Domain Modelling
A Foundation for Software Development[*]

Dines Bjørner

The Technical University of Denmark
Fredsvej 11, DK-2840 Holte, Danmark
E–Mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

Abstract. Domain modelling, as per the approach of this paper, offers the possibility of describing software application domains in a precise and comprehensive manner – well before requirements capture can take place. We endow domain modelling with appropriate analysis and description calculi and a systematic method for constructing domain models. The present paper is a latest exposé of the *domain science & engineering* as published in earlier papers and a book. It reports on our most recent simplifications to the *domain analysis & description* approach.

The Triptych Dogma

In order to *specify* software,
we must understand its requirements.

In order to *prescribe* requirements,
we must understand the domain.

So we must study, analyse and describe domains.

1 Introduction

This paper introduces the possibility of a *new phase of software development*, one that precedes requirements engineering, as well as *a new way of looking at the world around us!*

Today's well-managed software development projects usually start with some form of *requirements "capture"*. Now the possibility arises to precede this phase of requirements engineering with an initial phase of domain engineering.

The present paper is an improvement over previously published accounts [13, 16, 17]: builds upon a simpler domain ontology (Fig. 1 on page 4); has fewer domain concepts (Sects. 3 and 5); and presents a more rational way of "deriving"

[*] This paper is being submitted for evaluation as a contribution to the He JiFeng 80th anniversary Festschrift: Theories of Programming and Formal Methods: Essays Dedicated to Jifeng He on the Occasion of His 80th Birthday, Springer Verlag, 14–15 September 2023

behaviours from parts (Sect. 6). Taken together the presentation is thus made shorter and more precise.

The approach to the modelling of domains put forward in this paper has two major phases: modelling *external qualities* of the world as we see it, as it manifests itself to us, or otherwise, and modelling the *internal qualities*, as we may not see it, but qualities that can be measured and/or spoken about. The modelling of external qualities has a few steps. The major step of modelling of external qualities is that of deciding upon the atomic-, Cartesian- and set-oriented parts. A minor step, following the major step, is that of identifying a notion of *endurant state*. The modelling of internal qualities has a few more steps. The modelling of *unique identifiers*; the modelling of *mereologies*; the modelling of *attributes*; and the modelling of *'intentional pull'*. It is this structuring into manageable stages and steps that reassures us, i.e., me, that the approach is sound.

1.1 What is a Domain ?

By a *domain* we shall understand a *rationally describable* segment of a *discrete dynamics* fragment of a *human assisted reality*, i.e., of the world: its *endurants*, i.e., *solid and fluid entities:* whether *natural* ["God-given"] or *artefactual* ["man-made"], their *parts* and *living species entities:* whether *atomic* or *compound* parts, respectively whether *plant* or *animal* living species, including *humans* — as well as its *perdurants:* the *behaviours* of *parts* and *living species*.

Clearly this *characterisation* does not possess the rigour that should be common in software development. Terms such as *rationally describable*, *discrete dynamics* and *human assisted reality* must be not just assumed, but must, below, be made more precise. Yet precision defies us: The domains we shall study, analyse and describe are not amenable to such precision. *The world is not formal.*

Thus the *domain analysis & description* methodology that we shall be concerned with is not directed at *continuous dynamics* systems such as we find them in for example aerospace applications. And we shall not, in this paper be concerned with the *human assistance* aspects.

By *domain modelling* we mean the study, analysis and description of a domain.

If the domain already exists, then the modelling amounts to a faithful rendering of that domain but such that the resulting model, i.e., description, covers as wide a spectrum of domain instances as is deemed reasonable.[1] We shall, in this paper, assume already existing domains.

By *domain engineering* we mean the construction of domain models.[2]

[1] Thus a railway domain model should desirably cover such instances as the railways of Denmark and Norway and Sweden, each one individually.

[2] The approach taken here can, however, also be used to devise new domains.

1.2 Non-computable and Computable Specifications

When specifying[3] software we usually make use of a formal language – one whose semantics can be expressed mathematically. And the specification had better be *logically tractable*. Similarly for *prescribing requirements:* again a formal language can be deployed. And the requirement had better be *computable*. Typically, when we derive a *software specification*, $\mathcal{S}$, from a *requirements prescription*, $\mathcal{R}$, the *testing, model checking* and *proof* of some form of *correctness*, $\mathcal{D}, \mathcal{S} \models \mathcal{R}$, of the software design relies on not only on relations between the two documents: the $\mathcal{R}$ and $\mathcal{S}$, but also on the *domain description*, $\mathcal{D}$. But in *describing domains* we cannot assume computability. It is the task of *requirements engineering* to "derive" computable requirements from domain models. [17, *Chapter 9*] shows how. We refer to Sect. 8.2.3 on page 38 for summary comments.

1.3 Formal Method and Methodology

By a *method* we shall understand a set of *principles* for *selecting* and *applying* a number of *procedures, techniques* and *tools* for [effectively] *constructing* an *artefact*. By *methodology* we shall understand the *study* and *knowledge* of one or more *methods*. By a *formal method* we shall understand a *method* which *uses* one or more *formal specification languages* as per their intention: *specification* and *verification* (*formal tests, model checks* and *proofs of properties* of *domains descriptions*, *requirement prescriptions* and *software designs*). By a *formal specification language* we shall understand a language with a *formal syntax*, a *formal semantics* and a *proof system* with which to *describe & validate*[4] *domains, prescribe & validate requirements* and *specify (design) & validate software*.

Our domain analysis & description method has been developed, over the years, with this understanding of formal methods.

1.4 From Programming Languages to Domains

Domain stakeholders, those whose primary work is in and of the domain, name the entities of the domain and use these names, nouns and verbs, in communicating with other stakeholders. These utterings constitute a language, albeit an informal one. In a domain model we give *abstract syntax* to (roughly speaking) the *nouns*, Sects. 3 and 5, and *semantics* to (roughly speaking) the *verbs*, Sect. 6.

When, in comparison, we define the syntax and semantics of a programming language, that syntax and semantics covers all well-formed instances of programs in that language. Similarly, when, in consequence, we define the abstract syntax and semantics, i.e., a model, of a domain, that syntax and semantics covers all well-formed instances — we mean it, the model, to cover all well-formed instances of domains.

[3] By specifying software we mean specifying the design of the software. That design is derived from the software requirements.

[4] test, check and verify

1.5 A Review

We present a latest exposé of the *domain science & engineering* of [13, 16, 17, *2015–2021*]. The first inklings of this *applied science* were first reported in [3, *1995–1997*], Volume III, Part IV, Chapters 8–12, Pages 193–362 of [4, *2006*] cover several aspects of *domain engineering* – but not what we now consider the most important contribution to the field: namely that of the *analysis & description calculi*. First developments of the proposed *analysis and description calculi* were reported in [9, 10, *Kyiv 2010*]. The recently published papers and book [13, 16, 17, *2015–2021*] illustrates the fact that the details of the *calculi* may change. The present paper reports on our most recent simplification to the *domain analysis & description* approach and the few extensions, $\texttt{RSL}^+$, to the `RSL` specification language [32]. The domain modelling approach presented here has been honed over the last 30 years in numerous experiments. Some of these are reported in [18, 15, 19, 22].

1.6 An Overview

1.6.1 A Domain Analysis & Description Ontology Sections 3–6 represent the contribution of this paper. Figure 1 illustrates basic ideas of how we shall structure our *domain analysis & description*.

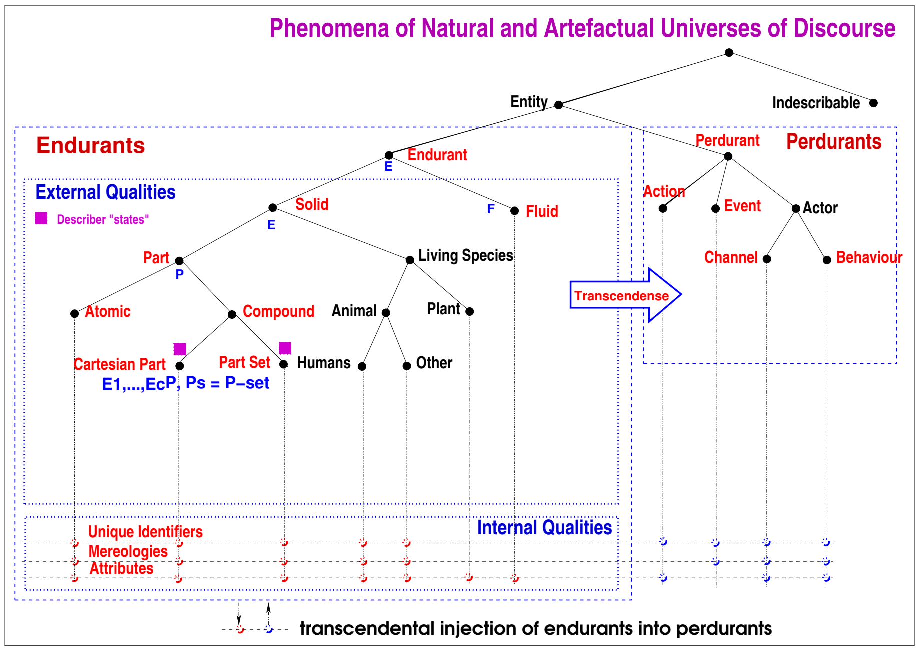

Fig. 1. An Analysis & Description Methodology Ontology

The *domain analyser cum describer*, i.e., the *domain modeller*, is confronted by a domain. How and where to start ! Figure 1 on the preceding page is intended to be read top-down, left-to-right. So it suggests that the *domain modeller*, starts by looking "at the whole domain ! ". That is, at the • right under the term Universes, between the `r` and the `s` !

1.6.2 Step-wise Analysis and Description Figure 1 then suggests, by the two lines emerging from that •, that the *domain modeller* poses the question, of the domain, `is it` (more or less) `rationally describable`, i.e., `is_entity(`ϕ`)`, or not. If the *domain modeller* decides yes, it is so, then the analysis "moves" on to the Entity •. Now the question is, is the entity being observed, an endurant or a perdurant, (to be explained below), and so on. We now assume that the analysis proceeds along the left hand side dashed line (· · ·- - - -· · ·) box labeled 'Endurants'.

The so-called *external quality analysis* of endurants ends when reaching either of the Atomic, Cartesian or Part Set •s.[5] At this point the description proceeds to that of the *internal qualities* of endurants. From Fig. 1 on the facing page You observe seven vertical [dashed] lines, emanating downwards from endurant bullets to cross three horizontal (bottom of the figure) lines. They "call" for the *domain modeller* to now analyse and describe the internal qualities of endurants: their *unique identification*, their *mereologies*, and their *attributes*.

Eventually the *domain modeller* has "traversed" the left hand side of Fig. 1 on the preceding page. At this point a *transcendental deduction* takes place: The *domain modeller* now *"morphs" manifest* endurant parts into *behaviours*. The focal point here are the part behaviour signatures and definitions. Figure 1 on the facing page's right hand side hints at the issues to be covered and that the internal qualities are being a crucial element of behaviour definitions.

1.6.3 The Analysis and Description Prompts Each • of Fig. 1 on the preceding page thus corresponds to an analysis or description prompt. There are two kinds of analysis prompts. Both are informal. The predicate analysis prompts and the function analysis prompts. There is two major kinds of description prompts. (α) external quality description prompts – with there being two such specific prompts: one for describing so-called *Cartesian* endurants (Sect. 3.3.1 on page 12), another for describing so-called *Part Set* endurants (Sect. 3.3.2 on page 13), and (β) internal quality description prompts with there being three such specific prompts: the *unique identifier* description prompt (Sect. 5.1.1 on page 16), the *mereology* description prompt (Sect. 5.2.1 on page 17), and the *attribute* description prompt (Sect. 5.3.2 on page 18). The predicate analysis prompts yield truth values. The function analysis prompts yield part endurants and the names of their type – which we shall call sorts. And the description

[5] We shall, in this paper, not exemplify living species endurants.

prompts yield domain description texts – here in a slight extended version of the RAISE[6] [33] specification language RSL [32].[7,8]

1.7 RSL, RSL-text and RSL$^+$

RSL is described in [32]. We use a subset of that RSL. Thus we shall not avail ourselves of the RSL module concepts of *object*, *class* and *scheme*. Basically, then, a specification expressed in RSL amounts to sequences of [alternating] *type*, *value* and *axiom* clauses – with, optionally, a single *channel* clause:

```
type            type            channel         type
    ...             ...             ...             ...
value           value           type            value
    ...             ...             ...             ...
axiom           axiom           value           axiom
    ...             ...             ...             ...
                                axiom
                ...                 ...
```

RSL-text is an addition to RSL. In describing domains in RSL we shall be introducing *description prompts* which are informal functions which yield values of type RSL-text, that is, proper RSL texts. Quoting an RSL text: " text ". shall denote an RSL-text.

RSL$^+$ designate RSL-text plus, in this paper, one extension. That extension is that of the type and values of type names. If T denotes a type, i.e., a possibly infinite set of values, then ηT denotes a value, the name of type T, with $\phi\mathbb{T}$ denoting the type of type names.

The domain analysis & description method is informally explained in a mixture of English and RSL$^+$. [12, *2014*] attempts a formalisation of an early version of RSL$^+$.

1.8 A Computer Science Philosophy

We shall base our *domain analysis & description* approach on the philosophy of Kai Sørlander [54–58]. The issue here is: In *studying, analysing & describing* domains one is confronted with the basic [metaphysical] question[s]: *which are the absolutely necessary conditions for describing any world ?*, that is: *what, if anything, is of such necessity, that it could under no circumstances be otherwise ?*, or: *which are the necessary characteristics of any possible world ?* In his work Sørlander rationally argues that space, time, Newton's laws, and a number of additional concepts are necessarily basic elements of any description of any domain.

[6] Rigorous Approach to Idustrial Software Engineering
[7] RSL: RAISE Specification Language
[8] Other formal specification languages are possible, f.ex.: VDM [23, 24, 30], Z [59], Alloy [43], or CafeOBJ [31].

1.9 Previous Work

We refer to Sect. 1.5 on page 4.

Axel van Lamsweerde [49, *2009*] and Michael A. Jackson [44, 45, *1995–2010*], as well as other *requirements engineering* researchers, do touch upon the issues of domains – such as that term is basically used here. But their *requirements analysis and prescription* do not "put it center stage", let alone mandate that the[ir] *requirements engineer* rely on an a priori established *domain description*. So they and others do not establish, as is the main focus of this contribution, calculi for the analysis & description of domains.

1.10 Structure of Paper

There are basically two parts to this paper. The main part consists of Sects. 2–3 and Sects. 5–6. They present a terse, comprehensive exposé of the *domain analysis & description* method of this paper. An appendix, the other part, Appendix 7, brings an example. For the domain modelling approach to be believable the example must open up for a realistic domain, one that is not "small".

• • •

We now explain *the domain description ontology* as a *structured set of concepts for modelling domains, a set that shows their properties and the relations between them. In simple terms, ontology seeks the classification and explanation of entities.*[9]

Figure 1 on page 4 is a graphical rendition of a *structured set of concepts for modelling domains*.

2 Universe of Discourse

Domain descriptions start with a terse sketch of the main facets of the domain followed by the naming of the domain.

1. Universe of Discourse: `calc_UoD`

Narration:
    Text
Formalisation:
    type UoD

1 *Example. Universe of Discourse: We refer to Sect. 7.1 on page 25.*

[9] *Google's English Dictionary* as provided by *Oxford Languages*.

3 External and Internal Qualities

Characterisation 1: External qualities: External qualities of endurants[10] of a domain are, in a simplifying sense, those properties of endurants that we can see, touch and which have spatial extent. They, so to speak, take form.

Characterisation 2: Internal qualities: Internal qualities of endurants of a domain are those which we may not be able to see or "feel" when touching an endurant, but they can, as we now 'mandate' them, be reasoned about. They have unique identifiers and mereologies,[11] And they have attributes that can be measured by some physical/chemical means, or be "spoken of" by intentional deduction.

3.1 Predicate Analysis of External Qualities of Endurants

Characterisation 3: Phenomenon: By a *phenomenon we shall understand a fact that is observed to exist or happen* ■ Examples of phenomena are: emotions of a human, the rivers, lakes, forests, mountains and valleys of mother nature; the railway tracks, their units, the locomotive of a railway system.

Domain Analysis Predicates: We shall define a number of domain analysis predicates. They are all referred to as prompts. Prompts are method tools. The domain modeller applies these to "real", i.e., actual world phenomena, that is, not to formal values. In the next 18 paragraphs we shall "reveal" a number of such predicates. First with a *reasonable definition (in slanted font)*, then with examples and some comments (in roman font).

Predicate Prompt 1: is_entity: *By an entity we shall understand something that can be observed, i.e., be seen or touched by humans, or that can be conceived as an abstraction of an entity; alternatively, a tangible or conceivable phenomenon is an entity [50, Vol. I, pg. 665]* ■ Some, but not necessarily all aspects of a river can be rationally described, hence can be still be considered entities. Similarly, many aspects of a road net can be rationally described, hence will be considered entities.

Predicate Prompt 2: is_endurant: Endurants are those quantities of domains that we can observe (see and touch), in space, as "complete" entities at no matter which point in time – "material" entities that persists, endures [50, Vol. I, pg. 656] ■ Street segments [links], street intersections [hubs], automobiles standing still in an automobile show room are endurants. Domain endurants, when eventually modelled in software, typically become data. Hence the careful analysis of domain endurants is a prerequisite for subsequent careful conception and analyses of data structures for software, including data bases.

Predicate Prompt 3: is_perdurant: *By a perdurant we shall understand an entity for which only a fragment exists if we look at or touch at any given snapshot in time. Were we to freeze time we would only see or touch a fragment of the perdurant [50, Vol. II, pg. 1552]* ■ Automobiles in action, container vessels sailing

[10] We refer to predicate prompt # 2 below for a definition of *endurant*.
[11] We refer to Sects. 5.1–5.2.

on the 7 seas and loading and unloading containers in harbours are examples of perdurants. Domain perdurants, when eventually modelled in software, typically become processes.

Endurants are either *solid* endurants, or are *fluid* endurants.

Predicate Prompt 4: is_solid: *By a solid endurant we shall understand an endurant which is separate, individual or distinct in form or concept, or, rephrasing: a body or magnitude of three-dimensions, having length, breadth and thickness [50, Vol. II, pg. 2046]* ■ Wells, pipes, valves, pumps, forks, joins, regulator, and sinks of a pipeline are solids.

Predicate Prompt 5: is_fluid: By a fluid endurant we shall understand an endurant which is prolonged, without interruption, in an unbroken series or pattern; or, rephrasing: a substance (liquid, gas or plasma) having the property of flowing, consisting of particles that move among themselves [50, Vol. I, pg. 774] ■ Fluids are otherwise liquid, or gaseous, or plasmatic, or granular[12], or plant products[13], et cetera. Specific examples of fluids are: water, oil, gas, compressed air, etc. A container, which we consider a solid endurant, may be *conjoined* with another, a fluid, like a gas pipeline unit may "contain" gas.

We analyse endurants into either of two kinds: *parts* and *living species*. The distinction between *parts* and *living species* is motivated in Kai Sørlander's Philosphy [54–58].

Predicate Prompt 6: is_part: By a part we shall understand a solid endurant existing in time and space and subject to laws of physics, including the causality principle and gravitational pull[14] ■

Natural and man-made parts are either atomic or compound.

Predicate Prompt 7: is_atomic: By an atomic part we shall understand a part which the domain analyser considers to be indivisible in the sense of not meaningfully, for the purposes of the domain under consideration, that is, to not meaningfully consist of sub-parts ■ The wells, pumps, valves, pipes, forks, joins and sinks of a pipeline can be considered atomic.

Predicate Prompt 8: is_compound: Compound parts are those which are either Cartesian-product- or are set- oriented parts ■

Predicate Prompt 9: is_Cartesian: Cartesian parts are those (compound parts) which consists of an "indefinite number" of two or more parts of distinctly named sorts ■ Some clarification may be needed. (i) In mathematics, as in RSL [32], a value is a Cartesian ("record") value if it can be expressed, for example as $(a, b, ..., c)$, where $a, b, ..., c$ are mathematical (or, which is the same, RSL) values. Let the sort names of these be $A, B, ..., C$ – with these being required to be distinct. We wrote "indefinite number": the meaning being that the number is fixed, finite, but not specific. (ii) The requirement: 'distinctly named' is prag-

[12] This is a purely pragmatic decision. "Of course" sand, gravel, soil, etc., are not fluids, but for our modelling purposes it is convenient to "compartmentalise" them as fluids!

[13] i.e., chopped sugar cane, threshed, or otherwise. See footnote 12.

[14] This characterisation is the result of our study of relations between philosophy and computing science, notably influenced by Kai Sørlander's Philosphy

matic. If the domain modeller thinks that two or more of the components of a Cartesian part [really] are of the same sort, then that person is most likely confused and must come up with suitably distinct sort names for these "same sort" parts! (iii) Why did we not write "definite number"? Well, at the time of first analysing a Cartesian part, the domain modeller may not have thought of all the consequences, i.e., analysed, the compound part. Additional sub-parts, of the Cartesian compound, may be "discovered", subsequently and can then, with the approach we are taking wrt. the modelling of these, be "freely" added subsequently! We refer to the road transport system example above. We there viewed (hubs, links and) automobiles as atomic parts. From another point of view we shall here understand automobiles as Cartesian parts: the engine train, the chassis, the car body, four doors (left front, left rear, right front, right rear), and the wheels. These may again be considered Cartesian parts.

Predicate Prompt 10: is_part_set: Part sets are those which, in a given context, are deemed to meaningfully consist of an indefinite number of sub-parts of the same sort ■ Examples of set parts are: the set of hubs of a road net hub aggregate, the set of links of a road net link aggregate, and the set of automobiles of an automobile aggregate – all of the road net transport that we are exemplifying.

Predicate Prompt 11: is_living_species: By a living species we shall understand a solid endurant, subject to laws of physics, and additionally subject to causality of purpose. Living species must have some form they can develop to reach; a form they must be causally determined to maintain. This development and maintenance must further engage in exchanges of matter with an environment ■ It must be possible that living species occur in two forms: plants, respectively animals. Although we have not yet come across domains for which the need to model the living species of plants were needed. Hence:

Predicate Prompt 12: is_plant: *Plants are living species which are characterised by development, form and exchange of matters with an environment* ■

Predicate Prompt 13: is_animal: Animals are living species which are additionally characterised by the ability of purposeful movement ■

Predicate Prompt 14: is_human: A human (a person) is an animal, with the additional properties of having language, being conscious of having knowledge (of its own situation), and responsibility ■

Characterisation 4: Manifest Part: By a manifest part we shall understand a part which 'manifests' itself either in a physical, visible manner, "occupying" an AREA or a VOLUME and a POSITION in SPACE, or in a conceptual manner forms an organisation in Your mind! ■ As we have already revealed, endurant parts can be transcendentally deduced into perdurant behaviours – with manifest parts indeed being so.

Predicate Prompt 15: is_manifest: *`is_manifest(e)` holds if e is manifest* ■

Characterisation 5: Structure: By a structure we shall understand an endurant concept that allows the domain modeller to rationally decompose a domain analysis and/or its description into manageable, logically relevant sections, but where these abstract endurants are not further reflected upon in the domain analysis *and description* ■ Structures are therefore not transcendentally deduced into perdurant behaviours.

Predicate Prompt 16: is_structure: `is_structure(e)` *holds if e is a structure* ■

Examples of structures arise as the result of our analysis of parts. Thus a road net could be modelled as the composite of two structures: a set of hubs and a set of links (the stretches between two adjacent hubs, i.e., road intersections), cf. Items 6– 7 on page 25.

Predicate Prompt 17: is_stationary: An endurant part is stationary if it never changes position in space ■

Predicate Prompt 18: is_mobile: An endurant part is mobile if it may possibly change position in space ■

We may need, occasionally, the distinction as now outlined:

Endurants are either *natural* endurants, or are *artefactual* endurants.

Predicate Prompt 19: is_natural: By a natural endurant we shall understand one which has been created by nature.

Predicate Prompt 20: is_artefactual: By an artefactual endurant we shall understand one which has been created by humans.

Discrete Dynamic and Artefactual Domains: In our initial characterisation of domains, Page 2, an emphasis was put on their *discrete dynamics and human assistedness*. The analysis and description calculi and, hence, our domain modelling, are therefore "geared" in that direction.

We are not offering to model time continuous domains. See Sect. 8.2.9 on page 39.

We summarise[15]:

2. Analysis Predicates

```
value
  is_entity: Φ → Bool         is_living_species: E → Bool   is_human: E → Bool
  is_endurants: E → Bool      is_atomic: E → Bool           is_manifest: E → Bool
  is_perdurant: E → Bool      is_compound: E → Bool         is_structure: E → Bool
  is_solid: E → Bool          is_animal: E → Bool           is_stationary: E → Bool
  is_fluid: E → Bool          is_plant: E → Bool            is_mobile: E → Bool
  is_part: E → Bool           is_Cartesian: E → Bool        is_natural: E → Bool
                              is_part_set: E → Bool         is_artefactual: E → Bool
```

2 *Example. Analysis Predicates: In the example of Appendix 7 on page 25–37 we do not [explicitly] show the "application" of analysis predicates. They are tacitly assumed.*

[15] Framed texts highlight *domain analysis & description* prompts.

3.2 Functional Analysis of External Qualities of Endurants

Given a compound endurant, that is, either a Cartesian or a part set, we analyse that compound, at the two •'s of Fig. 1 on page 4, into its constituent endurants, respectively parts, and the name of the sort:

3. determine_Cartesian_parts, determine_part_set

```
value
    determine_Cartesian_parts: E → (E1×ηΦ)×(E2×ηΦ)×...×(Ec×ηΦ)
    determine_Cartesian_parts(e) as (e1:ηE1,e2:ηE2,...,ec:ηEc)

    determine_part_set: E → P-set×ηΦ
    determine_part_set(e) as ({p1,p2,...,ps}:ηP,)
```

The above calculation function signatures and characterisations illustrate two extensions to RSL [32]: ηP expresses the name of a sort P, and $\eta\Phi$ expresses the type of sort names.

Again we emphasize that these calculations are performed by the domain modeller. They are used in subsequent schemas for describing external qualities of endurants.

3.3 Descriptions of External Qualities of Endurants

Similarly, again at the two ■'s of Fig. 1 on page 4, we are now ready to describe respectively Cartesian parts and part set parts.

3.3.1 Describing Cartesian Parts

4. descr_Cartesian

```
value
    descr_Cartesian: P → RSL-Text
    descr_Cartesian(p) ≡
            "Narrative:
                    [s] text on sorts
                    [o] text on observers
                    [a] text on axioms and/or proof obligations
             Formalisation:
                    [s] type
                                E1, E2, ..., Ec
                    [o] value
                                obs_E1: E→E1, obs_E2: E→E2, ..., obs_Ec: E→Ec
                    [a] axiom and/or proof obligation
                                A/P(...) "
```

3 *Example. Cartesians: We refer to Sect. 7.2.1 on page 25.*

3.3.2 Describing Part Sets

5. descr_part_set

```
value
    descr_part-set: P → RSL-Text
    descr_part_set(p) ≡
        "Narrative:
            [s] text on sorts
            [o] text on observers
            [a] text on axioms and/or proof obligations
         Formalisation:
            [s] type
                    P, Ps = P-set
            [o] value
                    obs_Ps: E→Ps
            [a] axiom and/or proof obligation
                    𝒜/𝒫(...) "
```

4 *Example. Part Sets: We refer to Sect. 7.2.2 on page 25.*

3.4 Endurant States

Characterisation 6: Endurant State: By an *endurant state* we shall understand any collection of endurant parts ■

6. obs_Σ

```
value
Σ = P-set
value
obs_Σ: E → Σ
obs_Σ(e) ≡
    if is_manifest(e)
     then
        is_atom(e) → {e},
        is_Cartesian(e) →
         let (p1:ηE1,p2:ηE2,...,pc:ηEc)=calc_cartesian_parts_and_sorts(e) in
         {p1,p2,...,pc}∪obs_Σ(p1)∪obs_Σ(p2)∪...∪obs_Σ(pc) end
        is_part-set(e) →
         let ({p1,p2,...,ps}:ηP)=calc_part_sets_parts_and_sort(e) in
         {p1,p2,...,ps}∪obs_Σ(p1)∪obs_Σ(p2)∪...∪obs_Σ(ps) end
```

```
 else {}
end
```

5 *Example. Endurant State Examples: We refer to Sect. 7.2.3 on page 26.*

3.5 An Explication, I

The concept of *analysis predicates* and *part observer functions* is due to McCarthy [52, *Sect.12-13*].

In [52] McCarthy introduces a notion of *abstract syntax*, Sect. 12, and *semantics*, Sect. 13. So far we have dealt, in our domain analysis, with syntax. There are three elements, according to McCarthy, to consider: the `is_...` predicates, the `obs_...` ["destructor"] functions, and, not shown, so far, in this paper, the `mk_...` constructor functions. For compound abstract syntactic entities they are related as follows:

```
is_Cartesian(p) ≡
    let (p1:ηP1,p2:ηP2,...,pc:ηPc) = calc_Cartesian_parts_and_sorts(p) in
    p = mk_Cartesian(obs_P1(p),obs_P2(p),...,obs_Pc(p)) end

is_part_set(p) ≡
    let ({p1,p2,...,ps},ηP1) = calc_part_sets_parts_and_sort(p) in
    p = mk_part_set({p1,p2,...,ps}) end
```

The `mk_...` constructors were not introduced above. The reason is simple; a pragmatic decision: As the domain modeller proceeds in their work they may, when encountering Cartesian compounds, be free to leave some components (of the Cartesian) out, components that they may later introduce. So really, the first of the identities above ought be expressed as

```
is_Cartesian(p) ≡
    let (p1η:P1,p2:ηP2,...,pc:ηPc,...) = calc_Cartesian_parts_and_sorts(p) in
    p = mk_Cartesian(obs_P1(p),obs_P2(p),...,obs_Pc(p),...) end
```

We continue this explication in Sect. 5.5 on page 20.

4 Space and Time

The concepts of space and time can be *transcendentally deduced*, by rational reasoning, as has been shown in [54–58, *Kai Sørlander*], from the facts of *symmetry, asymmetry, transitivity* and *intransitivity* relations.

They are therefore facts of every possible universe.

4.1 Space

There is one given space. As a type we name it $\mathbb{SPACE}$. We do not bother, here, about textual representation of spatial locations, but here is an example that would work in or near this globe we call our earth: `Latitude 55.805600, Longitude 12.448160, Altitude 35 m`[16].

Also, in this paper, we do not present models of $\mathbb{SPACE}$. But we do introduce such notions as (i) $\mathbb{POINT}$: as $\mathbb{SPACE}$ being some dense and infinite collection of points; (ii) $\mathbb{LOCATION}$: as the location in space of some point;

value record_$\mathbb{LOCATION}$: $\mathrm{E} \rightarrow \mathbb{LOCATION}$

(iii) $\mathbb{CURVE}$: as an infinite collection of points forming a mathematical curve – having a (finite or infinite) *length*; (iv) $\mathbb{SURFACE}$: as an infinite collection of points forming a mathematical surface – having a (finite or infinite) *area*; and (v) $\mathbb{VOLUME}$: as an infinite collection of points forming a mathematical volume – having a (finite or infinite) *volume*. We suggest it, as a domain science & engineering research topic, that somebody studies *a calculus or calculi of spatial modelling*.

4.2 Time

There is one given time. As a type we name it $\mathbb{TIME}$. We do not bother, here, about textual representation of time, but here is an example: `April 19, 2023: 13:43`[17]. But we do introduce such crucial notions as *time interval* $\mathbb{TI}$ and operations on $\mathbb{TIME}$ and $\mathbb{TI}$:

value
    $-$: $\mathbb{TIME}\times\mathbb{TIME}\rightarrow\mathbb{TI}$
    $+$: $\mathbb{TIME}\times\mathbb{TI}\rightarrow\mathbb{TIME}$
    $*$: Real$\times\mathbb{TI}\rightarrow\mathbb{TI}$

A crucial time-related operation is that of record_$\mathbb{TIME}$. It applies to "nothing": record_$\mathbb{TIME}$() and yields $\mathbb{TIME}$.

value record_$\mathbb{TIME}$: Unit $\rightarrow$ $\mathbb{TIME}$

5 Internal Qualities

We refer to the *Internal Qualities* characterisation on Page 8. We can justify the grouping of internal endurant qualities into three kinds: *unique identifiers*, cf. Sect. 5.1, *mereologies*, cf. Sect. 5.2, and *attributes*, cf. Sect. 5.3. To this we add the concept of *intentional pull*, cf. Sect. 5.4.

[16] The author's house location!
[17] The time this text was last compiled!

5.1 Unique Identification

On the basis of *philosophical reasoning*, within *metaphysics*, we [can] argue that parts are uniquely identifiable [54–58, *Kai Sørlander*]

5.1.1 Calculate Unique Identifiers

7. descr_unique_identifier

```
value
    descr_unique_identifier: P → RSL-Text
    descr_unique_identifier(p) ≡
         "Narrative:
              [s] text on unique identifier sort
              [o] text on unique identifier observer
              [a] text on axioms and/or proof obligations
          Formalisation:
              [s] type
                       PI
              [o] value
                       uid_P: P → PI
              [a] axiom and/or proof obligation
                       𝒜/𝒫(...) "
```

6 *Example. Unique Identifiers: We refer to Sect. 7.3.1 on page 26.*

5.1.2 Endurant Identifier States Given the endurant state values, for the whole domain or for respective, manifest part sorts, one can define corresponding unique identifier values.

7 *Example. Unique Identifier State: We refer to Sect. 7.3.2 on page 27.*

5.1.3 Axioms The number of manifest parts is the sames as the number of manifest part unique identifiers.

8 *Example. Unique Identifier Axiom: We refer to Sect. 7.3.3 on page 27.*

5.1.4 Endurant Retrieval Given a unique identifier, π, of a manifest part, p, of an endurant state, σ, of a domain one can retrieve that part:

```
value
    σ:Σ = gen_Σ(uod)
    retr_P: Π → Σ → P
    retr_P(π)(σ) ≡ let p:P • p ∈ σ ∧ uid_P(p)=π in p end
```

5.2 Mereology

Mereology is the study and knowledge of parts and part relations. It was first put forward, around 1916, by the Polish logician *Stanisław Leśniewski* [51, 26].

Which are the relations that can be relevant to being an endurant ? There are basically two relations: (i) physical ones, and (ii) conceptual ones. (i) Physically two or more endurants may be topologically either adjacent to one another, like rails of a line, or within an endurant, like links and hubs of a road net, or an atomic part is conjoined to one or more fluids, or a fluid is conjoined to one or more parts. The latter two could also be considered conceptual "adjacencies". (ii) Conceptually some parts, like automobiles, "belong" to an embedding endurant, like to an automobile club, or are registered in the local department of vehicles, or are intended to drive on roads.

5.2.1 Calculate Mereologies

8. descr_mereology

```
value
    descr_mereology: P → RSL-Text
    descr_mereology(p) ≡
        " Narrative:
              [ s ] text on mereology type
              [ o ] text on  mereology observer
              [ a ] text on axioms and/or proof obligations
          Formalisation:
              [ s ] type
                        MT = 𝓜(p)
              [ o ] value
                        mereo_P: P → MT
              [ a ] axiom and/or proof obligation
                        𝒜/𝒫(...) "
```

$\mathcal{M}$(p) is usually a type expression over unique identifiers of mereology-related parts.

9 *Example. Mereology: We refer to Sect. 7.4 on page 27.*

Given the definition of external qualities of a domain, and its unique identifier and mereology internal qualities one can analyse and describe many properties of that domain. The *routes* subsection (Page 28) of the mereology example, Example 9, illustrates one such property.

5.3 Attributes

Parts and fluids are typically recognised because of their spatial form and are otherwise characterised by their intangible, but measurable attributes. That is, whereas endurants, whether solid (as are parts) or fluids, are physical, tangible, in the sense of being spatial [or being abstractions, i.e., concepts, of spatial endurants], attributes are intangible: cannot normally be touched, or seen, but can be objectively measured. Thus, in our quest for describing domains where humans play an active rôle, we rule out subjective "attributes": feelings, sentiments, moods. Thus we shall abstain, in our domain science also from matters of psychology and aesthetics.

5.3.1 Functional Analysis of Attributes

Given a manifest part, p, that is, either an atom, or a Cartesian, or a part set, we calculate from that part, its constituent attributes values and types:

9. determine_attributes

```
value
  determine_attributes: P → (a1×ηA1)×(a2×ηA2)×...×(aa×ηAa)
```

5.3.2 Describe Attributes

10. descr_attributes

```
value
  descr_attributes: P → RSL-Text
    let ((_,ηA1),(_,ηA2),...,(_,ηAa))=determine_attributes(p:P) in
  descr_attributes(p) ≡
    "Narrative:
      [s] text on attribute types
      [o] text on attribute observers
      [a] text on axioms and/or proof obligations
    Formalisation:
      [s] type
              A1 [=...], A2 [=...], ..., Aa [=...],
      [o] value
              attr_A1: P→A1, attr_A2: P→A2, ..., attr_Aa: P→Aa,
      [a] axiom and/or proof obligation
              $\mathcal{A}/\mathcal{P}(...)$ "
      end
```

The domain modeller has thus determined/decided that A1, A2, ..., Aa are the "interesting" attributes of of parts of sort P. Attributes are often given a "concrete" form, hence the [= ...] where the ... is some type expression.

10 *Example. Attributes: We refer to Sect. 7.5 on page 29.*

5.3.3 Attribute Categories Michael A. Jackson has proposed a structure of attributes [44].

Attribute Category 1: Static: By a static attribute we shall understand an attribute whose values are constants, i.e., cannot change.

Attribute Category 2: Dynamic: By a dynamic attribute we shall understand an attribute whose values are variable, i.e., can change. Dynamic attributes are either inert, reactive or active attributes.

Attribute Category 3: Inert: By an inert attribute we shall understand a dynamic attribute whose values only change as the result of external stimuli where these stimuli prescribe new values.

Attribute Category 4: Reactive: By a reactive attribute we shall understand a dynamic attribute whose values, if they vary, change in response to external stimuli, where these stimuli either come from outside the domain of interest or from other endurants.

Attribute Category 5: Active: By an active attribute we shall understand a dynamic attribute whose values change (also) of its own volition. Active attributes are either autonomous, or biddable or programmable attributes.

Attribute Category 6: Autonomous: By an autonomous attribute we shall understand a dynamic active attribute whose values change only "on their own volition". The values of an autonomous attributes are a "law onto themselves and their surroundings".

Attribute Category 7: Biddable: By a biddable attribute we shall understand a dynamic active attribute whose values are prescribed but may fail to be observed as such.

Attribute Category 8: Programmable: By a programmable attribute we shall understand a dynamic active attribute whose values can be prescribed.

We modify Jackson's categorisation. This is done in preparation for our exposé of behaviour signatures, cf. Sect. 6.4.1 on page 23. Figure 2 shows groupings of some of M. A. Jackson's basic categories.

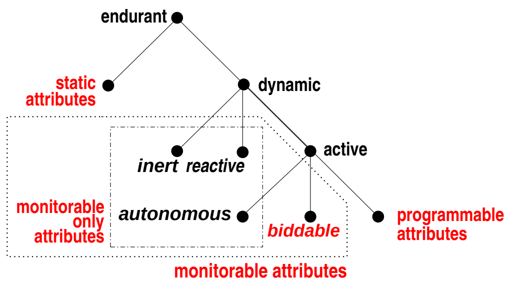

Fig. 2. An Attribute Ontology

Our motivation for modifying Jackson's attribute categories is as follows: when transcendentally deducing behaviours from parts we find that there are basically a need for distinguishing between only three major attribute categories the static, the monitorable, and the programmable attributes. Static attributes have their values passed "by value", as constants, programmable attributes have their values passed by "by reference", as variables who value can be changed, and monitorable attributes have their values passed by "by name" – as we shall see!

5.4 Intentional Pull

5.4.1 Characterisations Intentionality as a philosophical concept is defined by the Stanford Encyclopedia of Philosophy[18] as *"the power of minds to be about, to represent, or to stand for, things, properties and states of affairs."*

Intent is then a usually clearly formulated or planned intention. An example of *intent* is that of roads made for automobiles and automobiles meant for roads.

Intentional Pull[19]: Two or more artefactual parts of different sorts, but with overlapping sets of intents may excert an intentional "pull" on one another. This intentional "pull" may take many forms. Let p_x: X and p_y:Y be two parts of different sorts (X,Y), and with common intent, ι. Manifestations of these, their common intent, must somehow be subject to constraints, and these must be expressed predicatively. When a composite artefact has an intentionality then its constituents have individual intentionalities that relate to these. The composite road transport system has intentionality of the roads serving the automobiles, and the automobiles have the intent of being served by the roads.

11 *Example. Intentional Pull: Road Transport: We refer to Sect. 7.6 on page 30.*

12 *Example. Intentional Pull: Double-entry Bookkeeping: Double-entry bookkeeping, also known as double-entry accounting, is a method of bookkeeping that relies on a two-sided accounting entry to maintain financial information. Every entry to an account requires a corresponding and opposite entry to a different account. The double-entry system has two equal and corresponding sides known as debit and credit. A transaction in double-entry bookkeeping always affects at least two accounts, always includes at least one debit and one credit, and always has total debits and total credits that are equal.*[20] .

5.5 A Proof-theoretic Explication, II

We remind You of Sect. 3.5 on page 14.

[18] Jacob, P. (Aug 31, 2010). Intentionality. Stanford Encyclopedia of Philosophy (-`seop.illc.uva.nl/entries/intentionality/` October 15, 2014, retrieved April 3, 2018.

[19] The term *intentional pull* is chosen so as to connote with the term *gravitational pull*.

[20] https://en.wikipedia.org/wiki/Double-entry_bookkeeping

With the introduction of *analysis functions* and *observers* for *unique identifiers, mereology* and *attributes* we can now augment the `is_...`, `uid_...`, `mereo_...`, `attr_A...` observers introduced since Page 14.

```
is_manifest(p:P) ≡
        let ((_,ηA1),(_,ηA2),(_,ηAa)) = calc_attributes(p) in
        p = mk_P(uid_P(p),mereo_P(p),(attr_A1(p),attr_A2(p),...,attr_Aa(p))) end
```

6 Perdurants

A key point of our domain science & engineering approach is this: to every manifest part we transcendentally deduce a unique behaviour.

By transcendental *we shall understand the philosophical notion: the a priori or intuitive basis of knowledge, independent of experience.*

By a transcendental deduction *we shall understand the philosophical notion: a transcendental 'conversion' of one kind of knowledge into a seemingly different kind of knowledge.*

6.1 Channels

Part behaviours may interact. To express part behaviours and their interaction we use Hoare's CSP [38–40]. One may question this choice. In [7, 11, 14, *2009–2017*] we show *"that to every mereology there is a CSP expression"*. On that background we maintain that CSP is a reasonable choice — but invite the reader to suggest more appropriate mechanisms for handling behaviours and their communication.[21]

So, in general, we declare a RSL/CSP *channel*:

11. channel declaration

```
channel { ch[{ui,uj}] | ui,uj:UI • {ui,uj}⊆uis } : M
```

Here ch is the name of the indexed array of channels and the indexes are, in general, any two element sets of unique part identifiers. That is: For every pair of part behaviours – identified but their unique part identifiers (ui,uj) – there is a channel, say ch[{ui,uj}].

M is the type of the messages communicate between behaviours of index ui,uj.

We shall develop, in Sect. 6.2.2 on the following page, the specifics of the type, M, of channel messages.

[21] Please bear in mind that the use, here, of CSP, is in the following context: the CSP clauses are not to be "interpreted" on a computer where this "computerisation" has to be "shared" with other computations; hence CSP *synchcronisation & communication* is "ideal" and reflects reality.

6.2 Actors

By an *actor* we shall understand either an *action*, or an *event*, or a *behaviour*.

6.2.1 Actions

By an *action of a behaviour* we shall understand something which is local to a behaviour, and, which, when applied, potentially changes the states. Generally action clauses are expressed in `RSL` [32].

13 *Example. Road Transport Actions: We refer to Sect. 7.7.2.1 on page 33.*

6.2.2 Events

By an *event of a behaviour* we shall understand something that involves two behaviours, and, which, when applied, potentially changes the states of both behaviours. Event clauses are expressed using the `CSP` elements of `RSL`. That is, the `CSP` output "*!*" and input events "*?*":

```
ch[{ui,uj}] ! expr
let val = ch[{ui,uj}] ? ... end
```

14 *Example. Road Transport Events: We refer to Sect. 7.7.2.2 on page 33.*

6.3 State Access and Updates

We need define two functionals: one for changing the mereology of a part and another for changing the attribute value of a part. We therefore informally define the following functionals:

6.3.1 Update Mereologies

- part_update_mereology is a functional: it takes the following arguments: a part p of type P and a mereology value and yields a part of type P.
- The yielded result, p′, has the same unique identifier, as the argument part p,
- a new, the argument, mereology, as the argument part p,
- and the same attribute values for all attributes, as the argument part p.

```
value
    part_update_mereology: P → M → P
    part_update(p)(m) ≡
        let ((_,ηA1),(_,ηA2),...,(_,ηAa)) = determine_attributes(p) in
        let p′:P • uid_P(p′)=uid_P(p)∧mereo_P(p′)=m∧
            ∀ ηA:ηΦ•ηA∈{ηA1,ηA2,...,ηAa}⇒attr_A(p′)=attr_A(p) in
        p′ end end
```

6.3.2 Update Attributes

- part_update_attribute is a functional: it takes the following arguments: a part p of type P and a pair of an attribute name and value, and yields a part p′ of type P.
- The argument attribute name must be that of an attribute of the part.
- The yielded result p′ has the same unique identifier and mereology as the argument part p,
- and the same attribute values for all attributes, as the argument part p, except for argument attribute (name) for which it now yields the argument attribute value.

```
value
  part_update_attribute: $P \to \Phi A \times A \to P$
  part_update_attribute(p)($\eta$A,a) $\equiv$
    let ((_,$\eta$A1),(_,$\eta$A2),...,(_,$\eta$Aa)) = determine_attributes(p) in
        assert: $\eta A \in \{\eta A1,,\eta A2,...,,\eta Aa\}$
    let p′:P • uid_P(p)=uid_P(p′)$\wedge$mereo_P(p)=mereo_P(p′)$\wedge$
        $\forall\ \eta A:\eta\Phi \bullet \eta A \in \{\eta A1,,\eta A2,...,,\eta Aa\} \backslash \eta A \Rightarrow$attr_A(p′)=attr_A(p) in
    p′ end end
```

Examples of monitorable attributes are: an automobile's velocity and engine (cooler) temperature. Monitorable attributes usually change their values surreptitiously. That is, "behind the back", so-to-speak, of the part behaviour.

6.4 Behaviours

By a *behaviour* we shall understand a set of sequences of actions, events and behaviours.

6.4.1 Behaviour Signatures: We now come to a crucial point in our unrolling the *domain science & engineering method.* It is that of explaining the signature of behaviours, that is, the arguments ascribed to part behaviours. The general form of part p behaviour signatures is as follows.

12. Behaviour Signatures

```
value
p_behaviour: p:P→in,out {ch[{uid_P(p),ui}]|ui:UI•ui∈$uis \wedge \mathcal{M}ereo$(p)} Unit
```

Yes, that is it! The behaviour of a[ny] (manifest) part, p, is a function whose only argument is that part! The signature informs of the channels that p_behaviour may communicate with. The literal Unit informs that the behaviour may not yield any value, but, for example, go on "forever" having possibly effected a state change!

6.4.2 Behaviour Definitions:
Behaviours, besides their signatures, are defined. That is, a *behaviour definition 'body'* describes, in, for us, using `RSL` [32] with its embodiment of a variant of `CSP` [40], basically `CSP` clauses how it interacts with other behaviours, and, in basically `RSL`'s functional specification (read: programming) clauses, how it otherwise "goes about its business" !

In fragment I the focus is on the possible [action] update of either biddable or programmable attributes.

13. Behaviour Definition, I

```
p_behaviour(p) ≡
        let p′ = possible_update_of_biddable_and_programmable_attributes(p) in
        p_behaviour(p′) end
```

In fragment II the focus is on the possible [action] value access to any attributes.

14. Behaviour Definition, II

```
p_behaviour(p) ≡ ... attr_A(p) ... p_behaviour(p)
```

In fragment III the focus is on the possible interaction with other behaviours, hence illustrates two events as seen from one behaviour.

15. Behaviour Definition, III

```
p_behaviour(p) ≡
        ...
        let (val,ui) = ℰ(p) in ch[ {uid_P(p),ui} ] ! val end ;
        ...
        let uj = ℐ(p) in let (val′,uj) = ch[ {uid_P(p),uj} ] ? in
        ...
        p_behaviour(p) end end
```

15 *Example. Road Transport Behaviour Definitions: We refer to Sect. 7.7.4 on page 33.*

6.5 Domain Initialisation

By *domain initialisation* we mean the "start-up" of a behaviour for all manifest parts.

16 *Example. Road Transport Domain Initialisation: We refer to Sect. 7.8 on page 36.*

6.6 End of Domain Modelling Presentation

This concludes the four sections, Sects. 2, 3, 4 and 6, on domain modelling.

7 A ROAD TRANSPORT DOMAIN EXAMPLE

7.1 Naming and Sketch of Domain

We refer to Sect. 2 on page 7.
Narration:

1 The domain is referred to as RTD, the road transport domain.
2 The road transport domain comprises a set of automobiles and a road net of street intersections, called hubs, and [uninterrupted] street segments, called links. Automobiles drive in and out of hubs and links.

Formalisation:

```
type
1. RTD
```

7.2 Endurants: External Qualities

7.2.1 Cartesian Examples

We refer to Sect. 3.3.1 on page 12.

3 There is a road transport domain.

From road transport domains we can observe

4 a road net aggregate and
5 an automobile aggregate.

From the road net aggregate we can observe

6 an aggregate of hubs,
i.e., street intersections, and
7 an aggregate of links,
i.e., street segments (with no hubs).

```
type
3. RTD
4. RNA
5. AA
6. HA
7. LA
```

```
value
4. obs_RNA: RTD → RNA
5. obs_AA: RTD → AA
6. obs_HA: RNA → HA
7. obs_LA: RNA → LA
```

7.2.2 Part Sets

We refer to Sect. 3.3.2 on page 13.

8 There are hubs; from aggregate of hubs one can observe sets of hubs.
9 There are links; from aggregate of links one can observe sets of links.
10 There are automobiles; from aggregate of automobiles one can observe sets of automobiles.

type	value
8. H, Hs = H-set	8. obs_Hs: HA → Hs
9. L, Ls = L-set	9. obs_Ls: LA → Ls
10. A, As = A-set	10. obs_As: AA → As

7.2.3 Endurant States

We refer to Sect. 3.4 on page 13.

11 The singleton value *rtd* represents a road transport [domain] state.
12 The set value *hs* represents a state of all hubs of that road transport domain.
13 The set value *ls* represents a state of all links of that road transport domain.
14 The set value *as* represents a state of all automobiles of that road transport domain.

value
11. *rtd*:RTD,
12. *hs*:H-set = obs_Hs(obs_HA(obs_RNA(rtd))),
13. *ls*:L-set = obs_Ls(obs_LA(obs_RNA(rtd))),
14. *as*:A-set = obs_As(obs_AA(rtd))

7.3 Unique Identifiers

We refer to Sect. 5.1 on page 16.

7.3.1 Unique Identifiation

We shall only consider hubs, links and automobiles.

15 Hubs have unique identifiers.
16 Links have unique identifiers.
17 We define also a unique identifier observer for hubs and links.
18 Automobiles have unique identifiers.

type
15. HI
16. LI
18. AI
value
15. uid_H: H → HI
16. uid_L: L → LI
17. uid_HL: (H|L) → (HI|LI), uid_HL(hl) ≡ is_H(hl)→uid_H(hl),_→uid_L(hl)
18. uid_A: A → AI

7.3.2 Unique Identifier State

19 The variable *his* contains all unique hub identifiers of the road transport domain 3 on page 25.
20 The variable *lis* contains all unique link identifiers of the road transport domain 3 on page 25.
21 The variable *ais* contains all unique automobile identifiers of the road transport domain 3 on page 25.

variable
14. $his = \{ \text{uid\_H(h)} \mid \text{h:H} \bullet \text{h} \in hs \}$.
19. $lis = \{ \text{uid\_L(l)} \mid \text{l:L} \bullet \text{l} \in ls \}$.
20. $ais = \{ \text{uid\_A(a)} \mid \text{a:A} \bullet \text{a} \in as \}$.

7.3.3 Unique Identifier Axiom

22 No two hubs, links and automobiles have the same unique identifier.
23 *ps* is the set of all hubs, links and automobiles.
24 *uis* is the set of all unique hub, link and automobile identifiers.

axiom
22. card hs = card his,
22. card ls = card lis,
22. card as = card ais,
22. card hs + card ls + card as = card his + card lis + card ais
value
23. $ps = hs \cup ls \cup as$
24. $uis = his \cup lis \cup ais$
axiom
22. card ps = card uis

7.4 Mereology

We refer to Sect. 5.2 on page 17.

25 The mereology of any hub is a pair: the possibly empty set of the unique identifiers of links leading into and/or out from the hub, and the set of the unique identifiers of automobiles that are allowed to drive in the hub.
26 The mereology of any link is a pair: the two element set of the unique identifiers of the two hubs that are connected by the link, and the set of the unique identifiers of automobiles that are allowed to drive on the link.
27 The mereology of any automobile is the set of the unique identifiers of hubs in and links on which the automobile may be driving.

type	value
25. $H\_Mer = LI\text{-set} \times AI\text{-set}$	25. $mereo\_H: H \to H\_Mer$
26. $L\_Mer = HI\text{-set} \times AI\text{-set}$	26. $mereo\_L: L \to L\_Mer$
27. $A\_Mer = (HI\|LI)\text{-set}$	27. $mereo\_A: A \to A\_Mer$

28 Link and automobile identifiers of hub mereologies must be of the road transport domain.
29 Hub and automobile identifiers of hub mereologies must be of the road transport domain and there must be exactly two hub identifiers of those mereologies.
30 Hub and links identifiers of automobile mereologies must be of the road transport domain.

```
axiom
28.  $\forall$ (lis,ais):H_Mer•lis$\subseteq lis \wedge$ais$\subseteq ais$
29.  $\forall$ (his,ais):L_Mer•his$\subseteq his \wedge$ais$\subseteq ais \wedge$card his=2
30.  $\forall$ ris:A_Mer•ris$\subseteq his \cup lis$
```

7.4.1 Routes

31 By a *route* (of a road net) we shall understand
  a an alternating sequence of one or more hub and link identifiers
32 such that
  a basis clause 0: the empty list is a route;
  b basis clause 1: a singleton list of a hub or a link identifier of the road net is a route;
  c inductive clause: the concatenation of a route, r, and the tail of a route r′ where the last element of r is identical to the first element of r′ is a route; and
  d extremal clause: and only such routes that can be formed using the above clauses are routes.

```
type
31.   R′ = (HI|LI)$^*$
31a.  R = {| r:R′ | wf_R(r)($rtd$) |}
value
31a.  wf_R: R′ → RTD → Bool
31a.  wf_R(r)($rtd$) ≡
31a.      ∀ i,i+1:Nat • {i,i+1}⊆index(r) ⇒
31a.          let (ri,ri′) = (r[ i ],r[ i+1 ]) in
31a.          is_LI(ri)∧is_HI(ri′)∧ ...
31a.          is_HI(ri)∧is_LI(ri′)∧ ...
31a.          end

32.  routes: RTD×HI-set×LI-set → R-infset
```

```
32.  routes(rtd,his,lis) ≡
32.     let rs = { ⟨⟩ }
32.          ∪ { ⟨hi⟩ | hi:HI • hi ∈ his }
32.          ∪ { ⟨li⟩ | li:LI • li ∈ lis }
32.          ∪ { r⌢tl r′ | {r,r′}⊆rs ∧ r[ len r ]=hd r′ } in
32c.      rs  end
32.     pre: his={uid_H(h)|h:H•h ∈ obs_Hs(obs_AH(obs_RN(rtd)))} ∧
32.          lis={uid_L(l)|l:L•l ∈ obs_Ls(obs_AL(obs_RN(rtd)))}
```

7.5 Attributes

We refer to Sect. 5.3 on page 17.

7.5.1 Hubs, Links and Automobiles

Hub Attributes

33 Hubs have [traffic signal] states which are set of pairs, *li,lj*, of identifiers of the mereology links "signaling" that automobiles can connect from link *li* to link *lj*.
34 Hubs have [traffic signal] state spaces – designating the set of all possible hub states.
35 Hubs have a history; see Item 46 on page 31.

Link Attributes

36 Links have lengths.
37 Links have a history; see Item 47 on page 31.

Automobile Attributes

38 Automobiles have positions on the road net:
  a either *at a hub*,
  b or *on a link*, some fraction
  c down from an entry hub towards the exit hub.
39 Automobiles have a history; see Item 48 on page 31.

We postpone treatment of hub, link and automobile histories till Sect. 7.6.1.

```
type
33.   HΣ = (LI×LI)-set
34.   HΩ = HΣ-set
35.   H_Hist = ...
36.   LEN
37.   L_Hist = ...
38.   A_Pos = At_Hub | On_Link
38a.  At_Hub :: HI
38b.  On_Link :: LI × HI × F × HI
38c.  F = Real axiom ∀ f:F • 0<f<1
39.   A_Hist = ...
```

```
value
33. attr_HΣ: H → HΣ
34. attr_HΩ: H → HΩ
35. attr_H_Hist: A → H_Hist
36. attr_LEN: L → LEN
37. attr_L_Hist: A → L_Hist
38. attr_APos: A → A_Pos
39. attr_A_Hist: A → A_Hist
```

We omit treatment of such automobile attributes as speed, acceleration, engine temperature, energy (gas, oil, electricity) level, mileage and trip counters, GPS (map) position, road surface temperature, gear position (reverse, neutral, forward (1, 2, 3, 4, 5), hand brake position, clutch position, accelerator pressure, brake pedal position, etc.

40 The link identifiers of a hub state must be of the mereology of that hub.
41 A hub state must be in the hub state space.
42 The automobile position must be on the road net.

```
axiom
40. ∀ h:H • h ∈ hs • let hσ = attr_HΣ(h), (lis,_) = mereo_H(h) in
40.     ∀ (li,lj):(LI×LI) • (li,lj) ∈ hσ ⇒ {li,lj}⊆lis

41. ∀ h:H • h ∈ hs • attr_HΣ(h) ∈ attr_HΩ(h)

42. ∀ a:A • a ∈ as • let apos = attr_A_Pos(a) in
42.     cases apos of
42.         At_Hub(hi) → hi ∈ his,
42.         On_Link(li,fhi,_,thi) →
42.             let (his,ais) = mereo_L(retr_L(li,ls)) in
42.             {fhi,thi}⊆his ∧ uid_A(a) ∈ ais end
42.     end end
```

These were some well-formedness axioms. In Sect. 7.6.1 we shall treat well-formedness of hub, link and automobile histories.

7.5.2 Attribute Category Examples

Attribute categories are: HΣ (Item 33 on the preceding page) is a programmable attribute; HΩ (Item 34 on the previous page) is a static attribute; LEN (Item 36 on the preceding page) is a static attribute; A_Pos (Item 38 on the previous page) is a programmable attribute; GPS_Map is an inert attribute; Speed is a biddable attribute; Road_Surface_Temperature is an autonomous attribute; etcetera.

7.6 Intentional Pull

We refer to Sect. 5.4 on page 20.

7.6.1 Further Attributes We start by formulating the hub, link and automobile history attribute definitions.

43 Hubs and links are entered and left by automobiles, i.e., marked by corresponding events.
44 Automobile enters and leaves hubs, i.e., marked by corresponding events.
45 Automobile enters and leaves links, i.e., marked by corresponding events.
46 Hub histories are time-stamped sequences of automobile enter/leave events – in decreasing order (most recent events are listed first),
47 Link histories are time-stamped sequences of automobile enter/leave events – in decreasing order (most recent events are listed first),
48 Automobile histories are time-stamped sequences of hub and link enter/leave events – in decreasing order (most recent events are listed first),
49 For convenience we "lump" hub and link histories into hub-link histories.

```
type
43.  HL_OnOff = mkEnter(ai:AI) | mkLeave(ai:AI)
44.  A_OnOff_H = mkEnterHub(s:HI) | mkLeaveHub(s:HI)
45.  A_OnOff_L = mkEnterLink(s:LI) | mkLeaveLink(s:LI)
46.  H_Hist = (s_t:𝕋𝕀𝕄𝔼×s_oo:HL_OnOff)*
47.  L_Hist = (s_t:𝕋𝕀𝕄𝔼×s_oo:HL_OnOff)*
48.  A_Hist = (s_t:𝕋𝕀𝕄𝔼×s_oo:(OnOff_H|OnOff_L))*
49.  HL_Hist = H_Hist | L_Hist
value
49.  attr_HL_Hist: (H→H_Hist) | (L→L_Hist)
```

50 Automobile histories
  a alternate between being on hubs and being on links.
  b such that the *enter hub event time* is identical to the immediately "prior" *leave link event time*,
  c and such that these events are otherwise ordered in decreasing order of time.

```
axiom
50.  ∀ a_hist:A_Hist •
50.      ∀ i:Nat • {i,i+1}⊆inds a_hist ⇒
50.          let (e1,e2)=(s_oo(a_hist[ i ]),s_oo(a_hist[ i+1 ])),
50.              (t1,t2)=(s_t(a_hist[ i ]),s_t(a_hist[ i+1 ])) in
50.          case (e1,e2)
50b.             (mkLeaveHub(hi),mkEnterLink(li)) → t1=t2,
50c.             (mkLeaveLink(li),mkEnterHub(hi)) → t1=t2,
50c.             (mkLeaveLink(li),mkEnterLink(li′)) → t1>t2,
50c.             (mkLeaveHub(hi),mkEnterHub(hi′)) → t1>t2,
50a.             _ → false
50.          end end
```

We leave the (narrative and formal) expression of the well-formedness of hub and link histories to the reader!

The above indicates that one has to be very careful concerning well-formedness.

But we have not captured all of the constraints, i.e., well-formedness of the history attributes. Next we secure full care!

7.6.2 An Intentional Pull

51 For all automobiles,
- a if their traffic history records that the automobile was entering [leaving] a hub (link) at a certain time,
- b then that hub's (link's) traffic history shall record that that automobile left [entered] that hub (link) at exactly that time;

52 and vice versa, for all hubs an links:
- a if the hub or link traffic history records that an automobile was leaving [entering] that hub, respectively link at a certain time,
- b then that automobile's traffic history shall record that that automobile entered [left] that hub, resp. link, at exactly that time.

```
axiom
51.     ∀ a:A • a ∈ as ⇒
51a.        let a_hist=attr_A_Hist(a), ai=uid_A(a) in
51a.        ∀ (t,on_off_hl) • (t,on_off_hl) ∈ elems a_hist ⇒
51a.            let hli • s(on_off_hl) in
51b.            let hl:(H|L)•hl ∈ hs∪ls • uid_HL(hl) = hli in
51b.            let hl_hist = attr_HL_Hist(hl) in
51b.            ∃ ! i:Nat•i∈ inds hl_hist •
51b.                on_off_hl=mkEnter(hli) → hl_hist[ i ]=(t,mkLeave(ai))
51b.                on_off_hl=mkLeave(hli) → hl_hist[ i ]=(t,mkEnter(ai))
51.         end end end end
52.     ≡
52a.    ∀ hl:(H|L) • hl ∈ hs∪ls ⇒
52a.        let hl_hist=attr_HL_Hist(hl), hli=uid_HL(hl) in
52a.        ∀ (t,on_off_ai) • (t,on_off_ai) ∈ elems hl_hist ⇒
52a.            let ai = ai(on_off_ai) in let a:A•a∈as • uis_A(a)=ai in
52a.            let a_hist = attr_A_Hist(a) in
52a.            ∃ ! i:Nat•i∈ inds a_hist •
52a.                on_off_ai=mkEnter(ai) → on_off_ai[ i ]=(t,mkLeave(hli)),
52a.                on_off_ai=Leave(ai) → on_off_ai[ i ]=(t,Enter(hli))
52a.          end end end end
```

7.7 Perdurants

7.7.1 Channels

We refer to Sect. 6.1 on page 21.

channel { ch[{ui,uj}] | ui,uj:(HI|LI|AI) • {ui,uj}⊆his∪lis∪ais } : M

M will be defined in Sect. 7.7.2.2 on the facing page.

7.7.2 Domain Actions and Events

7.7.2.1 Domain Actions We refer to Sect. 6.2.1 on page 22.
Automobile actions are here simplified to be those of

53 remaining (staying) in a hub (Item 64a on the following page) and
54 remaining (staying) on a link (Item 65a on the next page).

7.7.2.2 Domain Events We refer to Sect. 6.2.2 on page 22.
Automobile events are here simplified to be those of

55 leaving a hub [in order to enter a link] (Item 66d on page 35 and Item 70 on page 36) and
56 entering a link [after having left a hub] (Item 66d on page 35 and Item 70 on page 36) and
57 leaving a link in order to enter a hub (Item 67c on page 35 and Item 75 on page 36).
58 entering a hub [after having left a link] (Item 67c on page 35 and Item 75 on page 36).

Thus contributions to M of Sect. 7.7.1 on the preceding page are:

```
type
55. mkLeaveH(hi:HI,li:LI,ai:AI)
56. mkEnterL(hi:HI,li:LI,ai:AI)
57. mkLeaveL(li:LI,hi:HI,ai:AI)
58. mkEnterH(li:LI,hi:HI,ai:AI)
```

7.7.3 Behaviour Signatures

We refer to Sect. 6.4.1 on page 23.

```
value
    hub: h:H → in,out { ch[{hi,ui}] | ui:(LI|AI)•ui∈lis∪ais } → Unit,
    link: l:L → in,out { ch[{li,ui}] | ui:(LI|AI)•ui∈lis∪ais } → Unit,
    auto: a:A → in,out { ch[{ai,ui}] | ui:(LI|HI)-set•ui∈lis∪his } → Unit.
```

7.7.4 Behaviour Definitions

We refer to Sect. 6.4.2 on page 24.

Automobile Behaviour

We omit consideration of the monitorable GPS_Map, Speed and Road_Surface_-Temperature attributes.

59 One interpretation of an automobile, auto,
60 focuses on its road position.
61 Either the automobile is at a hub,

62 or it is on a link.
63 There could be other focal points.

```
value
59.  auto(a) ≡
60.     auto_pos(a)(attr_A_Pos(p),attr_A_His(a))
63.    ⌈⌉ ...

61.  auto_pos(a)(At_Hub(hi),a_hist) ≡
61.      traversing_hub(a)(At_Hub(hi),a_hist)
61.      pre: attr_A_Pos(a)=At_Hub(hi) ∧ attr_A_Hist(a)=a_hist

62.  auto_pos(a)(On_Link(li,fhi,f,thi),a_hist) ≡
62.      traversing_link(a)(On_Link(li,fhi,f,thi),a_hist)
62.      pre: attr_A_Pos(a)=On_Link(li,fhi,f,thi) ∧ attr_A_Hist(a)=a_hist
```

64 In traversing a hub an automobile
  a is either, internal non-deterministically, ⌈⌉, moving on inside the hub
  b or, internal non-deterministically, entering a link from the hub.

```
value
64.  traversing_hub(a)(At_Hub(hi),a_hist) ≡
64a.      staying_at_H(a)(At_Hub(hi),a_hist)
64b.   ⌈⌉ entering_L(a)(At_Hub(hi),a_hist)
64.     pre: attr_A_Pos(a)=At_Hub(h) ∧ attr_A_Hist(a)=a_hist

64a.  staying_at_H(a)(At_Hub(hi),a_hist) ≡ auto(a)
```

65 In traversing a link an automobile
  a is either, internal non-deterministically, ⌈⌉, moving on inside the link
  b – possibly advancing a bit, i.e., increasing its fraction position “down” the link,
  c or, internal non-deterministically, entering a hub from the link.

```
value
65.  traversing_link(a)(On_Link(li,fhi,f,thi),a_hist) ≡
65a.     staying_on_L(a)(On_Link(li,fhi,f,thi),a_hist)
65c.  ⌈⌉ entering_H(a)(On_Link(li,fhi,f,thi),a_hist)
65.     pre: attr_A_Pos(a)=On_Link(li,fhi,f,thi) ∧ attr_A_Hist(a)=a_hist

65a.  staying_on_L(a)(On_Link(li,fhi,f,thi),a_hist) ≡
65b.     let f′:F • f≤f′<1 in assert: ∃ f′:F • f≤f′<1
65b.     let a′ = part_update(a)(ηA_Pos,On_Link(li,fhi,f′,thi))
65a.     auto(a′) end end
65a.     pre: attr_A_Pos(a)=On_Link(li,fhi,f,thi) ∧ attr_A_Hist(a)=a_hist
```

66 In entering a link
  a the automobile internal non-deterministically selects the link to be entered, and thus the next hub,
  b records the time,
  c updates its history and automobile position accordingly,
  d so informs the behaviour of the hub being left and the link being entered, while resuming being an automobile – with the updated history.

```
value
66.  entering_L(a)(At_Hub(fhi),a_hist) ≡
66a.     let li:LI•li∈lis∧li∈mereo_H(retr_H(fhi)(σ)),
66a.         thi:HI•thi∈his∧thi∈mereo_L(retr_L(li)(σ))\{fhi},[22]
66b.          τ = record_𝕋𝕀𝕄𝔼[23],
66b.          ai=uid_A(a) in
66a.     let a_pos = On_Link(fhi,li,0,thi) in
66c.     let a_hist′ = ⟨(a_pos,τ)⟩⌢a_hist in
66c.     let a′ = part_update(a)(ηA_Hist,a_hist′) in
66c.     let a″ = part_update(a′)(ηA_Pos,a_pos) in
66d.     (ch[{ai,fhi}] ! (mkLeaveH(fhi,li,ai),τ)
66d.      ‖ ch[{ai,li}] !( mkEnterL(li,fhi,ai),τ) ‖ auto(a″))
66.      end end end end end
66.      pre: attr_A_Pos(a)=At_Hub(fhi) ∧ attr_A_Hist(a)=a_hist
```

67 In entering a hub
  a the time is recorded,
  b the automobile history and position is updated,
  c and the behaviours of the link left link and hub entered are being so informed
    while the automobile resumes being an automobile – in the updated state.

```
value
67.  entering_H(a)(On_Link(li,fhi,f,thi),a_hist) ≡
67a.     let τ = record_𝕋𝕀𝕄𝔼,
67a.          ai = uid_A(a),
67a.          a_pos = at_Hub(thi) in
67a.     let a_hist′ = ⟨(a_pos,τ)⟩⌢a_hist in
67b.     let a′ = part_update(a)(ηA_Hist,(τ,a_hist′)) in
67b.     let a″ = part_update(a′)(ηA_Pos,a_pos) in
67c.     (ch[ai,li] ! (mkLeaveL(li,thi,ai),τ) ‖
67c.      ch[ai,thi] ! (mkEnterH(thi,li,ai),τ) ‖ auto(a″))
67.      end end end end
67.      pre: attr_A_Pos(a)=On_Link(li,fhi,f,thi) ∧ attr_A_Hist(a)=a_hist
```

[22] For retr_··· see Sect. 5.1.4 on page 16.
[23] For record_𝕋𝕀𝕄𝔼 see Sect. 4.2 on page 15.

68 The hub behaviour
69 externally non-deterministically ($\lbrack\rbrack$) offers
70 to accept, non-deterministically, a *leave* message,
71 from any automobile in its mereology;
72 it prepares for proper insertion of this event into its traffic history
73 updating to an augmented traffic history, and, hence, hub state;
74 resuming to be the hub behaviour in the updated state;
75 or to accept, non-deterministically, an *enter* message,
76 again from any automobile in its mereology;
77 updating to an augmented traffic history, and, hence, hub state;
78 resuming to be the hub behaviour in the updated state.

```
value
68.  hub(h) ≡ let hi = uid_HI(h) in
70.     ⌷ { let (mkLeaveH(hi′,li,ai),τ) = ch[{hi,ai}] ? in assert: hi′=hi
73.           let h_hist′ = ⟨(τ,mkEnter(ai))⟩^attr_H_Hist(h) in
73.           let h′ = part_update(ηH_Hist,h_hist′) in
74.           hub(h′)
71.        | ai:AI • ai ∈ ais end end end}
69.     ⌷
75.     ⌷ { let (mkEnterH(hi′,li,ai),τ) = ch[{hi,ai}] ? in assert: hi′=hi
77.           let h_hist′ = ⟨(τ,mkLeave(ai))⟩^attr_H_Hist(h) in
77.           let h′ = part_update(ηH_Hist,h_hist′) in
78.           hub(h′)
76.        | ai:AI • ai ∈ ais end end end } end
```

We leave the definition of link behaviours as an exercise!

7.8 Domain Initialisation

We refer to Sect. 6.5 on page 24.

We initialise a domain behaviour for all atomic endurants: hubs, links and automobiles.

79 The domain behaviour is the parallel composition of
80 the distributed parallel composition of all hub behaviours, with
81 the distributed parallel composition of all link behaviours, with
82 the distributed parallel composition of all automobile behaviours.

```
80.  ∥ { hub(b) | h:H • h ∈ hs }
79.  ∥
81.  ∥ { link(l) | l:L • l ∈ ls }
79.  ∥
82.  ∥ { auto(a) | a:A • a ∈ as }
```

7.9 Verification

It remains to verify that the automobile, hub and link behaviours and the road transport domain initialisation satisfy the appropriate axioms and the intentional pull.

End of Example

8 Closing

8.1 The Current Calculi

The treatment of behaviours of Sect. 6.4.2 differs very much from that of Sects. 7.6–7.7 of [17]. The present one is very short, but results in a repeated use of the part_update functional. Our domain modelling approach allows a wide spectrum, in-between these behaviour signature and definition styles, for expressing behaviours. What remains fixed in the treatment of endurants: both of their external qualities, and of their internal qualities.

8.2 Some Issues

A number of issues need be addressed.

8.2.1 A New View of Software Development ? Yes, we claim that this paper presents an additional view of software development! Aircraft designers and manufacturers employ professionally educated aeronautics engineers having state-of-the-art insight into aerodynamics. But, we claim, software companies do not, today, April 19, 2023, exhibit the same professionalism in their staffing. Software for health care (hospitals, etc.) are often developed by programmers with no previous professional insight into that area. Likewise for domains such as law, public administration, health care and tax administration. With sound methods for "deriving" requirements from domain models, cf. Sect. 8.2.7 on page 39, these software houses now have a possibility of becoming professional.

8.2.2 From Programming Language Semantics to Domain Models Domain models give semantics to the nouns (endurants) and verbs (perdurants) spoken by domain workers. Just like the development of compilers for programming languages were based on formal models of their semantics, so we can now give semantics to the nouns and verbs spoken by domain workers, and, from these, using rigorous development methods, similar to those used for compiler development [25, 28], develop trustworthy domain software.

8.2.3 Correctness: Verification, Checking, Testing This paper has not dealt with the issue of correctness of domain models. A number of endurant and perdurant $\mathcal{D}$escription prompts have indicated that axioms and assertions[24] need be expressed. For domain assertions their correctness must, of course, be shown – using whichever (testing, model checking and proof) techniques are adequate. The axioms and assertions carry over into $\mathcal{R}$equirements prescriptions and, from there, into software $\mathcal{S}$pecifications. Now the full-blown force of testing, model checking and proofs must be applied. As indicated in formula $\mathcal{D}, \mathcal{S} \models \mathcal{R}$, Sect. 1.2 on page 3, domain models now make proof obligations more clear.

8.2.4 No Recursive Domains ! Surprise, surprise ! Yes, there are no recursively defined endurant sorts. Domains do not contain "recursive endurants".[25]

8.2.5 Domain Facets There is more to domain engineering than this paper can cover. A main element of domain modelling is that of modelling also other than the *intrinsics* of domains – as so far covered. By a *domain facet* we shall understand *one amongst a finite set of generic ways of analysing a domain: a view of the domain, such that the different facets cover conceptually different views – and these views together cover the domain.*[26] [17, *Chapter 8*] covers methods for modelling additional facets – such as *support technology*, *rules & regulations*, *scripts* (or *contracts*), *license languages*, *management & organisation*, and *human behaviour*.

8.2.6 Algorithmics Algorithms are the hall-mark and corner-stone of computing. So where is *"algorithmics"* [34, 36, *Harel*] in all this ? ! The straight answer is: algorithm concerns are not concerns of domain modelling !

Domain models focus on expressing properties. They do so using abstraction in general, and simple combinations of *proof theoretic* and *model theoretic* means such as defining abstract types, here called sorts, comprehension over sets, sequences and maps $\{\mathsf{f(i)|i{:}D\bullet}\mathcal{P}\mathsf{(f,i)}\}$, $\langle\mathsf{f(i)|i{:}D\bullet}\mathcal{Q}\mathsf{(f,i)}\rangle$, and $[\,\mathsf{f(i)}\mapsto\mathsf{g(i)|i{:}D\bullet}\mathcal{R}\mathsf{(f,g,i)}\,]$. The predicates, $\mathcal{P}$, $\mathcal{Q}$ and $\mathcal{R}$ further raise the level of abstraction. It is in the efficient realization of these abstractions that algorithms play their part.

[24] i.e., proof obligations

[25] Some readers may object, but we insist ! If *trees* are brought forward, as an example of a recursively definable domain, then we argue: Yes, trees can be recursively defined. Trees can, as well, be defined as a variant of graphs, and you wouldn't claim, would you, that graphs are recursive ? We shall consider the living species of trees (that is, plants), as atomic. In defining attribute types You may wish to model certain attributes as 'trees'. Then, by all means, You may do so recursively. But natural trees, having roots and branches cannot be recursively defined, since proper "subtrees" of trees would then have roots !

[26] This characterisation clearly lacks sufficient formality. We refer to Sect. 8.2.16 on page 41 below.

8.2.7 Requirements In [17, *Chapter 9, 2021*] we show how to "derive", in a systematic manner, *requirements prescriptions* from *domain descriptions*. Requirements are for a *machine*[27] The *machine* is the *hardware* upon which the software to be developed is to be executed – as well as the [*auxiliary*] *software* "under which" that new software is performing (*operating system*, *database system*, *data communications software*, etc.). First requirements development proceeds in three stages: (i) a *domain requirements* stage in which requirements that can be expressed sôlely using terms from the domain are developed; (ii) an *interface requirements* stage in which requirements that can be express using terms from both the domain and the machine are developed; and (iii) a *domain requirements* stage in which requirements that can be expressed solely using terms from the machine are developed. [17] shows how *domain requirements* stage can be decomposed, sequentially, into *projection*, *initialisation*, *determination*, *extension* and *fitting* steps. For details on this and more we refer to [17].

8.2.8 Software Design [4, *2005-2006*] shows how to further develop software from their requirements prescriptions.

8.2.9 Continuity As remarked in Sect. 3.1 on page 11 the calculi of this paper do not address the issue of modelling continuous dynamic phenomena. This is clearly a weakness. The Integrated Formal Methods conferences [46] initially set out to spur research aimed at amalgamating continuous and discrete specifications. Not much progress has been made, except: `TLA`$^+$ offers some form of hybrid systems [47]; `Hybrid Event-B` [2] likewise; and for `Back's Action Systems` there is a hybrid version [1].[28] We also refer to [60, 61].

8.2.10 Modelling Concurrency We have used Hoare's `CSP` [40] to model concurrency. There are other, in this case, graphical languages for modelling concurrency. We refer to Chapters 12–15 of [5]. In these chapters I treat the modelling of four graphical specification languages: `Petri Nets` [53], `Message Sequence Charts` [41, 42], `State Charts` [35] and `Live Sequence Charts` [29, 37]. All of them are fascinating. Their graphics appeal to many of us – so I recommend to use them informally, aside, for the textual modelling shown in this paper. But they do not "merge" into formal, textual specification languages, like `VDM-SL, RSL, Z, Alloy`.

8.2.11 Modelling Temporality Although time is modelled, as part of internal attribute properties, we have not shown the modelling of temporality of behaviours. In Chapter 15 of [5] I show how to merge `Duration Calculus, DC` [62] with RSL-Text. Another fascinating such formal specification language is *Leslie Lamport*'s `TLA+: Temporal Logic of Actions` [48].

[27] – as suggested by Michael A. Jackson [44]

[28] I acknowledge the mentioning of these three references to one of the reviewers of the resent paper.

8.2.12 Domain Specific Languages A *domain specific language*, `DSL`, is a computer programming language specialised to a particular application domain. What we have shown here is not a `DSL`. Examples of `DSL`s could be programming languages for expressing calculations for railways or financial services or hospitals or other. [27, *Actulus*] reports on an actuarial programming language for life insurance and pensions. To give semantics for a specific `DSL` one invariably specifies a domain model. So that, then, is a rôle for domain modelling.

8.2.13 Three Rôles for Domain Models There are three rôles for domain models: (i) to just simply study and understand a domain – irrespective of any ensuing software for that domain; (ii) to serve as a basis for the development of a `DSL`; and (iii) to serve as a basis for the development of [other] software for the domain.

8.2.14 How Comprehensive should a Domain Model be ? Clearly domain models for any reasonable domain can potentially be very large in terms of pages of description. So the question is: *how much of the "domain at large" should be included in a domain description ?*. We cannot, of course, give a general answer to that question. But we can say that the domain model must at least encompass those domain entities that will, or might, be referred to in a requirements prescription. That is, if it is found when developing a *domain requirements*[29] of a *requirements prescription*, that terms thought to be of the domain was not covered by the *domain description*, then, obviously, that description must be augmented.

We do expect there to be, eventually, available for general use, a few, domain models for selected domains.

For physics Newton and Leibniz[30] has given us a calculus with which to – more or less quickly – establish a model for some physical phenomenon. When control engineers then wish to set up some automatic control system for a phenomenon they first apply the Newton/Leibniz calculi to model the phenomenon, then, from that, somehow derive a *control model*. We advocate a similar approach, as already hinted at in our expressing the *Triptych Dogma* (Page 1).

The road transport domain modelled in Appendix 7 is one such domain. It has here been expressed in a way, devoid of any specific orientation. Based on the model of Appendix 7 we can envisage some such orientations as a *road pricing domain*, a *cadastral*[31] *map domain*, a *road development domain*, a *road maintenance domain*, et cetera.

8.2.15 Domain Laws Physics has excelled in our understanding the world we live in by its *laws* and by the *calculi* it has spawned – calculi that enables us to *explain* what has happened and to *predict* what will or might happen. Domain modelling has already lead to some *domain laws* – such as illustrated by

[29] Cf. Sect. 8.2.7 on the previous page
[30] https://en.wikipedia.org/wiki/Leibniz%E2%80%93Newton_calculus_controversy
[31] https://eng.gst.dk/danish-cadastre-office/cadastral-map

for example *intentional pulls*, cf. Sect. 5.4 on page 20 (approx. half a page) and Appendix 7.6 on page 30 (two pages). The study of *intentional pull* in domains has just started! Its counterpart in physics, *gravitational pull*, is "behind" many laws of physics.

8.2.16 A Domain Modelling Science?

A science of domain modelling systematically builds and organizes knowledge about the ways and means of modelling domains such that that knowledge can explain what these models express. As an example of there not yet being a sufficient scientific knowledge of domains we refer to our informal coverage of the concept of *domain facets*, cf. footnote 26 on page 38. A formal understanding of domains and what "facet"–distinguishes them, could help sharpen the characterisation of Sect. 8.2.5 on page 38. Such a formal understanding was first reported in [12, *2014*]. Of more specific nature we suggest, next, studies of some specific issues.

(i) An *"integrated"* form of use of *differential equations* with the present RSL$^+$, i.e., the extension of our approach to domain modelling to cover more specifically issues of continuity.

(ii) A *"further detailed"* understanding of the concept of *intentional pull*.

(iii) A study of a possible *Calculus of Perdurants*.

(iv) A study of examples of domain models with an emphasis on *human interaction*.

(v) Formal models of the analysis predicates and functions and the description functions, cf. [12].

8.3 Acknowledgments

A referee of this paper, many thanks to all five (!), suggested the following, slightly edited acknowledgment:

Laudatio: Prof. He JiFeng

– He Jifeng's work on a *Unifying Theory of Programming*, `UTP` – a monumental contribution – is seen as a domain model for programming languages covering a wide range of programming language paradigms.
– `UTP` is about unifying axiomatic, denotational and operational semantics all of which can be expressed in `RSL`. Hence, `RSL` could be used as a concrete language to define a *unifying theory of programming*.
– One could combine domain modelling and `UTP` in order to systematically develop and define formal *domain specific languages*, `DSL`s. It might result in a new unifying theory of `DSL`s.

I fully concur.

I gratefully acknowledge the opportunity given to me, to write this paper, during my PhD lectures, October–November 2022, at the TU Wien Informatics[32],

[32] https://informatics.tuwien.ac.at/

Vienna, Austria, by Prof. Laura Kovacs. I also gratefully acknowledge comments by Klaus Havelund, Kazuhiro Ogata and Wolfgang Reisig.

9 Bibliography

References

1. Ralph-Johan Back, Luigia Petre, and Ivan Porres. Generalizing action systems to hybrid systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 202–213, September 2000. www.researchgate.net/publication/221654900_Generalizing_Action_Systems_to_Hybrid_Systems.
2. Richard Banach and Michael Butler. Modelling Hybrid Systems in Event-B and Hybrid Event-B: A Comparison of Water Tanks. In K. Ogata, M. Lawford, and S. Liu, editors, *Formal methods and software engineering*, volume 10009 of *Lecture notes in computer science*. Springer, 2016. 18th ICFEM 2016, https://doi.org/10.1007/978-3-319-47846-3_7.
3. Dines Bjørner. UNI/IIST Reports on Domain Modelling. Research Report, UNU/IIST, 1995–1997. UNUIIST:46: New Software Technology Development, UNUIIST:47: Software Support for Infrastructure Systems, UNUIIST:48: Software Systems Engineering — From Domain Analysis to Requirements Capture [— an Air Traffic Control Example], UNUIIST:58: Infrastructure Software Systems, UNUIIST:59: New Software Development, UNUIIST:60: Models of Enterprise Management: Strategy, Tactics & Operations — Case Study Applied to Airlines and Manufacturing, UNUIIST:61: Federated GIS+DIS–based Decision Support Systems for Sustainable Development — a Conceptual Architecture, UNUIIST:96: odels of Financial Services & Industries.
4. Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling; Vol. 2: Specification of Systems and Languages; Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
5. 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. See [6, 8].
6. Dines Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Qinghua University Press, 2008.
7. Dines Bjørner. On Mereologies in Computing Science. In *Festschrift: Reflections on the Work of C.A.R. Hoare*, History of Computing (eds. Cliff B. Jones, A.W. Roscoe and Kenneth R. Wood), pages 47–70, London, UK, 2009. Springer. www.imm.dtu.-dk/~dibj/bjorner-hoare75-p.pdf.
8. Dines Bjørner. **Chinese:** *Software Engineering, Vol. 2: Specification of Systems and Languages*. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.
9. Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics, Part I of II: The Engineering Part*. *Kibernetika i sistemny analiz*, 2(4):100–116, May 2010.
10. Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics Part II of II: The Science Part*. *Kibernetika i sistemny analiz*, 2(3):100–120, June 2011.
11. Dines Bjørner. A Rôle for Mereology in Domain Science and Engineering. In *Mereology and the Sciences*, Synthese Library (eds. Claudio Calosi and Pierluigi Graziani), Amsterdam, The Netherlands, October 2014. Springer.

12. Dines Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model www.imm.dtu.dk/ dibj/2014/kanazawa/kanazawa-p.pdf. In Shusaku Iida and José Meseguer and Kazuhiro Ogata, editor, *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi*. Springer, May 2014.
13. Dines Bjørner. Manifest Domains: Analysis & Description www.imm.dtu.dk/ dibj/-2015/faoc/faoc-bjorner.pdf. *Formal Aspects of Computing*, 29(2):175–225, March 2017. Online: 26 July 2016.
14. Dines Bjørner. To Every Manifest Domain a `CSP` Expression www.imm.dtu.dk/˜dibj/-2016/mereo/mereo.pdf. *Journal of Logical and Algebraic Methods in Programming*, 1(94):91–108, January 2018.
15. Dines Bjørner. *An Assembly Plant Domain – Analysis & Description*, `www.imm.dtu.-dk/˜dibj/2021/assembly/assembly-line.pdf`. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, September 2019.
16. Dines Bjørner. Domain Analysis & Description – Principles, Techniques and Modelling Languages. www.imm.dtu.dk/˜dibj/2018/tosem/Bjorner-TOSEM.pdf. *ACM Trans. on Software Engineering and Methodology*, 28(2), April 2019. 68 pages.
17. Dines Bjørner. *Domain Science & Engineering – A Foundation for Software Development*. EATCS Monographs in Theoretical Computer Science. Springer, 2021. A revised version of this book is [21].
18. Dines Bjørner. *Rigorous Domain Descriptions. A compendium of draft domain description sketches carried out over the years 1995–2021. Chapters cover:*

    – *Graphs,*
    – *Railways,*
    – *Road Transport,*
    – *The "7 Seas",*
    – *The "Blue Skies",*
    – *Credit Cards,*
    – *Weather Information,*
    – *Documents,*
    – *Urban Planning,*
    – *Swarms of Drones,*
    – *Container Terminals,*
    – *A Retailer Market,*
    – *Shipping,*
    – *Rivers,*
    – *Canals,*
    – *Stock Exchanges,*
    – *Web Transactions, etc.*

    This document is currently being edited. Own: www.imm.dtu.dk/ dibj/2021/dd/dd.pdf, Fredsvej 11, DK-2840 Holte, Denmark, November 15, 2021.
19. Dines Bjørner. Documents: A Basis for Government. In *United Natonans Inst., Festschrift for Tomas Janowski and Elsa Estevez*. Guimaraes, Portgal, October, `www.-imm.dtu.dk/˜dibj/2022/janowski/docs.pdf`, 2022.
20. Dines Bjørner. Domain Modelling – A Primer. A short version of [21]. xii+227 pages[33], January 2023.
21. Dines Bjørner. Domain Science & Engineering – A Foundation for Software Development. Revised edition of [17]. xii+346 pages[34], January 2023.
22. Dines Bjørner. Pipelines: A Domain Science & Engineering Description. In *FSEN 2023: Fundamentals of Software Engineering, Teheran, Iran, May 35*. `www.imm.dtu.-dk/˜dibj/2023/tehran/tehran.pdf`, 2023.
23. Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.
24. Dines Bjørner and Cliff B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.

[33] Due to copyright reasons no URL is given to this document's possible Internet location.

[34] Due to copyright reasons no URL is given to this document's possible Internet location. A primer version, omitting certain chapters, is [20]

25. Dines Bjørner and Ole N. Oest, editors. *Towards a Formal Description of Ada*, volume 98 of *LNCS*. Springer, 1980.
26. Roberto Casati and Achille C. Varzi. *Parts and Places: the structures of spatial representation*. MIT Press, 1999.
27. David R. Christiansen, Klaus Grue, Henning Niss, Peter Sestoft, and Kristján S. Sigtryggsson. Actulus Modeling Language - An actuarial programming language for life insurance and pensions. Technical Report, edlund.dk/sites/default/files/Downloads/paper_actulus-modeling-language.pdf, Edlund A/S, Denmark, Bjerregårds Sidevej 4, DK-2500 Valby. (+45) 36 15 06 30. edlund@edlund.dk, http://www.edlund.dk/en/insights/scientific-papers, 2015. This paper illustrates how the design of pension and life insurance products, and their administration, reserve calculations, and audit, can be based on a common formal notation. The notation is human-readable and machine-processable, and specialised to the actuarial domain, achieving great expressive power combined with ease of use and safety.
28. Geert Bagge Clemmensen and Ole N. 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.
29. Werner Damm and David Harel. LSCs: Breathing life into Message Sequence Charts. *Formal Methods in System Design*, 19:45–80, 2001. Early version appeared as Weizmann Institute Tech. Report CS98-09, April 1998. An abridged version appeared in *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-based Distributed Systems* (FMOODS'99), Kluwer, 1999, pp. 293–312.
30. John Fitzgerald and Peter Gorm 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.
31. K. Futatsugi, A.T. Nakagawa, and T. Tamai, editors. *CAFE: An Industrial–Strength Algebraic Formal Method*, Sara Burgerhartstraat 25, P.O. Box 211, NL–1000 AE Amsterdam, The Netherlands, 2000. Elsevier. Proceedings from an April 1998 Symposium, Numazu, Japan.
32. 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.
33. Chris W. George, Anne Elisabeth Haxthausen, Steven Hughes, Robert Milne, Søren Prehn, and Jan Storbank Pedersen. *The RAISE Development Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.
34. David Harel. *Algorithmics —The Spirit of Computing*. Addison-Wesley, 1987.
35. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
36. David Harel. *The Science of Computing — Exploring the Nature and Power of Algorithms*. Addison-Wesley, April 1989.
37. David Harel and Rami Marelly. *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
38. Charles Anthony Richard Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8), Aug. 1978.
39. Charles Anthony Richard Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985.
40. Charles Anthony Richard Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985. Published electronically: usingcsp.com/cspbook.pdf (2004).

41. ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992.
42. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1999.
43. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.
44. Michael A. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press. Addison-Wesley, Reading, England, 1995.
45. Michael A. Jackson. Program Verification and System Dependability. In Paul Boca and Jonathan Bowen, editors, *Formal Methods: State of the Art and New Directions*, pages 43–78, London, UK, 2010. Springer.
46. K. Araki and others, editor. *IFM 1999–2013: Integrated Formal Methods*, LNCS Vols.: 1945, 2335, 2999, 3771, 4591, 5423, 6496, 7321, 7940, etc. Springer, 1999–2019.
47. Leslie Lamport. Hybrid Systems. In Hans Rischel and Anders P. Ravn, editors, *Workshop on Theory of Hybrid Systems*, Lecture Notes in Computer Science. Springer, October 1992. `https://lamport.azurewebsites.net/pubs/lamport-hybrid.pdf`.
48. Leslie Lamport. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA, 2002.
49. Axel van Lamsweerde. *Requirements Engineering: from system goals to UML models to software specifications*. Wiley, 2009.
50. W. Little, H.W. Fowler, J. Coulson, and C.T. Onions. *The Shorter Oxford English Dictionary on Historical Principles*. Clarendon Press, Oxford, England, 1973, 1987. Two vols.
51. E.C. Luschei. *The Logical Systems of Leśniewksi*. North Holland, Amsterdam, The Netherlands, 1962.
52. John McCarthy. Towards a Mathematical Science of Computation. In C.M. Popplewell, editor, *IFIP World Congress Proceedings*, pages 21–28, 1962.
53. Wolfgang Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Leitfäden der Informatik. Vieweg+Teubner, 1st edition, 15 June 2010. 248 pages; ISBN 978-3-8348-1290-2.
54. Kai Sørlander. *Det Uomgængelige – Filosofiske Deduktioner [The Inevitable – Philosophical Deductions, with a foreword by Georg Henrik von Wright]*. Munksgaard · Rosinante, Copenhagen, Denmark, 1994. 168 pages.
55. Kai Sørlander. *Under Evighedens Synsvinkel [Under the viewpoint of eternity]*. Munksgaard · Rosinante, Copenhagen, Denmark, 1997. 200 pages.
56. Kai Sørlander. *Den Endegyldige Sandhed [The Final Truth]*. Rosinante, Copenhagen, Denmark, 2002. 187 pages.
57. Kai Sørlander. *Indføring i Filosofien [Introduction to The Philosophy]*. Informations Forlag, Copenhagen, Denmark, 2016. 233 pages.
58. Kai Sørlander. *Den rene fornufts struktur [The Structure of Pure Reason]*. Ellekær, Slagelse, Denmark, 2022.
59. James Charles Paul Woodcock and James Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.
60. WanLing Xie, ShuangQing Xiang, and HuiBiao Zhu. A UTP approach for rTiMo. *Formal Aspects of Computing*, 30(6):713–738, 2018.
61. WanLing Xie, HuiBiao Zhu, and Xu QiWen. A process calculus BigrTiMo of mobile systems and its formal semantics. *Formal Aspects of Computing*, 33(2):207–249, March 2021.
62. Chao Chen Zhou and Michael R. Hansen. *Duration Calculus: A Formal Approach to Real–time Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer–Verlag, 2004.