From Domains to Requirements

On a Triptych of Software Development

Dines Bjørner
School of Informatics, The University of Edinburgh, Scotland and
Graduate School of Arts and Sciences, The University of Tokyo, Japan

ABSTRACT

Domain engineering, in the sense of this paper, is offered as a means to help secure that software engineers deliver *the right software* – where formalisation of relevant stages and steps of software development helps secure that *the software is right* [9]. In this paper we shall present the essence of a software development *triptych*: from domains via requirements to software design. We emphasize the two first phases: *domain engineering* and *requirements engineering*. We show the pragmatic stages of the construction of *domain descriptions*: the facets of *intrinsics, support technologies, management & organisation, rules & regulations, script (licenses and contracts)* and *human behaviour*. And we show how to construct main facets of *requirements prescriptions*: *domain requirements* and *interface requirements*. In this respect we focus in particular on the *domain requirements* development stages of *projection, instantiation, determination* and *extension*. The paper represents a summary as well as some significant improvements over the domain-to-requirements coverage of [4, Vol. 3, Parts IV–V].

Categories and Subject Descriptors

D.2 [**Software Engineering**]: General; D.2.1 [**Software Engineering**]: Requirements Engineering—*derivation from domain descriptions*; D.2.4 [**Software Engineering**]: Verification; D.2.10 [**Software Engineering**]: Methodologies; F.3 [**Theory of Computation**]: Logics and Meanings of Specification; D.3 [**Software Engineering**]: Specification Languages

Keywords

Domain Engineering, Requirements Engineering, Formal Specification

1. INTRODUCTION

Current software development, when it is pursued in a state-of-the-art, but still a conventional manner, starts with requirements engineering and proceeds to software design. Current software development practices appears to be focused on processes (viz.: "best practices': tools and techniques'). In a delightful paper: [16, CACM, April 2009] Daniel Jackson introduces the concept of *'a direct path to* [the development of] *dependable software'* – in contrast to the concept of processes. The current paper contributes to such *direct paths*. 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 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 (scheduling, rostering, signalling, 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 proper, including formal descriptions of their domains: for railways cf. `www.railwaydomain.org`, and for pipelines `pipelines.pdf`, logistics `logistics.pdf` and for container lines `container-paper.pdf` – all at `www.-imm.dtu.dk/~db/`. The process knowledge and "best" practices of the triptych software engineering is well-founded in and takes place in the context of established domain model and an established, carefully phrased (and formalised) requirements model. The 24 hour 7 days a week trustworthy operation of many software systems is so crucial that utmost care must be taken to ensure that they fulfill all (and only) the customers expectations and are correct. Barry Boehm [9, 1981] has coined the statement: *it is the right software and the software is right*. Extra care must be taken to ensure those two "rights". And here it is not enough to only follow current "best process, technique and tool practices". Software engineers must follow – what is also clearly stated in [16, Daniel Jackson] – some form of direct path. This paper will illustrate some facets of such a direct approach. The phase, stage and step-wise, possibly iterative composition of what the Triptych approach offers has been so arranged as to provide *direct evidence* of the evolving software's dependability.

1.1 A Triptych of Software Engineering

Dogma: *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"[1], 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 relative completeness. $\mathcal{P}$roperties, as stated by domain stakeholders, are proved with respect to the domain description ($\mathcal{D}\models\mathcal{P}$). This phase is the most important, we think, when it comes to secure the first of the two "rights": that we are on our way to develop the right software.

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* points out which domain description parts are to be kept (*projected*) out of the *domain requirements*, and for those kept in, what *instantiations, determinations* and *extensions* are required. Similarly the requirements stakeholders, guided by the domain and requirements engineers, informs 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 – other than saying this: the design is "derived" from the requirements.

To ensure that the software being developed is right, that is, correct, we can then rigorously argue, informally, or formally – test, model check and/or prove, that the $\mathcal{S}$oftware is correct with respect to the $\mathcal{R}$equirements in the context of the $\mathcal{D}$omain: $\mathcal{D},\mathcal{S}\models\mathcal{R}$. These, the $\mathcal{D}$omain descriptions, the $\mathcal{R}$equirements prescriptions, the $\mathcal{S}$oftware design specification, and the rigorous correctness arguments (whether informal or formal) are examples of [16]'s concept of *direct evidence*.

1.2 What are Domains ?

By a domain we shall here understand a universe of discourse, an area of nature subject to laws of physics and study by physicists, or an area of human activity subject to its interfaces with other domains and to 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/shipping*, and *air/aircraft 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.*; *heating systems*; *heart pumps*; etc. 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.

[1]Section 5 [Item 5] will discuss renditions of "idealism"!

1.3 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 [18], 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 text alternates with clearly related formula text.

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 *intrinsics, support technologies, management and organisation, rules and regulations, scripts (licenses and contracts)* and *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 re-engineering* (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 *projection, instantiation, determination, extension* and *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. 4.2–4.2); and (5) the *identification* and *'elaboration'* of the technically determined *interface requirements facets* of *shared entity, shared action, shared event* and *shared behaviour* requirements principles and techniques (Sects. 4.3–4.3). We claim that the facets of (2, 3, 4) and (5) are all *novel*. In Sect. 5 we shall discuss these contributions in relation to the works and contributions of other researchers and technologists.

1.5 Structure of Paper

Before going into some details on domain enginering (Sect. 3) and requirements engineering (Sect. 4) we shall in the next section (Sect. 2) 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.

2. A SPECIFICATION ONTOLOGY

In order to describe domains we postulate the following related specification components: *entities*, *actions*, *events* and *behaviours*. Although not part of a proper domain description the examples of this section are necessary in order for the reader to better envisage what the domain descriptions and requirements prescriptions must "ultimately" cover.

[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 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
[2] ∀ (hs,ls):N • **card** hs≥2 ∧ **card** ks≥1
type
[3] HI, LI
value
[4] ωHI: H → HI, ωLI: L → LI
axiom
[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 express the following:

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
[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
∀ n:N,h:H•h ∈ ωHs(n)⇒wf_HΣ(ωHΣ(h))∧wf_HΩ(h,ωHΩ(h))
value
wf_HΣ: HΣ → **Bool**, wf_HΩ: H×HΩ → **Bool**
wf_HΣ(hσ) ≡ ∀ (li,hi,li′),(_,hi′,_):HT•(li,hi,li′)∈ hσ ⇒
{li,li′}⊆ωLIs(h)∧hi=ωHI(h)∧hi′=hi
wf_HΩ(h,hω) ≡ ∀ hσ:HΣ•hσ ∈ hω⇒wf_HΣ(hσ)∧hσ≠{} ⇒
let (li,hi,li′):HT•(li,hi,li′)∈ hσ **in** hi=ωHI(h) **end**

■ 1

[2] **Actions:** A set of entities form a domain state. It is the domain engineer which decides on such states. A function which, when applied to zero, one or more arguments and a state, results in a state change, is an action. (Arguments could be other entities or just values of entity attributes.)

Example 2. ***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 hub state space remains the same.

17 The result hub is in the hub state space — subject to some probability distribution.

value
p:**Real**, **axiom** 0<p≤1, typically p≃ $1-10^{-7}$
$\overline{p}$:**Real**, **axiom** $\overline{p}$=1−p

[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] ωHΩ(h)=ωHΩ(h′)∧
[17] ωHΣ(h′)=
[17] (⌈⌉{hσ′|hσ′:HΣ•hσ′∈ ωΩ(h)\{hσ}})$_{\overline{p}}$⌈⌉$_p$hσ

The non-deterministic internal choice operator expression $s_{\overline{p}}\lceil\rceil_p s'$ with probability p has value s' and with probability $\overline{p}$ has value s. The prefix internal choice operator expression $\lceil\rceil\{h\sigma_i, h\sigma_j, \ldots, h\sigma_k\}$ h nondeterministically as one of the values in the set $\{h\sigma_i, h\sigma_j, \ldots, h\sigma_k\}$, that is, is the same as $h\sigma_i \lceil\rceil h\sigma_j \lceil\rceil \ldots \lceil\rceil h\sigma_k$ ■ 2

[3] **Events:** Any domain 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 3. ***Events:***

18 A hub is in some state, $h\sigma$.

19 An action directs it to change to state $h\sigma'$ where $h\sigma' \neq h\sigma$.

20 But after that action the hub remains either in state $h\sigma$ or is possibly in a third state, $h\sigma''$ where $h\sigma'' \notin \{h\sigma, h\sigma'\}$.

21 Thus an "interesting event" has occurred !

```
∃ n:N,h:H,hσ,hσ′:HΣ•h ∈ ωHs(n)∧
  [19,20] {hσ,hσ′}⊆ωHΩ(h)∧card{hσ,hσ′}=2 ∧
  [18]    ωHΣ(h)=hσ ;
  [19]    let h′ = set_HΣ(h,hσ′) in
  [20]    ωHΣ(h′)∈ ωHΣ(h′)\{hσ′} ⇒
  [21]    "interesting event" end
```

It only makes sense to change hub states if there are more than just one single such state. ■ 3

[4] **Behaviours:** A behaviour is a set of zero, one or more sequences of actions, including events.

Example 4. ***Behaviours: Blinking Semaphores:***

22 Let h be a hub of a net n.

23 Let $h\sigma$ and $h\sigma'$ be two distinct states of h.

24 Let $ti : TI$ be some time interval.

25 Let h start in an initial state $h\sigma$.

26 Now let hub h undergo an ongoing sequence of n changes

26a from $h\sigma$ to $h\sigma'$ and

26b then, after a wait of ti seconds,

26c and then back to $h\sigma$.

```
type
  TI
value
  ti:TI
  n:Nat
  [26]  blinking: H × HΣ × HΣ → H
  [26]  blinking(h,hσ,hσ′,m) in
  [25]    let h′ = set_HΣ(h,hσ) in
  [26c]   wait ti ;
  [26a]   let h′′ = set_HΣ(h′,hσ′) in
  [26c]   wait ti ;
  [26]    if m=1 then h′′
  [26]    else blinking(h,hσ,hσ′,m−1) end end end
  [23]    pre {hσ,hσ′}⊆ωHΩ(h)∧hσ≠hσ′
  [26]        ∧ initial m=100
```

■ 4

3. DOMAIN ENGINEERING

We focus on the *facet* components of a domain description and leave it to other publications, for ex. [4, Vol. 3, Part IV, Chaps. 8–10], to cover such aspects of domain engineering as stakeholder identification and liaison, domain acquisition and analysis, terminologisation, verification, testing, model-checking, validation and domain theory formation. 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 stakeholder groups, rough-sketch their individual business processes.

Example 5. ***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 rough-sketch these.

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

28 **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.

29 **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.

30 **Toll road traffic:** State and province governments hire contractors to run toll road nets with toll booth plazas.

31 **Net revision: road (&c.) building:** State government and province and city councils contract road building contractors to extend (or shrink) road nets.

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. ■ 5

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, usually all of these other facets.

Example 6. ***Intrinsics:*** Most of the descriptions of Sect. 2 model intrinsics. We add a little more:

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

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

34 A link state is a set of link traversals.

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

```
value
  n:N
type
  [32]  LT′ = HI × LI × HI
  [33]  LT = {|lt:LT′•wfLT(lt)(n)|}
  [34]  LΣ′ = LT-set
  [34]  LΣ = {|lσ:LΣ′•wf_LΣ(lσ)(n)|}
  [35]  LΩ′ = LΣ-set
  [35]  LΩ = {|lω:LΩ′•wf_LΩ(lω)(n)|}
value
  [33]  wfLT: LT → N → Bool
  [33]  wfLT(hi,li,hi′)(n) ≡
  [33]    ∃ h,h′:H•{h,h′}⊆ωHs(n)∧
  [33]      ωHI(h)=hi∧ωHI(h′)=hi′∧
  [33]      li ∈ ωLIs(h)∧li ∈ ωLIs(h′)
```

The wf_LΣ and wf_LΩ can be defined like the corresponding functions for hub states and hub state spaces. ■ 6

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 7. ***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:

36 There are three colours: **red**, **yellow** and **green**.

37 Each hub traversal is extended with a colour and so is the hub state.

38 There is a notion of time interval.

39 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.

40 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
  [36] Colour == red | yellow | green
  [37] X = LI×HI×LI×Colour [crossings of a hub]
  [37] HΣ = X-set [hub states]
  [38] TI [time interval]
  [39] Signalling = (HΣ × TI)*
  [40] Semaphore = (HΣ × HΣ) -m~-> Signalling
value
  [37] ωHΣ: H → HΣ
  [40] ωSemaphore: H → Sema,
  [41] chg_HΣ: H × HΣ → H
  [41] chg_HΣ(h,hσ) as h'
  [41]   pre hσ ∈ ωHΩ(h) post ωHΣ(h')=hσ
  [39] chg_HΣ_Seq: H × HΣ → H
  [39] chg_HΣ_Seq(h,hσ) ≡
  [39]   let sigseq = (ωSemaphore(h))(ωΣ(h),hσ) in
  [39]   sig_seq(h)(sigseq) end
  [39] sig_seq: H → Signalling → H
  [39] sig_seq(h)(sigseq) ≡
  [39]   if sigseq=⟨⟩ then h else
  [39]   let (hσ,tδ) = hd sigseq in let h' = chg_HΣ(h,hσ);
  [39]   wait tδ;
  [39]   sig_seq(h')(tl sigseq) end end end
```

■ 7

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 (simple, one step behaviours): 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 8. ***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:

41 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.

42 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. ■ 8

We omit exemplification of regulations.

3.5 Scripts, Licenses and Contracts

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

Example 9. ***Timetable Scripts:***

43 Time is considered discrete. Bus lines and bus rides have unique names (across any set of time tables).

44 A *T*ime*T*able associates *B*us *L*ine *Id*entifiers (*blid*) to sets of *Journies*.

45 *Journies* are designated by a pair of a *BusRoute* and a set of *BusRides*.

46 A *BusRoute* is a triple of the *Bus Stop* of origin, a list of zero, one or more intermediate *Bus Stop*s and a destination *Bus Stop*.

47 A set of *BusRides* associates, to each of a number of *B*us *Id*entifiers (*bid*) a *Bus Sched*ule.

48 A *Bus Sched*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.

49 A *Bus Stop* (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.

50 A *Frac*tion is a **Real** properly between 0 and 1.

51 The *Journies* must be *w*ell_*f*ormed in the context of some net.

52 A set of journies is well-formed if

- 53 the bus stops are all different,
- 54 a bus line is embedded in some line of the net, and
- 55 all defined bus trips of a bus line are equivalent.

```
type
  [43] T, BLId, BId
  [44] TT = BLId -m~-> Journies
  [45] Journies' = BusRoute × BusRides
  [46] BusRoute = BusStop × BusStop* × BusStop
  [47] BusRides = BId -m~-> BusSched
  [49] BusSched = T × T* × T
  [50] BusStop == mkBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
  [51] Frac = {|r:Real•0<r<1|}
  [45] Journies = {|j:Journies'•∃ n:N • wf_Journies(j)(n)|}
value
  [52] wf_Journies: Journies → N → Bool
  [52] wf_Journies((bs1,bsl,bsn),js)(hs,ls) ≡
  [53]   diff_bus_stops(bs1,bsl,bsn) ∧
  [54]   is_net_embedded_bus_line(⟨bs1⟩^bsl^⟨bsn⟩)(hs,ls) ∧
  [55]   commensurable_bus_trips((bs1,bsl,bsn),js)(hs,ls)
```

■ 9

Timetables are used in the next example.

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 license (contract) actions incur moral obligations (respectively legal responsibilities).

Example 10. ***Contracts:*** An example contract can be 'schematised':

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.

Concrete examples of actions can be schematised:

(a) cid: **conduct bus ride** (blid,bid) **to start at time** t
(b) cid: **cancel bus ride** (blid,bid) **at time** t
(c) cid: **insert bus ride like** (blid,bid) **at time** t

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

(d) cid: **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 PageSlide .

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

We omit formalising the semantics of these syntaxes. A formalisation could be expressed (in CSP [14]) with each bus, each licensee (and licensor), time and the road net bus traffic being processes, etc. ■ 10

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 notions of *strategic*, *tactical* and *operational funds*, $\mathbb{F}_{\mathcal{S},\mathcal{T},\mathcal{O}}$, and other *resources*, $\mathbb{R}$, a notion of *contexts*, $\mathbb{C}$, and a notion of *states*, $\mathbb{S}$. Contexts bind resources to bindings from locations to disjoint time intervals (allocation and scheduling), states bind resource identifiers to resource values. Simplified types of the strategic, tactical and operational actions are now of the following types: executive functions apply to contexts, states and funds and obtain and redistribute funds; strategic functions apply to contexts and strategic funds and create new contexts and states and consume some funds; tactical functions apply to resources, contexts, states tactical funds and create new contexts while consuming some tactical funds; etcetera.

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

Example 11. ***Management:*** We relate to Example 10:

56 The **conduct, cancel** and **insert bus ride** actions are operational functions.

57 The actual **subcontract** actions are tactical functions;

58 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.

59 Forming new timetables, in consort with the contractor, is a strategic function.

We omit formalisations. ■ 11

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 our actions and responding to events.

Example 12. ***Human Behaviour:*** Cf. Examples 10–11:

60 no failures to conduct a bus ride must be classified as diligent;

61 rare failures to conduct a bus ride must be classified as sloppy if no technical reasons were the cause;

62 occasional failures ··· as delinquent;

63 repeated patterns of failures ··· as criminal.

We omit showing somewhat "lengthy" formalisations. ■ 12

3.8 Discussion

We have briefly outlined six concepts of domain facets 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 basic description is approximately 30 pages; for "small" parts of railway systems we easily get up to 100–200 pages – both including formalisations. 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. 5.

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.

We make a distinction between goals and requirements. Goals are what we expect satisfied by the software implemented from the requirements. Goals are usually expressed in terms of properties. Requirements can then be proved to satisfy the $\mathcal{G}$oals: $\mathcal{D}, \mathcal{R} \models \mathcal{G}$. [21, Lamsweerde] focus on goals. We shall assume that the (goal and) requirements engineer elicit both $\mathcal{G}$oals and $\mathcal{R}$equirements from requirements stakeholders. But we shall focus only on domain and interface requirements such as "derived" from domain descriptions.

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 the required 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. The goals help us formulate the BPR prescriptions.

Example 13. ***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).

64 **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 15 brings more details.)

65 **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.

66 **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.

67 **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 — with receipt 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).

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

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

70 **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.

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

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

73 **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". ■ 13

4.2 Domain Requirements

For the phase of domain requirements the requirements stakeholders "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 possibly, i.e., most likely regularly updated). Domain requirements are "derived" from the domain description as covered in Items [1–5]. The goals then determine the derivations: which projections, instantiations, determinations, etcetera, to perform.

[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 14. ***Projection:*** Our requirements is for a simple toll road: a linear sequence of links and hubs outlined in Example 13: see Items [1–11 of Example 1 and Items [32–35] of Example 6. ■ 14

[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 15. ***Instantiation:*** Here the toll road net topology as outlined in Example 13 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
  $\mathsf{N}' = (\mathsf{H} \times \mathsf{L}) \times \mathsf{H} \times ((\mathsf{L} \times \mathsf{L}) \times \mathsf{H} \times (\mathsf{H} \times \mathsf{L}))^*$
  $\mathsf{N}'' = \{|\mathsf{n}{:}\mathsf{N}' \bullet \mathsf{wf}(\mathsf{n})|\}$
value
  wf_N'': $\mathsf{N}' \rightarrow \mathbf{Bool}$
  wf_N''((h,l),h',llhpl) ≡ ... 6 lines ... !
  αN: $\mathsf{N}'' \rightarrow \mathsf{N}$
  αN((h,l),h',llhpl) ≡ ... 2 lines ... !

wf_N″ secures linearity; αN allows abstraction from more concrete N″ to more abstract N. ■ 15

[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 16. ***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
  LΣ = (HI×HI)-**set**, LΩ = LΣ-**set**
  HΣ = (LI×LI)-**set**, HΩ = HΣ-**set**
  N′ = (H × L) × H × ((L × L) × H × (H × L))*
value
  ωLΣ: L → LΣ, ωLΩ: L → LΩ
  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**

■ 16

[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 are now are not only feasible but also computable in reasonable time.

Example 17. ***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. ■ 17

[5] **Fitting:** By *domain requirements fitting* we understand an operation which takes two or more (say n) domain requirements prescriptions, d_{r_i}, that are claimed to share entities, actions, events and/or behaviours and map these into $n+1$ domain requirements prescriptions, δ_{r_i}, where one of these, $\delta_{r_{n+1}}$ capture the shared phenomena and concepts and the other n prescriptions, δ_{r_i}, 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 $\delta_{r_{n+1}}$, prescribe interfaces between δ_{r_i} and $\delta_{r_{n+1}}$ for $i : \{1..n\}$.

Example 18. ***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 here suggested 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. ■ 18

Discussion: This section has 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 respectively. Again goals help state which phenomena and concepts are to be shared.

[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-dialogues.

Example 19. ***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. ■ 19

[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 stakeholder) in order to proceed in its computation.

Example 20. ***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.* ■ *20*

[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 21. ***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. ■ 21

[4] **Behaviour Interfaces:** By a shared behaviour we understand a sequence of zero, one or more shared actions and shared events.

Example 22. ***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 more toll 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 21. ■ 22

Discussion: The discussion of Sect. 4.2 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

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 dependability seems to be subjectable to rigorous, formal treatment. We refer to [4, Vol. 3, Part V, Chap. 19, Sect. 19.6] for an extensive (30 page) survey.

The **discussions** of Sects. 4.2 and 4.3 carry over to this paragraph. That is, once the machine has been installed it, the machine, is part of the new domain !

5. DISCUSSION

We discuss a number of issues that were left open above. [1] **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 stakeholder identification and liaison, domain and, to some extents also requirements, especially goal acquisition and analysis, terminologisation, and techniques for domain and requirements and goal validation and [goal] verification ($\mathcal{D}, \mathcal{R} \models \mathcal{G}$). We refer, instead, to [4, Vol.3, Part IV (Chaps. 9, 12–14) and Part V (Chaps. 18, 20–23)]. [2] **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 [25]. 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. [3] **"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 stakeholder ideas". 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. [4] **What Can Be Described and Prescribed:** The issue of *"what can be described"* has been a constant challenge to philosophers. In [24] Russell covers his first *Theory of Descriptions* (stemming from the early 1900s), and in [23] a revision, as *The Philosophy of Logical Atomism*. The issue is not that straightforward. In [6, 7] 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 the introduction of *observer* functions and by axioms 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] **What Have We Achieved – and What Not:** Section 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 [5, ICTAC 2007]. [6] **Relation to Other Work:** The most obvious 'other' work is that of [19, Problem Frames]. In [19] 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 [21, Axel van

Lamsweerde] appears to represent the most definitive work on Requirements Engineering today. Much of its requirements and goal acquisition and analysis techniques carries over to main aspects of domain acquisition and analysis techniques and the goal-related techniques of [21] apply to determining which projections, instantiation, determination and extension operations to perform on domain descriptions. [7] **"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 controlled, contained iterated and "spiralling" manner and such that it is at all times clear which development step is what: $\mathcal{D}$, $\mathcal{R}$ or $\mathcal{S}$! [8] **Description Languages:** We have used the RSL specification language, [11, 4], for the formalisations of this report, but any of the model-oriented approaches and languages offered by Alloy [17], Event B [1], RAISE [12], VDM [10] and Z [26], 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 [22], CSP [14], MSC [15], Statecharts [13], and/or some temporal logic, for example either DC [27] or TLA+ [20]. Research into how such diverse textual and diagrammatic languages can be combined is ongoing [3]. [9] $\mathcal{D}, \mathcal{S} \models \mathcal{R}$**:** 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 enables proofs that the software is right and the formalisation of the "derivation" of $\mathcal{R}$equirements from $\mathcal{D}$omain specifications help ensure that it is the right software [9]. [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 the triptych 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 [6] above.

6. CONCLUSION

We have put forward the methodological steps of a different approach to requirements engineering than currently 'en vogue'. We claim that our approach, as it follows from the **dogma** expressed in Sect. 1, is logical, that is, follows as a necessity. Although our "derivation" of requirements from domain descriptions differ from current requirements development approaches the triptych approach, in fact, integrates most of their techniques "smoothly" into either domain engineering or the triptych requirements engineering. The 'triptych' approach has been in partial use since the early 1990s, including at the United Nations University's International Institute for Software Technology (www.iist.unu.edu [2]). This paper presents this triptych is a clearer form than presented in [4]. 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.

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 5 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 [8] covers ASM, B/event B, CafeObj, CASL, DC, RAISE, TLA+, VDM and Z.

[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] B. Aichernig and T. Maibaum, editors. *UNU-IIST 10^{th} Anniversary Colloquium on Formal Methods at the Crossroads, from Panacea to Foundational Support; Lisboa, Portugal*, volume 2757 of *Lecture Notes in Computer Science*. Springer, March 2002.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] D. Bjørner and M. C. Henson, editors. *Logics of Specification Languages*. EATCS Series, Monograph in Theoretical Computer Science. Springer, Heidelberg, Germany, 2008.

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

[10] J. S. Fitzgerald and P. G. Larsen. *Developing Software using VDM-SL*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 1RU, England, 1997.

[11] 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.

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

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

[14] T. 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).

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

[16] D. Jackson. A Direct Path to Dependable Software. *CACM: Communications of the ACM*, 52(4):78–88, April 2009.

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

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

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

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

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

[22] W. Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Institut für Informatik, Humboldt Universität zu Berlin, Unter den Linden 6, 10099 Berlin, Germany, 1 Oktober 2009. 276 pages. http://www2.informatik.hu-berlin.de/top/pnene_buch/pnene_buch.pdf.

[23] 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.

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

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

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

[27] 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.

Compiled: December 30, 2009: 08:20 ECT

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