The Manifest Domain Analysis & Description Approach to Implicit and Explicit Semantics

Invited Talk for the IMPEX Workshop, Xi'an, China, 16 November 2017

Dines Bjørner
Fredsvej 11, DK-2840 Holte, Danmark
E–Mail: bjorner@gmail.com, URL: www.imm.dtu.dk/˜db

November 16, 2017: 00:23 am

Abstract

The domain analysis & description calculi introduced in [1] is shown to alleviate the issue of implicit semantics [2]. The claim is made that domain descriptions, whether informal, or as also here, formal, amount to an explicit semantics for what is otherwise implicit if not described! I claim that [1] provides an answer to the claim in both [2, 3] that "The contexts of the systems in these cases are treated as second-class citizens ...", respectively "In general, modeling languages are not equipped with resources, concepts or entities handling explicitly domain engineering features and characteristics (domain knowledge) in which the modeled systems evolve".

Caveat !

When I wrote this paper I was unaware of [3, Yamine Ait-Ameur and Dominique Méry, *Making explicit domain knowledge in formal system development*, Science of Computer Programming, 121, 120–127]. I was first made aware of and given this paper Nov. 14, 2017. I apologize.

Contents

1 Introduction

1.1 On the Issues of Implicit and Explicit Semantics

In [2] the issues of implicit and explicit semantics are analysed. It appears, from [2], that when an issue of software requirements or of the context, or, as we shall call it, the domain, is not prescribed or described to the extent that is relied upon in the software design, then it is referred to as an issue of implicit semantics. Once prescribed, respectively described, that issue becomes one of explicit semantics. In this paper we offer **a calculus for analysing & describing domains (i.e., contexts), a calculus that allows you to systematically and formally describe domains**.

1.2 A Triptych of Software Engineering

The dogma is:

- *before* **software** *can be* **designed** *we must understand its* **requirements***;*
- and *before we can* **prescribe** *the* **requirements** *we must understand the* **domain,** *that is,* **describe** *the domain.*

A strict, but not a necessary, interpretation of this dogma thus suggests that software development “ideally” proceeds in three phases:

- First a phase of **domain engineering** in which an analysis of the application domain leads to a description of that domain.[1]
- Then a phase of **requirements engineering** in which an analysis of the domain description leads to a prescription of requirements to software for that domain.
- And, finally, a phase of **software design** in which an analysis of the requirements prescription leads to software for that domain.

Proof of program, i.e., software code, correctness can be expressed as:

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

which we read as: proofs that $\mathcal{S}$oftware is correct with respect to $\mathcal{R}$equirements implies references to the $\mathcal{D}$omain.

1.3 Contexts [2] ≡ Domains [1]

Often the domain is referred to as the **context**. We treat contexts, i.e., domain descriptions as first class citizens [2, Abstract, Page 1, lines 9–10]. By emphasizing the formalisation of domain descriptions we thus focus on the *explicit* semantics. Our approach, [1], summarised in Sect. 2 of this paper, thus represents a formal approach to the description of contexts (i.e., domains) [2, Abstract, Page 1, line 12]. By a **domain**, i.e., a context, **description**, we shall here understand an **explicit semantics** of what is usually not specified and, when not so, referred to as **implicit semantics**[2].

1.4 Semantics

I use the term ‘semantics’ rather than the term ‘knowledge’. The reason is this: The entities are what we can meaningfully speak about. That is, the names of the endurants and perdurants, of their being atomic or composite, discrete or continuous, parts, components or materials, their unique identifications, mereologies and attributes, and the types, values and use of operations over these, form the language spoken by practitioners in the domain. It is this language its base syntactic quantities and semantic domains we structure and ascribe a semantics.

1.5 Method & Methodology

By a **method** I understand a set of principles for selecting and applying techniques and tools for constructing a manifest or an abstract artifact. By **methodology** I understand the study and knowledge of methods. **My work is almost exclusively in the area of methods and methodology.**

---

[1] This phase is often misunderstood. On one hand we expect domain stakeholders, e,g,, *bank* associations and university economics departments, to establish “a family” of *bank* domain descriptions: taught when traing and educating new employees, resp. students. Together this ’family’ covers as much as is known about *banking*. On the other hand we expect each new *bank* application (software) development to “carve” out a “sufficiently large” description of the domain it is to focus on. Please replace the term *bank* with an apppropriate term for the domain for which You are to develop software.

[2] “The contexts . . . are treated as second-class citizens: in general, the modelling is implicit and usually distributed between the requirements model and the system model.” [2, Abstract, Page 1, lines 9–12].

1.6 Computer & Computing Sciences

By **computer science** I understand the study and knowledge about the things that can exist inside computing devices.

By **computing science** I understand the study and knowledge about how to construct the things that can exist inside computing devices. Computing science is also often referred to as *programming methodology*. **My work is almost exclusively in the area of computing science.**

2 The Analysis & Description Prompts

We present a calculus of analysis and description prompts[3]. The presentation here is a very short, 12 pages, version of [1, Sects. 2–4, 31 pages]. These prompts are tools that the domain analyser & describer uses. The domain analyser & describer is in the domain, sees it, can touch it, and then applies the prompts, in some orderly fashion, to what is being observed. So, on one hand, there is the necessarily informal domain, and, on the other hand, there are the seemingly formal prompts and the *"suggestions for something to be said"*, i.e., written down: narrated and formalised. See Fig. 1. The figure suggests a number of **analysis** and

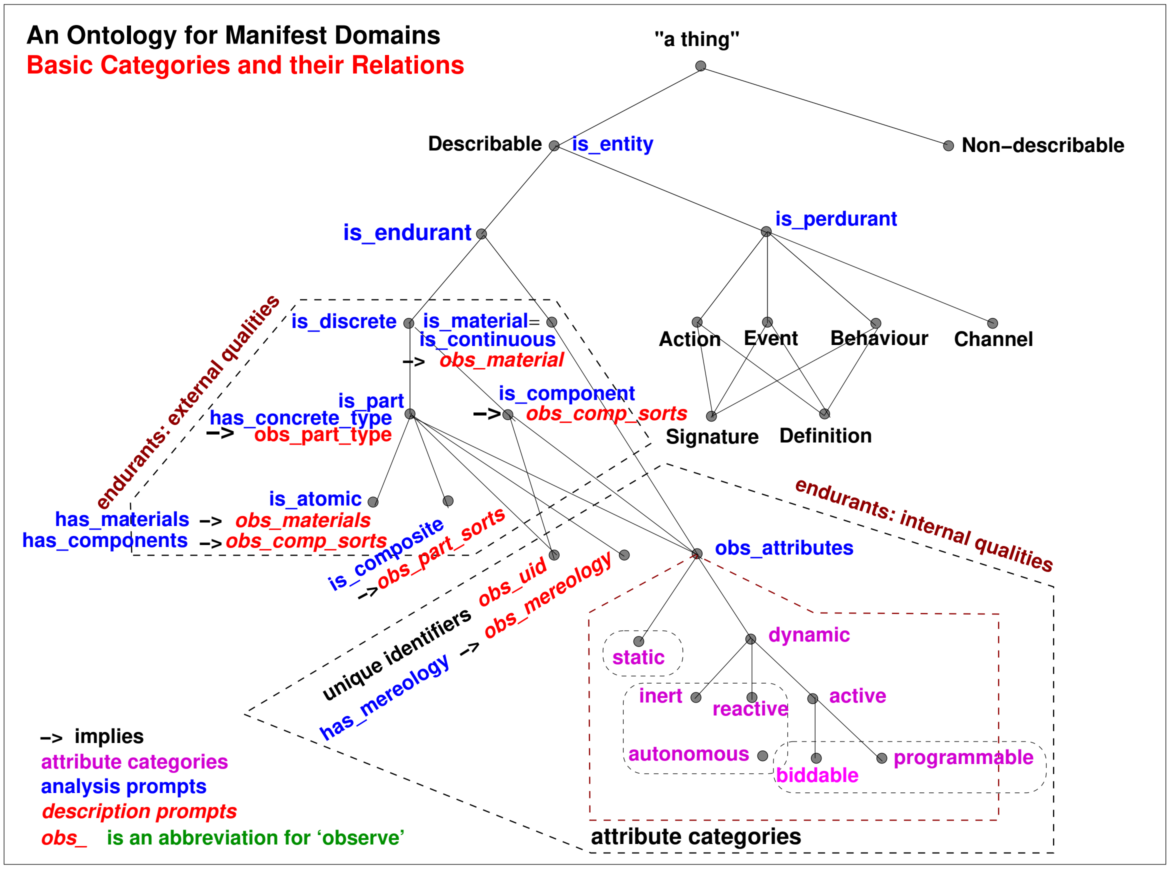


Figure 1: An Ontology for Manifest Domains

---

[3] Prompt, as a verb: to move or induce to action; to occasion or incite; inspire; to assist (a person speaking) by *"suggesting something to be said"*.

description prompts. The domain analyser & describer is “positioned” at the top, the “root”. If what is observed can be conceived and described then it **is an entity**. If it can be described as a “complete thing” at no matter which given snapshot of time then it **is an endurant**. If it is an entity but for which only a fragment exists if we look at or touch them at any given snapshot in time, then it **is a perdurant**.

2.1 Endurants: Parts, Components and Materials

Endurants are either **discrete** or **continuous**. With discrete endurants we can choose to associate, or to not associate *mereologies*[4]. If we do we shall refer to them as **parts**, else we shall call them **components**. With continuous endurants we do not associate mereologies. The continuous endurants we shall also refer to as *(gaseous or liquid)* **materials**. Parts are either **atomic** or **composite** and all parts have *unique identifiers*, *mereology* and *attributes*. If the observed part, $p{:}P$, **is_composite** then we can observe the part sorts and values, $P_1, P_2, ..., P_m$ respectively $p_1, p_2, ..., p_m$ of p. “Applying” **observe_part_sorts** to p yields an informal (i.e., a **narrative**) and a **formal** description:

Schema: Composite Parts

- **Narrative:**
  - ...
- **Formal:**
  - **type**
    - $P_1, P_2, ..., P_m$,
  - **value**
    - obs_P$_i$: $P \rightarrow P_i$,

  repeated for all m part sorts P_is” !

Aircraft **Example 1:** The Pragmatics

The *pragmatics*[5] of this ongoing example is this: We are dealing with ordinary passenger aircraft. We are focusing on that tiny area of concern that focus on passengers being informed of the progress of the flight, once in the air: where is the aircraft: its current position somewhere above the earth; its current speed and direction and possible acceleration (or deceleration); We do not bother about what time it is – etc. We abstract from the concrete presentation of this information.

Aircraft **Example 2:** Parts

1 An *aircraft* is composed from several parts of which we focus on

  a a *position* part,

---

[4] — ‘mereology’ will be explained next

[5] Pragmatics is here used in the sense outlined in [4, Chapter 7, Pages 145–148].

   b a *travel dynamics* part, and
   c a *display* part.

type
1 AC, PP, TD, DP
value
1a obs_PP: AC → PP
1b obs_TD: AC → TD
1c obs_DP: AC → DP

We have just summarised the analysis and description aspects of endurants in *extension* (their "form"). We now summarise the analysis and description aspects of endurants in *intension* (their "contents"). There are three kinds of intensional *qualities* associated with parts, two with components, and one with materials. Parts and components, by definition, have *unique identifiers*; parts have *mereologies*, and all endurants have *attributes*.

2.2 Internal Qualities

2.2.1 Unique Identifiers

Unique identifiers are further undefined tokens that uniquely identify parts and components. The description language observer **uid_P**, when applied to parts $p{:}P$ yields the unique identifier, $\pi{:}\Pi$, of p. So the **observe_part_sorts**(p) invocation also yields the description text:

Schema: Unique Identifiers

- ... [added to the narrative and]
- **type**
  - $\Pi_1, \Pi_2, ..., \Pi_m$;
- **value**
  - uid_Π_i : $P_i \to \Pi_i$,

  repeated for all m part sorts P_is and added to the formalisation.

Aircraft **Example 3:** Unique Identifiers

2 position, travel dynamic and display parts have unique identifiers.

type
2 PPI, TDI, DPI
value
2 uid_PP: PP → PPI
2 uid_TD: TD → TDI

```
2   uid_DP: DP → DPI
```

2.2.2 Mereology

Mereology is the study and knowledge of parts and part relations. The mereology of a part is an expression over the unique identifiers of the (other) parts with which it is related, hence **mereo_P:** $P \to \mathcal{E}(\Pi_j, ..., \Pi_k)$ where $\mathcal{E}(\Pi_j, ..., \Pi_k)$ is a type expression. So the **observe_part_sorts**(p) invocation also yields the description text:

Schema: Mereology

- ... [added to the narrative and]
- **value**
  - mereo_P_i : $P_i \to \mathcal{E}_i(\Pi_{i_j}, ..., \Pi_{i_k})$ [added to the formalisation]

Aircraft **Example 4:** Mereology

We shall omit treatment of aircraft mereologies.

3 The position part is related to the display part.

4 The travel dynamics part is related to the display part.

5 The display part is related to both the position and the travel dynamics parts.

```
value
3  mereo_PP: PP → DPI
4  mereo_TD: TP → DPI
4  mereo_DP: DP → PPI×TDI
```

2.2.3 Attributes

Attributes are the remaining qualities of endurants. The analysis prompt **obs_attributes** applied to an endurant yields a set of type names, $A_1, A_2, ..., A_t$, of attributes. They imply the additional description text:

Schema: Attributes

- **Narrative:**
  - ...
- **Formal:**
  - **type**
    - $A_1, A_2, ..., A_t$

◈ **value**

⊗ attr_A$_i$: $E \to A_i$

repeated for all t attribute sorts A_is !

Aircraft **Example 5:** Position Attributes

6 Position parts have longitude, latitude and altitude attributes.

type
6 LO, LA, AL
value
6 attr_LO: PP → LO
6 attr_LA: PP → LA
6 attr_AL: PP → AL

These quantities: longitude, latitude and altitude are “actual” quantities, they mean what they express, they are not *r*ecordings or *d*isplays of these quantities; to express those we introduce separate types.

Aircraft **Example 6:** Travel Dynamics Attributes

7 Travel dynamics parts have velocity[6] and acceleration[7].

type
7 VEL, ACC
value
7 attr_VEL: TD → VEL
7 attr_ACC: TD → ACC

These quantities: velocity and acceleration, are “actual” quantities, they mean what they express, they are not *r*ecordings or *d*isplays of these quantities; to express those we introduce separate types.

1

Aircraft **Example 7:** Quantity Recordings

8 On one hand there are the actual location and dynamics quantities (i.e., values),

9 on the other hand there are their recodings,

10 and there are conversion functions from actual to recorded values.

type
8 LO, LA, AL, VEL, ACC

---

[6] Velocity is a *vector* of *speed* and *orientation* (i.e., *direction*)

[7] Acceleration is a vector of change of speed per time unit and orientation.

```
9   rLO, rLA, rAL, rVEL, rACC
value
10  a2rLO: LO → rLO, a2rLA: LA → rLA, a2rAL: AL → rAL
10  a2rVEL: VEL → rVEL, a2rACC: ACC → rACC
```

There are, of course, no functions that convert recordings to actual values!

Aircraft **Example 8:** Display Attributes

11 Display parts have display modified longitude, latitude and altitude, and velocity and acceleration attributes – with functions that convert between these, recorded and displayed, attributes.

```
type
11  dLO, dLA, dAL
11  dVEL, dACC
value
11  attr_dLO: DP → dLO
11  attr_dLA: DP → dLA
11  attr_dAL: DP → dAL
11  attr_dVEL: DP → dVEL
11  attr_dACC: DP → dACC
11  r2dLO,d2rLO: rLO ↔ dLO
11  r2dLA,d2rLA: rLA ↔ dLA
11  r2dAL,d2rAL: rAL ↔ dAL
11  r2dVEL,d2rVEL: rVEL ↔ dVEL
11  r2dACC,d2rACC: rACC ↔ dACC
axiom
    ∀ rlo:rLO • d2rLO(r2dLO(rlo))=rlo etcetera !
```

2.2.4 Attribute Categories

Michael A. Jackson [5] categorizes and defines attributes as either *static* or *dynamic*, with dynamic attributes being either *inert*, *reactive* or *active*. The latter are then either *autonomous*, *biddable* or *programmable*. This categorization has a strong bearing on how these (f.ex., part) attributes are dealt with when now interpreting parts as behaviours.

Aircraft **Example 9:** Attribute Categories

12 Longitude, latitude, altitude, velocity and acceleration are all reactive attributes – they change in response to the bidding of aircraft attributes that we have not covered[8].

13 Their display modified forms are all programmable attributes.

```
attribute categories
12  reactive:     LO,LA,AL,VEL,ACC
13  programmable: dLO,dLA,dAL,dVEL,dACC
```

---

[8] – for example: *thrust*, *weight*, *lift*, *drag*, *rudder position*, and *aileron position* – plus dozens of other – attributes

2.3 Description Axioms and Proof Obligations

In [1] we show that the description prompts may result in axioms or proof obligations. We refer to [1] for details. Here we shall, but show one example of an axiom.

Aircraft **Example 10:** An Axiom

14 The displayed attributes must at any time be displayings of the corresponding recorded position and travel dynamics attributes.

```
axiom
14  □ ∀ ac:AC •
14     let (pp,td,di) = (obs_PP(ac),obs_TD(ac),obs_DP(ac)) in
14     let (lo,la,at) = (attr_LO(pp),attr_LA(pp),attr_AT(pp)),
14          (vel,acc,dir) = (attr_VEL(td),obs_ACC(td)),
14          (dlo,dla,dat) = (attr_dLO(di),attr_dLA(di),attr_dAT(di)),
14          (dvel,dacc) = (attr_dVEL(di),obs_dACC(di)) in
14     (dlo,dla,dat) = (r2dLO(a2rLO(lo)),r2dLA(a2rLA(la)),r2dAL(a2rAL(at)))
14  ∧ (dvel,dacc) = (r2dVEL(a2rVEL(vel)),r2dACC(a2rACC(acc)))
14     end end
```

2.4 From Manifest Parts (Endurants) to Domain Behaviours (Perdurants)

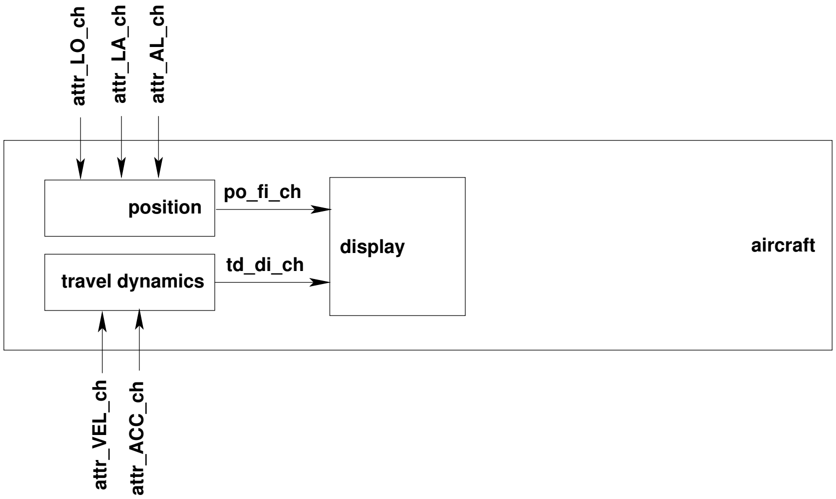
[1] then presents a *compiler* which to manifest *parts* associate *behaviours*. These are then specified as `CSP` [6] *processes*.

2.4.1 The Idea — by means of an example

The term *aircraft* can have the following "meanings": the *aircraft*, as an *endurant*, parked at the airport gate, i.e., as a *composite part;* the *aircraft*, as a *perdurant*, as it flies through the skies, i.e., as a *behaviour;* and the *aircraft*, as an *attribute*, of an airline timetable.

Aircraft **Example 11:** An Informal Story

An aircraft has the following behaviours: the *position* behaviour; it observes the aircraft location attributes: *longitude*, *latitude* and *altitude*, record and communicate these, as a triple, to the *display* behaviour; the *travel dynamics* behaviour; it observes the aircraft travel dynamics attributes *velocity* and *acceleration*, record and communicate these, as a triple, to the *display* behaviour; and the *display* behaviour receives two tuplets of attribute value recordings from respective *position* and *travel dynamics* behaviours and display these recorded attribute values: *longitude, latitude, altitude, velocity* and *acceleration* in some form.



The six actual *position* and *travel dynamics* attribute values *longitude, latitude, altitude, velocity* and *acceleration* are recorded, by appropriate instruments. In the above figure this is indicated by **in**put channels attr_LO_ch, attr_LA_ch, attr_AL_ch, attr_VEL_ch and attr_ACC_ch.

2.4.2 Channels and Communication

Behaviours sometimes synchronise and usually communicate. We use the `CSP` [6] notation (adopted by `RSL`) to model behaviour communication. Communication is abstracted as the sending, ch ! m, and receipt, ch ?, of messages, m:M, over channels, ch.

type M
channel ch:M

Aircraft **Example 12:** Channels

For this example we focus only on communications from the *position* and *travel dynamics* behaviours to the *display* behaviour.

15 The messages sent from the *position* behaviour to the *display* behaviour are triplets of recorded longitude, latitude and altitude values.

16 The messages sent from the *travel dynamics* behaviour to the *display* behaviour are duplets of of recorded velocity and acceleration values.

17 There is a channel, po_di_ch, that allows communication of messages from the position behaviour to the display behaviour.

18 There is a channel, td_di_ch, that allows communication of messages from the travel dynamics behaviour to the display behaviour.

19 For each of the reactive attributes there is a corresponding channel.

type

```
15   PM = rLO × rLA × rAL
16   TDM = rVEL × rACC
channel
17   po_di_ch:PM
18   td_di_ch:TDM
19   attr_LO_ch:LO, attr_LA_ch:LA, attr_AL_ch:AL
19   attr_VEL_ch:VEL, attr_ACC_ch:ACC
```

2.4.3 Behaviour Signatures

We shall only cover behaviour signatures when expressed in `RSL/CSP` [7]. The behaviour functions are now called processes. That a behaviour function is a never-ending function, i.e., a process, is "revealed" in the function signature by the "trailing" **Unit**:

behaviour: ... → ... **Unit**

That a process takes no argument is "revealed" by a "leading" Unit:

behaviour: **Unit** → ...

That a process accepts channel, viz.: ch, inputs, including accesses an external attribute A, is "revealed" in the function signature as follows:

behaviour: ... → **in** ch ... , resp. **in** attr_A_ch

That a process offers channel, viz.: ch, outputs is "revealed" in the function signature as follows:

behaviour: ... → **out** ch ...

That a process accepts other arguments is "revealed" in the function signature as follows:

behaviour: ARG → ...

where ARG can be any type expression:

T, T→T, T→T→T, etcetera

where T is any type expression.

2.4.4 Translation of Part Qualities

Part qualities, that is: *unique identifiers, mereologies* and *attributes*, are translated into behaviour arguments – of one kind or another, i.e., (...). Typically we can choose to *index* behaviour names, b by the *unique identifier*, id, of the part based on which they were translated, i.e., b_{id}. *Mereology values* are usually static, and can, as thus, be treated like we treat static attributes (see next), or can be set by their behaviour, and are then treated like we treat programmable attributes (see next), i.e., (...). *Static attributes* become behaviour definition (body) constant values. *Inert, reactive* and *autonomous attributes* become references to channels, say *ch_dyn*, such that when an inert, reactive and autonomous attribute value is required it is expressed as *ch_dyn* ?. *Programmable* and *biddable attributes* become arguments which are passed on to the tail-recursive invocations of the behaviour, and possibly updated as specified [with]in the body of the definition of the behaviour, i.e., (...).

2.4.5 Part Behaviour Signatures

We can, without loss of generality, associate with each part a behaviour; parts which share attributes (and are therefore referred to in some parts' mereology), can communicate (their "sharing") via channels. A behaviour signature is therefore:

$\mathsf{beh}_{\pi:\Pi}$: me:MT×sa:SA→ca:CA→**in** *ichns*(ea:EA) **in**,**out** *iochs*(me) **Unit**

where (i) π:Π is the unique identifier of part p, i.e., π=**uid_**P(p), (ii) me:ME is the mereology of part p, me = **obs_mereo_**P(p), (iii) sa:SA lists the static attribute values of the part, (iv) ca:CA lists the biddable and programmable attribute values of the part, (v) *ichns*(ea:EA) refer to the external attribute *i*nput *ch*an*n*el*s*, and where (vi) *iochs*(me) are the input/output channels serving the attributes shared between the part p and the parts designated in its mereology me.

Aircraft **Example 13:** Part Behaviour Signatures, I/II

We omit the signature of the aircraft behaviour.

20 The signature of the *position* behaviour lists its unique identifier, mereology, no static and no controllable attributes, but its three reactive attributes (as input channels) and its (output) channel to the *display* behaviour.

21 The signature of the *travel dynamics* behaviour lists its unique identifier, mereology, no static and no controllable attributes, but its three reactive attributes (as input channels) and its (output) channel to the *display* behaviour..

22 The signature of the *display* behaviour lists its unique identifier, its mereology, no static attribute, but the programmable display attributes, assembled in a pair of a triplet and duplet, and its two input channels from the *position*, respectively the *travel dynamics* behaviours.

Aircraft **Example 14:** Part Behaviour Signatures, I/II

```
type
22  DA = (dLA×dLO×dAL)×(dVEL×dACC)
value
20  position: PI × DPI →
20        in attr_LO_ch,attr_LA_ch,attr_AL_ch, out po_di_ch  Unit
21  travel_dynamics: TDI × DPI →
21        in attr_VEL_ch,attr_ACC_ch,attr_DIR_ch, out td_di_ch  Unit
22  display: DI × (PPI×TDI) → DA → in po_di_ch, td_di_ch  Unit
```

2.4.6 Behaviour Compilations

Composite Behaviours Let P be a composite sort defined in terms of sub-sorts P_1, P_2, ..., P_n. The process definition compiled from p:P, is composed from a process description, $\mathcal{M}cP_{\textbf{uid}\_P(p)}$, relying on and handling the unique identifier, mereology and attributes of part p operating in parallel with processes $p_1, p_2, \ldots, p_n$ where p_1 is compiled from p_1:P_1, p_2 is compiled from p_2:P_2, ..., and p_n is compiled from p_n:P_n. The domain description "compilation" schematic below "formalises" the above.

Process Schema: Abstract `is_composite(p)`

value

compile_process: P → RSL-**Text**

compile_process(p) ≡

$\mathcal{M}P_{\textbf{uid}\_P(p)}$(**obs_mereo**_P(p),$\mathcal{S}_\mathcal{A}$(p))($\mathcal{C}_\mathcal{A}$(p))

‖ compile_process(**obs_part**_P_1(p))

‖ compile_process(**obs_part**_P_2(p))

‖ ...

‖ compile_process(**obs_part**_P_n(p))

The text macros: $\mathcal{S}_\mathcal{A}$ and $\mathcal{C}_\mathcal{A}$ were informally explained above. Part sorts P_1, P_2, ..., P_n are obtained from the `observe_part_sorts` prompt.

Aircraft **Example 15:** Aircraft Behaviour, I/II

23 Compiling a composite aircraft part results in the parallel composition
  - a the compilation of the atomic position part,
  - b the compilation of the atomic travel dynamics part, and
  - c the compilation of the atomic display part.

We omit compiling the aircraft core behaviour.

24 Compilation of atomic parts entail no further compilations.

Aircraft **Example 15:** Aircraft Behaviour, II/II

value
23 compile(ac) ≡
23a compile(obs_PP(p))
23b ‖ compile(obs_TD(p))
23c ‖ compile(obs_DI(p))

Atomic Behaviours

Process Schema: `is_atomic(p)`

value

compile_process: P $\rightarrow$ RSL-**Text**

compile_process(p) $\equiv$

$\mathcal{M}P_{\mathsf{uid}\_P(p)}$(**obs_mereo**_P(p),$\mathcal{S}_{\mathcal{A}}$(p))($\mathcal{C}_{\mathcal{A}}$(p))

Aircraft **Example 16:** Atomic Behaviours

25 We initialise the display behaviour with a further undefined value.

value

23a compile(obs_PP(p))$\equiv$

23a position(uid_PP(p),mereo_PP(p))

23b compile(obs_TD(p)) $\equiv$

23b travel_dynamics(uid_TD(p),mereo_TD(p))

25 init_DA:DA = ...

23c compile(obs_DI(p)) $\equiv$

23c display(.uid_DI(p),mereo_DI(p))(init_DA)

In the above we have already subsumed the *atomic behaviour definitions*, see next, and directly inserted the $\mathcal{F}$ definitions.

2.4.7 Atomic Behaviour Definitions

Process Schema IV: Atomic Core Processes

value

$\mathcal{M}P_{\pi:\Pi}$: me:MT$\times$sa:SA $\rightarrow$ ca:CA $\rightarrow$

in $ichns$(ea:EA) **in**,**out** $iochs$(me) **Unit**

$\mathcal{M}P_{\pi:\Pi}$(me,sa)(ca) $\equiv$

let (me$'$,ca$'$) = $\mathcal{F}_{\pi:\Pi}$(me,sa)(ca) **in**

$\mathcal{M}P_{\pi:\Pi}$(me$'$,sa)(ca$'$) **end**

$\mathcal{F}_{\pi:\Pi}$: me:MT$\times$sa:SA $\rightarrow$ CA $\rightarrow$

in $ichns$(ea:EA) **in**,**out** $iochs$(me) $\rightarrow$ MT$\times$CA

Aircraft **Example 17:** Position Behaviour Definition

26 The *position* behaviour offers to receive the *longitude*, *latitude* and the *altitude* attribute values

27 and to offer them to the *display* behaviour,

28 whereupon it resumes being the *position* behaviour.

```
value
20  position(pπ,dπ) ≡
26      let (lo,la,al) = (attr_LO_ch?,attr_LA_ch?,attr_AL_ch?) in
27      po_di_ch ! (a2rLO(lo),a2rLA(la),a2rAL(al)) ;
28      position(pπ,dπ) end
```

Aircraft **Example 18:** Travel Dynamics Behaviour Definition

29 The *travel_dynamics* behaviour offers to receive the recorded *velocity* and the *acceleration* attribute values

30 and to offer these to the *display* behaviour,

31 whereupon it resumes being the *travel_dynamics* behaviour.

```
value
21  travel_dynamics(tdπ,dπ) ≡
29      let (vel,acc)=(attr_VEL_ch?,attr_ACC_ch?) in
30      td_di_ch ! (a2rVEL(vel),a2rACC(acc)) ;
31      travel_dynamics(tdπ,dπ) end
```

Aircraft **Example 19:** Display Behaviour Definition

32 The *display* behaviour offers to receive the reactive attribute tuplets from the *position* and the *travel_dynamics* behaviours while

33 resuming to be that behaviour albeit now with these as their updated display.

34 The **conv**ersion functions are extensions of the ones introduced earlier.

```
value
22  display(dπ,(dπ,tdπ))(d_pos,d_tdy) ≡
32      let (pos_d′,tdy_d′) = (po_di_ch?,td_di_ch?) in
33      display(dπ,(dπ,tdπ))(conv(pos_d′),conv(c_tdy_d′)) end
type
34  dMPD = dLO × dLA × dAL
34  dMTD = dVEL × dACC
value
34  conv: MPD → dMPD
34  conv(rlo,rla,ral) ≡ (r2dLO(rlo),r2dLA(rla),r2dAL(ral))
34  conv: MTD → dMTD
34  conv(rvel,racc) ≡ (r2dVEL(rvel),r2dACC(racc))
```

2.5 A Proof Obligation

We refer, again, to [1] for more on proof obligations.

Aircraft **Example 20:** A Proof Obligation

The perdurant descriptions of Items 15–34 is a model of the axiom expressed in Item 14.

3 Calculations in Classical Domains: Some Simple Observations

This section covers three loosely related topics: Sect. 3.1 muses over properties of some attribute values. Then, Sect. 3.2 we recall some facts about types, scales and values of measurable units in physics. The previous leads us, in Sect. 3.3 to consider further detailing the concept of attributes such as we have covered it in Sect. 2.2.3, Pages and in [1]. The reason for covering these topics is that most attribute values are represented in "final" programs as numbers of one kind or another and that type checking in most software is with respect to these numbers.

3.1 Some Observations on Some Attribute Values

Let us, seemingly randomly, examine some simple, e.g., arithmetic, operations in classical domains. By *time* is often meant absolute time. So a time could be *November 16, 2017: 00:23 am.* One can not add two times. One can speak of a time being earlier, or before another time. *October 23, 2017: 10:01 am* is earlier, $\leq$, than *November 16, 2017: 00:23 am.* One can speak of the time interval between *October 23, 2016: 8:01 am* and *October 24, 2017: 10:05 am* being *1 year, 1 day, 2 hours and 4 minutes,* that is: *October 24, 2017: 10:05 am* $\ominus$ *October 23, 2016: 8:01 am* = *1 year, 1 day, 2 hours and 4 minutes* One can add a *time interval* to a *time* and obtain a *time.* One can multiply a *time interval* with a *real*[9] We can formalize the above:

type

$\mathbb{T} = \mathsf{Month}\times\mathsf{Day}\times\mathsf{Year}\times\mathsf{Hour}\times\mathsf{Minute}\times\mathsf{Sec}\ldots$

$\mathbb{TI} = \mathsf{Days}\times\mathsf{Hours}\times\mathsf{Minutes}\times\mathsf{Seconds}\times\ldots$

$\mathsf{Month} = \{|1,2,3,4,5,6,7,8,9,10,11,12|\}$

$\mathsf{Day} = \{|1,2,3,4,\ldots,28,29,30,31|\}$

$\mathsf{Hour},\mathsf{Hours} = \{|0,1,2,3,\ldots,21,22,23|\}$

$\mathsf{Minute},\mathsf{Minutes} = \{|0,1,2,3,\ldots.,56,57,58,59|\}$

$\mathsf{Second},\mathsf{Seconds} = \{|0,1,2,3,\ldots.,56,57,58,59|\}$

...

$\mathsf{Days} = \mathbf{Nat}$

value

$<,\leq,=,\geq,>\colon\ \mathbb{T} \times \mathbb{T} \rightarrow \mathbf{Boole}$

$-\colon\ \mathbb{T} \times \mathbb{T} \rightarrow \mathbb{TI}\ \mathbf{pre}\ \mathsf{t}-\mathsf{t}'\colon\ \mathsf{t}'\leq\mathsf{t}$

$<,\leq,=,\geq,>\colon\ \mathbb{TI} \times \mathbb{TI} \rightarrow \mathbf{Bool}$

$-,+\colon\ \mathbb{TI} \times \mathbb{TI} \rightarrow \mathbb{TI}$

$*\colon\ \mathbb{TI} \times \mathbf{Real} \rightarrow \mathbb{TI}$

$/\colon\ \mathbb{TI} \times \mathbb{TI} \rightarrow \mathbf{Real}$

One can not add temperatures – makes no sense in physics! But one can take the mean value of two (or more) temperatures. One can subtract temperatures obtaining positive or negative temperature intervals. One can take the mean of any number of temperature, but would

---

[9] The time interval could, e.g., be converted into seconds, then the integer number standing for seconds can be multiplied by r and the result be converted "back" into years, days, hours, minutes and seconds — whatever it takes!

probably be well advised to have these represent regular sampling, or at least time-stamped. One can also define *rate of change of temperature*.

```
type
    Temp, MeanTemp, Degrees, TempIntv = Degrees
value
    mean: Temp-set × Nat → MeanTemp
    −: Temp × Temp → TempIntv
type
    TST = (Temp × 𝕋)-set
value
    avg: TST → MeanTemp
type
    TimeUnit = {|"year","month","day"",hour",...|}
    RoTC = TempIntv × TimeUnit
```

Etcetera. We leave it to the reader to speculate on which operations one can perform on a persons' attributes: height, weight, birth date, name, etc. And similarly for other domains. It is time to "lift" these observations. After the examples above we should inquire as to which kind of units we may operate upon. For the sake of our later exposition it is enough that we look in some detail at the "universe" of physics.

3.2 Physics Attributes

3.2.1 SI: The International System of Quantities

In physics we operate on values of attributes of manifest, i.e., physical phenomena. The type of some of these attributes are recorded in well known tables, cf. Tables 1–3.

Table 1 shows the base units of physics.

Base quantity	Name	Type
length	meter	m
mass	kilogram	kg
time	second	s
electric current	ampere	A
thermodynamic temperature	kelvin	K
amount of substance	mole	mol
luminous intensity	candela	cd

Table 1: Base Units

Table 2 on the facing page shows the units of physics derived from the base units.

Table 3 on page 20 shows further units of physics derived from the base units.

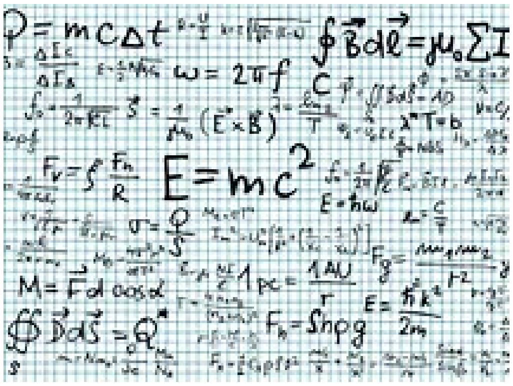
Table 4 on page 20 shows standard prefixes for SI units of measure.

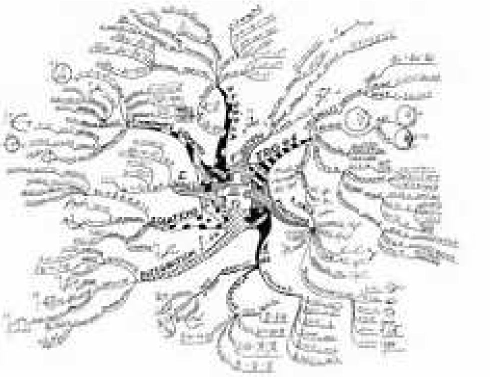
Table 5 on page 21 shows fractions of SI units of measure.

These "pictures" are meant as an eye opener, a "teaser".

Name	Type	Derived Quantity	Derived Type
radian	rad	angle	m/m
steradian	sr	solid angle	m^{2}×m^{-2}
Hertz	Hz	frequency	s^{-1}
newton	N	force, weight	kg×m×s^{-2}
pascal	Pa	pressure, stress	N/m^{2}
joule	J	energy, work, heat	N×m
watt	W	power, radiant flux	J/s
coulomb	C	electric charge	s×A
volt	V	voltage, electromotive force	W/A (kg×m^{2}×s^{-3}×A^{-1})
farad	F	capacitance	C/V (kg^{-1}×m^{-2}×s^{4}×A^{2})
ohm	Ω	electrical resistance	V/A (kg×m^{2}×s^{3}×A^{2})
siemens	S	electrical conductance	A/V (kg1×m^{2}×s^{3}×A^{2}
weber	Wb	magnetic flux	V×s (kg×m^{2}×s^{-2}×A^{-1})
tesla	T	magnetic flux density	Wb/m^{2} (kg×s^{2}×A^{-1})
henry	H	inductance	Wb/A (kg×m^{2}×s^{-2}×A^{2})
degree Celsius	^{o}C	temperature relative to 273.15 K	K
lumen	lm	luminous flux	cd×sr (cd)
lux	lx	illuminance	lm/m^{2} (m^{2}×cd)

Table 2: Derived Units





And these formulas likewise!

$$\text{Efficiency} = \frac{W}{Q_h}$$

$$\frac{Q_c}{Q_h} = \frac{T_c}{T_h}$$

$$\text{Efficiency} = 1 - \left(\frac{Q_c}{Q_h}\right) = 1 - \left(\frac{T_c}{T_h}\right)$$

$$\text{Coefficient of performance} = \frac{Q_h}{W}$$

$$\text{Coefficient of performance} = \frac{1}{1-\left(Q_c \div Q_h\right)} = \frac{1}{1-\left(T_c \div T_h\right)}$$

Carnot Engine

$$p = \frac{m}{V}$$

$$P = \frac{F}{A}$$

$$\Delta P = pgh$$

$$F_{buoyancy} = W_{water\ displaced}$$

$$p_1 A_1 v_1 = p_2 A_2 v_2$$

$$P_1 + \frac{1}{2} p v_1^2 + p g y_1 = P_2 + \frac{1}{2} p v_2^2 + p g y_2$$

Bernoulli Flow

The point in bringing this material is that when modelling, i.e., describing domains we

Name	Explanation	Derived Type
area	square meter	m^2
volume	cubic meter	m^3
speed, velocity	meter per second	m/s
acceleration	meter per second squared	m/s^2
wave number	reciprocal meter	m-1
mass density	kilogram per cubic meter	kg/m^3
specific volume	cubic meter per kilogram	m3/kg
current density	ampere per square meter	A/m^2
magnetic field strength	ampere per meter	A/m
amount-of-substance concentration	mole per cubic meter	mol/m3
luminance	candela per square meter	cd/m^2
mass fraction	kilogram per kilogram	kg/kg = 1

Table 3: Further Units

Prefix name		deca	hecto	kilo	mega	giga	tera	peta	exa	zetta	yotta
Prefix symbol		da	h	k	M	G	T	P	E	Z	Y
Factor	10^0	10^1	10^2	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}	10^{21}	10^{24}

Table 4: Standard Prefixes for SI Units of Measure

must be extremely careful in not falling into the trap of modelling physics, etc., types as we do in programming!

3.2.2 What Are We to Learn from this Exposition ?

We see from the previous section , Sect. 3.2, that physics units can be highly “structured”[10]. What Are We to Learn from this Exposition ? I think it is this: It is customary, in programs of languages from `Algol 60` via `Pascal` to `Java`, to assign `float` or `double`[11] **type**s, as in `Java`, to [constants or] variables that for example represent values of physics. *So rather completely different types of physics units are all cast into a same, simple-minded, “number” type.* **No chance, really, for any meaningful type checking.**

3.3 Attribute Types, Scales and Values: Some Thoughts

This section further elaborates on the treatment of attributes given in Sect. 2.2.3, Pages 7–9. The elaboration is only sketched. It need be studied, in detail.

The elaboration is this: The attr_A observer function, for a part p of sort P, such as defined in Sect. 2.2.3 (Page 8)yields values of type A. In the revised understanding of attributesthe attr_A observer is now to yield both the type, AT, and the value, AV, of attribute A:

type
    AT, AV
value
    attr_A: P → AT × AV

---

[10] For example, Newton: $kg \times m \times s^{-2}$, Volt = $kg \times m^2 \times s^{-3} \times A^{-1}$, etc.

[11] representing single-, resp. double-precision 32-bit IEEE 754 floating point values

Prefix name		deci	centi	milli	micro	nano	pico	femto	atto	zepto	yocto
Prefix symbol		d	c	m	μ	n	p	f	a	z	y
Factor	10^0	10^{-1}	10^{-2}	10^{-3}	10^{-6}	10^{-9}	10^{-12}	10^{-15}	10^{-18}	10^{-21}	10^{-24}

Table 5: Fractions

You may think of A being defined by AT × AV.

The revision is further that a domain analysis & description of the operations over attributes values, θ:

$$\theta\text{: } \mathsf{A}_i \times \mathsf{A}_j \times \ldots \times \mathsf{A}_k \to \mathsf{V}$$

be carefully checked – such as hinted at in Sect. 3.1 on page 17.

Whether such operator-checks be researched and documented "once-and-for-all" for given "standard" domains, by domain scientists, or per domain model, by domain engineers, in connection with specific software development projects is left for you to decide! These operator-checks, if not pursued, results in implicit semantics, and if pursued, results in explicit semantics.

4 Conclusion

4.1 What Have We Achieved ?

We have suggested that the issue of implicit semantics [2] be resolved by providing a carefully analysed and described domain model [1] prior to requirements capture and software design, a both informally annotated and formally specified model that goes beyond [1] in its treatment of attributes in that these are now endowed with types [and possibly scales (or fractions)] and that each specific domain model analyses and formalises the constraints that operations upon attribute values are carefully analysed, statically.

4.2 Domain Descriptions as Basis for Requirements Prescriptions

This paper covers but one aspect of software development.

- [8] covers additional facets of domain analysis & description.
- **[9] offers a systematic approach to requirements engineering based on domain descriptions. It is this approach that justifies our claim that domain modelling "alleviate the issue of implicit semantics."**
- [10] presents an operational/denotational semantics of the manifest domain analysis & description calculus of [1].
- [11][12] shows that to every manifest mereology there corresponds a `CSP` expression.
- [12] muses over issues of software simulators, demos, monitors and controllers.

---

[12] Accepted for publication in *Journal of Logical and Algebraic Methods in Programming*, 2018.

4.3 What Next ?

Well, there is a lot of fascinating research to be done now. Studying analysis & description techniques for attribute types, values and constraints. And for engineering their support.

4.4 Thanks

to J. Paul Gibson and Dominique Méry for inviting me, to J. Paul Gibson for organising my flights, hotel and registration, and to Dominique Méry for his patience in waiting for my written contribution.

5 Bibliographical Notes

5.1 References to Draft Domain Descriptions

- *Urban Planning* [13],
- *Documents* [14],
- *Credit Cards* [15],
- *Weather Information Systems* [16],
- *The Tokyo Stock Exchange* [17],
- *Pipelines* [18],
- *Road Transportation* [19],
- *Transaction-based Web Software* [20],
- *"The Market"* [21],
- *Container [Shipping] Lines* [22] and
- *Railway Systems* [23, 24, 25, 26, 27].

I apologise for the numerous references to own reports and publications.

5.2 References

[1] Dines Bjørner. Manifest Domains: Analysis & Description. *Formal Aspects of Computing*, 29(2):175–225, March 2017. DOI 10.1007/s00165-016-0385-z http://link.springer.com/article/10.1007/s00165-016-0385-z.

[2] Yamine Ait-Ameur, J Paul Gibson, and Dominique Méry. On Implicit and Explicit Semantics: Integration issues in proof-based development of systems. In T. Maragaria and B. Steffen, editors, *Leveraging Applications ofcFormal Methods, Verification and Validation. Specialized Techniques and Applications. ISoLA 2014*, number 8803 in Lecture Notes in Computer Science, pages 604–618, Berlin, Heidelberg, 2014. Springer.

[3] Yamine Ait-Ameur and Dominique Méry. Making explicit domain knowledge in formal system development. *Science of Computer Programming*, (121):120–127, 2016.

[4] 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.

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

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

[7] 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.

[8] Dines Bjørner. Domain Facets: Analysis & Description. 2016. Extensive revision of [28]. http://www.imm.dtu.dk/˜dibj/2016/facets/faoc-facets.pdf.

[9] Dines Bjørner. From Domain Descriptions to Requirements Prescriptions – A Different Approach to Requirements Engineering. 2016. Extensive revision of [29].

[10] Dines Bjørner. Domain Analysis and Description – Formal Models of Processes and Prompts. 2016. Extensive revision of [30]. http://www.imm.dtu.dk/˜dibj/2016/process/process-p.pdf.

[11] Dines Bjørner. To Every Manifest Domain a `CSP` Expression — A Rôle for Mereology in Computer Science. *Journal of Logical and Algebraic Methods in Programming*, Accepted for publication. 2018. http://www.imm.dtu.dk/˜dibj/2016/mereo/mereo.pdf.

[12] Dines Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. Technical report, Fredsvej 11, DK–2840 Holte, Denmark, 2016. Extensive revision of [31]. http://www.imm.dtu.dk/˜dibj/2016/demos/faoc-demo.pdf.

[13] Dines Bjørner. Urban Planning Processes. Research Note, July 2017. http://www.imm.dtu.dk/˜dibj/2017/up/urban-planning.pdf.

[14] Dines Bjørner. What are Documents ? Research Note, July 2017. http://www.imm.dtu.dk/˜dibj/2017/docs/docs.pdf.

[15] Dines Bjørner. A Credit Card System: Uppsala Draft. Technical Report: Experimental Research, Fredsvej 11, DK–2840 Holte, Denmark, November 2016. http://www.imm.dtu.dk/˜dibj/2016/credit/accs.pdf.

[16] Dines Bjørner. Weather Information Systems: Towards a Domain Description. Technical Report: Experimental Research, Fredsvej 11, DK–2840 Holte, Denmark, November 2016. http://www.imm.dtu.dk/˜dibj/2016/wis/wis-p.pdf.

[17] Dines Bjørner. The Tokyo Stock Exchange Trading Rules. R&D Experiment, Fredsvej 11, DK-2840 Holte, Denmark, January and February, 2010. Version 1, 78 pages: many auxiliary appendices. http://www2.imm.dtu.dk/ db/todai/tse-1.pdf, Version 2, 23 pages: omits many appendices and corrects some errors.. http://www2.imm.dtu.dk/ db/todai/tse-2.pdf.

[18] Dines Bjørner. Pipelines – a Domain Description. http://www.imm.dtu.dk/˜dibj/pipe-p.pdf. Experimental Research Report 2013-2, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.

[19] Dines Bjørner. Road Transportation – a Domain Description. http://www.imm.dtu.dk/˜dibj/road-p.pdf. Experimental Research Report 2013-4, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.

[20] Dines Bjørner. On Development of Web-based Software: A Divertimento of Ideas and Suggestions. Technical, Technical University of Vienna, August–October 2010. http://www.imm.dtu.dk/˜dibj/wfdftp.pdf.

[21] Dines 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. http://www2.imm.dtu.dk/ db/themarket.pdf.

[22] Dines Bjørner. A Container Line Industry Domain. Techn. report, Fredsvej 11, DK-2840 Holte, Denmark, June 2007. Extensive Draft. http://www2.imm.dtu.dk/ db/container-paper.pdf.

[23] Dines Bjørner. Formal Software Techniques in Railway Systems. In Eckehard Schnieder, editor, *9th IFAC Symposium on Control in Transportation Systems*, pages 1–12, Technical University, Braunschweig, Germany, 13–15 June 2000. VDI/VDE-Gesellschaft Mess– und Automatisieringstechnik, VDI-Gesellschaft für Fahrzeug– und Verkehrstechnik. Invited talk.

[24] Dines Bjørner, Chris W. George, and Søren Prehn. Computing Systems for Railways — A Rôle for Domain Engineering. Relations to Requirements Engineering and Software for Control Applications. In *Integrated Design and Process Technology. Editors: Bernd Kraemer and John C. Petterson*, P.O.Box 1299, Grand View, Texas 76050-1299, USA, 24–28 June 2002. Society for Design and Process Science. Extended version. http://www2.imm.dtu.dk/ db/pasadena-25.pdf.

[25] Dines Bjørner. Dynamics of Railway Nets: On an Interface between Automatic Control and Software Engineering. In *CTS2003: 10th IFAC Symposium on Control in Transportation Systems*, Oxford, UK, August 4-6 2003. Elsevier Science Ltd. Symposium held at Tokyo, Japan. Editors: S. Tsugawa and M. Aoki. Final version. http://www2.imm.dtu.dk/ db/ifac-dynamics.pdf.

[26] Martin Pěnička, Albena Kirilova Strupchanska, and Dines Bjørner. Train Maintenance Routing. In *FORMS'2003: Symposium on Formal Methods for Railway Operation and Control Systems*. L'Harmattan Hongrie, 15–16 May 2003. Conf. held at Techn.Univ. of Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany. Final version. http://www2.imm.dtu.dk/ db/martin.pdf.

[27] Albena Kirilova Strupchanska, Martin Pěnička, and Dines Bjørner. Railway Staff Rostering. In *FORMS2003: Symposium on Formal Methods for Railway Operation and Control Systems*. L'Harmattan Hongrie, 15–16 May 2003. Conf. held at Techn.Univ. of Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany. Final version. http://www2.imm.dtu.dk/ db/albena.pdf.

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

[29] Dines 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.

[30] Dines Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model. In Shusaku Iida, José Meseguer, and Kazuhiro Ogata, editors, *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi*. Springer, May 2014.

[31] Dines 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.