A New Foundation for Computing Science

Are we Studying the Right Things ?

Dines Bjørner[*]
Fredsvej 11, DK-2840 Holte, Denmark
DTU, DK-2800 Kgs. Lyngby, Denmark
E–Mail: bjorner@gmail.com, URL: www.imm.dtu.dk/˜dibj

September 4, 2015

Abstract

We argue that computing systems requirements must be based on precisely described domain models. We further argue that domain science & engineering offers a new dimension in computing. We review our work in this area and we hint at a research and experimental engineering programme for the first two phases of the triptych of domain enginering, requirements engineering and software design.

1 Introduction

This author can refer to some substantial evidence [19, 21, 33] that using formal specifications in software development brings some substantial benefits. Section 2 recalls a first, 1981–1984, instance of such benefits. Yet, as also outlined in [15, *Bjørner & Havelund: 40 Years of Formal Methods — 10 Obstacles and 3 Possibilities*], "propagation" of formal methods into a wider industry seems lacking. Although [35, Woodcock et al.] lacks a reference to the formal methods project covered in Sect. 2, it is a fair reference to a number of projects supporting the author's "benefits" claim.

1.1 The Domain Engineering Claim

In this paper we wish, however, to not "push" the *formal methods* claim, but to "push" a, or the, *domain science & engineering* claim: *in order to* **design software** *one must have a good grasp of its requirements; in order to* **prescribe requirements** *one must have a good grasp of the underlying domain; so we expect that behind every serious software development there lies a stable* **domain description**. This, then, is the purpose of this paper: to "tout" the concept of domain science and engineering, emphasizing, in this paper, the latter.

1.2 Aim of Paper

So this is neither a *theory* nor a *programming methodology* paper. It is a review paper: *"where do we stand ?"* with respect to being able to develop correct software and software that meets customers' expectations ?; and *"how can those two issues: 'correctness' and 'meeting expectations' be improved ?"*.

[*] This paper is the background paper for a 15 minute presentation to be given at KPS 2015, Parkhotel Pörtschach, Wörthersee, Austria, October 5–7, 2015. The presentation is, obviously, expected to be approximately 12–15 slides !

1.3 Structure of Paper

Section 2 brings two examples: one of arbitrary, but well-formed transportation nets (illustrated by a road net), the other of arbitrary, but well-formed pipelines with the flow (laws) of liquid materials. The purpose of Sect. 2 is to review a 44 man-year project using formal methods ("lightly"). We bring this example — of a now more than 30 year old project (1981–1984) — to show an early use of a carefully narrated formal domain description, a project that we claim to have been a very successful one. Section 3 overviews our concept of *TripTych* development: from domain descriptions, via requirements prescriptions, to software design. We emphasize the domain science & engineering aspects. Section 4 Discusses our claim that this *TripTych* suggests a new foundation for computing science.

2 A Background Development

We sketch the structure of a successful 44 man year project which developed a commercial compiler according to the *TripTych* approach and using formal specifications — with success measured in therms of meeting customers' expectations and being correct.

2.1 The 1981–1984 DDC Ada Compiler Development Project

In the spring semester (6 months) of 1980 five MSc students worked out their MSc theses: **A Formal Description of Ada**. The four theses were published as [19]. That work became the basis for a full-scale industry-size project: *The DDC*[1] *ADA Compiler Project*, funded, in part by the *CEC*, the *Commission of the European Countries*. The project was carried out according to abstraction and refinement principles — as far as the ⋯ dotted box: the leftmost dynamic semantics (quadruple of) boxes[2] as well as the *A-code Language* and *Compiling Algorithm* is concerned — laid down in [2], and can be diagrammed as shown in Fig. 1 We explain the approach taken to develop, using formal specifications, an industry-strength, commercial compiler for the *US DoD*[3] *Ada* programming language. We do so using Fig. 1 as a reference point. Each box represents a specification and denotes a mathematical object. Each directed line between boxes represents a step of development, from a higher to a lower level of abstraction, and denotes a proof (of correctness, also a mathematical object). There were three phases of development: the *domain* engineering phase, the *requirements engineering* phase, and the *software design* phase. They are clearly marked in Fig. 1. First a formal description was developed for Ada. This phase is referred to as the 'Domain'. It had four stages: first the *Abstract Syntax*, then (developed "concurrently") the *Higher-order Static Semantics*, the *"Denotational" Dynamic Sequential Semantics* and the *"Operational" Dynamic Parallel Semantics*. Then a phase, *Requirements*, consisting of several stages. The refinement work represented by each of the boxes, were conditioned by various requirements. But we show such only for two boxes: dashed, labelled pointed lines. The *Higher-order Static Semantics* is refined in two stages: first a *Resumption Static Semantics* and then a *First-order Static Semantics*. The *"Denotational" Dynamic Sequential Semantics* was, in principle, refined in three stages: a *1st-order Functional Interpreter*, a *Imperative Stack* dynamic semantics and a *Macro-expansion* dynamic semantics. From the *Operational "Parallel" Semantics* was developed an operational *Run-time Semantics* for the concurrency constructs of Ada. From the *Macro-expansion* semantics was developed the design of an *A*[da] *Code Language* which was given a semantics commensurate with the specification language and *Macro-expansion* semantics. And from the *Macro-expansion* semantics and the *A Code Language* was developed a *Compiling Algorithm* which to every construct of Ada prescribed a sequence of *A Code*. The *Run-time Interpreter* was developed from the *Operational "Parallel" Ada Semantics*. Two *requirements assumptions* were: the compiler should execute within a 128 KB addressing space, and the compiled code should likewise execute within a 128 KB addressing space. Therefore the compiler need be decomposed into a number of passes where a pass was defined as that of a linear

[1] DDC: Dansk Datamatik Center was an industry-operated R&D centre, 1979–1989.

[2] *"Denotational" Sequental Ada*, *1st-order Functional Interpreter*, *Imperative Stack* and *Macro-Expansion*

[3] DoD: Department of Defense

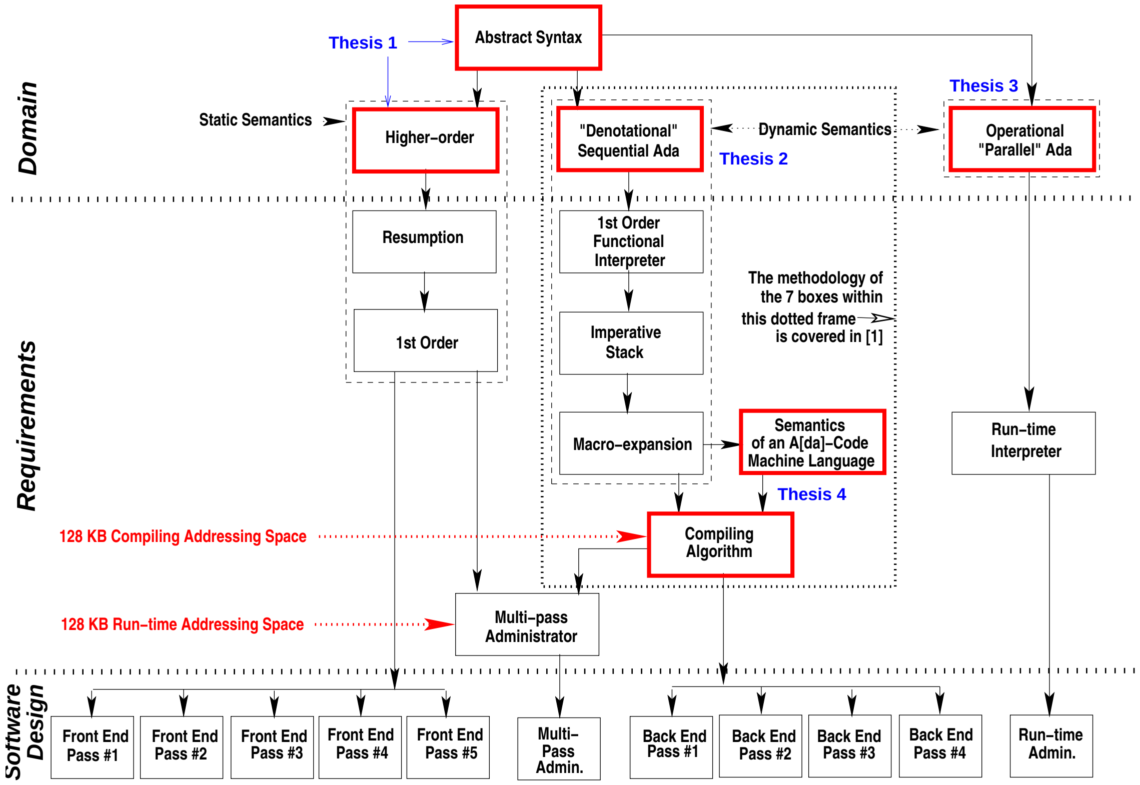

Figure 1: The Ada Compiler Software Development Graph. Bold-faced Boxes published in [19]

reading of of the Ada program text either left-to-right (forwards), or right-to-left (backwards), and in either pre-, in- or post-order[4]. From the combined *1st-order Static Semantics* and the *Compiling Algorithm* was, after careful analysis of these, developed a specification for a multi-pass administrator. The multi-pass analysis and synthesis resulted in five passes for the statics checks (i.e., "front-end"), and four passes for the code generator (i.e., "back-end"). These concluded the domain and requirements phases which were all specified in VDM [16, 31] for a total of approximately 10.000, respectively 56,000 lines of VDM and formula annotations. The nine compiler *Passes*, *Multi-pass Administrator*, and the *Run-time Administrator* were all coded from their specifications in a subset of the Ada language for which a compiler was developed in parallel with the full-Ada development!

2.2 A Review

2.2.1 Resources

The above project took place more than 30 years ago! Approximately the following man-power resources were used: For the *Domain* phase: seven people, one year; for the *Requirements* phase (exclusive of the *Multi-pass Administrator*: eleven people, one year; for the *Multi-pass Administrator*: six people, half a year; and for the rest (nine *Passes* and the *Administrators*): 12 people, 14 months. The subset Ada compiler development consumed seven man years. Thus a total of 42 man years was spent on effective development and its management, 2 man years on management of donors, funding and marketing.

[4] Pre-order: visiting program phrase tree nodes when first encountered; in-order: any time encountered, or post-order: when last encountered.

2.2.2 Formal Methods "Lite"

VDM was the prime "carrier" of the Ada compiler development. The domain and the requirements phases were specified in VDM. No properties of these specifications were formalised let alone proved. The first 10 years of use by industry on three continents (China, Japan, USA and Europe) revealed few, and only trivial errors: less than 2% of original development resources were spent on error corrections with average "repair" times being in the order of 1–2 days.

2.2.3 Epilogue

The above-outlined Ada compiler development project was reported in [21, 33]. The use of formal methods was clear. But 'formal methods' were not used in any other sense than formal specifications. Properties of and relationships between stage specifications, i.e., boxes, were not formalised. And yet, the project must be judged an unqualified success for formal methods. It took far fewer manpower resources than any other Ada compiler development project in those days. It had far, far fewer "bugs" than any comparable software development project in those days or since. Yet there were no tools available: No VDM syntax checker, No specification analyser. No nothing!

3 The Triptych of Software Engineering

We suggest a `TripTych` view of software engineering: *before software can be designed and coded we must have a reasonable grasp of "its" requirements; and before requirements can be prescribed we must have a reasonable grasp of "the underlying" domain.* To us, therefore, software engineering contains the three sub-disciplines:

- domain engineering,
- requirements engineering and
- software design.

3.1 What's New ?

So *"What's New ?"* in this ? Well, as far as the surveyed compiler development is concerned, nothing: that is how one should develop compilers — although it seems that it was done only once ![5] What can we learn from the example of Sect. 2 ? We can postulate that when there is a formal understanding of the domain — and of the stages from domain to requirements and on to software design, then software can be developed with greater assurance of meeting users' expectations and be correct than if not ! So that is what we are therefore proposing: to treat the domain, the application area for software development, as "a language" whose terms designate phenomena in the domain and "spoken/uttered" about by practitioners in the domain. So we consider a domain description to be the description of the syntax and the semantics of a language.

3.2 Domain Science & Engineering

3.2.1 What is a Domain ?

A **domain** is a human- and artifact-assisted arrangement of *endurant*, that is spatially "stable", and *perdurant*, that is temporally "fleeting" entities.

[5] Most textbooks in compiler development do not cover neither static nor dynamic semantics formally — and they certainly do not motivate the run-time stack stack/unstack operations upon procedure calls and returns such as done in [2].

3.2.2 Example Domains

To help understand the above delineation of the 'domain' concept we list some examples for which we can also refer to some either published or reported domain descriptions:

Example 1 . Manifest Domain Names: Examples of suggestive names of manifest domains are: *air traffic, banks, container lines, documents, hospitals, manufacturing, pipelines, railways* and *road nets*.

3.2.3 Comparison to Other Sciences and Their Engineering

We focus on the natural sciences and their engineerings: civil (or construction) engineering (buildings, roads, bridges, tunnels, etc.), mechanical engineering, chemical engineering, electrical (power engineering), electronics engineering (VLSI, IT hardware, etc.) and radio engineering (radio waves, transmitters, receivers, etc.). For all of these related technologies engineers are properly educated, knows the underlying sciences, that is, the domains of their artifacts. Not so, today, 2015, for software engineers for the domains listed in Example 1. Software engineers asked to develop software for either of air traffic control, banking. container lines, health care, railways, road pricing, etcetera, are expected to find out, themselves, what the relevant domain is, how it behaves, etc. No wonder that it often fails!

3.2.4 Domain Descriptions: Internet References

Now, we would not postulate the above without firm evidence. *"Proof in the pudding"* sort-of-evidence that domains can indeed be properly, informally and formally described. We shall first mention some existing descriptions before we exemplify fragments of such descriptions.

We list a number of reports all of which document descriptions of domains. These descriptions were carried out, by the present author, in order to research and develop the domain analysis and description concepts now summarised in the present paper. These reports ought now be revised, some slightly, others less so, so as to follow all of the prescriptions of the current paper. Except where a URL is given in full, please prefix the web reference with: `http://www2.compute.dtu.dk/~dibj/`.

1 *A Railway Systems Domain* D.Bjørner et al.

- Scheduling and Rescheduling of Trains; C.W.George and S.Prehn, 1996, `amore/scheduling.pdf`
- Formal Software Techniques in Railway Systems; 2000, `amore/dines-fac.pdf`
- Dynamics of Railway Systems; 2000, `amore/ifac-dynamics.pdf`
- Railway Staff Rostering; A.Strupchanska et al., 2003, `amore/albena-amore.pdf`
- Train Maintenance Routing; M.Peñicka et al., 2003, `amore/martin-amore.pdf`
- Train Composition and Decomposition: Domain and Requirements (draft), P.Karras et al., 2003, `amore/panos-amore.pdf`

2 *Models of IT Security. Security Rules & Regulations*, `it-security.pdf`, 2006. See [13]. A sketch is given of the IT security rules laid down by ISO

3 *A Container Line Industry Domain*, `container-paper.pdf`, 2007

4 *The "Market": Consumers, Retailers, Wholesalers, Producers*, `themarket.pdf`, 2007 See [3].

5 *What is Logistics ?* `logistics.pdf`, 2009

6 *A Domain Model of Oil Pipelines*, `pipeline.pdf`, 2009

7 *Transport Systems*, `comet/comet1.pdf`, 2010

8 *The Tokyo Stock Exchange*, `todai/tse-1.pdf` and `todai/tse-2.pdf`, 2010

9 *On Development of Web-based Software. A Divertimento*, `wfdftp.pdf`, 2010

10 *Documents (incomplete draft)*, `doc-p.pdf`, See [12]. 2013

3.2.5 An Example: Road Nets, Vehicles and Traffic

Parts The root domain, $\Delta_{\mathcal{D}}$, whose description is to be exemplified, is that of a composite traffic system (1a.) with a road net, (1b.) with a fleet of vehicles and (1c.) of whose individual position on the road net we can speak, that is, monitor.

1 We analyse the traffic system into

- a a composite road net,
- b a composite fleet (of vehicles), and
- c an atomic monitor.

2 The road net consists of two composite parts,

- a an aggregation of hubs and
- b an aggregation of links.

type

1. $\Delta_\triangle$

1a. $N_\triangle$

1b. $F_\triangle$

1c. $M_\triangle$

value

1a. $\textbf{obs\_part\_N}_\triangle: \Delta_\triangle \to N_\triangle$

1b. $\textbf{obs\_part\_F}_\triangle: \Delta_\triangle \to F_\triangle$

1c. $\textbf{obs\_part\_M}_\triangle: \Delta_\triangle \to M_\triangle$

type

2a. $HA_\triangle$

2b. $LA_\triangle$

value

2a. $\textbf{obs\_part\_HA}_\triangle: N_\triangle \to HA_\triangle$

2b. $\textbf{obs\_part\_LA}_\triangle: N_\triangle \to LA_\triangle$ ■

3 Hub aggregates are sets of hubs.

4 Link aggregates are sets of links.

5 Fleets are sets of vehicles.

6 We introduce some auxiliary functions.

- a links extracts the links of a network.
- b hubs extracts the hubs of a network.

type

3. $H_\triangle, HS_\triangle = H_\triangle\text{-}\underline{\textbf{set}}$

4. $L_\triangle, LS_\triangle = L_\triangle\text{-}\underline{\textbf{set}}$

5. $V_\triangle, VS_\triangle = V_\triangle\text{-}\underline{\textbf{set}}$

value

3. $\textbf{obs\_part\_HS}_\triangle: HA_\triangle \to HS_\triangle$

4. $\textbf{obs\_part\_LS}_\triangle: LA_\triangle \to LS_\triangle$

5. $\textbf{obs\_part\_VS}_\triangle: F_\triangle \to VS_\triangle$

6a. $\mathsf{links}_\triangle: \Delta_\triangle \to L\text{-}\underline{\textbf{set}}$

6a. $\mathsf{links}_\triangle(\delta_\triangle) \equiv \textbf{obs\_part\_LS}(\textbf{obs\_part\_LA}(\delta_\triangle))$

6b. $\mathsf{hubs}_\triangle: \Delta_\triangle \to H\text{-}\underline{\textbf{set}}$

6b. $\mathsf{hubs}_\triangle(\delta_\triangle) \equiv \textbf{obs\_part\_HS}(\textbf{obs\_part\_HA}(\delta_\triangle))$ ■

Unique Identifiers We cover the unique identifiers of all parts, whether needed or not.

7 Nets, hub and link aggregates, hubs and links, fleets, vehicles and the monitor all

- a have unique identifiers
- b such that all such are distinct, and
- c with corresponding observers.

8 We introduce some auxiliary functions:

- a xtr_lis extracts all link identifiers of a traffic system.
- b xtr_his extracts all hub identifiers of a traffic system.
- c given an appropriate link identifier and a net get_link 'retrieves' the designated link.
- d given an appropriate hub identifier and a net get_hub 'retrieves' the designated hub.

type

7a. NI, HAI, LAI, HI, LI, FI, VI, MI

value

7c. $\textbf{uid\_NI}: N_\triangle \to NI$

7c. $\textbf{uid\_HAI}: HA_\triangle \to HAI$

7c. $\textbf{uid\_LAI}: LA_\triangle \to LAI$

7c. $\textbf{uid\_HI}: H_\triangle \to HI$

7c. $\textbf{uid\_LI}: L_\triangle \to LI$

7c. $\textbf{uid\_FI}: F_\triangle \to FI$

7c. $\textbf{uid\_VI}: V_\triangle \to VI$

7c. $\textbf{uid\_MI}: M_\triangle \to MI$

axiom

7b. $NI \bigcap HAI = \emptyset$, $NI \bigcap LAI = \emptyset$, $NI \bigcap HI = \emptyset$, etc.

where axiom 7b is expressed semi-formally, in mathematics.

```
value
8a.  xtr_lis: $\Delta_\triangle$ → LI-set
8a.  xtr_lis($\delta_\triangle$) ≡
8a.      let ls = links($\delta_\triangle$) in {uid_LI(l)|l:L•l ∈ ls} end
8b.  xtr_his: $\Delta_\triangle$ → HI-set
8b.  xtr_his($\delta_\triangle$) ≡
8b.      let hs = hubs($\delta_\triangle$) in {uid_HI(h)|h:H•k ∈ hs} end
8c.  get_link: LI → $\Delta_\triangle$ $\overset{\sim}{\to}$ L
8c.  get_link(li)($\delta_\triangle$) ≡
8c.      let ls = links($\delta_\triangle$) in
8c.      let l:L • l ∈ ls ∧ li=uid_LI(l) in l end end
8c.      pre: li ∈ xtr_lis($\delta_\triangle$)
8d.  get_hub: HI → $\Delta_\triangle$ $\overset{\sim}{\to}$ H
8d.  get_hub(hi)($\delta_\triangle$) ≡
8d.      let hs = hubs($\delta_\triangle$) in
8d.      let h:H • h ∈ hs ∧ hi=uid_HI(h) in h end end
8d.      pre: hi ∈ xtr_his($\delta_\triangle$)  ■
```

Mereology

9 Links are connected to exactly two distinct hubs.

10 Hubs are connected to zero or more links.

11 For a given net the link and hub identifiers of the mereology of hubs and links must be those of links and hubs, respectively, of the net.

```
type
9.   LM′ = HI-set, LM = {|his:HI-set • card(his)=2|}
10.    HM = LI-set
value
9.   mereo_L: L → LM
10.    mereo_H: H → HM
axiom [Well−formedness of Road Nets, N]
11.    ∀ n:N,l:L,h:H•  l ∈ obs_part_Ls(obs_part_LC(n))∧h ∈ obs_part_Hs(obs_part_GC(n))
11.       let his=mereology_H(l), lis=mereology_H(h) in
11.       his⊆∪{uid_H(h) | h ∈ obs_part_Hs(obs_part_HC(n))}
11.     ∧ lis⊆∪{uid_H(l) | l ∈ obs_part_Ls(obs_part_LC(n))} end
```

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

- **Hubs:** *location*s[6] are considered static, *wear and tear* (condition of road surface) is considered inert, *hub state*s and *hub state space*s are considered programmable;
- **Links:** *length*s and *location*s are considered static, *wear and tear* (condition of road surface) is considered inert, *link state*s and *link state space*s are considered programmable;

[6] By location we mean a cadestral/geodetic position.

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

We treat one attribute each for hubs, links, vehicles and the monitor. First we treat hubs.

12 Hubs

 a have *hub states* which are sets of pairs of identifiers of links connected to the hub[7],

 b and have *hub state spaces* which are sets of hub states[8].

13 For every net,

 a link identifiers of a hub state must designate links of that net.

 b Every hub state of a net must be in the hub state space of that hub.

14 Hubs have geodetic and cadestral location.

15 We introduce an auxiliary function: xtr_lis extracts all link identifiers of a hub state.

```
type
12a.  HΣ = (LI×LI)-set
12b.  HΩ = HΣ-set
value
12a.  attr_HΣ: H → HΣ
12b.  attr_HΩ: H → HΩ
axiom
13.   ∀ δ:Δ,
13.      let hs = hubs(δ) in
13.      ∀ h:H • h ∈ hs •
13a.          xtr_lis(h)⊆xtr_lis(δ)
13b.          ∧ attr_Σ(h) ∈ attr_Ω(h)
13.      end
type
14.  HGCL
value
14.  attr_HGCL: H → HGCL
15.  xtr_lis: H → LI-set
15.  xtr_lis(h) ≡
15.     {li | li:LI,(li′,li″):LI×LI •
15.          (li′,li″) ∈ attr_HΣ(h) ∧ li ∈ {li′,li″}}
```

Then links.

16 Links have lengths.

17 Links have geodetic and cadestral location.

18 Links have states and state spaces:

 a States modeled here as pairs, (hi', hi''), of identifiers the hubs with which the links are connected and indicating directions (from hub h' to hub h''.) A link state can thus have 0, 1, 2, 3 or 4 such pairs.

 b State spaces are the set of all the link states that a link may enjoy.

```
type
16.   LEN
17.   LGCL
18a.  LΣ = (HI×HI)-set
18b.  LΩ = LΣ-set
value
16.   attr_LEN: L → LEN
17.   attr_LGCL: L → LGCL
18a.  attr_LΣ: L → LΣ
18b.  attr_LΩ: L → LΩ
axiom
18.  ∀ n:N •
18.     let ls = xtr−links(n), hs = xtr_hubs(n) in
18.     ∀ l:L•l ∈ ls ⇒
```

[7] A hub state "signals" which input-to-output link connections are open for traffic.

[8] A hub state space indicates which hub states a hub may attain over time.

```
18a.        let lσ = attr_LΣ(l) in
18a.        0≤card lσ≤4
18a.      ∧ ∀ (hi′,hi″):(HI×HI)•(hi′,hi″) ∈ lσ ⇒
18a.            {get_H(hi′)(n),get_H(hi″)(n)}=mereo_L(l)
18b.      ∧ attr_LΣ(l) ∈ attr_LΩ(l) ■
18.     end end
```

Then vehicles.

19 Every vehicle of a traffic system has a position which is either ‘on a link’ or ‘at a hub’.

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

```
type
19.    VPos = onL | atH
19a.   onL :: LI HI HI R
19b.   R = Real      axiom ∀ r:R • 0≤r≤1
19c.   atH :: HI LI LI
value
19.   attr_VPos: V△ → VPos
axiom
19a.   ∀ n△:N△, onL(li,fhi,thi,r):VPos •
19a.       ∃ l△:L△•l△∈obs_part_LS(obs_part_N△(n△))
19a.           ⇒ li=uid_L△(l)∧{fhi,thi}=mereo_L△(l△),
19c.   ∀ n△:N△, atH(hi,fli,tli):VPos •
19c.       ∃ h△:H△•h△∈obs_part_HS△(obs_part_N(n△))
19c.           ⇒ hi=uid_H△(h△)∧(fli,tli) ∈ attr_LΣ(h△)
```

And finally monitors. We consider only one monitor attribute.

20 The monitor has a vehicle traffic attribute.

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

```
type
20.   Traffic = VI →m (T × VPos)*
value
20.   attr_Traffic: M → Traffic
axiom
20b.   ∀ δ:Δ •
20b.      let m = obs_part_M△(δ) in
20b.      let tf = attr_Traffic(m) in
20b.      dom tf ⊆ xtr_vis(δ) ∧
```

```
20b.        ∀ vi:VI • vi ∈ dom tf •
20b.            let tr = tf(vi) in
20b.            ∀ i,i+1:Nat • {i,i+1}⊆dom tr •
20b.                let (t,vp)=tr(i),(t′,vp′)=tr(i+1) in
20b.                t<t′
20(b)i.              ∧ case (vp,vp′) of
20(b)i.                    (onL(li,fhi,thi,r),onL(li′,fhi′,thi′,r′))
20(b)i.                       → li=li′∧fhi=fhi′∧thi=thi′∧r≤r′
20(b)i.                        ∧ li ∈ xtr_lis(δ)
20(b)i.                        ∧ {fhi,thi} = mereo_L(get_link(li)(δ)),
20(b)ii.                   (atH(hi,fli,tli),atH(hi′,fli′,tli′))
20(b)ii.                      → hi=hi′∧fli=fli′∧tli=tli′
20(b)ii.                       ∧ hi ∈ xtr_his(δ)
20(b)ii.                       ∧ (fli,tli) ∈ mereo_H(get_hub(hi)(δ)),
20(b)iii.                  (onL(li,fhi,thi,1),atH(hi,fli,tli))
20(b)iii.                     → li=fli∧thi=hi
20(b)iii.                      ∧ {li,tli} ⊆ xtr_lis(δ)
20(b)iii.                      ∧ {fhi,thi}=mereo_L(get_link(li)(δ))
20(b)iii.                      ∧ hi ∈ xtr_his(δ)
20(b)iii.                      ∧ (fli,tli) ∈ mereo_H(get_hub(hi)(δ)),
20(b)iv.                   (atH(hi,fli,tli),onL(li′,fhi′,thi′,0))
20(b)iv.                      →  etcetera,
20b.                  _  → false
20b.        end end end end end ■
```

3.2.6 Another Example: Pipelines

Parts

21 A pipeline consists of an indefinite number of pipeline units.

22 A pipeline units is either a well, or a pipe, or a pump, or a valve, or a fork, or a join, or a sink.

23 All these unit sorts are atomic and disjoint.

```
type
21.   PL, U, We, Pi, Pu, Va, Fo, Jo, Si
21.   Well, Pipe, Pump, Valv, Fork, Join, Sink
value
21.   obs_part_Us: PL → U-set
type
22.   U == We | Pi | Pu | Va | Fo | Jo | Si
23.  We::Well, Pi::Pipe, Pu::Pump, Va::Valv, Fo:Fork, Jo::Join, Si::Sink
```

Unique Identifiers

24 Every pipeline unit has a unique identifier.

```
type
24.  UI
value
24.  uid_U: U → UI
```

Materials

25 Applying `obs_material_sorts_U` to any pipeline unit, u:U, yields

a a type clause stating the material sort LoG for some further undefined liquid or gaseous material, and b a material observer function signature.

```
    type
25a     LoG
    value
25b     obs_mat_LoG: U → LoG
```

Mereology Pipeline units serve to conduct fluid or gaseous material. The flow of these occur in only one direction: from so-called input to so-called output.

26 Wells have exactly one connection to an output unit.

27 Pipes, pumps and valves have exactly one connection from an input unit and one connection to an output unit.

28 Forks have exactly one connection from an input unit and exactly two connections to distinct output units.

29 Joins have exactly one two connection from distinct input units and one connection to an output unit.

30 Sinks have exactly one connection from an input unit.

31 Thus we model the mereology of a pipeline unit as a pair of disjoint sets of unique pipeline unit identifiers.

```
type
31.  UM′=(UI-set×UI-set)
31.  UM={|(iuis,ouis):UI-set×UI-set•iuis ∩ ouis={}|}
value
31.  mereo_U: UM
axiom [Well−formedness of Pipeline Systems, PLS (0)]
    ∀ pl:PL,u:U • u ∈ obs_part_Us(pl) ⇒
            let (iuis,ouis)=mereo_U(u) in
            case (card iuis,card ouis) of
26.             (0,1) → is_We(u),
27.             (1,1) → is_Pi(u)∨is_Pu(u)∨is_Va(u),
28.             (1,2) → is_Fo(u),
29.             (2,1) → is_Jo(u),
30.             (1,0) → is_Si(u)
            end end
```

Attributes Let us postulate a[n attribute] sort Flow. We now wish to examine the flow of liquid (or gaseous) material in pipeline units. We use two types

32 F for "productive" flow, and L for wasteful leak.

Flow and leak is measured, for example, in terms of volume of material per second. We then postulate the following unit attributes "measured" at the point of in- or out-flow or in the interior of a unit.

33 current flow of material into a unit input connector,

34 maximum flow of material into a unit input connector while maintaining laminar flow,

35 current flow of material out of a unit output connector,

36 maximum flow of material out of a unit output connector while maintaining laminar flow,

37 current leak of material at a unit input connector,

38 maximum guaranteed leak of material at a unit input connector,

39 current leak of material at a unit input connector,

40 maximum guaranteed leak of material at a unit input connector,

41 current leak of material from "within" a unit, and

42 maximum guaranteed leak of material from "within" a unit.

```
type
32.  F, L
value
33.  attr_cur_iF: U → UI → F
34.  attr_max_iF: U → UI → F
35.  attr_cur_oF: U → UI → F
36.  attr_max_oF: U → UI → F
37.  attr_cur_iL: U → UI → L
38.  attr_max_iL: U → UI → L
39.  attr_cur_oL: U → UI → L
40.  attr_max_oL: U → UI → L
41.  attr_cur_L: U → L
42.  attr_max_L: U → L
```

The maximum flow attributes are static attributes and are typically provided by the manufacturer as indicators of flows below which laminar flow can be expected. The current flow attributes are dynamic attributes

Intra Unit Flow and Leak Law

43 For every unit of a pipeline system, except the well and the sink units, the following law apply.

44 The flows into a unit equal

  a the leak at the inputs

  b plus the leak within the unit

  c plus the flows out of the unit

  d plus the leaks at the outputs.

```
axiom [Well−formedness of Pipeline Systems, PLS (1)]
43.  ∀ pls:PLS,b:B\We\Si,u:U •
43.      b ∈ obs_part_Bs(pls)∧u=obs_part_U(b)⇒
43.      let (iuis,ouis) = mereo_U(u) in
44.      sum_cur_iF(iuis)(u) =
44a.         sum_cur_iL(iuis)(u)
44b.       ⊕ attr_cur_L(u)
44c.       ⊕ sum_cur_oF(ouis)(u)
44d.       ⊕ sum_cur_oL(ouis)(u)
43.      end
```

45 The sum_cur_iF (cf. Item 44) sums current input flows over all input connectors.

46 The sum_cur_iL (cf. Item 44a) sums current input leaks over all input connectors.

47 The sum_cur_oF (cf. Item 44c) sums current output flows over all output connectors.

48 The sum_cur_oL (cf. Item 44d) sums current output leaks over all output connectors.

```
45.  sum_cur_iF: UI-set → U → F
45.  sum_cur_iF(iuis)(u) ≡ ⊕ {attr_cur_iF(ui)(u)|ui:UI•ui ∈ iuis}
46.  sum_cur_iL: UI-set → U → L
46.  sum_cur_iL(iuis)(u) ≡ ⊕ {attr_cur_iL(ui)(u)|ui:UI•ui ∈ iuis}
47.  sum_cur_oF: UI-set → U → F
47.  sum_cur_oF(ouis)(u) ≡ ⊕ {attr_cur_iF(ui)(u)|ui:UI•ui ∈ ouis}
48.  sum_cur_oL: UI-set → U → L
48.  sum_cur_oL(ouis)(u) ≡ ⊕ {attr_cur_iL(ui)(u)|ui:UI•ui ∈ ouis}
     ⊕: (F|L) × (F|L) → F
```

where ⊕ is both an infix and a distributed-fix function which adds flows and or leaks

Inter Unit Flow and Leak Law

49 For every pair of connected units of a pipeline system the following law apply:

  a the flow out of a unit directed at another unit minus the leak at that output connector

  b equals the flow into that other unit at the connector from the given unit plus the leak at that connector.

```
      axiom [Well−formedness of Pipeline Systems, PLS (2)]
49.     ∀ pls:PLS,b,b′:B,u,u′:U•
49.        {b,b′}⊆obs_part_Bs(pls)∧b≠b′∧u′=obs_part_U(b′)
49.         ∧ let (iuis,ouis)=mereo_U(u),(iuis′,ouis′)=mereo_U(u′),
49.              ui=uid_U(u),ui′=uid_U(u′) in
49.           ui ∈ iuis ∧ ui′ ∈ ouis′ ⇒
49a.             attr_cur_oF(u′)(ui′) − attr_leak_oF(u′)(ui′)
49b.           = attr_cur_iF(u)(ui) + attr_leak_iF(u)(ui)
49.           end
49.     comment: b′ precedes b
```

From the above two laws one can prove the **theorem:** what is pumped from the wells equals what is leaked from the systems plus what is output to the sinks.

3.2.7 Domain Descriptions: Methodology

By a **method** we shall understand a set of **principles** for **selecting** and **applying** **techniques** and **tools** for **constructing artifacts** By **methodology** we shall understand the **study** and **knowledge** of **methods**.

The tools of the domain description method centers around two kinda of **prompts**. By a **prompt** we shall understand something that induces an action, an occasion or incitement to inspire, or an assist suggesting something to be expressed. There are two kinds of prompts: **analysis prompts** and **description prompts**. The analysis prompts to be summarised below can be thought of as predicates that the domain engineer applies to phenomena of the domain yielding true, false or undefined answers. The description prompts to be summarised below are applied, by the domain engineer, to phenomena of the domain for which preceding analysis prompts has yielded truth answers. Thus the *domain analysis & description process* alternates between analysis prompts and description prompts. The **domain description method** is here specialised to **manifest domains** [11]. First the domain engineer cum scientist examines a perceived domain phenomena, ϕ: is_entity(ϕ), and if **true**, then inquires which of is_endurant(ϕ) or is_perdurant(ϕ) holds. If is_endurant(ϕ) holds then the domain analyser inquires as to whether is_discrete(ϕ) or is_continuous(ϕ) holds. If is_discrete(ϕ) holds then is_part(ϕ) holds, otherwise either of is_material or is_component holds. If is_part(ϕ) then either is_atomic(ϕ) or is_composite(ϕ). If is_composite(ϕ) holds then observe_parts(ϕ) yields some parts that can now be analysed, eventually leading the domain analyser to conclude that the part ϕ can be described. By applying observe_part_sorts(ϕ) to a composite domain δ we then obtain its constituent parts — as exemplified in formula lines 1.–1c. and similarly formula lines 2a.–2b. Some composite parts may be modelled by concrete types: has_concrete_type(ϕ) in which case observe_part_types(ϕ) will yield those concrete types as exemplified in formula lines 3.–5 , and in formula lines 21 . Once the atomic and composite parts of a domain has been settled their properties: unique identifiers, mereology and attributes can be analysed and described. First their uniqueness: observe_unique_identifiers, such as f.ex. illustrated by formula lines 7a.–7b. Once all parts have been identified one can inquire as to their mereology: how parts relate to other parts: if has_mereology(ϕ) holds then observe_mereology(ϕ) yields which specific other parts, of same or other sorts, such as for example in formula lines 9.–10 or formula lines 31. Finally a last set of properties of parts can be investigated, namely their attributes. Any part, ϕ, may have any number of attributes. The analysis prompt attribute_names(ϕ) yields names of attributes. — with the description prompt observe_attributes(ϕ) yielding their description — as in formula lines 12a.–12b. or in formula lines 16.–18b.

There are other aspects to the methodology analysing and describing endurants: gaseous or liquid materials being contained in parts, and perdurants actions, events and behaviours. We shall not cover these here, but refer to [10, Manifest Domains: Analysis & Description] and [11, From Domain Descriptions to Requirements Prescriptions].

3.2.8 Domain Science

There are a number of issues that need be researched.

A Prompt Semantics: The analysis and description prompts need be precisely, that is, mathematically defined. Such a semantics is a first step towards securing a foundation for our approach. We refer to [8].

Laws of Domain Descriptions: A semantics of the analysis and description prompts and thus their applications is expected to satisfy the following law: Analysing ($\mathcal{A}$) and/or describing ($\mathcal{D}$) two otherwise unrelated composite parts, p_i and p_j, shall yield the same results whether p_i is treated before p_j or vice-versa: $\mathcal{A}(p_i);\mathcal{A}(p_j)$ and $\mathcal{A}(p_j);\mathcal{A}(p_i)$, respectively $\mathcal{D}(p_i);\mathcal{D}(p_j)$ and $\mathcal{D}(p_j);\mathcal{D}(p_i)$. There are many others such laws.

Laws of Domains: Given an appropriate domain description it should be possible to prove certain laws about that domain. **An example:** Assume a railway system with trains operating according to a timetable that prescribes train departures from and arrivals at any station according to a 24 hour cycle, and assume that all trains function precisely. Now we would expect the following law to hold over any 24 hour period: The of trains *arriving* at a station, minus the number of trains *ending* their journey at that station, plus number of trains *starting* their journey at that station, equals the number of trains *leaving* that station •

•••

Physics is characterised by its laws. So should man-assisted domains. A proper theory of domain description should invite domain laws to be identified and proved. There is a rich world *"out there"*.

3.2.9 What Can Be Described ?

Even if we limit ourselves to physically manifest domains [11], that is, entities that we can observe, i.e., see, in cases even touch, there are such which we do not yet know how to describe objectively, that is, mathematically. Moreover, we cannot give a precise delineation of which domains, or aspects of domains, are describable.

An example: We have described aspects of a pipeline system, Sect. 3.2.6. We have even postulated (implementable) functions for observing the flow and leaks of the material (oil, gas, or other) conducted by pipeline units. We also know, but do not show, how to formalise the fluid dynamics of these flows, namely in terms of partial differential equations (PDEs) based on Bernoulli and Navier–Stokes models through individual pipeline units. But we have yet to show how to combine our "discrete mathematics" models with hose of fluid dynamics. One problem here is that our discrete mathematics descriptions model an infinite variety of pipelines, that is, arbitrary compositions of pipeline units, whereas, conventionally, PDEs, model the dynamics only of specific, single units. It has been suggested[9] that perhaps the Wiener–Feynman–Dirac–Wheeler concept of *Path Integrals.*[10] may be a way to solve the problem •

Our domain models are just abstractions ! One cannot expect any domain description to "completely" model a domain. There are simply too many properties to describe. And there are domain properties that we can informally describe in words, but cannot yet formalise. Domain description is (therefore) a matter of choice, of abstraction level and of what to include in the description and what to leave out !

3.3 Requirements Engineering

We would not advocate the *TripTych* to software development unless we had a method for "deriving" requirements from domain descriptions. And from formal requirements prescriptions we know how to design software such that $\mathcal{D},\mathcal{S} \models \mathcal{R}$, that is: the $\mathcal{S}$oftware can be proved correct — in the context of the $\mathcal{D}$omain — with respect to the $\mathcal{R}$equirements.

3.3.1 Three Kinds of Requirements

Our approach to the "derivation" of requirements is based on the following decomposition of requirements into three kinds: *domain requirements*, *interface requirements* and *machine requirements* where the *machine* is the hardware and software to be developed

3.3.2 Domain Requirements

By *domain requirements* we shall understand such requirements that can be expressed sôlely using terms of the domain that is, terms defined in the domain description.

[9] Jakob Bohr, Technical University of Denmark

[10] The path integral formulation of quantum mechanics is a description of quantum theory which generalizes the action principle of classical mechanics. It replaces the classical notion of a single, unique trajectory for a system with a sum, or functional integral, over an infinity of possible trajectories to compute a quantum amplitude `wikipedia.org/wiki/Path_integral_formulation`.

The "derivation"[11] of domain requirements prescriptions from domain descriptions is governed by a set of "derivation" operations. Examples of these 'derivation' operations are: *projection*, *instantiation*, *determination*, *extension* and *fitting*.

Projection means that we remove from the evolving requirements prescription those entity descriptions of the domain which are not to be considered when (further) prescribing the requirements. **An example:** From the example of the road net and traffic system we remove the vehicles and the monitor •

Instantiation means that we concretise, i.e., prescribe "less-abstract", those retained domain phenomena whose concretisation it is suitable to prescribe. **An example:** The general road net is instantiated to a "linear" toll-road system of a sequence of toll-road hubs connected, "up" and "down" the toll-road to neighbouring toll-road hubs, and, by means of toll-road plazas, to a remaining road net • What do we mean by: "it is suitable to prescribe" ? Well, first of all, we have to realize the following: requirements must only prescribe what can be computed. That means that entities whose realisability in terms of computable data structure or functions must eventually be so prescribed. Secondly, as requirements prescription may, and normally will proceed in stages, one (i.e., the requirements engineer) may decide to instantiate some entities while leaving other entities "untouched", only to return to the concretisation of these n a later stage. And so forth. It is all a matter of style and taste!

Determination means that there may be entities, i.e., endurants or perdurants, that are described to be non-deterministic in the domain but which, after projection and instantiation need be prescribed to be "less non-deterministic". **An example:** Whereas hubs in general allow traffic from any link incident upon that hub to any links emanating from that hub but so that signaling, as expressed in the hub states, may, at times, prevent some emanating links to be accessible from some incident links; a toll-road hub, in order to be an appropriate toll-road, must allow for free flow from any incident link to any emanating link •

Extension typically means that there may be entities that were "hitherto" not computationally feasible, but where new technologies or higher labour costs mandate their feasibility — thus making way for introducing these mew technologies into a this 'extended' domain. **An example** is that of the electronic sensing of vehicles entering or leaving a toll-road — thus enabling *"road pricing"* •

Fitting is necessitated when two or more requirements projects based on "the same" domain, and with "overlapping domain coverage" need be "harmonised". **An example:** One set of requirements are being prescribed for a *road state-of-repair and maintenance facility,* another set of requirements are being prescribed for a *road pricing system.* Now they must both rely on some sort of representation of the same road net •

3.3.3 Interface Requirements

By *interface requirements* we shall understand such requirements that can be expressed only using terms both of the domain and of the machine.

In order to structure the interface requirements we introduce a notion of *shared phenomena* whether endurants or perdurants. If a phenomenon is present in the domain and if it is also to be present in the machine to be designed then that phenomenon is said to be shared. As a result we structure interface requirements prescriptions around *shared endurants*, *shared actions*, *shared events* and *shared behaviours.*

Shared endurants pose two "problems" the initialisation of endurant data structures and their values, and the regular access to and update of endurant data. Both must be prescribed. Usually both require the interaction between the domain and the machine. **An example:** Road nets are shared between the domain and the machine. Initially all hubs and all links need be structured in some data structure, say a database. The shared endurant requirements must now specify which, usually composite database operations are to be used in establishing the database, and which are to be used in accessing and updating the endurants.

Shared actions imply an interaction between between the domain and the machine. That interaction is typically manifested by interaction between either humans of the domain or physical domain entities and the machine **Example: Human/Machine Interaction:** The payment of a road price fee today involves a human (say, with a credit card) and the machine, checking and accepting or rejecting the credit card, etcetera • **Example: Machine/Machine Interaction:** The electronic recording (within the machine) of a vehicle passing a toll-gate barrier (another part of the machine) and the vehicle itself (another machine, external to required machine) •

And so on, for *shared events* and *shared behaviours.*

[11] We put 'derivation' in double quotes because we do not mean 'automatic' derivation.

3.3.4 Machine Requirements

By *machine requirements* we shall understand such requirements that can be expressed sôlely using terms of the machine. Since that is the case: no "mention" of the domain in the machine requirements we shall omit covering this field.

3.4 Discussion

We have suggested that there are a set of principles and techniques for "deriving" a major set of requirements from domain descriptions. This, then, is an argument for taking domain modelling serious: there are principles and techniques for bringing you from domain descriptions to requirements prescriptions and from there on to software design. We refer to [10, 2015] for details.

4 Are We Studying the Right Things?

We claim to have justified our claim that *$\mathcal{S}$oftware* must be *designed* on the basis of *$\mathcal{R}$equirements prescriptions* that have been "derived" from *$\mathcal{D}$omain descriptions*, all of them formally. In this way we can secure that software fulfill users'/customers' expectations since the requirements are strongly related to the domain and is correct: $\mathcal{D},\mathcal{S} \models \mathcal{R}$.

It is the only way in which we can see these two, *expectations* and *correctness*, fulfilled.

4.1 Papers on Domain Science & Engineering

I mention but a few of my earlier papers related to domain science & engineering. In chronological order. For a comprehensive introduction to Domain Science & Engineering and to a novel approach to Requirements Engineering I refer to [11, 10] respectively.

[5, *Domain Engineering, 2008*] treats and aspect of domain modelling referred to as *domain facets*. We expect to revise [5].

[6, *Domains: Their Simulation, Monitoring and Control, 2011*]. The concepts of simulation, monitoring and simulation are analysed in the light of the domain–requirements–design *TripTych*.

[7, *A Rôle for Mereology in Domain Science and Engineering, 2009*]. Stanisław Leśhniewski's replacement of Bertrand Russells set theory axiomatisation is reviewed amd it is shown how part/sub-part relations can interpreted as a reation between (Hoare) CSP-processes.

[9, *Domain Engineering – A Basis for Safety Critical Software. 2014*]. Issues of *system safety criticality* that can be considered already before requirements engineering are here seen in the light of domain engineering.

[11, *Manifest Domains: Analysis & Description, 2014*] is the definitive paper on domain analysis and description.

[10, *From Domains to Requirements — A Different View of Requirements Engineering, 2015*] is the definitive paper on "derivation" of requirements prescriptions from domain descriptions. It is a complete rewrite of [4, From Domains to Requirements] and represents a complete rethinking of that paper.

4.2 A Research and Experimental Engineering Programme

In the papers on which the current paper is based a number of open problems have been identified.

4.2.1 The Mathematics of Analysis & Description Prompts

In [11, *Domain Analysis: Endurants – An Analysis & Description Process Model*] we present a formal semantics of the analysis and description process. In [10, *From Domain Descriptions to Requirements Prescriptions — a Different Approach to Requirements Engineering*] we present a $\boxed{\dots}$ The study of this area is elusive.

4.2.2 Analysis & Description Calculi for Other Domains

The analysis and description calculus of this paper appears suitable for manifest domains. For other domains other calculi appears necessary. There is the introvert, composite domain of systems software: operating systems, compilers, database management systems, Internet-related software, etcetera. The classical computer science and software engineering disciplines related to these components of systems software appears to have provided the necessary analysis and description "calculi." There is the domain of financial systems software accounting & bookkeeping, banking systems, insurance, financial instruments handling (stocks, etc.), etcetera. Etcetera. For each domain characterisable by a distinct set of analysis & description calculus prompts such calculi must be identified. It seems straightforward: to base a method for analysing & describing a category of domains on the idea of prompts like those developed in this paper.

4.2.3 On Domain Description Languages

We have in this paper expressed the domain descriptions in the RAISE [27] specification language RSL [26]. With what is thought of as basically inessential, editorial changes, one can reformulate these domain description texts in either of Alloy [30] or The B-Method [1] or VDM [17, 18, 23] or Z [36]. One could also express domain descriptions algebraically, for example in CafeOBJ [25, 22, 24, 20]. The analysis and the description prompts remain the same. The description prompts now lead to CafeOBJ texts.

We did not go into much detail with respect to perdurants, let alone behaviours. For all the very many domain descriptions, covered elsewhere, RSL (with its CSP sub-language) suffices. But there are cases where we have conjoined our RSL domain descriptions with descriptions in Petri Nets [34] or MSC [29] (Message Sequence Charts) or StateCharts [28]. Since this paper only focused on endurants there was no need, it appears, to get involved in temporal issues. When that becomes necessary, in a study or description of perdurants, then we either deploy DC: The Duration Calculus [37] or TLA+: Temporal Logic of Actions [32].

4.2.4 Commensurate Discrete and Continuous Models

The pipeline example hinted at co-extensive descriptions of discrete and continuous behaviours, the former in, for example, RSL, the latter in, typically, the calculus mathematics of partial different equations (PDEs). The problem that arises in this situation is the following: there will be, say variable identifiers, e.g., x, y, ..., z which in the RSL formalisation has one set of meanings, but which in the PDE "formalisation" has another set of meanings. Current formal specification languages[12] do not cope with continuity. Some research is going on. But to substantially cover, for example, the proper description of laminar and turbulent flows in networks (e.g., pipelines) requires more substantial results.

4.2.5 Interplay between Parts and Materials

The pipeline example revealed but a small fraction of the problems that may arise in connection with modeling the interplay between parts and materials. Subject to proper formal specification language and, for example PDE specification we may expect more interesting laws, as for example those of pipeline flows and even proof of these as if they were theorems. Formal specifications have focused on verifying properties of requirements and software designs. With co-extensive (i.e., commensurate) formal specifications of both discrete and continuous behaviours we may expect formal specifications to also serve as bases for predictions.

4.2.6 The Mathematics of Domain-to-Requirements Operators

In [10, *From Domain Specifications to Requirements Prescriptions – A Different View of Requirements Engineering*][13] we postulate that certain properties hold between domain requirements prescriptions "before"and "after" the application of the domain-to-requirements operations: *projection*, *instantiation*, *determination*, *extension* and *fitting*. These postulated properties need be studied further.

4.2.7 Further Work on Domain-to-Requirements and Interface Techniques

In [10, *From Domain Specifications to Requirements Prescriptions – A Different View of Requirements Engineering*] we have shown a number of techniques for domain-to-requirements operations, in particular those that yield domain requirements. In [10] we also show some techniques that pertain to interface requirements, but it seems more study is required.

[12] Alloy [30], Event B [1], RSL [26], VDM-SL [17, 18, 23], Z, etc.

[13] [10] is a complete rewrite/rethinking of [4].

4.3 Tony Hoare's Summary on 'Domain Modeling'

In a 2006 e-mail, in response, undoubtedly to my steadfast, perhaps conceived as stubborn insistence, on domain engineering, Tony Hoare summed up his reaction to domain engineering as follows, and I quote[14]:

"There are many unique contributions that can be made by domain modeling.

1 The models describe all aspects of the real world that are relevant for any good software design in the area. They describe possible places to define the system boundary for any particular project.

2 They make explicit the preconditions about the real world that have to be made in any embedded software design, especially one that is going to be formally proved.

3 They describe the whole range of possible designs for the software, and the whole range of technologies available for its realisation.

4 They provide a framework for a full analysis of requirements, which is wholly independent of the technology of implementation.

5 They enumerate and analyse the decisions that must be taken earlier or later in any design project, and identify those that are independent and those that conflict. Late discovery of feature interactions can be avoided."

4.4 Are We Studying the Right Things ?

By **computer science** we understand the study and knowledge about the phenomena that can "exist inside" computers. By **computing science** we understand the study and knowledge about how to construct those phenomena.

If we accept the *TripTych* dogma of basing software design on precise requirements prescriptions which are based on precise domain descriptions, then training, teaching and research in computer and computing science must be revised.

5 Acknowledgements

I thank Prof. Jens Knoop for challenging me to present this paper.

6 Bibliography

6.1 References

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

[2] D. Bjørner. Programming Languages: Formal Development of Interpreters and Compilers. In *International Computing Symposium 77 (eds. E. Morlet and D. Ribbens)*, pages 1–21. European ACM, North-Holland Publ.Co., Amsterdam, 1977.

[3] D. Bjørner. Domain Models of "The Market" — in Preparation for E–Transaction Systems. In *Practical Foundations of Business and System Specifications (Eds.: Haim Kilov and Ken Baclawski)*, The Netherlands, December 2002. Kluwer Academic Press. Final draft version.

[4] D. Bjørner. From Domains to Requirements. In *Montanari Festschrift*, volume 5065 of *Lecture Notes in Computer Science (eds. Pierpaolo Degano, Rocco De Nicola and José Meseguer)*, pages 1–30, Heidelberg, May 2008. Springer.

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

[6] D. Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. In *Rainbow of Computer Science, Festschrift for Hermann Maurer on the Occasion of His 70th Anniversary.*, Festschrift (eds. C. Calude, G. Rozenberg and A. Saloma), pages 167–183. Springer, Heidelberg, Germany, January 2011.

[14] E-Mail to Dines Bjørner, July 19, 2006

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

[8] D. Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model. In S. Iida, J. Meseguer, and K. Ogata, editors, *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi*. Springer, May 2014.

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

[10] D. Bjørner. From Domain Descriptions to Requirements Prescriptions – A Different Approach to Requirements Engineering. *Submitted for consideration by Formal Aspects of Computing*, 35 pages. 2016.

[11] D. Bjørner. Manifest Domains: Analysis & Description. *Expected published by Formal Aspects of Computing*, 44 pages. 2016.

[12] D. Bjørner. *[14] Chap. 7: Documents – A Rough Sketch Domain Analysis*, pages 179–200. JAIST Press, March 2009.

[13] D. Bjørner. *[14] Chap. 9: Towards a Model of IT Security — – The ISO Information Security Code of Practice – An Incomplete Rough Sketch Analysis*, pages 223–282. JAIST Press, March 2009.

[14] D. Bjørner. *Domain Engineering: Technology Management, Research and Engineering*. A JAIST Press Research Monograph # 4, 536 pages, March 2009.

[15] D. Bjørner and K. Havelund. 40 Years of Formal Methods — 10 Obstacles and 3 Possibilities. In *FM 2014, Singapore, May 14-16, 2014*. Springer, 2014. Distinguished Lecture.

[16] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978. This was the first monograph on $Meta\text{-}IV$.

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

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

[19] D. Bjørner and O. N. Oest, editors. *Towards a Formal Description of Ada*, volume 98 of *LNCS*. Springer, 1980.

[20] CafeOBJ. http://cafeobj.org/, 2014.

[21] G. Clemmensen and O. Oest. Formal specification and development of an Ada compiler – a VDM case study. In *Proc. 7th International Conf. on Software Engineering, 26.-29. March 1984, Orlando, Florida*, pages 430–440. IEEE, 1984.

[22] R. Diaconescu and K. Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. AMAST Series in Computing, 6. World Scientific, Singapore, 1998.

[23] J. Fitzgerald and P. G. Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.

[24] K. Futatsugi, D. Găină, and K. Ogata. Principles of proof scores in CafeOBJ. *Theor. Comput. Sci.*, 464:90–112, 2012.

[25] K. Futatsugi and A. Nakagawa. An overview of CAFE specification environment - an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In *Proc. of 1st International Conference on Formal Engineering Methods (ICFEM '97), November 12-14, 1997, Hiroshima, JAPAN*, pages 170–182. IEEE, 1997.

[26] 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.

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

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

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

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

[31] C. B. Jones. *Software Development: A Rigorous Approach*. Prentice-Hall, 1980.

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

[33] O. Oest. VDM From Research to Practice. In H.-J. Kugler, editor, *Information Processing '86*, pages 527–533. IFIP World Congress Proceedings, North-Holland Publ.Co., Amsterdam, 1986.

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

[35] J. Woodcock, J. B. P.G. Larsen, and J. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):19, 2009.

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

[37] 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.