Rôle of Domain Engineering in Software Development
and Why Current Requirements Engineering is Flawed!

Dines Bjørner

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

Abstract. We introduce the notion of domain descriptions (D) in order to ensure that software (S) is right and is the right software, that is, that it is correct with respect to written requirements (R) and that it meets customer expectations (D).
That is, before software can be designed (S) we must make sure we understand the requirements (R), and before we can express the requirements we must make sure that we understand the application domain (D): the area of activity of the users of the required software, before and after installment of such software.
We shall outline what we mean by informal, narrative and formal domain description, and how one can systematically, albeit not (in fact: never) automatically go from domain descriptions to requirements prescriptions.
As it seems that domain engineering is a relatively new discipline within software engineering we shall mostly focus on domain engineering and discuss its necessity.
The talk will show some formulas but they are really not meant to be read by the speaker, let alone understood, during the talk, by the listeners. They are merely there to bring home the point: Professional software engineering, like other professional engineering branches rely on and use mathematics.
And it is all very simple to learn and practise anyway !
We end this paper with, to some, perhaps, controversial remarks: Requirements engineering, as pursued today, researched, taught and practised, is outdated, is thus fundamentally flawed. We shall justify this claim.

1 The Software Development Dogma

1.1 The Dogma

The dogma is this: Before software can be designed we must understand the requirements. Before requirements can be finalised we must have understood the domain.

We assume that the reader knows what is meant by software design and requirements. But what do we mean by "the domain" ?

1.2 What Do We Mean by 'Domain'?

By a domain we shall loosely understand an 'area' of natural or human activity, or both, where the 'area' is "well-delineated" such as, for example, for physics: mechanics or electricity or chemistry or hydrodynamics; or for an infrastructure component: banking, railways, hospital health-care, "the market": consumers, retailers, wholesalers, producers and the distribution chain.

By a *domain* we shall thus, less loosely, understand a universe of discourse, small or large, a structure (i) of entities, that is, of "things", individuals, particulars some of which are designated as state components; (ii) of functions, say over entities, which when applied become possibly state-changing actions of the domain; (iii) of events, possibly involving entities, occurring in time and expressible as predicates over single or pairs of (before/after) states; and (iv) of behaviours, sets of possibly interrelated sequences of actions and events.

1.3 Dialectics

Now, let's get this "perfectly" straight ! Can we develop software requirements without understanding the domain ? Well, how much of the domain should we understand ? And how well should we understand it ?

Can we develop software requirements without understanding the domain ? No, of course we cannot ! But we, you, do develop software for hospitals (railways, banks) without understanding health-care (transportation, the financial markets) anyway ! In other engineering disciplines professionalism is ingrained: Aeronautics engineers understand the domain of aerodynamics; naval architects (i.e., ship designers) understand the domain of

hydrodynamics; telecommunications engineers understand the domain of electromagnetic field theory; and so forth.

Well, how much of the domain should we understand ? A basic answer is this: enough for us to understand formal descriptions of such a domain.

This is so in classical engineering: Although the telecommunications engineer has not herself researched and made mathematical models of electromagnetic wave propagation in the form of Maxwell's equations: Gauss's Law for Electricity, Gauss's Law for Magnetism, Faraday's Law of Induction, Ampéres Law:

$$\oint \vec{E} \cdot d\vec{A} = \frac{q}{\varepsilon_0} \quad \oint \vec{B} \cdot d\vec{A} = 0 \quad \oint \vec{E} \cdot \vec{ds} = -\frac{d\Phi_B}{dt} \quad \oint \vec{B} \cdot \vec{ds} = \mu_0 i + \frac{1}{c^2}\frac{\partial}{\partial t}\int \vec{E} \cdot d\vec{A}$$

the telecommunications engineer certainly understands these laws.

And how well should we understand it ? Well, enough, as an engineer, to manipulate the formulas, to further develop these for engineering calculations.

1.4 Conclusion

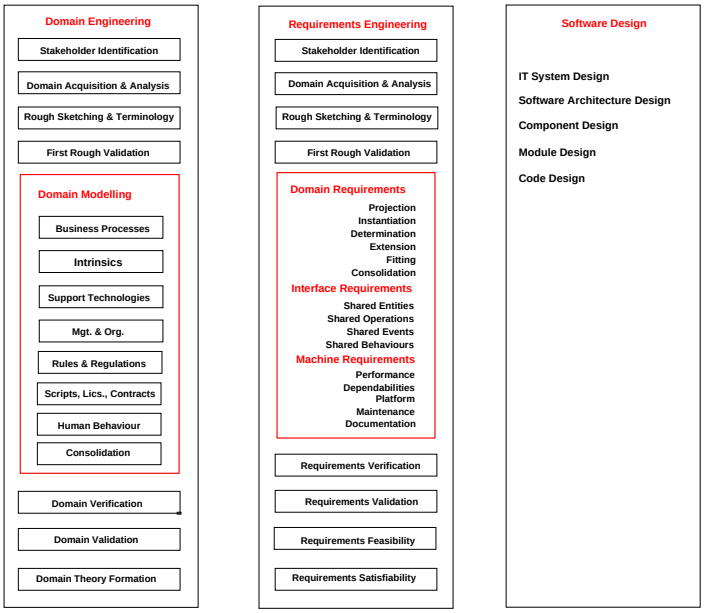
It is about time that software engineers consult precise descriptions, including formalisations of the application domains for software.

These domain models may have to be developed by computing scientists. Software engineers then "transform" these into requirements prescriptions and software designs.

2 The Triptych of Software Development

We recall the dogma: before software can be designed we must understand the requirements. Before requirements can be finalised we must have understood the domain.

We conclude from that, that an "ideal" software development proceeds, in three major development phases, as follows:



2.1 The Phase Results

- **Domain engineering**: The results of domain engineering include a domain model: a description, both informal, as a precise narrative, and formal, as a specification.
- **Requirements engineering**: The results of requirements engineering include a requirements model: a prescription, both informal, as a precise narrative, and formal, as a specification.
- **Software design**: The results of software design include executable code and all documentation that goes with it.

2.2 Relations to "Reality" and Phase Interrelations

- **Domain engineering**: The domain is described as it is.
- **Requirements engineering** The requirements are described as we would like the software to be, and the requirements must be clearly related to the domain description.
- **Software design** The software design specification must be correct with respect to the requirements.

2.3 Technicalities: An Overview

Domain Engineering Section 3 outlines techniques of domain engineering. But just as a preview: Based on extensive domain acquisition and analysis an informal and a formal domain model is established, a model which is centered around sub-models of: intrinsics, supporting technologies, management and organisation, rules and regulations, script [or contract] languages and human behaviours, which are then validated and verified.

Requirements Engineering Section 4 outlines techniques of requirements engineering. But just as a preview:Based on presentations of the domain model to requirements stakeholders requirements can now be "derived" from the domain model and as follows: First a domain requirements model is arrived at: projection of the domain model, instantiation of the domain model, determination of the domain model, extension of the domain model and fitting of several, separate domain requirements models; then an interface requirements model, and finally a machine requirements model. These are simultaneously verified and validated and the feasibility and satisfiability of the emerging model is checked.

Software Design We do not cover techniques of software design in detail — so only this summary. From the requirements prescription one develops, in stages and steps of transformation ("refinement"), first the system architecture, then the program (code) organisation (structure), and then, in further steps of development, the component design, the module design and the code. These stages and step can be verified, model checked and tested with respect to the previous phase of requirements prescription, respectively the previous software design stages and steps. One can then assert that the $\mathcal{S}$oftware design is correct with respect to the $\mathcal{R}$equirements in the context of the assumptions expressed about the $\mathcal{D}$omain:

$$\mathcal{D},\ \mathcal{S}\ \models\ \mathcal{R}$$

3 Domain Engineering

We shall focus only on the actual modelling, thus omitting any treatment of the preparatory administrative and informative work, the identification of and liaison with domain stakeholders, the domain acquisition and analysis, and the establishment of a domain terminology (document). So we go straight to the descriptive work. We first illustrate the ideas of modelling domain phenomena and concepts in terms of simple entities, operations, events and behaviours, then we model the domain in terms of domain facets. Also, at then end, we do not have time and paper space for any treatment of domain verification, domain validations and the establishment of a domain theory.

3.1 Simple Entities, Operations, Events and Behaviours

Without discussing our specification ontology, that is, the principles according to which we view the world around us, we just present the decomposition of phenomena and concepts into simple entities, operations, events and behaviours. All of these are "first class citizens", that is, are entities.

We now illustrate examples of each of these ontological categories.

Simple Entities A *simple entity* is something that has a distinct, separate existence, though it need not be a material existence, to which we apply functions. With simple entities we associate attributes, i.e., properties modelled as types and values. Simple entities can be considered either continuous or discrete, and, if discrete then either atomic or composite. It is the observer (that is, the specifier) who decides whether to consider a simple entity to be atomic or composite. Atomic entities cannot meaningfully be decomposed into sub-entities, but atomic entities may be analysed into (Cartesian) "compounds" of properties, that is, attributes. Attributes have name, type and value. Composite entities can be meaningfully decomposed into sub-entities, which are

entities. The composition of sub-entities into a composite entity "reveals" the, or a mereology of the composite entity: that is, how it is "put together".

Example 1: Transport Entities: Nets, Links and Hubs — Narrative

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) identifiers (which can be observed from the hub [respectively link]).

Example 2: Transport Entities: Nets, Links and Hubs — Formalisation

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
4a obs_HI: H → HI, obs_LI: L → LI
axiom
4b ∀ h,h′:H, l,l′:L • h≠h′⇒obs_HI(h)≠obs_HI(h′) ∧ l≠l′⇒obs_LI(l)≠obs_LI(l′)

Operations By an *operation* we shall understand something which when *applied* to some entities, called the *arguments* of the operation, *yields* an entity, called the *result* of the operation application (also referred to as the operation invocation). Operations have signatures, that is, can be grossly described by the Cartesian type of its arguments and the possibly likewise compounded type of its results. Operations may be total over their argument types, or may be just partial. We shall consider some acceptable operations as "never terminating" processes. We shall, for the sake of consistency, consider all operation invocations as processes (terminating or non-terminating), and shall hence consider all operationdefinitions as also designating process definitions.

We shall also use the term **function** to mean the same as the term operation.

By a *state* we shall loosely understand a collection of one or more simple entities whose value may change. By an *action* we shall understand an operation application which applies to and/or yields a state.

Example 3: Link Insertion Operation

5. To a net one can insert a new link in either of three ways:
   (a) Either the link is connected to two existing hubs — and the insert operation must therefore specify the new link and the identifiers of two existing hubs;
   (b) or the link is connected to one existing hub and to a new hub — and the insert operation must therefore specify the new link, the identifier of an existing hub, and a new hub;
   (c) or the link is connected to two new hubs — and the insert operation must therefore specify the new link and two new hubs.
   (d) From the inserted link one must be able to observe identifier of respective hubs.
6. From a net one can remove a link. The removal command specifies a link identifier.

type
5 Insert == Ins(s_ins:Ins)
5 Ins = 2xHubs | 1x1nH | 2nHs
5a 2xHubs == 2oldH(s_hi1:HI,s_l:L,s_hi2:HI)
5b 1x1nH == 1oldH1newH(s_hi:HI,s_l:L,s_h:H)
5c 2nHs == 2newH(s_h1:H,s_l:L,s_h2:H)

axiom
5d ∀ 2oldH(hi′,l,hi″):Ins • hi′≠hi″ ∧ obs_LIs(l)={hi′,hi″} ∧
∀ 1old1newH(hi,l,h):Ins • obs_LIs(l)={hi,obs_HI(h)} ∧
∀ 2newH(h′,l,h″):Ins • obs_LIs(l)={obs_HI(h′),obs_HI(h″)}

7. If the Insert command is of kind 2newH(h',l,h") then the updated net of hubs and links, has
   – the hubs hs joined, ∪, by the set {h′,h″} and
   – the links ls joined by the singleton set of {l}.
8. If the Insert command is of kind 1oldH1newH(hi,l,h) then the updated net of hubs and links, has
   8.1 : the hub identified by hi updated, hi′, to reflect the link connected to that hub.
   8.2 : The set of hubs has the hub identified by hi replaced by the updated hub hi′ and the new hub.
   8.2 : The set of links augmented by the new link.
9. If the Insert command is of kind 2oldH(hi',l,hi") then
9.1–.2 : the two connecting hubs are updated to reflect the new link,
   9.3 : and the resulting sets of hubs and links updated.

int_Insert(op)(hs,ls) ≡
$\star_i$ **case** op **of**
7 2newH(h′,l,h″) → (hs ∪ {h′,h″},ls ∪ {l}),
8 1oldH1newH(hi,l,h) →
8.1 **let** h′ = aLI(xtr_H(hi,hs),obs_LI(l)) **in**
8.2 (hs\{xtr_H(hi,hs)}∪{h,h′},ls ∪{l}) **end**,
9 2oldH(hi′,l,hi″) →
9.1 **let** hsδ = {aLI(xtr_H(hi′,hs),obs_LI(l)),
9.2 aLI(xtr_H(hi″,hs),obs_LI(l))} **in**
9.3 (hs\{xtr_H(hi′,hs),xtr_H(hi″,hs)}∪ hsδ,ls ∪{l}) **end**
$\star_j$ **end** $\star_k$ **pre** pre_int_Insert(op)(hs,ls)

Events Informally, by an *event* we shall loosely understand the occurrence of "something" that may either trigger an action, or is triggered by an action, or alter the course of a behaviour, or a combination of these.

An *event* can be characterised by a predicate, p and a pair of ("before") and ("after") of pairs of states and times: $p((t_b, \sigma_b), (t_a, \sigma_a))$. Usually the time interval $t_a - t_b$ is of the order $t_a \simeq (t_b) + \delta_{\text{tiny}}$.

Example 4: Transport Events

(i) A link, for some reason "ceases to exist"; for example: a bridge link falls down, or a level road link is covered by a mud slide, or a road tunnel is afire, or a link is blocked by some vehicle accident. (ii) A vehicle enters or leaves the net. (iii) A hub is saturated with vehicles.

Behaviours By a *behaviour* we shall informally understand a strand of (sets of) actions and events. In the context of domain descriptions we shall speak of behaviours whereas, in the context of requirements prescriptions and software designs we shall use the term processes.

By a *behaviour* we, more formally, understand a sequence, q of actions and/or events q_1, q_2, ..., q_i, q_{i+1}, ..., q_n such that the state resulting from one such action, q_i, or in which some event, q_i, occurs, becomes the state in which the next action or event, q_{i+1}, if it is an action, is effected, or, if it is an event, is the event state.

Example 5: Transport: Traffic Behaviour

10. There are further undefined vehicles.
11. Traffic is a discrete function from a 'Proper subset of Time' to pairs of nets and vehicle positions.
12. Vehicles positions is a discrete function from vehicles to vehicle positions.

type
10 Veh

11 TF = Time $\overrightarrow{m}$ (N × VehPos)
12 VehPos = Veh $\overrightarrow{m}$ Pos

13. There are positions, and a position is either on a link or in a hub.
    (a) A hub position is indicated just by a triple: the identifier of the hub in question, and a pair of (from and to) link identifiers, namely of links connected to the identified hub.
    (b) A link position is identified by a quadruplet: The identifier of the link, a pair of hub identifiers (of the link connected hubs), designating a direction, and a real number, properly between 0 and 1, denoting the relative offset from the from hub to the to hub.

type
13 Pos = HPos | LPos
13a) HPos == hpos(s_hi:HI,s_fli:LI,s_tli:LI)
13b) LPos == lpos(s_li:HI,s_fhi:LI,s_tli:LI,s_offset:Frac)
13b) Frac = {|r:**Real**•0<r<1|}

3.2 Domain Facets

By a domain facet we mean one amongst a finite set of generic ways of analysing a domain: a view of the domain, such that the different facets cover conceptually different views, and such that these views together cover the domain

We shall postulate the following domain facets: intrinsics, support technologies, management & organisation, rules & regulations, script languages [contract languages] and human behaviour. Each facet covers simple entities, operations, events and behaviours.

We shall now illustrate these.

Intrinsics By *domain intrinsics* we mean those phenomena and concepts of a domain which are basic to any of the other facets (listed earlier and treated, in some detail, below), with such domain intrinsics initially covering at least one specific, hence named, stakeholder view.

Example 6: Intrinsics, I

The links, hubs, hence the nets, and the identifiers of links and hubs are intrinsic phenomena, respectively concepts.

So are:

Example 7: Intrinsics, II

14. From any link of a net one can observe the two hubs to which the link is connected.
    (a) We take this 'observing' to mean the following: From any link of a net one can observe the two distinct identifiers of these hubs.
15. From any hub of a net one can observe the one or more links to which are connected to the hub.
    (a) Again: by observing their distinct link identifiers.
16. Extending Item 14: the observed hub identifiers must be identifiers of hubs of the net to which the link belongs.
17. Extending Item 15: the observed link identifiers must be identifiers of links of the net to which the hub belongs.

value
14a obs_HIs: L → HI-**set**,
15a obs_LIs: H → LI-**set**,
axiom
14b ∀ l:L • **card** obs_HIs(l)=2 ∧
15b ∀ h:H • **card** obs_LIs(h)=1 ∧

```
  ∀ (hs,ls):N •
  14a)  ∀ h:H • h ∈ hs ⇒ ∀ li:LI • li ∈ obs_LIs(h) ⇒
          ∃ l′:L • l′ ∈ ls ∧ li=obs_LI(l′) ∧ obs_HI(h) ∈ obs_HIs(l′) ∧
  15a)  ∀ l:L • l ∈ ls ⇒
          ∃ h′,h″:H • {h′,h″}⊆hs ∧ obs_HIs(l)={obs_HI(h′),obs_HI(h″)}
  16  ∀ h:H • h ∈ hs ⇒ obs_LIs(h) ⊆ iols(ls)
  17  ∀ l:L • l ∈ ls ⇒ obs_HIs(h) ⊆ iohs(hs)
value
  iohs: H-set → HI-set, iols: L-set → LI-set
  iohs(hs) ≡ {obs_HI(h)|h:H•h ∈ hs}
  iols(ls) ≡ {obs_LI(l)|l:L•l ∈ ls}
```

Support Technologies By *domain support technologies* we mean ways and means of concretesing certain observed (abstract or concrete) phenomena or certain conceived concepts in terms of (possibly combinations of) human work, mechanical, hydro mechanical, thermo-mechanical, pneumatic, aero-mechanical, electro-mechanical, electrical, electronic, telecommunication, photo/opto-electric, chemical, etc. (possibly computerised) sensor, actuator tools.

In this example of a support technology we shall illustrate an abstraction of the kind of semaphore signalling one encounters at road intersections, that is, hubs. The example is indeed an abstraction: we do not model the actual "machinery" of road sensors, hub-side monitoring & control boxes, and the actuators of the green/yellow/red sempahore lamps. But, eventually, one has to, all of it, as part of domain modelling.

Example 8: Hub Sempahores

To model signalling we need to model hub and link states.

A hub (link) state is the set of all traversals that the hub (link) allows. A hub traversal is a triple of identifiers: of the link from where the hub traversal starts, of the hub being traversed, and of the link to where the hub traversal ends. A link traversal is a triple of identifiers: of the hub from where the link traversal starts, of the link being traversed, and of the hub to where the link traversal ends.

A hub (link) state space is the set of all states that the hub (link) may be in. A hub (link) state changing operation can be designated by the hub and a possibly new hub state (the link and a possibly new link state).

```
type
  LΣ′ = L_Trav-set
  L_Trav = (HI × LI × HI)
  LΣ = {| lnkσ:LΣ′ • syn_wf_LΣ{lnkσ} |}
  HΣ′ = H_Trav-set
  H_Trav = (LI × HI × LI)
  HΣ = {| hubσ:HΣ′ • wf_HΣ{hubσ} |}
  HΩ = HΣ-set, LΩ = LΣ-set
value
  obs_LΣ: L → LΣ, obs_LΩ: L → LΩ
  obs_HΣ: H → HΣ, obs_HΩ: H → HΩ
axiom
  ∀ h:H • obs_HΣ(h) ∈ obs_HΩ(h) ∧ ∀ l:L • obs_LΣ(l) ∈ obs_LΩ(l)
value
  chg_HΣ: H × HΣ → H, chg_LΣ: L × LΣ → L
  chg_HΣ(h,hσ) as h′
    pre hσ ∈ obs_HΩ(h) post obs_HΣ(h′)=hσ
  chg_LΣ(l,lσ) as l′
    pre lσ ∈ obs_LΩ(h) post obs_HΣ(l′)=lσ
```

Well, so far we have indicated that there is an operation that can change hub and link states. But one may debate whether those operations shown are really examples of a support technology. (That is, one could equally well claim that they remain examples of intrinsic facets.) We may accept that and then ask the question: How to effect the described state changing functions ? In a simple street crossing a semaphore

does not instantaneously change from red to green in one direction while changing from green to red in the cross direction. Rather there is are intermediate sequences of, for example, not necessarily synchronised green/yellow/red and red/yellow/green states to help avoid vehicle crashes and to prepare vehicle drivers. Our "solution" is to modify the hub state notion.

type
  Colour == red | yellow | green
  X = LI×HI×LI×Colour [crossings **of** a hub]
  HΣ = X-**set** [hub states]
value
  obs_HΣ: H → HΣ, xtr_Xs: H → X-**set**
  xtr_Xs(h) ≡
    {(li,hi,li′,c)|li,li′:LI,hi:HI,c:Colour•{li,li′}⊆obs_LIs(h)∧hi=obs_HI(h)}
axiom
  ∀ n:N,h:H • h ∈ obs_Hs(n) ⇒ obs_HΣ(h)⊆xtr_Xs(h) ∧
    ∀ (li1,hi2,li3,c),(li4,hi5,li6,c′):X •
      {(li1,hi2,li3,c),(li4,hi5,li6,c′)}⊆obs_HΣ(h) ∧
      li1=li4 ∧ hi2=hi5 ∧ li3=li6 ⇒ c=c′

We consider the colouring, or any such scheme, an aspect of a support technology facet. There remains, however, a description of how the technology that supports the intermediate sequences of colour changing hub states.

We can think of each hub being provided with a mapping from pairs of "stable" (that is non-yellow coloured) hub states (hσ_i,hσ_f) to well-ordered sequences of intermediate "un-stable' (that is yellow coloured) hub states paired with some time interval information $\langle(h\sigma', t\delta'), (h\sigma'', t\delta''), \ldots, (h\sigma'^{\cdots\prime}, t\delta'^{\cdots\prime})\rangle$ and so that each of these intermediate states can be set, according to the time interval information,[1] before the final hub state (hσ_f) is set.

type
  TI [time interval]
  Signalling = (HΣ × TI)*
  Sema = (HΣ × HΣ) $\xrightarrow[m]{}$ Signalling
value
  obs_Sema: H → Sema, chg_HΣ: H × HΣ → H, chg_HΣ_Seq: H × HΣ → H
  chg_HΣ(h,hσ) **as** h′ **pre** hσ ∈ obs_HΩ(h) **post** obs_HΣ(h′)=hσ
  chg_HΣ_Seq(h,hσ) ≡
    let sigseq = (obs_Sema(h))(obs_Σ(h),hσ) **in** sig_seq(h)(sigseq) **end**

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

Management and Organisation

Management By *domain management* we mean people (i) who determine, formulate and thus set standards (cf. rules and regulations, a later lecture topic) concerning strategic, tactical and operational decisions; (ii) who ensure that these decisions are passed on to (lower) levels of management, and to "floor" staff; (iii) who make sure that such orders, as they were, are indeed carried out; (iv) who handle undesirable deviations in the carrying out of these orders cum decisions; and (v) who "backstop" complaints from lower management levels and from floor staff.

Organisation By *domain organisation* we mean the structuring of management and non-management staff levels; the allocation of strategic, tactical and operational concerns to within management and non-management staff levels; and hence the "lines of command": who does what and who reports to whom — administratively and functionally.

Examples Formalisation of the next example is found in Sect. 3.2, Pages 13–15.

Example 9: Bus Transport Management & Organisation

In Sect. 3.2, Pages 13–15, we illustrate what is there called a contract language. "Programs" in that language are either contracts or are orders to perform the actions permitted or obligated by contracts. The language in question is one of managing bus traffic on a net. The **management & organisation** of bus traffic involves contractors issuing contracts, contractees acting according to contracts, busses (owned or leased) by contractees, and the bus traffic on the (road) net. Contractees, i.e., bus operators, `"start"` buses according to a contract timetable, `"cancel"` buses if and when deemed necessary, `"insert"` rush-hour and other buses if and when deemed necessary, and, acting as contractors, `"sub-contract"` sub-contractees to operate bus lines, for example, when the issuing contractor is not able to operate these bus lines, i.e., not able to fulfill contractual obligations, due to unavailability of buses or staff. Clearly the programs of bus contract languages are "executed" according to **management** decisions and the sub-contracting "hierarchy" reflects **organisational** facets.

Rules and Regulations Human stakeholders act in the domain, whether clients, workers, managers, suppliers, regulatory authorities, or other. Their actions are guided and constrained by rules and regulations. These are sometimes implicit, that is, not "written down". But we can talk about rules and regulations as if they were explicitly formulated.

The main difference between rules and regulations is that rules express properties that must hold and regulations express state changes that must be effected if rules are observed broken.

Rules and regulations are directed not only at human behaviour but also at expected behaviours of support technologies.

Rules and regulations are formulated by enterprise staff, management or workers, and/or by business and industry associations, for example in the form of binding or guiding national, regional or international standards[2], and/or by public regulatory agencies.

Domain Rules By a *domain rule* we mean some text which prescribes how people or equipment are expected to behave when dispatching their duty, respectively when performing their functions.

Domain Regulations By a *domain regulation* we mean some text which prescribes what remedial actions are to be taken when it is decided that a rule has not been followed according to its intention.

Two Informal Examples The two informal examples will be followed up by sketches of formalisation.

Example 10: Trains at Stations: Available Station Rule and Regulation

- Rule: In China the arrival and departure of trains at, respectively from, railway stations is subject to the following rule:

  In any three-minute interval at most one train may either arrive to or depart from a railway station.
- Regulation: *If it is discovered that the above rule is not obeyed*, then there is some regulation which prescribes administrative or legal management and/or staff action, as well as some correction to the railway traffic.

[2] Viz.: ISO (International Organisation for Standardisation, www.iso.org/iso/home.htm), CENELEC (European Committee for Electrotechnical Standardization, www.cenelec.eu/Cenelec/Homepage.htm), etc.

Example 11: Trains Along Lines: Free Sector Rule and Regulation

- Rule: In many countries railway lines (between stations) are segmented into blocks or sectors. The purpose is to stipulate that if two or more trains are moving along the line, then:
  There must be at least one free sector (i.e., without a train) between any two trains along a line.
- Regulation: *If it is discovered that the above rule is not obeyed*, then there is some regulation which prescribes administrative or legal management and/or staff action, as well as some correction to the railway traffic.

A Formal Example We shall develop the above example (11, Page 10) into a partial, formal specification. That is, not complete, but "complete enough" for the reader to see what goes on.

Example 12: Continuation of Example 11 Page 10

We start by analysing the text of the rule and regulation. The rule text: *There must be at least one free sector (i.e., without a train) between any two trains along a line.* contains the following terms: free (a predicate), sector (an entity), train (an entity) and line (an entity). We shall therefore augment our formal model to reflect these terms. We start by modelling sectors and sector descriptors, lines and train position descriptors, we assume what a train is,, and then we model the predicate free.

```
type
  Sect′ = H × L × H,
  SectDescr = HI × LI × HI
  Sect = {|(h,l,h′):Sect′ • obs_HIs(l)={obs_HI(h),obs_HI(h′)}|}
  SectDescr = {|(hi,li,hi′):SectDescr′ •
                  ∃ (h,l,j′):Sect•obs_HIs(l)={obs_HI(h),obs_HI(h′)}|}
  Line′ = Sect*,
  Line = {|line:Line′•wf_Line(line)|}
  TrnPos′ = SectDescr*
  TrnPos = {|trnpos′:TrnPos′•∃ line:Line•conv_Line_to_TrnPos(line)=trnpos′|}
value
  wf_Line: Line′ → Bool
  wf_Line(line) ≡
    ∀ i:Nat • {i,i+1}⊆inds(line) ⇒
      let (_,l,h)=line(i),(h′,l′,_)=line(i+1) in h=h′ end
  conv_Line_to_TrnPos: Line → TrnPos
  conv_Line_to_TrnPos(line) ≡
    ⟨(obs_HI(h),obs_LI(l),obs_HI(h′))|1≤i≤len line∧line(i)=(h,l,h′)⟩
```

The function lines yield all lines of a net.

```
value
  lines: N → Line-set
  lines(hs,ls) ≡
    let lns = {⟨(h,l,h′)⟩|h,h′:H,l:L•proper_line((h,l,h′),(hs,ls))}
              ∪ {ln⌢ln′|ln,l′:Line•{ln,ln′}⊆lns∧adjacent(ln,ln′)} in
    lns end
```

The function lines makes use of an auxiliary function:

```
  adjacent: Line × Line → Bool
  adjacent((_,l,h),(h′,l′,_)) ≡ h=h′
    pre {obs_LI(l),obs_LI(l′)}⊆ obs_LIs(h)
```

We reformulate traffic in terms of train positions.

```
type
  TF = T m→ (N × (TN m→ TrnPos))
```

We formulate a necessary property of traffic, namely that its train positions correspond to actual lines of the net.

```
value
  wf_TF: TF → Bool
  wf_TF(tf) ≡
    ∀ t:T•t ∈ dom tf ⇒
      let ((hs,ls),trnposs) = tf(t) in
      ∀ trn:TN • trn ∈ dom trnposs ⇒
        ∃ line:Line • line ∈ lines(hs,ls) ∧
          trnposs(trn) = conv_Line_to_TrnPos(line) end
```

Nothing prevents two or more trains from occupying overlapping train positions. They have "merely" – and regrettably – crashed. But such is the domain. So wf_TF(tf) is not part of an axiom of traffic, merely a desirable property.

```
value
  has_free_Sector: TN × T → TF → Bool
  has_free_Sector(trn,(hs,ls),t)(tf) ≡
    let ((hs,ls),trnposs) = tf(t) in
    (trn ∉ dom trnposs ∨ (tn ∈ dom trnposs(t) ∧
     ∃ ln:Line • ln ∈ lines(hs,ls) ∧
       is_prefix(trnposs(trn),ln))(hs,ls)) ∧
       ∼∃ trn′:TN • trn′ ∈ dom trnposs ∧ trn′≠trn ∧
         trnposs(trn′)=conv_Line_to_TrnPos(⟨follow_Sect(ln)(hs,ls)⟩)
    end
    pre exists_follow_Sect(ln)(hs,ls)

  is_prefix: Line × Line → N → Bool
  is_prefix(ln,ln′)(hs,ls) ≡ ∃ ln″:Line • ln″ ∈ lines(hs,ls) ∧ ln⌢ln″=ln′
```

The test ln″ ∈ lines(hs,ls) in the definition of is_prefix is not needed for the cases where that function is invoked as only shown here.

The function follow_Sect yields the sector following the argument line, if such a sector exists.

```
  exists_follow_Sect: Line → Net → Bool
  exists_follow_Sect(ln)(hs,ls) ≡
    ∃ ln′:Line•ln′ ∈ lines(hs,ls)∧ln⌢ln′ ∈ lines(hs,ls)
    pre ln ∈ lines(hs,ls)
  follow_Sect: Line → Net ∼→ Sect
  follow_Sect(ln)(hs,ls) ≡
    let ln′:Line•ln′ ∈ lines(hs,ls)∧ln⌢ln′ ∈ lines(hs,ls) in hd ln′ end
    pre line ∈ lines(hs,ls)∧exists_follow_Sect(ln)(hs,ls)
```

We doubly recursively define a function free_sector_rule(tf)(r). tf is that part of the traffic which has yet to be "searched" for non-free sectors. Thus tf is "counted" up from a first time t till the traffic tf is empty. That is, we assume a finite definition set tf . r is like a traffic but without the net. Initially r is the empty traffic. r is "counted" up from "earliest" cases of trains with no free sector ahead of them. The recursion stops, for a given time when there are no more train positions to be "searched" for that time; and when the "to-be-searched" traffic is empty.

```
type
```

```
  TNPoss = T m→ (TN → TrnPos)
value
  free_sector_rule: TF × TF → TNPoss
  free_sector_rule(tf)(r) ≡
    if tf=[ ] then r else
    let t:T•t ∈ dom tf∧smallest(t)(tf) in
    let ((hs,ls),trnposs)=tf(t) in
    if trnposs=[ ] then free_sector_rule(tf\{t})(r) else
    let tn:TN•tn ∈ dom trnposs in
    if exists_follow_Sect(trnposs(tn))(hs,ls)∧∼has_free_Sector(tn,(hs,ls),t)(tf)
      then
        let r′ = if t ∈ dom r then r else r ∪ [t↦[ ]] end in
        free_sector_rule(tf†[t↦((hs,ls),trnposs\{tn})])
                        (r†[t↦r(t)∪[tn↦trnposs(tn)]]) end
      else
        free_sector_rule(tf†[t↦((hs,ls),trnposs\{trn})])(r)
    end end end end end end

  smallest(t)(tf) ≡ ∼∃ t′:T• t′isin dom tf∧t′<t pre t ∈ dom tf
```

Script Languages [Contract Languages] By a *domain* script *language* we mean the definition of a set of licenses and actions where these licenses when issued and actions when performed have morally obliging power.

By a *domain* contract *language* a domain script language whose licenses and actions have legally binding power, that is, their issuance and their invocation may be contested in a court of law.

A Script Language Some common, visual forms of bus timetables are shown in Fig. 1.

Fig. 1. Some bus timetables: Spain, India and Norway

The next examples exemplify narrative and formal description of syntax of bus timetables as well as formal description of semantics of bus timetables.

Example 13: Narrative Syntax of a Bus Timetable Script Language

18. Time is a concept covered earlier. Bus lines and bus rides have unique names (across any set of time tables). Hub and link identifiers, HI, LI, were treated from the very beginning.
19. A TimeTable associates to Bus Line Identifiers a set of Journies.
20. Journies are designated by a pair of a BusRoute and a set of BusRides.
21. A BusRoute is a triple of the Bus Stop of origin, a list of zero, one or more intermediate Bus Stops and a destination Bus Stop.
22. A set of BusRides associates, to each of a number of Bus Identifiers a Bus Schedule.
23. A Bus Schedule a triple of the initial departure Time, a list of zero, one or more intermediate bus stop Times and a destination arrival Time.

24. A Bus Stop (i.e., its position) is a Fraction of the distance along a link (identified by a Link Identifier) from an identified hub to an identified hub.
25. A Fraction is a **Real** properly between 0 and 1.
26. The Journies must be well_formed in the context of some net.

Example 14: Formal Syntax of a Bus Timetable Script Language

type
18. T, BLId, BId
19. TT = BLId $\overrightarrow{m}$ Journies
20. Journies′ = BusRoute × BusRides
21. BusRoute = BusStop × BusStop* × BusStop
22. BusRides = BId $\overrightarrow{m}$ BusSched
23. BusSched = T × T* × T
24. BusStop == mkBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
25. Frac = {|r:**Real**•0<r<1|}
26. Journies = {|j:Journies′•∃ n:N • wf_Journies(j)(n)|}

Example 15: Semantics of a Bus Timetable Script Language

type
  Bus
value
  obs_X: Bus → X
type
  BusTraffic = T $\overrightarrow{m}$ (N × (BusNo $\overrightarrow{m}$ (Bus × BPos)))
  BPos = atHub | onLnk | atBS
  atHub == mkAtHub(s_fl:LIs_hi:HI,s_tl:LI)
  onLnk == mkOnLnk(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
  atBSt == mkAtBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
  Frac = {|r:**Real**•0<r<1|}
value
  gen_BusTraffic: TT → BusTraffic-**infset**
  gen_BusTraffic(tt) **as** btrfs
    post ∀ btrf:BusTraffic • btrf ∈ btrfs ⇒ on_time(btrf)(tt)

We omit definition of several functions, including the interesting on_time predicate.

A Contract Language We shall, as for the timetable script, just hint at a contract language.

Example 16: Informal Syntax of Bus Transport Contracts

An example contract can be 'schematised':

con_id: **contractor** corn **contracts contractee** ceen
  to perform operations `"start","cancel","insert","subcontract"`
  with respect to bus timetable tt.

Example 17: Formal Syntax of a Bus Transport Contracts

type
  CId, CNm

Contract = CId × CNm × CNm × Body
Body = Op-**set** × TT
Op == "conduct" | "cancel" | "insert" | "subcontract"

an example contract:

(cid,cor,cee,({"start","cancel","insert","subcontract"},tt))

Example 18: Informal Syntax of a Bus Transport Actions

Example actions can be schematised:

(a) cid: **start bus ride** (blid,bid) **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 (Page 13) shown earlier is almost like an action; here is the action form:

(d) cid: **contractee** cee **is granted a license** cid′
to perform operations {"start","cancel","insert",subcontract"}
with respect to timetable tt′.

Example 19: Formal Syntax of a Bus Transport Actions

type
Action = CNm × CId × (SubLic | SmpAct) × Time
SmpAct = Start | Cancel | Insert
DoRide == mkSta(s_blid:BLId,s_bid:BId)
Cancel == mkCan(s_blid:BLId,s_bid:BId)
Insert = mkIns(s_blid:BLId,s_bid:BId)
SubCon == mkCon(s_cid:ConId,s_cee:CNm,s_body:(s_ops:Op-**set**,s_tt:TT))

examples:
(a) (cee,cid,mkRid(blid,id),t)
(b) (cee,cid,mkCan(blid,id),t)
(c) (cee,cid,mkIns(blid,id),t)
(d) (cee,cid,mkCon(cid′,({"start","cancel","insert","subcontract"},tt′),t))

where: cid′ = generate_ConId(cid,cee,t)

Example 20: Semantics of a Bus Transport Contract Language: States

type
Body = Op-**set** × TT
ConΣ = RcvConΣ×SubConΣ×CorBusΣ
RcvConΣ = CNm $\overrightarrow{m}$ (CId $\overrightarrow{m}$ (Body×TT))
SubConΣ = CNm $\overrightarrow{m}$ (CId $\overrightarrow{m}$ Body)
BusNo
BusΣ = FreeBusesΣ × ActvBusesΣ × BusHistsΣ
FreeBusesΣ = BusStop $\overrightarrow{m}$ BusNo-**set**
ActvBusesΣ = BusNo $\overrightarrow{m}$ BusInfo
BusInfo = BLId×BId×CId×CNm×BusTrace
BusHistsΣ = Bno $\overrightarrow{m}$ BusInfo*

BusTrace = (Time×BusStop)*
CorBusΣ = CNm $\overrightarrow{m}$ (CId $\overrightarrow{m}$ ((BLId×BId) $\overrightarrow{m}$ (BNo×BusTrace)))
AllBs=CNm $\overrightarrow{m}$ BusNo-**set**

Example 21: Semantics of a Bus Transport Contract Language: Constants and Functions

value
cns:CNm-**set**, busnos:BNo-**set**, ibσ:IBΣs=CNm $\overrightarrow{m}$ BusΣ,
rcor,icee:CNm • rcor $\notin$ cns$\wedge$icee $\in$ cns, itr:BusTraffic,
rcid:ConId, iops:Op-**set**={"subcontract"}, itt:TT, t_0:Time
allbs:AllBs • **dom** allbs=cns $\cup$ {rcor}$\wedge\cup$ **rng** allbs=busnos,
icon:Contract=(rcid,rcor,icee,(iops,itt)),
icσ:ConΣ=([icee $\mapsto$ [rcid $\mapsto$ [icee $\mapsto$ icon]]]
$\cup$ [cee $\mapsto$ [ ] | cee:CNm • cee $\in$ cnms\{icee}],[ ],[ ]),
system: **Unit** $\rightarrow$ **Unit**
system() $\equiv$
cntrcthldr(icee)(ilσ(icee),ibσ(icee))
‖(‖{cntrcthldr(cee)(ilσ(cee),ibσ(cee))|cee:CNm•cee $\in$ cns\{icee}})
‖(‖{bus_ride(b,cee)(rcor,"nil")
| cee:CNm,b:BusNo•cee $\in$ **dom** allbs $\wedge$ b $\in$ allbs(cee)})
‖time_clock(t_0) ‖ bus_traffic(itr)

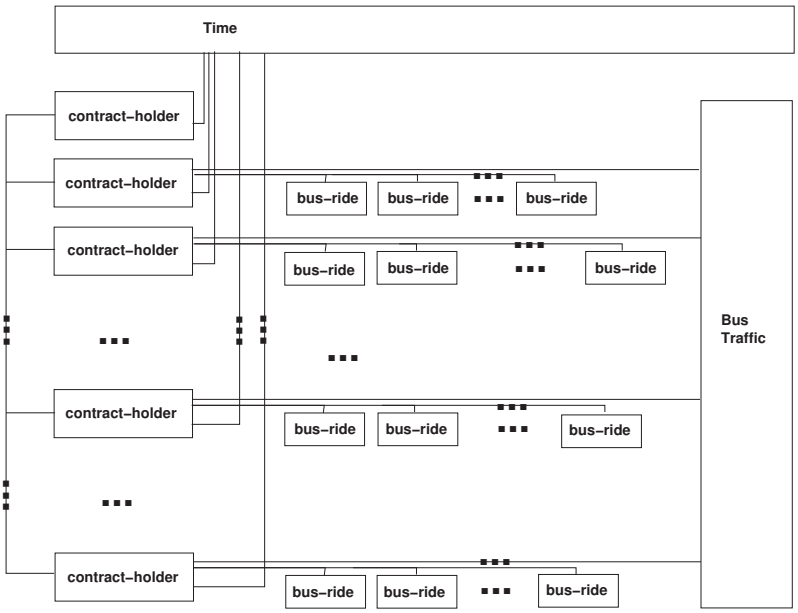


Fig. 2. An organisation

The thin lines of Fig. 2 denote communication "channels".

Human Behaviour By *human behaviour* we mean any of a quality spectrum of carrying out assigned work: from (i) **careful, diligent** and **accurate**, via (ii) **sloppy** dispatch, and (iii) **delinquent** work, to (iv) outright **criminal** pursuit.

Example 22: A Diligent Operation

The int_Insert operation of Page 5 was expressed without stating necessary pre-conditions:

27. The insert operation takes an Insert command and a net and yields either a new net or **chaos** for the case where the insertion command "is at odds" with, that is, is not semantically well-formed with respect to the net.
28. We characterise the "is not at odds", i.e., is semantically well-formed, that is: pre_int_Insert(op)(hs,ls), as follows: it is a propositional function which applies to Insert actions, op, and nets, (hs.ls), and yields a truth value if the below relation between the command arguments and the net is satisfied.
    Let (hs,ls) be a value of type N.
29. If the command is of the form 2oldH(hi′,l,hi′) then
    ⋆1 hi′ must be the identifier of a hub in hs,
    ⋆2 l must not be in ls and its identifier must (also) not be observable in ls, and
    ⋆3 hi″ must be the identifier of a(nother) hub in hs.
30. If the command is of the form 1oldH1newH(hi,l,h) then
    ⋆1 hi must be the identifier of a hub in hs,
    ⋆2 l must not be in ls and its identifier must (also) not be observable in ls, and
    ⋆3 h must not be in hs and its identifier must (also) not be observable in hs.
31. If the command is of the form 2newH(h′,l,h″) then
    ⋆1 h′ — left to the reader as an exercise (see formalisation !),
    ⋆2 l — left to the reader as an exercise (see formalisation !), and
    ⋆3 h″ — left to the reader as an exercise (see formalisation !).

```
value
  28′ pre_int_Insert: Ins → N → Bool
  28″ pre_int_Insert(Ins(op))(hs,ls) ≡
  ⋆2   s_l(op)∉ ls ∧ obs_LI(s_l(op)) ∉ iols(ls) ∧
         case op of
  29       2oldH(hi′,l,hi″) → {hi′,hi″}⊆iohs(hs),
  30       1oldH1newH(hi,l,h) → hi ∈ iohs(hs)∧h∉ hs∧obs_HI(h)∉ iohs(hs),
  31       2newH(h′,l,h″) → {h′,h″}∩ hs={}∧{obs_HI(h′),obs_HI(h″)}∩ iohs(hs)={}
         end
```

These must be carefully expressed and adhered to in order for staff to be said to carry out the link insertion operation accurately.

Example 23: A Sloppy via Delinquent to Criminal Operation

We replace systematic checks (∧) with partial checks (∨), etcetera, and obtain various degrees of sloppy to delinquent, or even criminal behaviour.

```
value
  28′ pre_int_Insert: Ins → N → Bool
  28″ pre_int_Insert(Ins(op))(hs,ls) ≡
  ⋆2   s_l(op)∉ ls ∧ obs_LI(s_l(op)) ∉ iols(ls) ∧
         case op of
  29       2oldH(hi′,l,hi″) → hi′ ∈ iohs(hs)∨hi″isin iohs(hs),
  30       1oldH1newH(hi,l,h) → hi ∈ iohs(hs)∨h∉ hs∨obs_HI(h)∉ iohs(hs),
  31       2newH(h′,l,h″) → {h′,h″}∩ hs={}∨{obs_HI(h′),obs_HI(h″)}∩ iohs(hs)={}
         end
```

Dialectics So now you should have a practical and technical "feel" for domain engineering: What it takes to express a domain model.

But there is lots' more: We have not shown you (i) the rôle of domain stakeholders: (i.1) how to identify them, (i.2) how to involve them and (i.3) how they help validate resulting domain descriptions. (ii) the domain (ii.1) knowledge acquisition and (ii.2) analysis processes, (ii) the domain (ii.1) model verification and (ii.2) validation and processes, and (iii) the domain theory R&D process.

Can we agree that we cannot, as professional software engineers, start on gathering requirements, let alone prescribing these before we have understood the domain ? Can we agree that, "ideally", we must therefore first R&D the domain model before we can embark on any requirements prescription process ?

By "ideally" we mean the following: Ideally domain engineering should fully precede requirements engineering, but for many practical reasons[3] we must co-develop domain descriptions "hand-in-hand" with requirements prescriptions. And that is certainly feasible, when done with care. So we shall, for years assume this to be the case.

Pragmatics While the software industry "humps along": co-developing domain descriptions and requirements with their clients, or, for COTS, with their marketing departments, private and public research centres should and will embark on large scale (5–8 manyears/year), long range projects (5–8 year) foundational research and development (R&D) of infrastructure component domain models of the financial service industry: banking (all forms); insurance (all forms); portfolio management; securities trading: brokers, traders, commodities and stock etc. exchanges; transportation: road, rail, air, and sea; healthcare: physicians, hospitals, clinics, pharmacies, etc.; "the market": consumers, retailers, wholesalers, and the supply chain; etcetera.

3.3 Further on the Modelling of Domains

[8] Part IV, Chaps. 8–16 covers techniques of domain modelling.

4 Requirements Engineering

We cannot possibly, within the confines of a seminar talk and a reasonably sized paper cover, however superficially, both informal and formal examples of requirements engineering.

Instead we shall just briefly mention the major stages and sub-stages of requirements modeling:

- **Domain Requirements:** those which can be expressed sôlely using terms from the domain description;
- **Interface Requirements:** those which can be expressed using terms both from the domain description and from IT; and
- **Machine Requirements:** those which can be expressed sôlely using terms from IT.

IEEE Definition of Requirements

By IT requirements we understand (cf. IEEE Standard 610.12): *"A condition or capability needed by a user to solve a problem or achieve an objective on a computing machine"*.

By computing *machine* we shall understand a, or the, combination of computer (etc.) *hardware* and *software* that is the target for, or result of the required computing systems development.

4.1 Domain Requirements

Domain Requirements

By *domain requirements* we mean such which can be expressed sôlely using terms from the domain description

To construct the domain requirements the domain engineer together with the various groups of requirements stakeholder "apply" the following "domain-to-requirements" operations to a copy of the domain description: **projection**, **instantiation**, **determination**, **extension** and **fitting**. First we briefly charaterise these.

[3] Among the many practical reasons for not first fully developing a domain model are: (a) it takes literally "ages" to develop a complete domain model, (b) in fact one will never achieve complete domain models, and (c) software houses and their clients cannot wait for this software!

The Domain-to-Requirements Operations The 'domain-to-requirements' operations cannot be automated. They increasingly "turn" the copy of the domain description into a domain requirements prescription.

- **Projection** removes, from that emerging requirements document all the domain phenomena and concepts for which the customer does not need IT support.
- **Instantiation** makes a number of entities: *simple, operations, events and behaviours*, less abstract, more concrete.
- **Determination** makes the emerging requirements entities more determinate, that is, removes undesired non-determinism.
- **Extension** introduces new, computable entities that were not possible in the non-IT domain.
- **Fitting** merges the domain requirements prescription with those of other, more-or-less independent IT developments.

4.2 Interface Requirements

Interface Requirements

By *interface requirements* we mean such which those which can be expressed using terms from both the domain description and from IT, that is, terminology of hardware and of software.

When phenomena and concepts of the domain are also to be represented by the machine, these phenomena and concepts are said to be shared between the domain and the machine; the requirements therefore need be expressed both in terms of phenomena and concepts of the domain and in terms of phenomena and concepts of the machine.

Shared Phenomena and Concepts A shared phenomenon or concept is either a simple entity, an operation, an event or a behaviour.

Shared simple entities need to be initially input to the machine and their machine representation need to be regularly, perhaps real-time refreshed.

Shared operations need to be interactively performed by human or other agents of the domain and by the machine.

Shared events are shared in the sense that their occurrence in the domain must be made known to the machine.

Shared behaviours need to occur in the domain and in the machine by alternating means, that is, a protocol need be devised.

For each of these four kinds of interface requirements the requirements engineers work with the requirements stakeholders to determine the properties of these forms of sharing. These interface requirements are then narrated and formalised. They are always "anchored" in specific items of the domain description.

4.3 Machine Requirements

Machine Requirements

By *machine requirements* we mean those which can be expressed sôlely using terms from the machine, that is, terminology of hardware and of software.

We shall not cover any principles or techniques for developing machine requirements, but shall just list the very many issues that must be captured by a machine requirements.

- Performance
  - Storage
  - Time
  - Software Size
- Dependability
  - Accessibility
  - Availability
  - Reliability
  - Robustness
  - Safety
  - Security
- Maintenance
  - Adaptive
  - Corrective
  - Perfective
  - Preventive
- Platform (P)
  - Development P
  - Demonstration P
  - Execution P
  - Maintenance P
- Documentation Requirements
- Other Requirements

The machine requirements are usually not so easily, formalised, if at all, with today's specification language tools. Extra great care must therefore be exerted in their narration. Some formal modelling calculations, like fault (tree) analysis, can be made in order to justify quantitative requirements.

4.4 Further on the Modelling of Requirements

[8] Part V, Chaps. 17–24 covers techniques of requirements modelling.

5 Why "Current" Requirements Engineering (RE) is Flawed

Current, conventional RE has no scientific basis: "My" RE starts with a domain model. It provides the scientific basis. "Derivation" of domain and interface requirement provides a further scientific basis. The separation of concerns: domain model, in-and-by-itself, and the requirements projection, instantiation, determination, extension and fitting operators provide a basis for scientific analysis. Current, conventional RE does not have these bases. Current, research into and practice of conventional RE "must be stopped" if we are to pursue Software Engineering in a professionally responsible manner.

6 Conclusion

6.1 Summary — A Wrap Up

We have illustrated the triptych concept: from domains via requirements to software. We spent most time on domain engineering. We just sketched major requirements engineering concepts. And we assumed you know how to turn formal requirements into correct software designs !

6.2 Dialectics

So, are we clear on this: (i) that we must understand the domain before we express the requirements; (ii) that we can "derive" major parts of the requirements prescription from the domain description; (iii) that domains are far more "stable" than requirements; (iv) that prescribing requirements with no prior domain description is thoroughly unsound; (v) that describing [prescribing] domains [requirements] both informally (narratives) and formally (formal specifications) helps significantly towards consistent specifications; and (vi) that we must therefore embrace the triptych: from domains via requirements to software.

Implication: Theory-work So, get on with it ! Pick up one or another of the new domain engineering ideas: business processes, facets, domain theories, etc., or the new requirements engineering ideas: projection, instantiation, determination, extension and fitting, research them, write papers about it.

Implication: Engineering-work — Extrovert Applications But do it in connection with real life, actual domains: banking, insurance, stock exchange and brokerage, hospitalisation, bus & tax transport, rail transport, container line shipping, etcetera. That is, "build" some impressive domain theories !

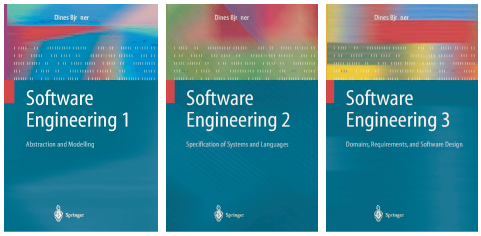
Implication: Engineering-work — Introspective Applications By introspective applications we mean such as providing software for, or such as the Internet, the Web, operating systems database management, data communication, etcetera, etcetera, Also these are lack proper domain descriptions.

6.3 For More on Domain and Requirements Engineering

For details on domain and requirements engineering we refer to:

Software Engineering:

– *Vol. 3: Domains, Requirements and Software Design*, XXX+766 pages.
Texts in Theoretical Computer Science, EATCS Series, 2006 Springer



and the upcoming book:

Software Engineering,
– *Vol. I: The Triptych Approach,*
– *Vol. II: A Model Development.*
To be submitted to Springer for evaluation, expected published 2009.

This book (draft) is the basis for lectures at Techn. Univ. of Graz, Austria Nov.-Dec. 2008; Univ. of Saarland, Germany March 2009; and Univ. of Tokyo, Japan Fall (Oct.-Nov.) 2009.

6.4 For More on Extrovert Applications

We refer to some indicative Internet-based reports:

– air traffic `www.imm.dtu.dk/~db/brisbane.pdf` and `/airtraffic.pdf`;
– container line industry: `www.imm.dtu.dk/~db/container-paper.pdf`;
– the 'Market': `www.imm.dtu.dk/~db/themarket.pdf`;
– IT security: `www.imm.dtu.dk/~db/5lectures/it-system-security-ISO.pdf`;
– oil industry: Appendix: `www.imm.dtu.dk/~db/de-p.pdf`;
– railways: `www.railwaydomain.org/`;
– transportation (in general): Appendix: `www.imm.dtu.dk/~db/tseb.pdf`;
– etcetera.

6.5 Introvert Applications

Software Engineering Archeology In general I would prefer to see precise domain models of the Internet, the Web, 'Cloud Computing', Windows Vista, Linux and idealised SQL[4] as the basis for requirements and software that claim that they are "based" on the Internet, the Web, 'Cloud Computing', Windows Vista, Linux and/or SQL.

Here is clearly a fascination engineering task.

I see the Internet as an instantiation of 'Cloud Computing'.

6.6 For More on Research Topics

A number of research topics of domain theory has been outlined in:

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

Acknowledgements. Thanks for inviting me to PSI'09. Indeed, very many THANKS.

[4] By idealised SQL I mean an SQL where relations are indeed sets, and hence that all results of SQL queries are sets. To my knowledge ORACLE SQL does not satisfy this simple property, but the FRONTBASE SQL92 system does (http://www.frontbase.com/cgi-bin/WebObjects/FrontBase)

7 Bibliographical Notes

Specification languages, techniques and tools, that cover the spectrum of domain and requirements specification, refinement and verification, are dealt with in Alloy: [43], ASM: [60, 61], B/event B: [1, 16], CafeOBJ: [22, 23, 18, 19], CSP [38, 63, 64, 39], DC [68, 69] (Duration Calculus), Live Sequence Charts [17, 34, 45], Message Sequence Charts [40–42], RAISE [25, 27, 6–8, 24] (RSL), Petri nets [44, 55, 58, 57, 59], Statecharts [30, 31, 33, 35, 32], Temporal Logic of Reactive Systems [48, 49, 54, 56], TLA+ [46, 47, 50, 51] (Temporal Logic of Actions), VDM [11, 12, 21, 20], and Z [65–67, 37, 36]. Techniques for integrating "different" formal techniques are covered in [2, 28, 14, 13, 62]. The recent book on Logics of Specification Languages [10] covers ASM, B/event B, CafeObj, CASL, DC, RAISE, TLA+, VDM and Z.

References

1. Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings.* Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, England, 1996.
2. Keijiro Araki, Andy Galloway, and Kenji Taguchi, editors. *IFM 1999: Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, York, UK, June 1999. Springer. Proceedings of 1st Intl. Conf. on IFM.
3. Dines Bjørner. Programming in the Meta-Language: A Tutorial. In Dines Bjørner and Cliff B. Jones, editors, *The Vienna Development Method: The Meta-Language, [11]*, LNCS, pages 24–217. Springer–Verlag, 1978.
4. Dines Bjørner. Software Abstraction Principles: Tutorial Examples of an Operating System Command Language Specification and a PL/I-like On-Condition Language Definition. In Dines Bjørner and Cliff B. Jones, editors, *The Vienna Development Method: The Meta-Language, [11]*, LNCS, pages 337–374. Springer–Verlag, 1978.
5. Dines Bjørner. The Vienna Development Method: Software Abstraction and Program Synthesis. In *Mathematical Studies of Information Processing*, volume 75 of *LNCS*. Springer–Verlag, 1979. Proceedings of Conference at Research Institute for Mathematical Sciences (RIMS), University of Kyoto, August 1978.
6. Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling.* Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
7. Dines Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages.* Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. Chapters 12–14 are primarily authored by Christian Krog Madsen.
8. Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design.* Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
9. Dines Bjørner. Domain Theory: Practice and Theories, Discussion of Possible Research Topics. In *ICTAC'2007*, volume 4701 of *Lecture Notes in Computer Science (eds. J.C.P. Woodcock et al.)*, pages 1–17, Heidelberg, September 2007. Springer.
10. Dines Bjørner and Martin C. Henson, editors. *Logics of Specification Languages — see [61, 16, 19, 52, 29, 24, 51, 20, 36].* EATCS Monograph in Theoretical Computer Science. Springer, Heidelberg, Germany, 2008.
11. Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer–Verlag, 1978. This was the first monograph on *Meta-IV*. [3–5].
12. Dines Bjørner and Cliff B. Jones, editors. *Formal Specification and Software Development.* Prentice-Hall, 1982.
13. Eerke A. Boiten, John Derrick, and Graeme Smith, editors. *IFM 2004: Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, London, England, April 4-7 2004. Springer. Proceedings of 4th Intl. Conf. on IFM. ISBN 3-540-21377-5.
14. Michael J. Butler, Luigia Petre, and Kaisa Sere, editors. *IFM 2002: Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, Turku, Finland, May 15-18 2002. Springer. Proceedings of 3rd Intl. Conf. on IFM. ISBN 3-540-43703-7.
15. Dominique Cansell and Dominique Méry. Logical Foundations of the B Method. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [60, 18, 53, 26, 50, 37] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
16. Dominique Cansell and Dominique Méry. *Logics of Specification Languages*, chapter The event-B Modelling Method: Concepts and Case Studies, pages 47–152 in [10]. Springer, 2008.
17. Werner Damm and David Harel. LSCs: Breathing life into Message Sequence Charts. *Formal Methods in System Design*, 19:45–80, 2001. Early version appeared as Weizmann Institute Tech. Report CS98-09, April 1998. An abridged version appeared in *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-based Distributed Systems* (FMOODS'99), Kluwer, 1999, pp. 293–312.
18. Ražvan Diaconescu, Kokichi Futatsugi, and Kazuhiro Ogata. CafeOBJ: Logical Foundations and Methodology. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [60, 15, 53, 26, 50, 37] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
19. Răzvan Diaconescu. *Logics of Specification Languages*, chapter A Methodological Guide to the CafeOBJ Logic, pages 153–240 in [10]. Springer, 2008.
20. John S. Fitzgerald. *Logics of Specification Languages*, chapter The Typed Logic of Partial Functions and the Vienna Development Method, pages 453–487 in [10]. Springer, 2008.

21. John S. Fitzgerald and Peter Gorm Larsen. *Developing Software using VDM-SL*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 1RU, England, 1997.
22. K. Futatsugi, A.T. Nakagawa, and T. Tamai, editors. *CAFE: An Industrial–Strength Algebraic Formal Method*, Sara Burgerhartstraat 25, P.O. Box 211, NL–1000 AE Amsterdam, The Netherlands, 2000. Elsevier. Proceedings from an April 1998 Symposium, Numazu, Japan.
23. Kokichi Futatsugi and Razvan Diaconescu. *CafeOBJ Report The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. AMAST Series in Computing – Vol. 6. World Scientific Publishing Co. Pte. Ltd., 5 Toh Tuck Link, SINGAPORE 596224. Tel: 65-6466-5775, Fax: 65-6467-7667, E-mail: wspc@wspc.com.sg, 1998.
24. Chris George and Anne E. Haxthausen. *Logics of Specification Languages*, chapter The Logic of the RAISE Specification Language, pages 349–399 in [10]. Springer, 2008.
25. Chris W. George, Peter Haff, Klaus Havelund, Anne Elisabeth Haxthausen, Robert Milne, Claus Bendix Nielsen, Søren Prehn, and Kim Ritter Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.
26. Chris W. George and Anne E. Haxthausen. The Logic of the RAISE Specification Language. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [60, 15, 18, 53, 50, 37] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
27. Chris W. George, Anne Elisabeth Haxthausen, Steven Hughes, Robert Milne, Søren Prehn, and Jan Storbank Pedersen. *The RAISE Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.
28. Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors. *IFM 2000: Integrated Formal Methods*, volume of *Lecture Notes in Computer Science*, Schloss Dagstuhl, Germany, November 1-3 2000. Springer. Proceedings of 2nd Intl. Conf. on IFM.
29. Michael R. Hansen. *Logics of Specification Languages*, chapter Duration Calculus, pages 299–347 in [10]. Springer, 2008.
30. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
31. David Harel. On visual formalisms. *Communications of the ACM*, 33(5), 514–530 1988.
32. David Harel and Eran Gery. Executable object modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.
33. David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark B. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *Software Engineering*, 16(4):403–414, 1990.
34. David Harel and Rami Marelly. *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
35. David Harel and Amnon Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.
36. Martin C. Henson, Moshe Deutsch, and Steve Reeves. *Logics of Specification Languages*, chapter Z Logic and Its Applications, pages 489–596 in [10]. Springer, 2008.
37. Martin C. Henson, Steve Reeves, and Jonathan P. Bowen. Z Logic and its Consequences. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [60, 15, 18, 53, 26, 50] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
38. Tony Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985.
39. Tony Hoare. Communicating Sequential Processes. Published electronically: `http://www.usingcsp.com/-cspbook.pdf`, 2004. Second edition of [38]. See also `http://www.usingcsp.com/`.
40. ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992.
41. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1996.
42. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1999.
43. Daniel Jackson. *Software Abstractions Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.
44. Kurt Jensen. *Coloured Petri Nets*, volume 1: Basic Concepts (234 pages + xii), Vol. 2: Analysis Methods (174 pages + x), Vol. 3: Practical Use (265 pages + xi) of *EATCS Monographs in Theoretical Computer Science*. Springer–Verlag, Heidelberg, 1985, revised and corrected second version: 1997.
45. Jochen Klose and Hartmut Wittke. An automata based interpretation of Live Sequence Charts. In T. Margaria and W. Yi, editors, *TACAS 2001*, LNCS 2031, pages 512–527. Springer-Verlag, 2001.
46. Leslie Lamport. The Temporal Logic of Actions. *Transactions on Programming Languages and Systems*, 16(3):872–923, 1995.
47. Leslie Lamport. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA, 2002.
48. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive Systems: Specifications*. Addison Wesley, 1991.
49. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive Systems: Safety*. Addison Wesley, 1995.
50. Stephan Merz. On the Logic of TLA+. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [60, 15, 18, 53, 26, 37] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.

51. Stephan Merz. *Logics of Specification Languages*, chapter The Specification Language TLA$^+$, pages 401–451 in [10]. Springer, 2008.
52. T. Mossakowski, A. Haxthausen, D. Sannella, and A. Tarlecki. *Logics of Specification Languages*, chapter CASL – the Common Algebraic Specification Language, pages 241–298 in [10]. Springer, 2008.
53. Till Mossakowski, Anne E. Haxthausen, Don Sanella, and Andzrej Tarlecki. CASL — The Common Algebraic Specification Language: Semantics and Proof Theory. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [60, 15, 18, 26, 50, 37] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
54. Ben C. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, Cambridge, England, 1986.
55. Carl Adam Petri. *Kommunikation mit Automaten*. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.
56. Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, IEEE CS FoCS, pages 46–57. Providence, Rhode Island, IEEE CS, 1977. .
57. Wolfang Reisig. *A Primer in Petri Net Design*. Springer Verlag, March 1992. 120 pages.
58. Wolfgang Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs in Theoretical Computer Science*. Springer Verlag, May 1985.
59. Wolfgang Reisig. *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets*. Springer Verlag, December 1998. xi + 302 pages.
60. Wolfgang Reisig. The Expressive Power of Abstract State Machines. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [15, 18, 53, 26, 50, 37] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
61. Wolfgang Reisig. *Logics of Specification Languages*, chapter Abstract State Machines for the Classroom, pages 15–46 in [10]. Springer, 2008.
62. Judi M.T. Romijn, Graeme P. Smith, and Jaco C. van de Pol, editors. *IFM 2005: Integrated Formal Methods*, volume 3771 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands, December 2005. Springer. Proceedings of 5th Intl. Conf. on IFM. ISBN 3-540-30492-4.
63. A. W. Roscoe. *Theory and Practice of Concurrency*. C.A.R. Hoare Series in Computer Science. Prentice-Hall, 1997. Now available on the net: http://www.comlab.ox.ac.uk/people/bill.roscoe/publications/68b.pdf.
64. Steve Schneider. *Concurrent and Real-time Systems — The CSP Approach*. Worldwide Series in Computer Science. John Wiley & Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, January 2000.
65. J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, January 1988.
66. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
67. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.
68. Chao Chen Zhou and Michael R. Hansen. *Duration Calculus: A Formal Approach to Real–time Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer–Verlag, 2004.
69. Chao Chen Zhou, Charles Anthony Richard Hoare, and Anders P. Ravn. A Calculus of Durations. *Information Proc. Letters*, 40(5), 1992.