very briefly cover aspects of *machine requirements*.

And finally a phase of **software design.** We shall not cover this phase in this paper.

[4] Section 5.8 will discuss practical renditions of "idealism"!

Methodology

The paper is a methodology paper – where a method is seen as a set of principles (applied by engineers, not machines) for selecting and applying (often with some tool support) techniques (and tools) for the efficient construction of some artifact – here software.

This paper shall be seen as an adjoint to current research in domain analysis, requirements techniques and specification. We think our techniques go well-in-hand with those of [20, 21, 17, 23].

• • •

We do not claim that our concept of domain engineering is new, only (I) that we have contributed with a check list (Items *(a)–(f)* as mentioned in the abstract) which is of help to the domain engineer; and (II) that the idea of some systematic form of "derivation" of parts of the requirements prescription from the domain description is reasonably new; we, in particular suggest that the domain description to requirements prescription "operations" (Items *(g)–(o)* as mentioned in the abstract) are new. But, again, much of the inspiration for these are due to [20, 21, 17, 23].

1.1 What are Domains

By a domain we shall here understand a universe of discourse, an area of nature subject to laws of physics and studies by physicists, or an area of human activity (subject to its interfaces with nature). There are other domains which we shall ignore. We shall focus on the human-made domains. "Large scale" examples are *the financial service industry: banking, insurance, securities trading, portfolio management, etc.*, *health care: hospitals, clinics, patients, medical staff, etc.*, *transportation:* road, rail/train, sea, and air transport (vehicles, transport nets, etc.); *oil and gas systems:* pumps, pipes, valves, refineries, distribution, etc. "Intermediate scale" examples are *automobiles:* manufacturing or monitoring and control, etc.; and *heating systems*. The above explication was "randomised": for some domains, to wit, *the financial service industry*, we mentioned major functionalities, for others, to wit, *health care*, we mentioned major entities. An objection can be raised, namely that the above characterisation – of what a domain is – is not sufficiently precise. We shall try, in the next section, to partially meet this objection.

1.2 What is a Domain Description

By a *domain description* we understand a description of the *entities*, the *actions*, the *events* and the *behaviours* of the domain, including its *interfaces* to other domains. A domain description describes the domain **as it is**. A domain description does not contain requirements let alone references to any software. Michael Jackson, in [20], refers to domain descriptions as *indicative* (stating objective fact), requirements prescriptions as *optative* (expressing wish or hope) and software specifications as *imperative* ("do it!"). A description is *syntax*. The meaning (*semantics*) of a domain description is usually a set of *domain models*. We shall take domain models to be *mathematical structures (theories)*. The form of domain descriptions that we shall advocate "come in pairs": precise, say, English, i.e., narrated text (narratives) alternates with clearly related formula text.

1.3 Description Languages

Besides using as precise a subset of a national language, as here English, as possible, and in enumerated expressions and statements, we "pair" such narrative elements with corresponding enumerated clauses of a formal specification language. We shall be using the RAISE Specification Language, RSL, [12, 13, 3], in our formal texts. But any of the model-oriented approaches and languages offered by Alloy [19], Event B [1], VDM [7, 8, 11] and Z [29], should work as well. No single one of the above-mentioned formal specification languages, however, suffices. Often one has to carefully combine the above with elements of Petri Nets [24], CSP: Communicating Sequential Processes [16], MSC: Message Sequence Charts [18], Statecharts [15], and some temporal logic, for example either DC: Duration Calculus [30] or TLA+ [22]. Research into how such diverse textual and diagrammatic languages can be meaningfully and proof-theoretically combined is ongoing [2].

1.4 Contributions of This Paper

We claim that the major contributions of the triptych approach to software engineering as presented in this paper are the following: (1) the clear *identification* of domain engineering, or, for some, its clear *separation* from requirements engineering (Sects. 3 and 4); (2) the *identification* and *'elaboration'* of the pragmatically determined domain *facets* of *(a) intrinsics, (b) support technologies, (c) rules and regulations, (d) scripts (licenses and contracts)*, (e) management and organisation, and *(f) human behaviour* whereby 'elaboration' we mean that we provide principles and techniques for the construction of these facet description parts (Sects. 3.2–3.7); (3) the *re-identification* and *'elaboration'* of the concept of *business process reengineering* (Sect. 4.1) on the basis of the notion of *business processes* as first introduced in Sect. 3.1; (4) the *identification* and *'elaboration'* of the technically determined *domain requirements facets* of *(g) projection, (h) instantiation, (i) determination, (j) extension* and *(k) fitting* requirements principles and techniques – and, in particular the *"discovery"* that these requirements engineering stages are strongly dependent on necessary and sufficient domain descriptions (Sects. 4.2.1–4.2.5); and (5) the *identification* and *'elaboration'* of the technically determined *interface requirements facets* of *(l) shared simple entity, (m) shared action, (n) shared event* and *(o) shared behaviour* requirements principles and techniques (Sects. 4.3.1–4.3.4). We claim that the facets of (2, 3, 4) and (5) are all relatively new. In Sect. 5 we shall discuss these contributions in relation to the works and contributions of other researchers and technologists.

1.5 Relation to Other Engineering Disciplines

An aeronautics engineer – to be hired by Boeing to their design team for a next generation aircraft – must be pretty well versed in applied mathematics and in aerodynamics. A radio communications engineer – to be hired by Ericsson to their design team for a next generation mobile telephony antennas – must be likewise pretty well versed in applied mathematics and in the physics of electromagnetic wave propagation in matter. And so forth. Software engineers hired for the development of software for hospitals, or for railways, know little, if anything, about health care, respectively rail transportation (schedulimg, rostering, etc.). The Ericsson radio communications engineer can be expected to understand Maxwell's Equations, and to base the design of antenna characteristics on the transformation and instantiation of these equations. It is therefore quite reasonable to expect the domain-specific software engineer

to understand formalisation of their domains, to wit: *railways:* `www.railwaydomain.org`, and *pipelines:* `pipelines.pdf`, *logistics:* `logistics.pdf`, *transport nets:* `comet1.pdf`, *stock exchanges:* `tse-2.pdf` and *container lines:* `container-paper.pdf` – these latter five at `www.-imm.dtu.dk/~db/`.

1.6 Structure of Paper

Before going into some details on domain enginering (Sect. 3) and requirements engineering (Sect. 4) we shall in the next section cover the basic concepts of specifications, whether domain descriptions or requirements prescriptions. These are: entities, actions, events and behaviours. Section 5 then discuses the contributions of the triptych approach as covered in this paper.

1.7 Examples and Formalisation

We bring 23 examples. These examples take up about 50% of the paper space. Most of these have both narrative, informal (English) text and formal texts. In principle all examples should have formal texts. But page space concerns dictated their absence.The reader, however, need not read the formalised parts of the examples ! They are expressed in the `RAISE` [13] Specification Language (`RSL` [12])

2 A Specification Ontology

In order to describe domains we postulate the following related specification components: *entities*, *actions*, *events* and *behaviours*.

2.1 Entities

By an entity we shall understand a phenomenon we can point to in the domain or a concept formed from such phenomena.

Example 1 ***Entities:*** The example is that of aspects of a transportation net. You may think of such a net as being either a road net, a rail net, a shipping net or an air traffic net. Hubs (or junctions) are then street intersections, train stations, harbours, respectively airports. Links are then street segments between immediately adjacent intersections, rail tracks between train stations, sea lanes between harbours, respectively air lanes between airports.

1. There are hubs and links.
2. There are nets, and a net consists of a set of two or more hubs and one or more links.
3. There are hub and link identifiers.
4. Each hub (and each link) has an own, unique hub (respectively link) identifier (which can be observed (ω) from the hub [respectively link]).

type
1. H, L,
2. N = H-**set** × L-**set**

axiom [nets−hubs−links−1]
2. ∀ (hs,ls):N • **card** hs≥2 ∧ **card** ls≥1
type
3. HI, LI
value
4. ωHI: H → HI, ωLI: L → LI
axiom [nets−hubs−links−2]
4. ∀ h,h′:H, l,l′:L • h≠h′ ⇒ ωHI(h)≠ωHI(h′) ∧ l≠l′⇒ωLI(l)≠ωLI(l′)

In order to model the physical (i.e., domain) fact that links are delimited by two hubs and that one or more links emanate from and are, at the same time, incident upon a hub we need to be able to express the following: that we can observe identifiers of hubs connected to a link from that link, and identifiers of links connected to a hub from that hub.

5. From any link of a net one can observe the two hubs to which the link is connected. We take this 'observing' to mean the following: from any link of a net one can observe the two distinct identifiers of these hubs.
6. From any hub of a net one can observe the identifiers of one or more links which are connected to the hub.
7. Extending Item 5.: the observed hub identifiers must be identifiers of hubs of the net to which the link belongs.
8. Extending Item 6.: the observed link identifiers must be identifiers of links of the net to which the hub belongs.

value
5. ωHIs: L → HI-**set**,
6. ωLIs: H → LI-**set**,
axiom [net−hub−link−identifiers−1]
5. ∀ l:L • **card** ωHIs(l)=2 ∧
6. ∀ h:H • **card** ωLIs(h)≥1 ∧
∀ (hs,ls):N •
5. ∀ h:H • h ∈ hs ⇒ ∀ li:LI • li ∈ ωLIs(h)
⇒ ∃ l′:L • l′ ∈ ls ∧ li=ωLI(l′) ∧ ωHI(h) ∈ ωHIs(l′) ∧
6. ∀ l:L • l ∈ ls ⇒ ∃ h′,h″:H • {h′,h″}⊆hs ∧ ωHIs(l)={ωHI(h′),ωHI(h″)}
7. ∀ h:H • h ∈ hs ⇒ ωLIs(h) ⊆ iols(ls)
8. ∀ l:L • l ∈ ls ⇒ ωHIs(h) ⊆ iohs(hs)
value
iohs: H-**set** → HI-**set**, iols: L-**set** → LI-**set**
iohs(hs) ≡ {ωHI(h)|h:H•h ∈ hs}
iols(ls) ≡ {ωLI(l)|l:L•l ∈ ls}

In the above extensive example we have focused on just five entities: nets, hubs, links and their identifiers. The nets, hubs and links can be seen as separable phenomena. The hub and link identifiers are conceptual models of the fact that hubs and links are connected — so the identifiers are abstract models of 'connection', i.e., the mereology of nets, that is, of how nets are composed. These identifiers are attributes of entities.

Links and hubs have been modelled to possess link and hub identifiers. A link's "own" link identifier enables us to refer to the link, A link's two hub identifiers enables us to refer to the connected hubs. Similarly for the hub and link identifiers of hubs and links.

9. A hub, h_i, state, $h\sigma$, is a set of hub traversals.
10. A hub traversal is a triple of link, hub and link identifiers $(l_{i_{in}}, h_{i_i}, l_{i_{out}})$ such that $l_{i_{in}}$ and $l_{i_{out}}$ can be observed from hub h_i and such that h_{i_i} is the identifier of hub h_i.
11. A hub state space is a set of hub states such that all hub states concern the same hub.

type
9. HT = (LI×HI×LI)
10. HΣ = HT-**set**
11. HΩ = HΣ-**set**

value
10. ωHΣ: H → HΣ
11. ωHΩ: H → HΩ

axiom [hub−states]
∀ n:N,h:H•h ∈ ωHs(n)⇒wf_HΣ(h)∧wf_HΩ(h)

value
wf_HΣ: H → **Bool**, wf_HΩ: H → **Bool**
wf_HΣ(h) ≡ ∀ (li,hi,li′),(_,hi′,_):HT•(li,hi,li′)∈ ωHΣ(h) ⇒
{li,li′}⊆ωLIs(h)∧hi=ωHI(h)∧hi′=hi
wf_HΩ(h) ≡ ∀ hσ:HΣ•hσ ∈ ωHΩ(h)⇒ hσ≠{} ⇒
∀ (li,hi,li′):HT•(li,hi,li′)∈ hσ ⇒ hi=ωHI(h)

2.2 Actions

A set of entities form a state. It is the domain engineer which decides on such states. A function application, one which applies to zero, one or more arguments and a state results in a state changes, is an action.

Example 2 ***Deterministic Actions:***

12. Our example action is that of setting the state of hub.
13. The setting applies to a hub
14. and a hub state in the hub state space
13. and yields a "new" hub.
15. The before and after hub identifier remains the same.
16. The before and after link identifiers remain the same.
17. The before and after hub state space remains the same.
18. The result hub state is that being set (i.e., the argument hub state).

value
12. set_HΣ: H × HΣ → H
13. set_HΣ(h,hσ) **as** h′
14. **pre** hσ ∈ ωHΩ(h)
15. **post** ωHI(h)=ωHI(h′)∧
16. ωLIs(h)=ωLIs(h′)∧
17. ωHΩ(h)=ωHΩ(h′)∧
18. ωHΣ(h′)=hσ

Example 3 ***Non-deterministic Actions:***

19. The result hub state is now a possible hub state:

value
12. set_HΣ: H × HΣ → H
13. set_HΣ(h,hσ) **as** h′
14. **pre** hσ ∈ ωHΩ(h)
15. **post** ωHI(h)=ωHI(h′)∧
16. ωLIs(h)=ωLIs(h′)∧
17. ωHΩ(h)=ωHΩ(h′)∧
19. ωHΣ(h′) ∈ possible_Σs(h)

possible_Σs: H → HΣ-**set**
possible_Σs(h) ≡
let hi = ωHI(h), lis = ωLIs(h) **in** {(li,hi,li′)|li,li′:LI•{li,li′}⊆lis} **end**

2.3 Events

Any state change is an event. A situation in which a (specific) state change was expected but none (or another) occurred is an event. Some events are more "interesting" than other events. Not all state changes are caused by actions of the domain.

Example 4 ***Events:***

20. A hub is in some state, $h\sigma$.
21. An action directs it to change to state $h\sigma'$ where $h\sigma' \neq h\sigma$.
22. But after that action the hub remains either in state $h\sigma$ or is possibly in a third state, $h\sigma''$ where $h\sigma'' \notin \{h\sigma,h\sigma'\}$.
23. Thus an "interesting event" has occurred !

∃ n:N,h:H,hσ,hσ′:HΣ•h ∈ ωHs(n)∧
21.–22. {hσ,hσ′}⊆ωHΩ(h)∧**card**{hσ,hσ′}=2 ∧
20. ωHΣ(h)=hσ ;
21. **let** h′ = set_HΣ(h,hσ′) **in**
22. ωHΣ(h′)∈ ωHΣ(h′)\{hσ′} ⇒
23. "interesting event" **end**

It only makes sense to change hub states if there are more than just one single such state.

2.4 Behaviours

A behaviour is a set of zero, one or more sequences of actions, including events.

Example 5 ***Behaviours:***

24. Let h be a hub of a net n.
25. Let $h\sigma$ and $h\sigma'$ be two distinct states of h.
26. Let $ti : TI$ be some time interval.
27. Let h start in an initial state $h\sigma$.
28. Now let hub h undergo an ongoing sequence of n changes
    (a) from $h\sigma$ to $h\sigma'$ and
    (b) then, after a wait of ti seconds,
    (c) and then , after another wait of ti seconds, back to $h\sigma$.
    (d) After n blinks a pause, $tp : TI$, is made and blinking restarts.

```
type
  TI
value
  ti,tp:TI [ axiom tp>>ti ]
  n:Nat,
  28. blinking: H × HΣ × HΣ × Nat → Unit
  28. blinking(h,hσ,hσ′,m) in
  27.    let h′ = set_HΣ(h,hσ) in
  28c.   wait ti ;
  28a.   let h″ = set_HΣ(h′,hσ′) in
  28c.   wait ti ;
  28.    if m=1
  28.       then skip
  28.       else blinking(h,hσ,hσ′,m−1) end end end
  28.    wait tp ;
  28d.   blinking(h,hσ,hσ′,n)
  25.  pre {hσ,hσ′}⊆ωHΩ(h)∧hσ≠hσ′
  28.        ∧ initial m=n
```

3 Domain Engineering

We focus on the *facet* components of a domain description and leave it to other publications, for ex. [3, Vol. 3, Part IV, Chaps. 8–10], to cover such aspects of domain engineering as stake-holder identification and liaison, domain acquisition and analysis, terminologisation,

verification, testing, model-checking, validation and domain theory. By understanding, first, the *facet* components the domain engineer is in a better position to effectively establish the regime of stakeholders, pursue acquisition and analysis, and construct a necessary and sufficient terminology. The domain description components each cover their domain facet. We outline six such facets: intrinsics, support technology, rules and regulations, scripts (licenses and contracts), management and organisation, and human behaviour. But first we cover a notion of business processes.

3.1 Business Processes

By a business process we understand a set of one or more, possibly interacting behaviours which fulfill a business objective. We advocate that domain engineers, typically together with domain stake-holder groups, rough-sketch their individual business processes.

Example 6 ***Some Transport Net Business Processes:*** With respect to one and the same underlying road net we suggest some business-processes and invite the reader to further rough-sketch these.

29. **Private citizen automobile transports:** Private citizens use the road net for pleasure and for business, for sightseeing and to get to and from work.

    A private citizen automobile transport "business process rough-sketch" might be:

    > A car owner drives to work: *Drives out, onto the street, turns left, goes down the street, straight through the next three intersections, then turns left, two blocks straight, etcetera, finally arrives at destination, and finally turns into a garage.*

30. **Public bus (&c.) transport:** Province and city councils contract bus (&c.) companies to provide regular passenger transports according to timetables and at cost or free of cost.

    A public bus transport "business process rough-sketch" might be:

    > A bus drive from station of origin to station of final destination: *Bus driver starts from station of origin at the designated time for this drive; drives to first passenger stop; open doors to let passenger in; leaves stop at timetable-designated time; drives to next stop adjusting speed to traffic conditions and to "keep time" as per the timetable; repeats this process: "from stop to stop", letting passengers off and on the bus; after having (thus, i.e., in this manner) completed last stop "turns" bus around to commence a return drive.*

31. **Road maintenance and repair:** Province and city councils hire contractors to monitor road (link and hub) surface quality, to maintain set standards of surface quality, and to "emergency" re-establish sudden occurrences of low quality. *Now provide your own rough sketch description.*

32. **Toll road traffic:** State and province governments hire contractors to run toll road nets with toll booth plazas. *Now provide your own rough sketch description.*

33. **Net revision: road (&c.) building:** State government and province and city councils contract road building contractors to extend (or shrink) road nets. *Now provide your own rough sketch description.*

The detailed description of the above rough-sketched business process synopses now becomes part of the domain description as partially exemplified in the previous and the next many examples.

Rough-sketching such business processes helps bootstrap the process of domain acquisition. We shall return to the notion of business processes in Sect. 4.1 where we introduce the concept of *business process re-engineering*.

3.2 Intrinsics

By intrinsics we shall understand the very basics, that without which none of the other facets can be described, i.e., that which is common to two or more of these other facets.

Example 7 ***Intrinsics:*** Most of the descriptions of Sect. 2 model intrinsics. We add a little more. We wish to describe link traversals.

34. A link traversal is a triple of a (from) hub identifier, an along link identifier, and a (towards) hub identifier

35. such that these identifiers make sense in any given net.

36. A link state is a set of link traversals.

37. And a link state space is a set of link states.

value
  n:N
type
  34. $\mathsf{LT}' = \mathsf{HI} \times \mathsf{LI} \times \mathsf{HI}$
  35. $\mathsf{LT} = \{|\mathsf{lt}{:}\mathsf{LT}' \bullet \mathsf{wfLT}(\mathsf{lt})(\mathsf{n})|\}$
  36. $\mathsf{L}\Sigma' = \mathsf{LT}$**-set**
  36. $\mathsf{L}\Sigma = \{|\mathsf{l}\sigma{:}\mathsf{L}\Sigma' \bullet \mathsf{wf\_L}\Sigma(\mathsf{l}\sigma)(\mathsf{n})|\}$
  37. $\mathsf{L}\Omega' = \mathsf{L}\Sigma$**-set**
  37. $\mathsf{L}\Omega = \{|\mathsf{l}\omega{:}\mathsf{L}\Omega' \bullet \mathsf{wf\_L}\Omega(\mathsf{l}\omega)(\mathsf{n})|\}$
value
  35. wfLT: LT $\to$ N $\to$ **Bool**
  35. wfLT(hi,li,hi′)(n) $\equiv$
  35. $\quad \exists$ h,h′:H•{h,h′}$\subseteq\omega$Hs(n)$\wedge\omega$HI(h)=hi$\wedge\omega$HI(h′)=hi′$\wedge$li $\in \omega$LIs(h)$\wedge$li $\in \omega$LIs(h′)

The wf_LΣ and wf_LΩ can be defined like the corresponding functions for hub states and hub state spaces.

3.3 Support Technologies

By support technologies we shall understand the ways and means by which humans and/or technologies support the representation of entities and the carrying out of actions.

Example 8 ***Support Technologies:*** Some road intersections (i.e., hubs) are controlled by semaphores alternately shining **red**–**yellow**–**green** in carefully interleaved sequences in each of the in-directions from links incident upon the hubs. Usually these signalings are initiated as a result of road traffic sensors placed below the surface of these links. We shall model just the signaling:

38. There are three colours: **red**, **yellow** and **green**.
39. Each hub traversal is extended with a colour and so is the hub state.
40. There is a notion of time interval.
41. Signaling is now a sequence, $\langle(h\sigma', t\delta'), (h\sigma'', t\delta''), \ldots, (h\sigma'^{\cdots\prime}, t\delta'^{\cdots\prime})\rangle$ such that the first hub state $h\sigma'$ is to be set first and followed by a time delay $t\delta'$ whereupon the next state is set, etc.
42. A semaphore is now abstracted by the signalings that are prescribed for any change from a hub state $h\sigma$ to a hub state $h\sigma'$.

type

38. Colour == red | yellow | green
39. X = LI×HI×LI×Colour [crossings **of** a hub]
39. HΣ = X-**set** [hub states]
40. TI [time interval]
41. Signalling = (HΣ × TI)*
42. Semaphore = (HΣ × HΣ) $\xrightarrow{m}$ Signalling

value

39. ωHΣ: H → HΣ
42. ωSemaphore: H → Semaphore
13. chg_HΣ: H × HΣ → H

13.–18. chg_HΣ(h,hσ) **as** h′ **pre** hσ ∈ ωHΩ(h) **post** ωHΣ(h′)=hσ

41. chg_HΣ_Seq: H × HΣ → H
41. chg_HΣ_Seq(h,hσ) ≡
41. **let** sigseq = (ωSemaphore(h))(ωΣ(h),hσ) **in** sig_seq(h)(sigseq) **end**
41. sig_seq: H → Signalling → H
41. sig_seq(h)(sigseq) ≡
41. **if** sigseq=⟨⟩ **then** h **else**
41. **let** (hσ,tδ) = **hd** sigseq **in let** h′ = chg_HΣ(h,hσ) **in**
41. **wait** tδ;
41. sig_seq(h′)(**tl** sigseq) **end end end**

3.4 Rules and Regulations

By a rule we shall understand a text which describe how the domain is (i.e., people and technology are) expected to behave. The meaning of a rule is a predicate over "before/after" states of actions: if the predicate holds then the rule has been obeyed. By a regulation we shall understand a text which describes actions to be performed should its corresponding rule fail to hold. The meaning of a regulation is therefore a state-to-state transition, one that brings the domain into a rule-holding "after" state.

Example 9 ***Rules:*** We give two examples related to railway systems where train stations are the hubs and the rail tracks between train stations are the links:

43. Trains arriving at or leaving train stations:
    (a) (In China:) No two trains
    (b) must arrive at or leave a train station
    (c) in any two minute time interval.
44. Trains travelling "down" a railway track. We must introduce a notion of links being a sequence of adjacent sectors.
    (a) Trains must travel in the same direction;
    (b) and there must be at least one "free-from-trains" sector
    (c) between any two such trains.

We omit showing somewhat "lengthy" formalisations.

We omit exemplification of regulations.

3.5 Scripts, Licenses and Contracts

3.5.1 Scripts

By a **script** we understand a set of pairs of rules and regulations.

Example 10 ***Timetable Scripts:***

45. Time is considered discrete. Bus lines and bus rides have unique names (across any set of timetables).
46. A *TimeTable* associates *B*us *L*ine *Id*entifiers (*blid*) to sets of *Journies*.
47. *Journies* are designated by a pair of a *BusRoute* and a set of *BusRides*.
48. A *BusRoute* is a triple of the *BusStop* of origin, a list of zero, one or more intermediate *BusStop*s and a destination *BusStop*.
49. A set of *BusRides* associates, to each of a number of *B*us *Id*entifiers (*bid*) a *BusSched*ule.
50. A *BusSched*ule is a triple of the initial departure *T*ime, a list of zero, one or more intermediate bus stop *T*imes and a destination arrival *T*ime.

51. A *BusStop* (i.e., its position) is a *Frac*tion of the distance along a link (identified by a *L*ink *I*dentifier) *f*rom an *i*dentified *h*ub *t*o an *i*dentified *h*ub.
52. A *Frac*tion is a **Real**, properly between 0 and 1 (incl.).
53. The *Journies* must be *w*ell_*f*ormed in the context of some net.
54. A set of journies is well-formed if
    (a) the bus stops are all different,
    (b) a bus line is embedded in some line of the net, and
    (c) all defined bus trips of a bus line are equivalent.

type
45. T, BLId, BId
46. TT = BLId $\overrightarrow{m}$ Journies
47. Journies′ = BusRoute × BusRides
48. BusRoute = BusStop × BusStop* × BusStop
49. BusRides = BId $\overrightarrow{m}$ BusSched
50. BusSched = T × T* × T
51. BusStop == mkBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
52. Frac = {|r:**Real**•0<r<1|}
53. Journies = {|j:Journies′•∃ n:N • wf_Journies(j)(n)|}

value
54. wf_Journies: Journies → N → **Bool**
54. wf_Journies((bs1,bsl,bsn),js)(hs,ls) ≡
54a. diff_bus_stops(bs1,bsl,bsn) ∧
54b. is_net_embedded_bus_line(⟨bs1⟩^bsl^⟨bsn⟩)(hs,ls) ∧
54c. commensurable_bus_trips((bs1,bsl,bsn),js)(hs,ls)

Timetables are used in the next example.

3.5.2 Licenses and Contracts

By a **license** (a contract) language we understand a pair of languages of licenses and of the set of actions allowed by the license – such that non-allowable actions incur moral obligations whereas, for **contract**s, they incur legal responsibilities.

Example 11 ***Contracts:*** An example contract can be 'schematised':

55. cid: **contractor** cor **contracts sub-contractor** cee
    to perform operations
    {"conduct","cancel","insert","subcontract"}
    with respect to timetable tt.

We assume a context (a global state) in which all contract actions (including contracting) takes place and in which the implicit net is defined.

We bring only abstractions of actions. That is, we hint at "conduct", "cancel", "insert", and "subcontract" such actions. For specific deployments of this *bus transport contract language* we need to be more specific.

56. Concrete examples of actions can be schematised:
    (a) **conduct bus ride** (blid,bid) **to start at time** t
    (b) **cancel bus ride** (blid,bid) **at time** t
    (c) **insert bus ride like** (blid,bid) **at time** t

The schematised license shown earlier is almost like an action; here is the action form:

57. **contractor** cnm′ **is granted a contract** cid′
    to perform operations
    {"conduct","cancel","insert",sublicense"}
    with respect to timetable tt′.

All actions are being performed by a sub-contractor in a context which defines that sub-contractor *cnm*, the relevant net, say *n*, the base contract, referred here to by *cid* (from which this is a sublicense), and a timetable *tt* of which *tt*′ is a subset. contract name *cnm*′ is new and is to be unique. The subcontracting action can (thus) be simply transformed into a contract as shown on Page 15.

type
56. Action = CNm × CId × (SubCon | SmpAct) × Time
56. SmpAct = Conduct | Cancel | Insert
56a. Conduct == μConduct(s_blid:BLId,s_bid:BId)
56b. Cancel == μCancel(s_blid:BLId,s_bid:BId)
56c. Insert = μInsert(s_blid:BLId,s_bid:BId)
57. SubCon == μSubCon(s_cid:CId,s_cnm:CNm,s_body:(s_ops:Op-**set**,s_tt:TT))

We omit formalising the semantics of these syntaxes. A formalisation could be expressed (in CSP [16]) with each bus, each licensee (and licensor), time and the road net bus traffic being processes, etc.

3.6 Management and Organisation

By management we shall understand the set of behaviours which perform strategic, tactical and operational actions. By organisation we shall understand the decomposition of these behaviours into, for example, clearly separate strategic, tactical and operational "areas", possibly further decomposed by geographical and/or "subject matter" concerns. To explain differences between strategic, tactical and operational issues we introduce a notion of $\mathcal{S}trategic$, $\mathcal{T}actical$ and $\mathcal{O}perational$ $\mathcal{F}unds$, $\mathcal{F}_{\mathcal{S},\mathcal{T},\mathcal{O}}$, and other *resources*, $\mathcal{R}$, a notion of *contexts*, $\mathcal{C}$, and a notion of *states*, $\mathcal{S}$. Contexts bind resources to bindings from locations to disjoint time intervals (allocation and scheduling), states bind resource identifiers to maps from resource attribute names to resource values. Simplified types of the strategic, tactical and operational actions are now:

type
$\mathcal{R}, \mathcal{RID}, \mathcal{RAN}, \mathcal{RVAL}, \mathcal{F}_{\mathcal{S}}, \mathcal{F}_{\mathcal{T}}, \mathcal{F}_{\mathcal{O}}$
$\mathcal{C} = \mathcal{R} \xrightarrow{m} ((\mathcal{T} \times \mathcal{T}) \xrightarrow{m} \mathcal{L})$
$\mathcal{S} = \mathcal{RID} \xrightarrow{m} (\mathcal{RAN} \xrightarrow{m} \mathcal{RVAL})$

value
$\omega\mathcal{RID}$: $\mathcal{R} \to \mathcal{RID}$
$\omega\mathcal{RVAL}s$: $\mathcal{R} \to (\mathcal{RAN} \xrightarrow[m]{} \mathcal{RVAL})$
Executive_functions: $\mathcal{C} \times \mathcal{S} \times \mathcal{F}_{\mathcal{S},\mathcal{T},\mathcal{O}} \to \mathcal{F}_{\mathcal{S},\mathcal{T},\mathcal{O}}$
Strategic_functions: $\mathcal{C} \times \mathcal{F}_{\mathcal{S}} \to \mathcal{F}_{\mathcal{S}} \times \mathcal{R} \times \mathcal{C} \times \mathcal{S}$
Tactic_functions: $\mathcal{R} \times \mathcal{C} \times \mathcal{S} \times \mathcal{F}_{\mathcal{T}} \to \mathcal{C} \times \mathcal{F}_{\mathcal{T}}$
Operational_functions: $\mathcal{C} \times \mathcal{S} \times \mathcal{F}_{\mathcal{O}} \to \mathcal{S} \times \mathcal{F}_{\mathcal{O}}$

where we have omitted arguments pertinent to specific functions. Executive functions redistribute financial assets. The above can be the basis for a worthwhile study of a theory of executive, strategic, tactical and operational management.

Example 12 ***Management:*** We relate to Example 11:

58. The **conduct, cancel** and **insert bus ride** actions are operational functions.
59. The actual **subcontract** actions are tactical functions;
60. but the decision to carry out such a tactical function may very well be a strategic function as would be the acquisition or disposal of busses.
61. Forming new timetables, in consort with the contractor, is a strategic function.

We omit formalisations.

3.7 Human Behaviour

By human behaviour we shall understand those aspects of the behaviour of domain stakeholders which have a direct bearing on the "functioning" of the domain, in a spectrum from diligent via sloppy to delinquent and outright criminal neglect in the observance of maintaining entities, carrying out actions and responding to events.

Example 13 ***Human Behaviour:*** Cf. Examples 11–12. Under the assumption that there is no technical reasons for not conducting a, or some bus rides, and then with respect to any one specific bus driver:

62. Conducting all bus rides must be classified as diligent;
63. rare failures to conduct a bus ride must be classified as sloppy;
64. occasional failures $\cdots$ as delinquent;
65. repeated patterns of failures $\cdots$ as criminal.

We omit showing somewhat "lengthy" formalisations.

3.8 Discussion

We have ever so briefly outlined six domain facet concepts and we have exemplified each of these. Real-scale domain descriptions are, of course, much larger than what we can show. Typically, say for the domain of logistics, a description is 30 pages; for "small" parts of railway systems we easily get up to 100–200 pages of the kind shown here. The reader should now have gotten a reasonably clear idea as to what constitutes a domain description. As mentioned, in the introduction to Sect. 3, we shall not cover post-modelling activities such a validation and domain theory formation. The latter is usually part of the verification (theorem proving, model checking and formal testing) of the formal domain description. Final validation of a domain description is with respect to the narrative part of the narrative/formalisation pairs of descriptions. The reader should also be able to form a technical opinion about what can be formalised, and that not all can be formalised within the framework of a single formal specification language, cf. Sect. 1.3.

4 Requirements Engineering

Whereas a domain description presents a domain **as it is**, a requirements prescription presents a domain **as it would be** if some required machine was implemented (from these requirements). The machine is the hardware plus software to be designed from the requirements. That is, the *machine* is what the requirements are about. We distinguish between three kinds of requirements: (Sect. 4.2) the domain requirements are those requirements which can be expressed solely using terms of the domain; (Sect. 4.4) the machine requirements are those requirements which can be expressed solely using terms of the machine and (Sect. 4.3) the interface requirement are those requirements which must use terms from both the domain and the machine in order to be expressed.

4.1 Business Process Re-engineering

In Sect. 3.1 we very briefly covered a notion of business processes. These were the business processes of the domain before installation of possible computing systems. The potential of installing computing systems invariably requires revision of established business processes. Business process re-engineering (BPR) is a development of new business processes – whether or not complemented by computing and communication. BPR, such as we advocate it, proceeds on the basis of an existing domain description and outlines needed changes (additions, deletions, modifications) to entities, actions, events and behaviours following the six domain facets outlined in Sects. 3.2–3.7.

Example 14 ***Rough-sketching a Re-engineered Road Net:*** Our sketch centers around a toll road net with toll booth plazas. The BPR focuses first on entities, actions, events and behaviours (Sect. 2), then on the six domain facets (Sects. 3.2–3.7).

66. **Re-engineered Entities:** We shall focus on a linear sequence of toll road intersections (i.e., hubs) connected by pairs of one-way (opposite direction) toll roads (i.e., links). Each toll road intersection is connected by a two way road to a toll plaza. Each toll plaza contains a pair of sets of entry and exit toll booths. (Example 16 brings more details.)

67. **Re-engineered Actions:** Cars enter and leave the toll road net through one of the toll plazas. Upon entering, car drivers receive, from the entry booth, a plastic/paper/electronic ticket which they place in a special holder in the front window. Cars arriving at intermediate toll road intersections choose, on their own, to turn either "up" the toll road or "down" the toll road — with that choice being registered by the electronic ticket. Cars arriving at a toll road intersection may choose to "circle" around that intersection one or more times — with that choice being registered by the electronic ticket. Upon leaving, car drivers "return" their electronic ticket to the exit booth and pay the amount "asked" for.

68. **Re-engineered Events:** A car entering the toll road net at a toll both plaza entry booth constitutes an event. A car leaving the toll road net at a toll both plaza entry booth constitutes an event. A car entering a toll road hub constitutes an event. A car entering a toll road link constitutes an event.

69. **Re-engineered Behaviours:** The journey of a car,from entering the toll road net at a toll booth plaza, via repeated visits to toll road intersections interleaved with repeated visits to toll road links to leaving the toll road net at a toll booth plaza, constitutes a behaviour — withreceipt of tickets, return of tickets and payment of fees being part of these behaviours. Notice that a toll road visitor is allowed to cruise "up" and "down" the linear toll road net – while (probably) paying for that pleasure (through the recordings of "repeated" hub and link entries).

70. **Re-engineered Intrinsics:** Toll plazas and abstracted booths are added to domain intrinsics.

71. **Re-engineered Support Technologies:** There is a definite need for domain-describing the failure-prone toll plaza entry and exit booths.

72. **Re-engineered Rules and Regulations:** Rules for entering and leaving toll booth entry and exit booths must be described as must related regulations. Rules and regulations for driving around the toll road net must be likewise be described.

73. **Re-engineered Scripts:** No need.

74. **Re-engineered Management and Organisation:** There is a definite need for domain describing the management and possibly distributed organisation of toll booth plazas.

75. **Re-engineered Human Behaviour:** Humans, in this case car drivers, may not change their behaviour in the spectrum from diligent and accurate via sloppy and delinquent to outright traffic-law breaking – so we see no need for any "re-engineering".

4.2 Domain Requirements

For the phase of domain requirements the requirements stake-holders "sit together" with the domain cum requirements engineers and read the domain description, line-by-line, in order to "derive" the domain requirements. They do so in five rounds (in which the BPR rough sketch is both regularly referred to and most likely regularly updated). Domain requirements are "derived" from the domain description as covered in Sect. 4.2.1–4.2.5.

4.2.1 Projection

By *domain projection* we understand an operation that applies to a domain description and yields a domain requirements prescription. The latter represents a projection of the former in which only those parts of the domain are present that shall be of interest in the ongoing requirements development.

Example 15 ***Projection:*** Our requirements is for a simple toll road: a linear sequence of links and hubs outlined in Example 14: see Items 1.–11. of Example 1 and Items 34.–37. of Example 7.

4.2.2 Instantiation

By *domain instantiation* we understand an operation that applies to a (projected) domain description, i.e., a requirements prescription, and yields a domain requirements prescription, where the latter has been made more specific, usually by constraining a domain description

Example 16 Instantiation: Here the toll road net topology as outlined in Example 14 is introduced: a straight sequence of toll road hubs pairwise connected with pairs of one way links and with each hub two way link connected to a toll road plaza.

type
  H, L, P = H
  $N' = (H \times L) \times H \times ((L \times L) \times H \times (H \times L))^*$
  $N'' = \{|n{:}N' \bullet wf(n)|\}$
value
  wf_N'': $N' \rightarrow$ **Bool**
  wf_N''((h,l),h',llhpl) ≡ ... 6 lines ... !
  αN: $N'' \rightarrow$ N
  αN((h,l),h',llhpl) ≡ ... 2 lines ... !

wf_N'' secures linearity; αN allows abstraction from more concrete N'' to more abstract N.

4.2.3 Determination

By *domain determination* we understand an operation that applies to a (projected and possibly instantiated) domain description, i.e., a requirements prescription, and yields a domain requirements prescription, where (attributes of) entities, actions, events and behaviours have been made less indeterminate.

Example 17 ***Determination:*** Pairs of links between toll way hubs are open in opposite directions; all hubs are open in all directions; links between toll way hubs and toll plazas are open in both directions.

type
  $L\Sigma$ = (HI×HI)-**set**, $L\Omega$ = $L\Sigma$-**set**
  $H\Sigma$ = (LI×LI)-**set**, $H\Omega$ = $H\Sigma$-**set**
  $N' = (H \times L) \times H \times ((L \times L) \times H \times (H \times L))^*$
value
  $\omega L\Sigma$: $L \rightarrow L\Sigma$, $\omega L\Omega$: $L \rightarrow L\Omega$

ωHΣ: H → HΣ, ωHΩ: H → HΩ
axiom
∀ ((h,l),h′,llhhl:⟨(l′,l″),h″,(h‴,l‴)⟩^llhhl′):N″ •
ωLΣ(l)={(ωHI(h),ωHI(h′)),(ωHI(h′),ωHI(h))}∧
ωLΣ(l‴)={(ωHI(h″),ωHI(h‴)),(ωHI(h‴),ωHI(h″))}∧
∀ i,i+1:**Nat** • {i,i+1}⊆**inds** llhhl ⇒
let ((li,li′),hi,(hi″,li″))=llhhl(i), (_,hj,(hj″,lj″))=llhhl(i+1) **in**
ωLΩ(li)= {{(ωHI(hi),ωHI(hj))}}∧ωLΩ(li′)={{(ωHI(hj),ωHI(hi))}}∧
ωHΩ(hi)= { ... } ... 3 lines **end**

4.2.4 Extension

By *domain extension* we understand an operation that applies to a (projected and possibly determined and instantiated) domain description, i.e., a (domain) requirements prescription, and yields a (domain) requirements prescription. The latter prescribes that a software system is to support, partially or fully, entities, operations, events and/or behaviours that were not feasible (or not computable in reasonable time) in a domain without computing support, but which now are not only feasible but also computable in reasonable time.

Example 18 ***Extension:*** We extend the domain by introducing toll road entry and exit booths as well as electronic ticket hub sensors and actuators. There should now follow a careful narrative and formalisation of these three machines: the car driver/machine "dialogues" upon entry and exit as well as the sensor/car/actuator machine "dialogues" when cars enter hubs. The description should first, we suggest, be ideal; then it should take into account failures of booth equipment, electronic tickets, car drivers, and of sensors and actuators.

4.2.5 Fitting

By *domain requirements fitting* we understand an operation which takes n domain requirements prescriptions, d_{r_i}, that are claimed to share m independent sets of tightly related sets of entities, actions, events and/or behaviours and map these into $n+m$ domain requirements prescriptions, δ_{r_j}, where m of these, $\delta_{r_{n+k}}$ capture the shared phenomena and concepts and the other n prescriptions, δ_{r_ℓ}, are like the n "input" domain requirements prescriptions, d_{r_i}, except that they now, instead of the "more-or-less" shared prescriptions, that are now consolidated in $\delta_{r_{n+k}}$, prescribe interfaces between δ_{r_i} and $\delta_{r_{n+k}}$ for $i : \{1..n\}$.

Example 19 ***Fitting:*** We assume three ongoing requirements development projects, all focused around road transport net software systems: (i) road maintenance, (ii) toll road car monitoring and (iii) bus services on ordinary plus toll road nets. The main shared phenomenon is the road net, i.e., the links and the hubs. The consolidated, shared road net domain requirements prescription, $\delta_{r_{n+1}}$, is to become a prescription for the domain requirements for shared hubs and links. Tuples of these relations then prescribe representation of all hub, respectively all link attributes – common to the three applications. Functions (including actions) on hubs and links become database queries and updates. Etc.

4.2.6 Discussion

The last page or so have very briefly surveyed and illustrated domain requirements. The reader should take cognizance of the fact that these are indeed "derived" from the domain description. They are not domain descriptions, but, once the business process re-engineering has been adopted and the required software has been installed, then the domain requirements become part of a revised domain description !

4.3 Interface Requirements

By interface requirements we understand such requirements which are concerned with the phenomena and concepts *shared* between the domain and the machine. Thus such requirements can only be expressed using terms from both the domain and the machine. We tackle the problem of "deriving", i.e., constructing interface requirements by tackling four "smaller" problems: those of "deriving" interface requirements for entities, actions, events and behaviours.

4.3.1 Entity Interfaces

Entities that are shared between the domain and the machine must initially be input to the machine. Dynamically arising entities must likewise be input and all such machine entities must have their attributes updated, when need arise. Requirements for shared entities thus entail requirements for their representation and for their human/machine and/or machine/machine transfer dialogue.

Example 20 ***Shared Entities:*** Main shared entities are those of hubs and links. We suggest that eventually a relational database be used for representing hubs links in relations. As for human input, some man/machine dialogue based around a set of visual display unit screens with fields for the input of hub, respectively link attributes can then be devised. Etc.

4.3.2 Action Interfaces

By a shared action we mean an action that can only be partly computed by the machine. That is, the machine, in order to complete an action, may have to inquire with the domain (some measurable, time-varying entity attribute value, or some domain stake-holder) in order to proceed in its computation.

Example 21 ***Shared Actions:*** In order for a car driver to leave an exit toll both the following component actions must take place: the driver inserts the electronic pass in the exit toll booth machine; the machine scans and accepts the ticket and calculates the fee for the car journey from entry booth via the toll road net to the exit booth; the driver is alerted to the cost and is requested to pay this amount; once paid the exit booth toll gate is raised. *Notice that a number of details of the new support technology is left out. It could either be elaborated upon here, or be part of the system design.*

4.3.3 Event Interfaces

By a shared event we mean an event whose occurrence in the domain need be communicated to the machine – and, vice-versa, an event whose occurrence in the machine need be communicated to the domain.

Example 22 ***Shared Events:*** The arrival of a car at a toll plaza entry booth is an event that must be communicated to the machine so that the entry booth may issue a proper pass (ticket). Similarly for the arrival at a toll plaza exit booth so that the machine may request the return of the pass and compute the fee. The end of that computation is an event that is communicated to the driver (in the domain) requesting that person to pay a certain fee after which the exit gate is opened.

4.3.4 Behaviour Interfaces

By a shared behaviour we understand a sequence of zero, one or more shared actions and events.

Example 23 ***Shared Behaviour:*** A typical toll road net use behaviour is as follows: Entry at some toll plaza: receipt of electronic ticket, placement of ticket in special ticket "pocket" in front window, the raising of the entry booth toll gate; drive up to [first] toll road hub (with electronic registration of time of occurrence), drive down a selected link (with electronic registration of time of occurrence of entry to and exit from link), then a repeated number of zero, one or moretoll road hub and link visits – some of which may be "repeats" – ending with a drive down from a toll road hub to a toll plaza with the return of the electronic ticket, etc. – cf. Example 22.

4.3.5 Discussion

The discussion of Sect. 4.2.6 carries over to this section. That is, once the machine has been installed it, the machine, is part of the new domain !

4.4 Machine Requirements

Domains, other than the introspective machine domain itself, has no bearing on machine requirements we shall not cover this stage of requirements development other than saying that it consists of the following concerns: performance requirements (storage, speed, other resources), dependability requirements (availability, accessibility, integrity, reliability, safety, security), maintainability requirements (adaptive, extensional, corrective, perfective, preventive), portability requirements (development platform, execution platform, maintenance platform, demo platform) and documentation requirements. Only performance and dependability seems to be subjectable to rigorous, formal treatment. We refer to [3, Vol. 3, Part V, Chap. 19, Sect. 19.6] for an extensive (30 page) survey.

The **discussions** of Sects. 4.2.6 and 4.3.5 carry over to this paragraph. That is, once the machine has been installed it, the machine, is part of the new domain !

5 Discussion

5.1 What Have We Achieved – and What Not

Item 4. of Sect. 1.4 made some claims. We think we have substantiated them all, albeit ever so briefly. Each of the domain facets (intrinsics, support technologies, management and organisation, rules and regulations, scrips [licenses and contracts] and human behaviour) and each of the requirements facets (projection, instantiation, determination, extension and

fitting) provide rich grounds for both specification methodology studies and and for more theoretical studies [4].

5.2 What Have We Omitted

Our coverage of domain and requirements engineering has focused on modelling techniques for domain and requirements facets. We have omitted the important software engineering tasks of stake-holder identification and liaison, domain and, to some extents also requirements acquisition and analysis, terminologisation, and techniques for domain and requirements validation and verification. We refer, instead, to [3, Vol.3, Part IV (Chaps. 9, 12–14) and Part V (Chaps. 18, 20–23)].

5.3 Domain Engineering Can Be Pursued Just By Itself

One can pursue domain engineering just for the sake of understanding a domain. As for physics.

5.4 Domain Descriptions Are Not Normative

The description of, for example, "the" domain of the New York Stock Exchange would describe the set of rules and regulations governing the submission of sell offers and buy bids as well as those of clearing ('matching') sell offers and buy bids. These rules and regulations appears to be quite different from those of the Tokyo Stock Exchange [27]. A normative description of stock exchanges would abstract these rules so as to be rather un-informative. And, anyway, rules and regulations changes and business process re-engineering changes entities, actions, events and behaviours. For any given software development one may thus have to rewrite parts of existing domain descriptions, or construct an entirely new such description.

5.5 "Requirements Always Change"

This claim is often used as a hidden excuse for not doing a proper, professional job of requirements prescription, let alone "deriving" them, as we advocate, from domain descriptions. Instead we now make the following counterclaims [1] "domains are far more stable than requirements" and [2] "requirements changes arise more as a result of business process re-engineering than as a result of changing stake-holder ideas". Cases (1), where it seems that domains are changing, are most often examples of business process re-engineering. Closer studies of a number of domain descriptions, for example of a *financial service industry*, reveals that the domain in terms of which an "ever expanding" variety of financial products are offered, are, in effect. based on a small set of very basic domain functions which have been offered for well-nigh centuries ! We claim that thoroughly developed domain descriptions and thoroughly "derived" requirements prescriptions tend to stabilise the requirements re-design, but never alleviate it.

5.6 What Can Be Described and Prescribed

The issue of *"what can be described"* has been a constant challenge to philosophers. In [26] Russell covers his first *Theory of Descriptions* (stemming from the early 1900s), and in [25] a revision, as *The Philosophy of Logical Atomism*. The issue is not that straightforward. In

[5, 6] we try to broach the topic from the point of view of the kind of domain engineering presented in this paper. Our approach is simple; perhaps too simple ! We can describe what can be observed. We do so, first by postulating *types* of observable phenomena and of derived concepts; then by introducing *observer* functions and *axiom*s over these, that is, over values of postulated types and observers. To this we add defined functions – usually described by pre/post-conditions. The narratives refer to the "real" phenomena whereas the formalisations refer to related phenomenological concepts. The narrative/formalisation problem is that one can 'describe' phenomena without always knowing how to formalise them.

5.7 Relation to Other Works

The most obvious 'other' work is that of [21, M.A.Jackson: Problem Frames]. In [21], Jackson, like is done here, departs radically from conventional requirements engineering. In his approach understandings of the domain, the requirements and possible software designs are arrived at, not hierarchically, but in parallel, interacting streams of decomposition. Thus the 'Problem Frame' development approach iterates between concerns of domains, requirements and software design. "Ideally" our approach pursues domain engineering prior to requirements engineering, and, the latter, prior to software design. But see next.

The recent book [23, Axel van Lamsweerde] appears to represent the most definite work on Requirements Engineering today. It covers goal modelling, time-based descriptions of system behaviour, scenarios, and requirements analysis. Much of this "carries" over, inter alia, to both domain and ("our") requirements acquisition.

5.8 "Ideal" Versus Real Developments

The term 'ideal' has been used in connection with 'ideal development' from domain to requirements. We now discuss that usage. Ideally software development could proceed from developing domain descriptions via "deriving" requirements prescriptions to software design, each phase involving extensive formal specifications, verifications (formal testing, model checking and theorem proving) and validation. More realistically, less comprehensive domain description development (D) may alternate with both requirements development (R) work and with software design (S) – in some hopefully controlled, contained "spiralling" manner and such that it is, at all times, clear which $\mathcal{D}$, $\mathcal{R}$ or $\mathcal{S}$ development step is taken.

5.9 A Reference Model for Domains, Goals, Requirements and Software

In [14] Gunter, Gunter, Jackson and Zave suggests a reference model of requirements and specifications. In this section we shall present a simplified version of that model while also covering cases not covered in [14].

$\mathcal{D} \models \mathcal{P}$ expresses that one can derive properties from a formal domain description. Whether they are also properties of the "actual" domain now depends on experimental evidence.

$\mathcal{D}, \mathcal{R} \models \mathcal{G}$ expresses that one can argue that the requirements, in the context of the domain, entails the goals.

$\mathcal{D}, \mathcal{S} \models \mathcal{R}$ expresses that in a proof of correctness of $\mathcal{S}$oftware design with respect to $\mathcal{R}$equirements prescriptions one often has to refer to assumptions about the $\mathcal{D}$omain. Formalising our understandings of the $\mathcal{D}$omain, the $\mathcal{R}$equirements and the $\mathcal{S}$oftware design en-

ables proofs that the software is right and the $\mathcal{D}$omain formalisation of the "derivation" of $\mathcal{R}$equirements from $\mathcal{D}$omain specifications help ensure that it is the right software [9].

Our triptych treatment of software development differs from that of [14] in the following:

MORE TO COME

5.10 Domain Versus Ontology Engineering

In the information science community an ontology is a "formal, explicit specification of a shared conceptualisation". Most of the information science ontology work seems aimed primarily at axiomatisations of properties of entities. Apart from that there are many issues of "ontological engineering" that are similar to "our kind" of domain engineering; but then, we claim, that domain engineering goes well beyond ontological engineering and makes free use of whatever formal specification languages are needed, cf. Item [3] of Sect. 1.3.

6 Conclusion

We have put forward the methodological steps of another approach to requirements engineering than currently 'en vogue'. We claim that our approach, as it follows from the **dogma** expressed at the opening of Sect. 1, is logical, that is, follows as a necessity. The 'triptych' approach has been in partial use since the early 1990s, notably at the United Nations University's International Institute for Software Technology (`www.iist.unu.edu`). This paper presents this triptych in a clearer form than presented in [3]. The (six) domain, the (five) domain requirements and the (4) interface requirements facets each have nice theories and each has a simple set of methodological principles and techniques – some covered in [3, 4, 5, 6] others to be further researched.

Acknowledgements: I thank Alan Bundy of The University of Edinburgh and Tetsuo Tamai of The University of Tokyo for their letting me use time when visiting them to write this paper.

7 Bibliographical Notes

Section 1.3 gives most relevant references to formal specification languages (techniques and tools) that cover the spectrum of domain and requirements specification, refinement and verification. The recent book on Logics of Specification Languages [10] covers `ASM, B/event B, CafeObj, CASL, DC, RAISE, TLA+, VDM` and `Z`.

References

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

[2] K. Araki et al., editors. *IFM 1999–2009: Integrated Formal Methods*, volume 1945, 2335, 2999, 3771, 4591, 5423 (only some are listed) of *Lecture Notes in Computer Science*. Springer, 1999–2009.

[3] D. Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling; Vol. 2: Specification of Systems and Languages; ol. 3: Domains, Requirements and Software Design.* Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.

[4] D. Bjørner. Domain Theory: Practice and Theories, Discussion of Possible Research Topics. In *ICTAC'2007*, volume 4701 of *Lecture Notes in Computer Science (eds. J.C.P. Woodcock et al.)*, pages 1–17, Heidelberg, September 2007. Springer. Final Version[5].

[5] D. Bjørner. An Emerging Domain Science – A Rôle for Stanisław Leśniewski's Mereology and Bertrand Russell's Philosophy of Logical Atomism. *Higher-order and Symbolic Computation*, 2009. Final Version[6].

[6] D. Bjørner. On Mereologies in Computing Science. In *Festschrift for Tony Hoare*, History of Computing (ed. Bill Roscoe), London, UK, 2009. Springer. Final Version[7].

[7] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.

[8] D. Bjørner and C. B. Jones, editors. *Formal Specification and Software Development.* Prentice-Hall, 1982.

[9] B. Boehm. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, NJ., USA, 1981.

[10] Dines Bjørner and Martin C. Henson, editor. *Logics of Specification Languages.* EATCS Series, Monograph in Theoretical Computer Science. Springer, Heidelberg, Germany, 2008.

[11] J. Fitzgerald and P. G. Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development.* Cambridge University Press, Cambridge, UK, Second edition, 2009.

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

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

[14] C. A. Gunter, E. L. Gunter, M. A. Jackson, and P. Zave. A Reference Model for Requirements and Specifications. *IEEE Software*, 17(3):37–43, May–June 2000.

[15] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[16] C. 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).

[5] http://www.imm.dtu.dk/ db/ictac-paper.pdf
[6] http://www.imm.dtu.dk/ db/domain.pdf
[7] http://www2.imm.dtu.dk/ db/bjorner-hoare75-p.pdf

[17] D. M. Hoffman and D. M. Weiss, editors. *Software fundamentals: collected papers by David L. Parnas*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[18] ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992, 1996, 1999.

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

[20] M. A. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press. Addison-Wesley, Reading, England, 1995.

[21] M. A. Jackson. *Problem Frames — Analyzing and Structuring Software Development Problems*. ACM Press, Pearson Education. Addison-Wesley, England, 2001.

[22] L. Lamport. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA, 2002.

[23] A. Lamsweerde. *Requirements Engineering: from system goals to UML models to software specifications*. Wiley, 2009.

[24] W. Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Leitfäden der Informatik. Vieweg+Teubner, 1st edition, 15 June 2010. 248 pages; ISBN 978-3-8348-1290-2.

[25] B. Russell. The Philosophy of Logical Atomism. *The Monist: An International Quarterly Journal of General Philosophical Inquiry,*, xxxviii–xxix:495–527, 32–63, 190–222, 345–380, 1918–1919.

[26] B. Russell. *Introduction to Mathematical Philosophy*. George Allen and Unwin, London, 1919.

[27] T. Tamai. Social Impact of Information System Failures. *Computer, IEEE Computer Society Journal*, 42(6):58–65, June 2009.

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

[29] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.

[30] 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.