Under consideration for publication in Formal Aspects of Computing

Manifest Domains: Analysis and Description

Dines Bjørner[1]

[1] Fredsvej 11, DK-2840 Holte, Denmark. DTU, DK-2800 Kgs. Lyngby, Denmark. e-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~dibj

To the memory of Peter Lucas: 13 Jan. 1935 – 2 Feb. 2015

Abstract.
We show that manifest domains, an understanding of which are a prerequisite for software requirements prescriptions, can be precisely described: narrated and formalised. We show that such manifest domains can be understood as a collection of endurant, that is, basically spatial entities: parts, components and materials, and perdurant, that is, basically temporal entities: actions, events and behaviours. We show that parts can be modeled in terms of external qualities whether: atomic or composite parts, having internal qualities: unique identifications, mereologies, which model relations between parts, and attributes. We show that the manifest domain analysis endeavour can be supported by a calculus of manifest domain analysis prompts: `is_entity`, `is_endurant`, `is_perdurant`, `is_part`, `is_component`, `is_material`, `is_atomic`, `is_composite`, `has_components`, `has_materials`, `has_concrete_type`, `attribute_names`, `is_stationary`, etcetera.We show how the manifest domain description endeavour can be supported by a calculus of manifest domain description prompts: `observe_part_sorts`, `observe_part_type`, `observe_components`, `observe_materials`, `observe_unique_identifier`, `observe_mereology`, `observe_attributes`, `observe_location` and `observe_position`. We show how to model attributes, essentially following Michael Jackson, [Jac95], but with a twist: The model of attributes introduces the attribute analysis prompts `is_static_attribute`, `is_dynamic_attribute`, `is_inert_attribute`, `is_reactive_attribute`, `is_active_attribute`, `is_autonomous_attribute`, `is_biddable_attribute` and `is_programmable_attribute`. The twist suggests ways of modeling "access" to the values of these kinds of attributes: the static attributes by simply *"copying"* them, once, the reactive and programmable attributes by *"carrying"* them as function parameters whose values are kept always updated, and the remaining, the `external_attributes`, by inquiring, when needed, as to their value, as if they were always offered on `CSP`-like channels [Hoa85]. We show how to model essential aspects of perdurants in terms of their signatures based on the concepts of endurants. And we show how one can "compile" descriptions of endurant parts into descriptions of perdurant behaviours. We do not show prompt calculi for perdurants. The above contributions express a method with principles, techniques and tools for constructing domain descriptions. It is important to realise that we do not wish to nor claim that the method can describe all that it is interesting to know about domains.

Keywords: Domain Engineering, Manifest Domains, Analysis & Description, Prompt Calculi

Correspondence and offprint requests to: Dines Bjørner, Fredsvej 11, DK-2840 Holte, Denmark. e-mail: bjorner@gmail.com

1. Introduction

The broader subject of this paper is that of software development. The narrower subject is that of manifest domain engineering. We shall see software development in the context of the TripTych approach (next section). The contribution of this paper is twofold: the propagation of manifest domain engineering as a first phase of the development of a large class of software — and a set of principles, techniques and tools for the engineering of the analysis & descriptions of manifest domains. These principles, techniques and tools are embodied in a set of analysis and description prompts. We claim that this embodiment — in the form of prompts — is novel.

1.1. The TripTych Approach to Software Engineering

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

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

This paper contributes, we claim, to a methodology for domain analysis &[1] domain description. References [Bjø08, Bjø10b] show how to "refine" domain descriptions into requirements prescriptions, and reference [Bjø11c] indicates more general relations between domain descriptions and domain demos, domain simulators and more general domain specific software.

In branches of engineering based on natural sciences professional engineers are educated in these sciences. Telecommunications engineers know Maxwell's Laws. Maybe they cannot themselves "discover" such laws, but they can "refine" them into designs, for example, for mobile telephony radio transmission towers. Aeronautical engineers know laws of fluid mechanics. Maybe they cannot themselves "discover" such laws, but they can "refine" them into designs, for example, for the design of airplane wings. And so forth for other engineering branches.

Our point is here the following: software engineers must domain specialise. This is already done, to a degree, for designers of compilers, operating systems, database systems, Internet/Web systems, etcetera. But is it done for software engineering banking systems, traffic systems, health care, insurance, etc. ? We do not think so, but we claim it should be done.

The concept of **systems engineering** arises naturally in the TripTych approach. First: *domains can be claimed to be systems.* Secondly: *requirements are usually not restricted to software, but encompasses all the human and technological "assists" that must be considered.* Other than that we do not wish to consider domain analysis & description principles, techniques and tools specific to "systems engineering".

1.2. Method and Methodology

1.2.1. ***Method***

By a **method** we shall understand a "somehow structured" set of principles for selecting and applying a number of techniques and tools for analysing problems and synthesizing solutions for a given domain ⊙[2]

The 'somehow structuring' amounts, in this treatise on domain analysis & description, to the techniques and tools being related to a set of domain analysis & description "prompts", "issued by the method", prompting the domain engineer, hence carried out by the domain analyser & *describer*[3] — conditional upon the result of other prompts.

[1] When, as here, we write *A* & *B* we mean *A* & *B* to be one subject.

[2] Definitions and examples are delimited by ⊙ respectively □ symbols.

[3] We shall thus use the term domain engineer to cover both the analyser & the describer.

1.2.2. ***Discussion***

There may be other 'definitions' of the term 'method'. The above is the one that will be adhered to in this paper. The main idea is that there is a clear understanding of what we mean by, as here, a software development method, in particular a *domain analysis & description method.*

The **main principles** of the TripTych domain analysis and description approach are those of abstraction and both narrative and formal modeling. This means that evolving domain descriptions necessarily limit themselves to a subset of the domain focusing on what is considered relevant, that is, abstract "away" some domain phenomena.

The **main techniques** of the TripTych domain analysis and description approach are besides those techniques which are in general associated with formal descriptions, focus on the techniques that relate to the deployment of of the individual prompts.

And the **main tools** of the TripTych domain analysis and description approach are the analysis and description prompts and the description language, here the Raise Specification Language RSL [GHH+92].

A main contribution of this paper is therefore that of "painstakingly" elucidating the principles, techniques and tools of the domain analysis & description method.

1.2.3. ***Methodology***

By **methodology** we shall understand the study and knowledge about one or more methods[4] ⊙

1.3. Computer and Computing Science

By **computer science** we shall understand the study and knowledge of the conceptual phenomena that "exists" inside computers and, in a wider context than just computers and computing, of the theories "behind" their formal description languages ⊙ Computer science is often also referred to as theoretical computer science.

By **computing science** we shall understand the study and knowledge of how to construct and describe those phenomena ⊙ Another term for computing science is programming methodology.

This paper is about computing science. It is concerned with the construction of domain descriptions. It puts forward a calculus for analysing and describing domains. It does not theorize about this calculus. There are no theorems about this calculus and hence no proofs. We leave that to another study and paper.

1.4. What Is a Manifest Domain ?

By 'domain' we mean the same as 'problem domain' [JHJ07]. We offer a number of complementary delineations of what we mean by a manifest domain. But first some examples, "by name" !

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

A **manifest domain** is a human- and artifact-assisted arrangement of endurant, that is spatially "stable", and perdurant, that is temporally "fleeting" entities. Endurant entities are either parts or components or materials. Perdurant entities are either actions or events or behaviours ⊙

Example 2. Manifest Domain Endurants: Examples of (names of) endurants are ***Air traffic:*** *aircraft, airport, air lane.* ***Banks****: client, passbook.* ***Container lines:*** *container, container vessel, container terminal port.* ***Documents:*** *document, document collection.* ***Hospitals:*** *patient, medical staff, ward, bed, patient medical journal.* ***Pipelines:*** *well, pump, pipe, valve, sink, oil.* ***Railways:*** *simple rail unit, point, crossover, line, track, station.* ***Road nets:*** *link (street segment), hub (street intersection)* □

Example 3. Manifest Domain Perdurants: Examples of (names of) perdurants are ***Air traffic:*** *start (ascend) an aircraft, change aircraft course.* ***Banks:*** *open, deposit into, withdraw from, close (an account).* ***Container lines:*** *move container off or on board a vessel.* ***Documents:*** *open, edit, copy, shred.* ***Hospitals:*** *admit, diagnose, treat (patients).* ***Pipelines:*** *start pump, stop pump, open valve, close valve.* ***Railways:*** *switch rail point, start train.* ***Road nets:*** *set a hub signal, sense a vehicle* □

[4] Please note our distinction between method and methodology. We often find the two, to us, separate terms used interchangeably.

A **manifest domain** is further seen as a mapping from *entities* to *qualities*, that is, a mapping from manifest phenomena to usually non-manifest qualities ⊙

Example 4. Endurant Entity Qualities: Examples of (names of) endurant qualities: ***Pipeline:*** *unique identity of a pipeline unit, mereology (connectedness) of a pipeline unit, length of a pipe, (pumping) height of a pump, open/close status of a valve.* ***Road net:*** *unique identity of a road unit (hub or link), road unit mereology: identity of neighbouring hubs of a link, identity of links emanating from a hub, and state of hub (traversal) signal* □

Example 5. Perdurant Entity Qualities: Examples of (names of) perdurant qualities: ***Pipeline:*** *the signature of an open (or close) valve action, the signature of a start (or stop) pump action, etc.* ***Road net:*** *the signature of an insert (or remove) link action, the signature of an insert (or remove) hub action, the signature of a vehicle behaviour, etc.* □

We shall in the rest of this paper just write 'domain' instead of 'manifest domain'.

1.5. What Is a Domain Description ?

By a **domain description** we understand a collection of pairs of narrative and commensurate formal texts, where each pair describes either aspects of an endurant entity or aspects of a perdurant entity ⊙

What does it mean that some text describes a domain entity ?

For a text to be a **description text** it must be possible from that text to either, if it is a narrative, to reason, informally, that the *designated* entity is described to have some properties that the reader of the text can observe that the described entities also have; or, if it is a formalisation to prove, mathematically, that the formal text *denotes* the postulated properties ⊙

By a **domain description** we shall thus understand a text which describes the entities of the domain: whether endurant or perdurant, and when endurant whether discrete or continuous, atomic or composite; or when perdurant whether actions, events or behaviours. as well as the qualities of these entities. So the task of the domain analyser cum describer is clear: There is a domain: right in front of our very eyes, and it is expected that that domain be described.

1.6. Towards a Methodology of Manifest Domain Analysis & Description

Practicalities of Domain Analysis & Description. How does one go about analysing & describing a domain ? Well, for the first, one has to designate one or more domain analysers cum domain describers, i.e., trained domain scientists cum domain engineers. How does one get hold of a domain engineer ? One takes a software engineer and *educates* and *trains* that person in domain science & domain engineering. A derivative purpose of this paper is to unveil aspects of domain science & domain engineering. The education and training consists in bringing forth a number of scientific and engineering issues of domain analysis and of domain description. Among the engineering issues are such as: *what do I do when confronted with the task of domain analysis ?* and *with the task of description ?* and *when, where and how do I* `select` *and* `apply` *which* `techniques` *and which* `tools` ? Finally, there is the issue of *how do I, as a domain describer, choose appropriate abstractions and models ?*

The Four Domain Analysis & Description "Players". We can say that there are four 'players' at work here. (i) the domain, (ii) the domain analyser & describer, (iii) the domain analysis & description method, and (iv) the evolving domain analysis & description (document). (i) The *domain* is there. The domain analyser & describer cannot change the domain. Analysing & describing the domain does not change it[5]. During the analysis & description process the domain can be considered inert. (It changes with the installation of the software that has been developed from the requirements developed from the domain description.) In the physical sense the domain will usually contain entities that are static (i.e., constant), and entities that are dynamic (i.e., variable). (ii) The domain analyser & domain describer is a human, preferably a scientist/engineer[6], well-educated and trained in domain science & engineering. The domain analyser & describer observes the domain, analyses it according to a method and thereby produces a domain description.

[5] Observing domains, such as we are trying to encircle the concept of domain, is not like observing the physical world at the level of subatomic particles. The experimental physicists' instruments of observation change what is being observed.

[6] At the present time domain analysis appears to be partly an art, partly a scientific endeavour. Until such a time when domain analysis & description principles, techniques and tools have matured it will remain so.

(iii) As a concept the *method* is here considered "fixed". By 'fixed' we mean that its principles, techniques and tools do not change during a domain analysis & description. The domain analyser & describer may very well apply these principles, techniques and tools more-or-less haphazardly during domain analysis & description, flaunting the method, but the method remains invariant. The method, however, may vary from one domain analysis & description (project) to another domain analysis & description (project). Domain analysers & describers, may, for example, have become wiser from a project to the next. (iv) Finally there is the evolving *domain analysis & description*. That description is a text, usually both informal and formal. Applying a *domain description prompt* to the domain yields an *additional domain description text* which is added to the thus evolving *domain description*. One may speculate of the rôle of the "input" domain description. Does it change ? Does it help determine the additional domain description text ? Etcetera. Without loss of generality we can assume that the "input" domain description is changed[7] and that it helps determine the added text.

Of course, analysis & description is a trial-and-error, iterative process. During a sequence of analyses, that is, analysis prompts, the analyser "discovers" either more pleasing abstractions or that earlier analyses or descriptions were wrong, or that an entity either need be abstracted or made less abstract. So they are corrected.

An Interactive Domain Analysis & Description Dialogue. We see domain analysis & description as a process involving the above-mentioned four 'players', that is, as a dialogue between the domain analyser & describer and the domain, where the dialogue is guided by the method and the result is the description. We see the method as a 'player' which issues prompts: alternating between: *"analyse this"* (analysis prompts) and *"describe that"* (synthesis or, rather, description prompts).

Prompts In this paper we shall suggest a number of *domain analysis prompts* and a number of *domain description prompts*. The **domain analysis prompt**s (schematically: `analyse_named_condition`(e)) directs the analyser to inquire as to the truth of whatever the prompt "names" at wherever part (component or material), e, in the domain the prompt so designates. Based on the truth value of an analysed entity the domain analyser may then be prompted to describe that part (or material). The **domain description prompt**s (schematically: `observe_type_or_quality`(e)) directs the (analyser cum) describer to formulate both an informal and a formal description of the type or qualities of the entity designated by the prompt. The prompts form languages, and there are thus two languages at play here.

A Domain Analysis & Description Language. The 'Domain Analysis & Description Language' thus consists of a number of meta-functions, the prompts. The meta-functions have names (say `is_endurant`) and types, but have no formal definition. They are not computable. They are "performed" by the domain analysers & describers. These meta-functions are systematically introduced and informally explained in Sects. 2, 3 and 4.

The Domain Description Language. The 'Domain Description Language' is `RSL` [GHH+92], the `RAISE` Specification Language [GHH+95]. With suitable, simple adjustments it could also be either of `Alloy` [Jac06], `Event B` [Abr09], `VDM-SL` [BJ78, BJ82, FL98] or `Z` [WD96]. We have chosen `RSL` because of its simple provision for defining sorts, expressing axioms, and postulating observers over sorts.

Domain Descriptions: Narration & Formalisation Descriptions *must* be readable and *must* be mathematically precise.[8] For that reason we decompose domain description fragments into clearly identified[9] "pairs" of narrative texts and formal texts.

1.7. One Domain – Many Models ?

Will two or more domain engineers cum scientists arrive at "the same domain description" ? No, almost certainly not ! What do we mean by "the same domain description" ? To each proper description we can associate a mathematical meaning, its semantics. Not only is it very unlikely that the syntactic form of the domain descriptions are the same or even "marginally similar". But it is also very unlikely that the two (or more) semantics are the same; that is, that all properties that can be proved for one domain model can be proved also for the other. Why will different domain

[7] for example being "stylistically" revised.

[8] One must insist on formalised domain descriptions in order to be able to verify that domain descriptions satisfy a number of properties not explicitly formulated as well as in order to verify that requirements prescriptions satisfy domain descriptions.

[9] The "clear identification" is here achieved by narrative text item and corresponding formula line numbers.

models emerge ? Two different domain describers will, undoubtedly, when analysing and describing independently, focus on different aspects of the domain. One describer may focus attention on certain phenomena, different from those chosen by another describer. One describer may choose some abstractions where another may choose more concrete presentations. Etcetera. We can thus expect that a set of domain description developments lead to a set of distinct models. As these domain descriptions are communicated amongst domain engineers cum scientists we can expect that iterated domain description developments within this group of developers will lead to fewer and more similar models. Just like physicists, over the centuries of research, have arrived at a few models of nature, we can expect there to develop some consensus models of "standard" domains. We expect, that sometime in future, software engineers, when commencing software development for a "standard domain", that is, one for which there exists one or more "standard models", will start with the development of a domain description based on "one of the standard models" — just like control engineers of automatic control "repeat" an essence of a domain model for a control problem.

Example 6. One Domain – Three Models: In this paper we shall bring many examples from a domain containing automobiles. (i) One domain model may focus on roads and vehicles, with roads being modeled in terms of atomic hubs (road intersections) and atomic links (road sections between immediately neighbouring hubs), and with automobiles being modeled in terms of atomic vehicles. (ii) Another domain model considers hubs of the former model as being composite, consisting, in addition to the "bare" hub, also of a signaling part — with automobiles remaining atomic vehicles, (iii) A third model focuses on vehicles, now as composite parts consisting of composite and atomic sub-parts such as they are relevant in the assembly-line manufacturing of cars[10] □

1.8. Formal Concept Analysis

Domain analysis involves that of concept analysis. As soon as we have identified an entity for analysis we have identified a concept. The entity is usually a spatio-temporal, i.e., a physical thing. Once we speak of it, it becomes a concept. Instead of examining just one entity the domain analyser shall examine many entities. Instead of describing one entity the domain describer shall describe a class of entities. Ganter & Wille's [GW99] addresses this issue.

1.8.1. ***A Formalisation***

This section is a transcription of Ganter & Wille's [GW99] *Formal Concept Analysis, Mathematical Foundations*, the 1999 edition, Pages 17–18.

Some Notation: By $\mathscr{E}$ we shall understand the type of entities; by $\mathbb{E}$ we shall understand a phenomenon of type $\mathscr{E}$; by $\mathscr{Q}$ we shall understand the type of qualities; by $\mathbb{Q}$ we shall understand a quality of type $\mathscr{Q}$; by $\mathscr{E}$**-set** we shall understand the type of sets of entities; by $\mathbb{ES}$ we shall understand a set of entities of type $\mathscr{E}$**-set**; by $\mathscr{Q}$**-set** we shall understand the type of sets of qualities; and by $\mathbb{QS}$ we shall understand a a set of qualities of type $\mathscr{Q}$**-set**.

Definition: 1. Formal Context: A **formal context** $\mathbb{K} := (\mathbb{ES}, \mathbb{I}, \mathbb{QS})$ consists of two sets; $\mathbb{ES}$ of entities and $\mathbb{QS}$ of qualities, and a relation $\mathbb{I}$ between $\mathbb{E}$ and $\mathbb{Q}$ ⊙

To express that $\mathbb{E}$ is in relation $\mathbb{I}$ to a Quality $\mathbb{Q}$ we write $\mathbb{E} \cdot \mathbb{I} \cdot \mathbb{Q}$, which we read as *"entity* $\mathbb{E}$ ***has*** *quality* $\mathbb{Q}$*"* ⊙Example endurant entities are a specific vehicle, another specific vehicle, etcetera; a specific street segment (link), another street segment, etcetera; a specific road intersection (hub), another specific road intersection, etcetera, a monitor. Example endurant entity qualities are (a vehicle) has mobility, (a vehicle) has velocity (≥0), (a vehicle) has acceleration, etcetera; (a link) has length (>0), (a link) has location, (a link) has traffic state, etcetera.

Definition: 2. Qualities Common to a Set of Entities: For any subset, $s\mathbb{ES} \subseteq \mathbb{ES}$, of entities we can define $\mathscr{D}\mathscr{Q}$ for "derive[d] set of qualities".

$\mathscr{D}\mathscr{Q} : \mathscr{E}$**-set** $\to (\mathscr{E}$**-set** $\times\ \mathscr{I} \times\ \mathscr{Q}$**-set**$) \to \mathscr{Q}$**-set**
$\mathscr{D}\mathscr{Q}(s\mathbb{ES})(\mathbb{ES}, \mathbb{I}, \mathbb{QS}) \equiv \{\mathbb{Q} \mid \mathbb{Q}{:}\mathscr{Q}, \mathbb{E}{:}\mathscr{E} \bullet \mathbb{E} \in s\mathbb{ES} \wedge \mathbb{E} \cdot \mathbb{I} \cdot \mathbb{Q}\}$
pre: $s\mathbb{ES} \subseteq \mathbb{ES}$

The above expresses: *"the set of qualities common to entities in* $s\mathbb{ES}$*"* ⊙

Definition: 3. Entities Common to a Set of Qualities: For any subset, $s\mathbb{QS} \subseteq \mathbb{QS}$, of qualities we can define $\mathscr{D}\mathscr{E}$ for "derive[d] set of entities".

[10] The road nets of the first two models can be considered a zeroth model.

$\mathscr{D}\mathscr{E}$: $\mathscr{Q}$**-set** $\to$ ($\mathscr{E}$**-set** $\times$ $\mathscr{I}$ $\times$ $\mathscr{Q}$**-set**) $\to$ $\mathscr{E}$**-set**
$\mathscr{D}\mathscr{E}(s\mathbb{QS})(\mathbb{ES},\mathbb{I},\mathbb{QS}) \equiv \{\mathbb{E} \mid \mathbb{E}{:}\mathscr{E},\ \mathbb{Q}{:}\mathscr{Q} \bullet \mathbb{Q} \in s\mathbb{Q} \wedge \mathbb{E}\cdot\mathbb{I}\cdot\mathbb{Q}\}$,
pre: $s\mathbb{QS} \subseteq \mathbb{QS}$

The above expresses: *"the set of entities which have all qualities in* $s\mathbb{QS}$*"* $\odot$

Definition: 4. Formal Concept: A **formal concept** of a *context* $\mathbb{K}$ is a pair:

- $(s\mathbb{Q}, s\mathbb{E})$ where
  - $\circ\!\circ$ $\mathscr{D}\mathscr{Q}(s\mathbb{E})(\mathbb{E},\mathbb{I},\mathbb{Q}) = s\mathbb{Q}$ and
  - $\circ\!\circ$ $\mathscr{D}\mathscr{E}(s\mathbb{Q})(\mathbb{E},\mathbb{I},\mathbb{Q}) = s\mathbb{E}$;
- $s\mathbb{Q}$ is called the **intent** of $\mathbb{K}$ and $s\mathbb{E}$ is called the **extent** of $\mathbb{K}$ $\odot$

1.8.2. ***Types Are Formal Concepts***

Now comes the "crunch": *In the* `TripTych` *domain analysis we strive to find formal concepts and, when we think we have found one, we assign a type (or a sort) and qualities to it !*

1.8.3. ***Practicalities***

There is a little problem. To search for all those entities of a domain which each have the same sets of qualities is not feasible. So we do a combination of two things: (i) we identify a small set of entities all having the same qualities and tentatively associate them with a type, and (ii) we identify certain nouns of our national language and if such a noun does indeed designate a set of entities all having the same set of qualities then we tentatively associate the noun with a type. Having thus, tentatively, identified a type we conjecture that type and search for counterexamples, that is, entities which refute the conjecture. This "process" of conjectures and refutations is iterated until some satisfaction is arrived at that the postulated type constitutes a reasonable conjecture.

1.8.4. ***Formal Concepts: A Wider Implication***

The formal concepts of a domain form Galois Connections [GW99]. We gladly admit that this fact is one of the reasons why we emphasise formal concept analysis. At the same time we must admit that this paper does not do justice to this fact. We have experimented with the analysis & description of a number of domains, and have noticed such Galois connections, but it is, for us, too early to report on this. Thus we invite the reader to study this aspect of domain analysis.

1.9. Structure of Paper

Sections 2–4 are the main sections of this paper. They cover the analysis and description of endurants and perdurants. Section 2 introduce the concepts of entities, endurant entities and perdurant entities. Section 3 can be considered the main contribution of this paper. It introduces the external qualities of parts, components and materials, and the internal qualities of unique part identifiers, part mereologies and part attributes. Section 4 complements Sect. 3. It covers a less systematic analysis and description of perdurants. Section 5 concludes the paper.

2. Entities

2.1. General

Definition 1. Entity: By an **entity** we shall understand a **phenomenon**, i.e., something that can be observed, i.e., be seen or touched by humans, *or* that can be conceived as an abstraction of an entity. We further demand that an entity can be objectively described $\odot$[11]

[11] **Definitions** and **examples** are delimited by $\odot$ respectively $\square$

Analysis Prompt 1. `is_entity:` The domain analyser analyses "things" (θ) into either entities or non-entities. The method can thus be said to provide the ***domain analysis prompt***:

- `is_entity` — where `is_entity(`θ`)` holds if θ is an entity $\Diamond$[12]

`is_entity` is said to be a *prerequisite prompt* for all other prompts.

Whither Entities: The "demands" that entities be observable and objectively describable raises some philosophical questions. Can sentiments, like feelings, emotions or "hunches" be objectively described ? This author thinks not. And, if so, can they be other than artistically described ? It seems that psychologically and aesthetically "phenomena" appears to lie beyond objective description. We shall leave these speculations for later.

2.2. Endurants and Perdurants

Definition 2. Endurant: By an **endurant** we shall understand an entity that can be observed or conceived and described as a "complete thing" at no matter which given snapshot of time. Were we to "freeze" time we would still be able to observe the entire endurant $\odot$

That is, endurants "reside" in space. Endurants are, in the words of Whitehead [Whi20], continuants.

Example 7. Traffic System Endurants: Examples of traffic system endurants are: traffic system, road nets, fleets of vehicles, sets of hubs (i.e., street intersections), sets of links (i.e., street segments [between hubs]), and individual hubs, links and vehicles □

Definition 3. Perdurant: By a **perdurant** we shall understand an entity for which only a fragment exists if we look at or touch them at any given snapshot in time, that is, were we to freeze time we would only see or touch a fragment of the perdurant $\odot$

That is, perdurants "reside" in space and time. Perdurants are, in the words of Whitehead [Whi20], occurrents.

Example 8. Traffic System Perdurants: Examples of road net perdurants are: *insertion* and *removal* of hubs or links (actions), *disappearance* of links (events), vehicles *entering* or *leaving* the road net (actions), vehicles *crashing* (events) and *road traffic* (behaviour) □

Analysis Prompt 2. `is_endurant:` The domain analyser analyses an entity, ϕ, into an endurant as prompted by the ***domain analysis prompt***:

- `is_endurant` — ϕ is an endurant if `is_endurant(`ϕ`)` holds.

`is_entity` is a *prerequisite prompt* for `is_endurant` $\Diamond$

Analysis Prompt 3. `is_perdurant:` The domain analyser analyses an entity ϕ into perdurants as prompted by the ***domain analysis prompt***:

- `is_perdurant` — ϕ is a perdurant if `is_perdurant(`ϕ`)` holds.

`is_entity` is a *prerequisite prompt* for `is_perdurant` $\Diamond$

In the words of Whitehead [Whi20] — as communicated by Sowa [Sow99, Page 70] — an endurant has stable qualities that enable its various appearances at different times to be recognised as the same individual; a perdurant is in a state of flux that prevents it from being recognised by a stable set of qualities.

Necessity and Possibility: It is indeed possible to make the endurant/perdurant distinction. But is it necessary ? We shall argue that it is 'by necessity' that we make this distinction. Space and time are fundamental notions. They cannot be dispensed with. So, to describe manifest domains without resort to space and time is not reasonable.

[12] **Analysis** prompt definitions and **description** prompt definitions and schemes are delimited by $\Diamond$ respectively $\otimes$.

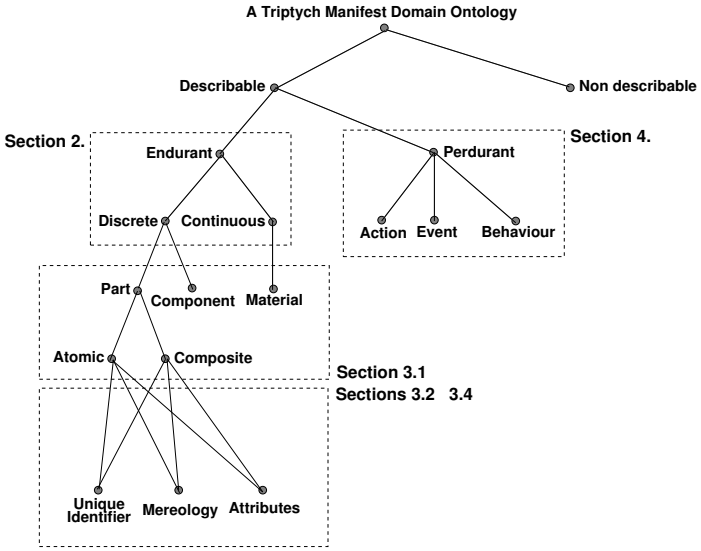

Fig. 1. An Upper Ontology for Domains

2.3. Discrete and Continuous Endurants

Definition 4. Discrete Endurant: By a **discrete endurant** we shall understand an endurant which is separate, individual or distinct in form or concept ⊙

Example 9. Discrete Endurants: Examples of discrete endurants are a road net, a link, a hub, a vehicle, a traffic signal, etcetera □

Definition 5. Continuous Endurant: By a **continuous endurant** we shall understand an endurant which is prolonged, without interruption, in an unbroken series or pattern ⊙

Example 10. Continuous Endurants: Examples of continuous endurants are water, oil, gas, sand, grain, etcetera □

Continuity shall here not be understood in the sense of mathematics. Our definition of 'continuity' focused on *prolonged, without interruption, in an unbroken series or pattern.* In that sense materials and components shall be seen as 'continuous',

Analysis Prompt 4. `is_discrete:` The domain analyser analyses endurants e into discrete entities as prompted by the ***domain analysis prompt***:

- `is_discrete` — e is discrete if `is_discrete(`e`)` holds ◊

Analysis Prompt 5. `is_continuous:` The domain analyser analyses endurants e into continuous entities as prompted by the ***domain analysis prompt***:

- `is_continuous` — e is continuous if `is_continuous(`e`)` holds ◊

2.4. An Upper Ontology Diagram of Domains

Figure 1 shows a so-called upper ontology for manifest domains. So far we have covered only a fraction of this ontology, as noted. By ontologies we shall here understand *"formal representations of a set of concepts within a domain and the relationships between those concepts"*. In Sect. 5.3 we shall review relations between our approach to modeling domains and that of many related modeling approaches, including the so-called ontology approach based on AI-models.

3. Endurants

This section brings a comprehensive treatment of the analysis and description of endurants.

3.1. Parts, Components and Materials

3.1.1. ***General***

Definition 6. Part: By a **part** we shall understand a discrete endurant which the domain engineer chooses to endow with **internal qualities** such as unique identification, mereology, and one or more attributes ⊙

We shall define the terms 'unique identification', 'mereology', and 'attributes' shortly.

Example 11. Parts: Example 7 on Page 8 illustrated, and examples 15 on the facing page and 16 on the next page shall illustrate parts □

Definition 7. Component: By a **component** we shall understand a discrete endurant which we, the domain analyser cum describer chooses to **not** endow with **internal qualities** ⊙

Example 12. Components: Examples of components are: chairs, tables, sofas and book cases in a living room, letters, newspapers, and small packages in a mail box, machine assembly units on a conveyor belt, boxes in containers of a container vessel, etcetera □

"At the Discretion of the Domain Engineer": We emphasise the following analysis and description aspects: (a) The domain is full of observable phenomena. It is the decision of the domain analyser cum describer whether to analyse and describe some such phenomena, that is, whether to include them in a domain model. (b) The borderline between an endurant being (considered) discrete or being (considered) continuous is fuzzy. It is the decision of the domain analyser cum describer whether to model an endurant as discrete or continuous. (c) The borderline between a discrete endurant being (considered) a part or being (considered) a component is fuzzy. It is the decision of the domain analyser cum describer whether to model a discrete endurant as a part or as a component. (d) In Sect. 4.11 we shall show how to "compile" parts into processes. A factor, therefore, in determining whether to model a discrete endurant as a part or as a component is whether we may consider a discrete endurant as also representing a process.

Definition 8. Material: By a **material** we shall understand a continuous endurant ⊙

Example 13. Materials: Examples of material endurants are: air of an air conditioning system, grain of a silo, gravel of a barge, oil (or gas) of a pipeline, sewage of a waste disposal system, and water of a hydro-electric power plant. □

Example 14. Parts Containing Materials: Pipeline units are here considered discrete, i.e., parts. Pipeline units serve to convey material □

3.1.2. ***Part, Component and Material Analysis Prompts***

Analysis Prompt 6. `is_part`: The domain analyser analyse endurants, *e*, into part entities as prompted by the ***domain analysis prompt***:

- `is_part` — *e* is a part if `is_part(e)` holds ◊

We remind the reader that the outcome of `is_part(e)` is very much dependent on the domain engineer's intention with the domain description, cf. Sect. 3.1.1.

Analysis Prompt 7. `is_component`: The domain analyser analyse endurants *e* into component entities as prompted by the ***domain analysis prompt***:

- `is_component` — *e* is a component if `is_component(e)` holds ◊

We remind the reader that the outcome of `is_component(e)` is very much dependent on the domain engineer's intention with the domain description, cf. Sect. 3.1.1.

Analysis Prompt 8. `is_material`: The domain analyser analyse endurants *e* into material entities as prompted by the ***domain analysis prompt***:

- `is_material` — e is a material if `is_material(`e`)` holds ◊

We remind the reader that the outcome of `is_material(`e`)` is very much dependent on the domain engineer's intention with the domain description, cf. Sect. 3.1.1 on the facing page.

3.1.3. ***Atomic and Composite Parts***

A distinguishing quality of parts is whether they are atomic or composite. Please note that we shall, in the following, examine the concept of parts in quite some detail. That is, parts become the domain endurants of main interest, whereas components and materials become of secondary interest. This is a choice. The choice is based on pragmatics. It is still the domain analyser cum describers' choice whether to consider a discrete endurant a part or a component. If the domain engineer wishes to investigate the details of a discrete endurant then the domain engineer choose to model the discrete endurant as a part otherwise as a component.

Definition 9. Atomic Part: Atomic parts are those which, in a given context, are deemed to *not* consist of meaningful, separately observable proper *sub-part*s ⊙

A **sub-part** is a *part* ⊙

Example 15. Atomic Parts: Examples of atomic parts of the above mentioned domains are: aircraft[13] (of air traffic), demand/deposit accounts (of banks), containers (of container lines), documents (of document systems), hubs, links and vehicles (of road traffic), patients, medical staff and beds (of hospitals), pipes, valves and pumps (of pipeline systems), and rail units and locomotives (of railway systems) □

Definition 10. Composite Part: Composite parts are those which, in a given context, are deemed to *indeed* consist of meaningful, separately observable proper *sub-part*s ⊙

Example 16. Composite Parts: Examples of composite parts of the above mentioned domains are: airports and air lanes (of air traffic), banks (of a financial service industry), container vessels (of container lines), dossiers of documents (of document systems), routes (of road nets), medical wards (of hospitals), pipelines (of pipeline systems), and trains, rail lines and train stations (of railway systems). □

Analysis Prompt 9. `is_atomic`: The domain analyser analyses a discrete endurant, i.e., a part p into an atomic endurant:

- `is_atomic(`p`)`: p is an atomic endurant if `is_atomic(`p`)` holds ◊

Analysis Prompt 10. `is_composite`: The domain analyser analyses a discrete endurant, i.e., a part p into a composite endurant:

- `is_composite(`p`)`: p is a composite endurant if `is_composite(`p`)` holds ◊

`is_discrete` is a **prerequisite prompt** of both `is_atomic` and `is_composite`.

Whither Atomic or Composite: If we are analysing & describing vehicles in the context of a road net, cf. Example 7 on Page 8, then we have chosen to abstract vehicles as atomic; if, on the other hand, we are analysing & describing vehicles in the context of an automobile maintenance garage then we might very well choose to abstract vehicles as composite — the sub-parts being the object of diagnosis by the auto mechanics.

3.1.4. ***On Observing Part Sorts and Types***

We use the term 'sort' when we wish to speak of an abstract type [ST12], that is, a type for which we do not wish to express a model[14]. We shall use the term 'type' to cover both abstract types and concrete types.

3.1.5. ***On Discovering Part Sorts***

Recall from Sect. 1.8.2 on Page 7 that we "equate" a formal concept with a type (i.e., a sort). Thus, to us, a part sort is a set of all those entities which all have exactly the same qualities. Our aim now is to present the basic principles that

[13] An aircraft from the point of view of airport management are atomic. From the point of view of aircraft manufacturers they are composite.
[14] for example, in terms of the concrete types: sets, Cartesians, lists, maps, or other.

let the domain analyser decide on part sorts. We observe parts one-by-one. *(α) Our analysis of parts concludes when we have "lifted" our examination of a particular part instance to the conclusion that it is of a given sort, that is, reflects a formal concept.*

Thus there is, in this analysis, a "eureka", a step where we shift focus from the concrete to the abstract, from observing specific part instances to postulating a sort: from one to the many.

Analysis Prompt 11. `observe_parts`: The ***domain analysis prompt***:

- `observe_parts`(p)

directs the domain analyser to observe the sub-parts of p ◊

Let us say the sub-parts of p are: $\{p_1,p_2,\ldots,p_m\}$. *(β) The analyser analyses, for each of these parts, p_{i_k}, which formal concept, i.e., sort, it belongs to; let us say that it is of sort P_k; thus the sub-parts of p are of sorts $\{P_1,P_2,\ldots,P_m\}$. Some P_k may be atomic sorts, some may be composite sorts.*

The domain analyser continues to examine a finite number of other composite parts: $\{p_j, p_\ell, \ldots, p_n\}$. It is then "discovered", that is, decided, that they all consists of the same number of sub-parts $\{p_{i_1},p_{i_2},\ldots,p_{i_m}\}$, $\{p_{j_1},p_{j_2},\ldots,p_{j_m}\}$, $\{p_{\ell_1},p_{\ell_2},\ldots,p_{\ell_m}\}$, ..., $\{p_{n_1},p_{n_2},\ldots,p_{n_m}\}$, of the same, respective, part sorts. *(γ) It is therefore concluded, that is, decided, that $\{p_i,p_j,p_\ell,\ldots,p_n\}$ are all of the same part sort P with observable part sub-sorts $\{P_1,P_2,\ldots,P_m\}$.*

Above we have *type-font-highlighted* three sentences: (α,β,γ). When you analyse what they "prescribe" you will see that they entail a "depth-first search" for part sorts. The β sentence says it rather directly: *"The analyser analyses, for each of these parts, p_k, which formal concept, i.e., part sort it belongs to."* To do this analysis in a proper way, the analyser must ("recursively") analyse the parts "down" to their atomicity, and from the atomic parts decide on their part sort, and work ("recurse") their way "back", through possibly intermediate composite parts, to the p_ks.

3.1.6. ***Part Sort Observer Functions***

The above analysis amounts to the analyser first "applying" the domain analysis prompt `is_composite`(p) to a discrete endurant, where we now assume that the obtained truth value is **true**. Let us assume that parts p:P consists of sub-parts of sorts $\{P_1,P_2,\ldots,P_m\}$. Since we cannot automatically guarantee that our domain descriptions secure that P and each P_i ($1\leq i\leq m$) denotes disjoint sets of entities we must prove it.

Domain Description Prompt 1. `observe_part_sorts`: If `is_composite`(p) holds, then the analyser "applies" the ***domain description prompt***

- `observe_part_sorts`(p)

resulting in the analyser writing down the *part sorts and part sort observers* domain description text according to the following schema:

1. `observe_part_sorts` schema

Narration:
[s] ... narrative text on sorts ...
[o] ... narrative text on sort observers ...
[i] ... narrative text on sort recognisers ...
[p] ... narrative text on proof obligations ...
Formalisation:
type
[s] P,
[s] P_i [$1\leq i\leq m$] **comment:** P_i [$1\leq i\leq m$] abbreviates P_1, P_2, ..., P_m
value
[o] **obs_part**_P_i: P → P_i [$1\leq i\leq m$]
[i] **is**_P_i: ($P_1|P_2|...|P_m$) → **Bool** [$1\leq i\leq m$]

proof obligation [Disjointness of part sorts]
[p] ∀ p:($P_1|P_2|...|P_m$) •
[p] ⋀ {**is**_P_i(p) ≡ ⋀ {∼**is**_P_j(p) | j ∈ {1..m} \ {i}} | i ∈ {1..m}}

`is_composite` is a **prerequisite prompt** of `observe_part_sorts` △

We do not here state guidelines for discharging these kinds of proof obligations. But we will very informally sketch such discharges, see below.

Example 17. Composite and Atomic Part Sorts of Transportation: The following example illustrates the multiple use of the `observe_part_sorts` function: first to δ:Δ, a specific transport domain, Item 1, then to an $n:N$, the net of that domain, Item 2, and then to an $f:F$, the fleet of that domain, Item 3.

1 A transportation domain is viewed as composed from a net (of hubs and links), a fleet (of vehicles) and a monitor.
2 A transportation net is here seen as composed from a collection of hubs and a collection of links.
3 A fleet is here seen as a collection of vehicles.

The monitor is considered an atomic part.

type
1. Δ, N, F, M
value
1. **obs_part**_N: Δ → N, **obs_part**_F: Δ → F, **obs_part**_M: Δ → M
type
2. HS, LS
value
2. **obs_part**_HS: N → HS, **obs_part**_LS: N → LS
type
3. VS
value
3. **obs_part**_VS: F → VS

*A **proof obligation** has to be discharged, one that shows disjointedness of sorts N, F and M. An informal sketch is: entities of sort N are composite and consists of two parts: aggregations of hubs, HS, and aggregations of links, LS. Entities of sort F consists of an aggregation, VS, of vehicles. So already that makes N and F disjoint. M is an atomic entity — where N and F are both composite. Hence the three sorts N, F and M are disjoint* □

3.1.7. ***On Discovering Concrete Part Types***

Analysis Prompt 12. `has_concrete_type`: The domain analyser may decide that it is expedient, i.e., pragmatically sound, to render a part sort, P, whether atomic or composite, as a concrete type, T. That decision is prompted by the holding of the ***domain analysis prompt***:

- `has_concrete_type`(p).

`is_discrete` is a **prerequisite prompt** of `has_concrete_type` ◊

The reader is reminded that the decision as to whether an abstract type is (also) to be described concretely is entirely at the discretion of the domain engineer.

Domain Description Prompt 2. `observe_part_type`: Then the domain analyser applies the ***domain description prompt***:

- `observe_part_type`(p)[15]

to parts p:P which then yield the *part type and part type observers* domain description text according to the following schema:

2. `observe_part_type` schema

Narration:
[t_1] ... narrative text on sorts and types S_i ...
[t_2] ... narrative text on types T ...
[o] ... narrative text on type observers ...
Formalisation:

[15] `has_concrete_type` is a **prerequisite prompt** of `observe_part_type`.

type
[t$_1$] S_1, S_2, ..., S_m, ..., S_n,
[t$_2$] $T = \mathscr{E}(S_1,S_2,...,S_n)$
value
[o] **obs_part**_T: $P \to T$

where S_1,S_2,...,S_m,...,S_n may be any types, including part sorts, where $0 \leq m \leq n \geq 1$, where m is the number of new (atomic or composite) sorts, and where $n - m$ is the number of concrete types (like **Bool, Int, Nat**) or sorts already analysed & described. and $\mathscr{E}(S_1,S_2,...,S_n)$ is a type expression $\triangle$

The type name, T, of the concrete type, as well as those of the auxiliary types, S_1,S_2,...,S_m, are chosen by the domain describer: they may have already been chosen for other sort–to–type descriptions, or they may be new.

Example 18. Concrete Part Types of Transportation: We continue Example 17 on the previous page:

4 A collection of hubs is here seen as a set of hubs and a collection of links is here seen as a set of links.
5 Hubs and links are, until further analysis, part sorts.
6 A collection of vehicles is here seen as a set of vehicles.
7 Vehicles are, until further analysis, part sorts.

type
4. Hs = H-**set**, Ls = L-**set**
5. H, L
6. Vs = V-**set**
7. V
value
4. **obs_part**_Hs: HS $\to$ Hs, **obs_part**_Ls: LS $\to$ Ls
6. **obs_part**_Vs: VS $\to$ Vs $\square$

3.1.8. ***Forms of Part Types***

Usually it is wise to restrict the part type definitions, $T_i = \mathscr{E}_i(Q,R,...,S)$, to simple type expressions. T=A-**set** or $T=A^*$ or $T=ID \xrightarrow[m]{} A$ or $T=A_t|B_t|...|C_t$ where ID is a sort of unique identifiers, $T=A_t|B_t|...|C_t$ defines the disjoint types $A_t==mkA_t(s:A_s)$, $B_t==mkB_t(s:B_s)$, ..., $C_t==mkC_t(s:C_s)$, and where A, A_s, B_s, ..., C_s are sorts. Instead of $A_t==mkA_t(a:A_s)$, etc., we may write $A_t::A_s$ etc.

3.1.9. ***Part Sort and Type Derivation Chains***

Let P be a composite sort. Let P_1, P_2, ..., P_m be the part sorts "discovered" by means of `observe_part_sorts`(p) where p:P. We say that P_1, P_2, ..., P_m are (immediately) **derived** from P. If P_k is derived from P_j and P_j is derived from P_i, then, by transitivity, P_k is **derived** from P_i.

No Recursive Derivations We "mandate" that if P_k is derived from P_j then there can be no P derived from P_j such that P is P_j, that is, P_j cannot be derived from P_j.

That is, we do not allow recursive domain sorts.

It is not a question, actually of allowing recursive domain sorts. It is, we claim to have observed, in very many domain modeling experiments, that there are no recursive domain sorts !

3.1.10. ***Names of Part Sorts and Types***

The domain analysis and domain description text prompts `observe_part_sorts`, `observe_material_sorts` and `observe_part_type` — as well as the `attribute_names`, `observe_material_sorts`, `observe_unique_identifier`, `observe_mereology` and `observe_attributes` prompts introduced below — "yield" type names. That is, it is as if there is a reservoir of an indefinite-size set of such names from which these names are

"pulled", and once obtained are never "pulled" again. There may be domains for which two distinct part sorts may be composed from identical part sorts. In this case the domain analyser indicates so by prescribing a part sort already introduced.

Example 19. Container Line Sorts: Our example is that of a container line with container vessels and container terminal ports.

8 A container line contains a number of container vessels
and a number of container terminal ports,
as well as other parts.
9 A container vessel contains a container stowage area, etc.
10 A container terminal port contains a container stowage area, etc.
11 A container stowage areas contains a set of uniquely identified container bays.
12 A container bay contains a set of uniquely identified container rows.
13 A container row contains a set of uniquely identified container stacks.
14 A container stack contains a stack, i.e., a first-in, last-out sequence of containers.
15 Containers are further undefined.

After a some slight editing we get:

```
type
  CL
  VS, VI, V, Vs = VI →m V,
  PS, PI, P, Ps = PI →m P
value
  obs_part_VS: CL → VS
  obs_part_Vs: VS → Vs
  obs_part_PS: CL → PS
  obs_part_Ps: CTPS → CTPs
type
  CSA
value
  obs_part_CSA: V → CSA
  obs_part_CSA: P → CSA
```

```
type
  BAYS, BI, BAY, Bays=BI →m BAY
  ROWS, RI, ROW, Rows=RI →m ROW
  STKS, SI, STK, Stks=SI →m STK
  C
value
  obs_part_BAYS: CSA → BAYS,
  obs_part_Bays: BAYS → Bays
  obs_part_ROWS: BAY → ROWS,
  obs_part_Rows: ROWS → Rows
  obs_part_STKS: ROW → STKS,
  obs_part_Stks: STKS → Stks
  obs_part_Stk: STK → C*
```

Note that observe_part_sorts(v:V) and observe_part_sorts(p:P) both yield CSA □

3.1.11. ***More On Part Sorts and Types***

The above "experimental example" motivates the below. We can always assume that composite parts $p{:}P$ abstractly consists of a definite number of sub-parts.

Example 20.. We comment on Example 17, Page 13: Parts of type Δ and N are composed from three, respectively two abstract sub-parts of distinct types □
Some of the parts, say p_{i_z} of $\{p_{i_1},p_{i_2},\ldots,p_{i_m}\}$, of $p{:}P$, may themselves be composite.

Example 21.. We comment on Example 17: Parts of type N, F, HS, LS and VS are all composite □
There are, pragmatically speaking, two cases for such compositionality. Either the part, p_{i_z}, of type t_{i_z}, is is composed from a definite number of abstract or concrete sub-parts of distinct types.

Example 22.. We comment on Example 17: Parts of type N are composed from three sub-parts □
Or it is composed from an indefinite number of sub-parts of the same sort.

Example 23.. We comment on Example 17: Parts of type HS, LS and VS are composed from an indefinite numbers of hubs, links and vehicles, respectively □

Example 24. Pipeline Parts:

16 A pipeline consists of an indefinite number of pipeline units.
17 A pipeline units is either a well, or a pipe, or a pump, or a valve, or a fork, or a join, or a sink.
18 All these unit sorts are atomic and disjoint.

type
16. PL, U, We, Pi, Pu, Va, Fo, Jo, Si
16. Well, Pipe, Pump, Valv, Fork, Join, Sink
value
16. **obs_part**_Us: PL → U-**set**
type
17. U == We | Pi | Pu | Va | Fo | Jo | Si
18. We::Well, Pi::Pipe, Pu::Pump, Va::Valv, Fo:Fork, Jo::Join, Si::Sink

The experimental research report [Bjø13b] covers pipelines in some detail □

3.1.12. ***External and Internal Qualities of Parts***

By an **external part quality** we shall understand the `is_atomic`, `is_composite`, `is_discrete` and `is_continuous` qualities ⊙ By an **internal part quality** we shall understand the part qualities to be outlined in the next sections: `unique identification`, `mereology` and `attributes` ⊙ By **part qualities** we mean the sum total of `external endurant` and `internal endurant` qualities ⊙

3.1.13. ***Three Categories of Internal Qualities***

We suggest that the internal qualities of parts be analysed into three categories: (i) a category of unique part identifiers, (ii) a category of mereological quantities and (iii) a category of general attributes. Part mereologies are about sharing qualities between parts. Some such sharing expresses spatio-topological properties of how parts are organised. Other part sharing aspects express relations (like equality) of part attributes. We base our modeling of mereologies on the notion of unique part identifiers. Hence we cover **internal qualities** in the order (i–ii–iii).

3.2. Unique Part Identifiers

We introduce a notion of unique identification of parts. We assume (i) that all parts, p, of any domain P, have unique identifiers, (ii) that unique identifiers (of parts p:P) are abstract values (of the unique identifier sort PI of parts p:P), (iii) such that distinct part sorts, P_i and P_j, have distinctly named unique identifier sorts, say PI_i and PI_j, (iv) that all π_i:PI_i and π_j:PI_j are distinct, and (v) that the observer function **uid**_P applied to p yields the unique identifier, say π:PI, of p.

Representation of Unique Identifiers: Unique identifiers are abstractions. When we endow two parts (say of the same sort) with distinct unique identifiers then we are simply saying that these two parts are distinct. We are not assuming anything about how these identifiers otherwise come about.

Domain Description Prompt 3. `observe_unique_identifier`: We can therefore apply the ***domain description prompt***:

- `observe_unique_identifier`

to parts p:P resulting in the analyser writing down the *unique identifier type and observer* domain description text according to the following schema:

3. `observe_unique_identifier` schema

Narration:
[s] ... narrative text on unique identifier sort PI ...
[u] ... narrative text on unique identifier observer **uid**_P ...
[a] ... axiom on uniqueness of unique identifiers ...
Formalisation:
type
[s] PI
value
[u] **uid**_P: P → PI

axiom
[a] $\mathscr{U}$

$\mathscr{U}$ is a predicate over part sorts and unique part identifier sorts. The unique part identifier sort, PI, is unique, as are all part sort names, P ⊗

Example 25. Unique Transportation Net Part Identifiers: We continue Example 17 on Page 13.

19 Links and hubs have unique identifiers
20 and unique identifier observers.

type
19. LI, HI
value
20. **uid**_LI: L → LI
20. **uid**_HI: H → HI
axiom [Well−formedness of Links, L, and Hubs, H]
19. ∀ l,l′:L • **uid**_LI(l)=**uid**_LI(l′) ⇒ l=l′,
19. ∀ h,h′:H • **uid**_HI(h)=**uid**_HI(h′) ⇒ h=h′ □

Axiom 19, although expressed for links and hubs of road nets, applies in general: Two parts with the same unique part identifiers are indeed one and the same part.

3.3. Mereology

Mereology is the study and knowledge of parts and part relations. Mereology, as a logical/philosophical discipline, can perhaps best be attributed to the Polish mathematician/logician Stanisław Leśniewski [CV99, Bjø14a].

3.3.1. ***Part Relations***

Which are the relations that can be relevant for part-hood ? We give some examples. Two otherwise distinct parts may share attribute values. [16]

Example 26. Shared Timetable Mereology (I): Two or more distinct public transport busses may "run" according to the (identically) same, thus "shared", bus time table (cf. Example 37 on Page 23) □

Two otherwise distinct parts may be said to, for example, be topologically "adjacent" or one "embedded" within the other.

Example 27. Topological Connectedness Mereology: (i) two rail units may be connected (i.e., adjacent); (ii) a road link may be connected to two road hubs; (iii) a road hub may be connected to zero or more road links; (iv) distinct vehicles of a road net may be monitored by one and the same road pricing sub-system □

The above examples are in no way indicative of the "space" of part relations that may be relevant for part-hood. The domain analyser is expected to do a bit of experimental research in order to discover necessary, sufficient and pleasing "mereology-hoods" !

3.3.2. ***Part Mereology: Types and Functions***

Analysis Prompt 13. `has_mereology:` To discover necessary, sufficient and pleasing "mereology-hoods" the analyser can be said to endow a truth value, **true**, to the ***domain analysis prompt***:

- `has_mereology`

[16] For the concept of attribute value see Sect. 3.4.2 on Page 20.

When the domain analyser decides that some parts are related in a specifically enunciated mereology, the analyser has to decide on suitable mereology types and mereology observers (i.e., part relations).

We can define a **mereology type** as a type $\mathscr{E}$xpression over unique [part] identifier types. We generalise to unique [part] identifiers over a definite collection of part sorts, P1, P2, ..., Pn, where the parts p1:P1, p2:P2, ..., pn:Pn are not necessarily (immediate) sub-parts of some part p:P.

type
PI1, PI2, ..., PIn
MT = $\mathscr{E}$(PI1, PI2, ..., PIn),

Domain Description Prompt 4. `observe_mereology`: If `has_mereology`(p) holds for parts p of type P, then the analyser can apply the **domain description prompt**:

- `observe_mereology`

to parts of that type and write down the *mereology types and observer* domain description text according to the following schema:

4. `observe_mereology` schema

Narration:
[t] ... narrative text on mereology type ...
[m] ... narrative text on mereology observer ...
[a] ... narrative text on mereology type constraints ...
Formalisation:
type
[t] MT[17] = $\mathscr{E}$(PI1,PI2,...,PIm)
value
[m] **obs_mereo_**P: P → MT
axiom [Well−formedness of Domain Mereologies]
[a] $\mathscr{A}$(MT)

Here $\mathscr{E}$(PI1,PI2,...,PIm) is a type expression over possibly all unique identifier types of the domain description, and $\mathscr{A}$(MT) is a predicate over possibly all unique identifier types of the domain description. To write down the concrete type definition for MT requires a bit of analysis and thinking. `has_mereology` is a **prerequisite prompt** for `observe_mereology` △

Example 28. Road Net Part Mereologies: We continue Example 17 on Page 13 and Example 25 on the preceding page.

21 Links are connected to exactly two distinct hubs.
22 Hubs are connected to zero or more links.
23 For a given net the link and hub identifiers of the mereology of hubs and links must be those of links and hubs, respectively, of the net.

type
21. LM′ = HI-**set**, LM = {|his:HI-**set** • **card**(his)=2|}
22. HM = LI-**set**
value
21. **obs_mereo_**L: L → LM
22. **obs_mereo_**H: H → HM
axiom [Well−formedness of Road Nets, N]
23. ∀ n:N,l:L,h:H•
23. l ∈ **obs_part_**Ls(**obs_part_**LS(n))
23. ∧ h ∈ **obs_part_**Hs(**obs_part_**HS(n))

[17] MT will be used several times in Sect. 4.11.

23. ⇒ **obs_mereo_L**(l) ⊆ ∪{**uid_H**(h) | h ∈ **obs_part_Hs**(**obs_part_HS**(n))}
23. ∧ **obs_mereo_H**(h) ⊆ ∪{**uid_H**(l) | l ∈ **obs_part_Ls**(**obs_part_LS**(n))} □

Example 29. Pipeline Parts Mereology: We continue Example 24 on Page 15. Pipeline units serve to conduct fluid or gaseous material. The flow of these occur in only one direction: from so-called input to so-called output.

24 Wells have exactly one connection to an output unit.

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

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

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

28 Sinks have exactly one connection from an input unit.

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

type
29. UM′=(**UI-set**×**UI-set**)
29. UM={|(iuis,ouis):UM′•iuis ∩ ouis={}|}
value
29. **obs_mereo_U**: UM
axiom [Well−formedness of Pipeline Systems, PLS (0)]
∀ pl:PL,u:U • u ∈ **obs_part_Us**(pl) ⇒
let (iuis,ouis)=**obs_mereo_U**(u) **in**
case (**card** iuis,**card** ouis) **of**
24. (0,1) → **is_We**(u),
25. (1,1) → **is_Pi**(u)∨**is_Pu**(u)∨**is_Va**(u),
26. (1,2) → **is_Fo**(u),
27. (2,1) → **is_Jo**(u),
28. (1,0) → **is_Si**(u), _ → **false**
end end

Example 43 on Page 27 (axiom Page 27) and Example 44 on Page 27 (axiom Page 27) illustrates the need to constrain the sets of endurant entities denoted by definitions of part sort, unique identifier and mereology attribute definitions □

3.3.3. ***Formulation of Mereologies***

The **`observe_mereology`** domain descriptor, Page 18, may give the impression that the mereo type MT can be described "at the point of issue" of the **`observe_mereology`** prompt. Since the MT type expression may, in general, depend on any part sort the mereo type MT can, for some domains, "first" be described when all part sorts have been dealt with. In [Bjø14b] we we present a model of one form of evaluation of the `TripTych` analysis and description prompts.

3.4. Part Attributes

To recall: there are three sets of **internal qualities**: unique part identifiers, part mereology and attributes. Unique part identifiers and part mereology are rather definite kinds of internal endurant qualities. Part attributes form more "free-wheeling" sets of internal qualities.

3.4.1. ***Inseparability of Attributes from Parts***

Parts are typically recognised because of their spatial form and are otherwise characterised by their intangible, but measurable attributes. That is, whereas endurants, whether discrete (as are parts and components) or continuous (as are materials), 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[18], or seen[19], but can be objectively measured[20]. 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 aesthetics. We learned from Sect. 1.8 that a formal concept, that is, a type, consists of all the entities which all have the same qualities. Thus removing a quality from an entity makes no sense: the entity of that type either becomes an entity of another type or ceases to exist (i.e., becomes a non-entity) !

3.4.2. ***Attribute Quality and Attribute Value***

We distinguish between an attribute, as a logical proposition, and an attribute value, as a value in some not necessarily Boolean value space.

Example 30. Attribute Propositions and Other Values: A particular street segment (i.e., a link), say ℓ, satisfies the proposition (attribute) has_length, and may then have value length 90 meter for that attribute. Another link satisfies the same proposition but has another value; and yet another link satisfies the same proposition and may have the same value. That is: all links satisfies has_length and has some value for that attribute. A particular road transport domain, δ, has three immediate sub-parts: net, n, fleet, f, and monitor m; typically nets `has_net_name` and `has_net_owner` proposition attributes with, for example, `US Interstate Highway System` respectively `US Department of Transportation` as values for those attributes There may be other aspects of the net value n □

3.4.3. ***Endurant Attributes: Types and Functions***

Let us recall that attributes cover qualities other than unique identifiers and mereology. Let us then consider that parts have one or more attributes. These attributes are qualities which help characterise "what it means" to be a part. Note that we expect every part to have at least one attribute.

Example 31. Atomic Part Attributes: Examples of attributes of atomic parts such as a human are: *name, gender, birth-date, birth-place, nationality, height, weight, eye colour, hair colour,* etc. Examples of attributes of transport net links are: *length, location, 1 or 2-way link, link condition,* etc. □

Example 32. Composite Part Attributes: Examples of attributes of composite parts such as a road net are: *owner, public or private net, free-way or toll road, a map of the net,* etc. Examples of attributes of a group of people could be: *statistic distributions of gender, age, income, education, nationality, religion,* etc. □

We now assume that all parts have attributes. The question is now, in general, how many and, particularly, which.

Analysis Prompt 14. `attribute_names`: The ***domain analysis prompt*** `attribute_names` when applied to a part p yields the set of names of its attribute types:

- `attribute_names`(p): $\{\eta A_1, \eta A_2, ..., \eta A_n\}$.

η is a type operator. Applied to a type A it yields is name[21] ◊

We cannot automatically, that is, syntactically, guarantee that our domain descriptions secure that the various attribute types for an emerging part sort denote disjoint sets of values. Therefore we must prove it.

The Attribute Value Observer The "built-in" description language operator

- **attr_**A

applies to parts, p:P, where ηA∈`attribute_names`(p). It yields the value of attribute A of p.

[18] One can see the red colour of a wall, but one touches the wall.

[19] One cannot see electric current, and one may touch an electric wire, but only if it conducts high voltage can one know that it is indeed an electric wire.

[20] That is, we restrict our domain analysis with respect to attributes to such quantities which are observable, say by mechanical, electrical or chemical instruments. Once objective measurements can be made of human feelings, beauty, and other, we may wish to include these "attributes" in our domain descriptions.

[21] Normally, in non-formula texts, type A is referred to by ηA. In formulas A denote a type, that is, a set of entities. Hence, when we wish to emphasize that we speak of the name of that type we use ηA. But often we omit the distinction

Domain Description Prompt 5. `observe_attributes`: The domain analyser experiments, thinks and reflects about part attributes. That process is initated by the ***domain description prompt***:

- `observe_attributes`.

The result of that ***domain description prompt*** is that the domain analyser cum describer writes down the *attribute (sorts or) types and observers* domain description text according to the following schema:

5. `observe_attributes` schema

Narration:
[t] ... narrative text on attribute sorts ...
[o] ... narrative text on attribute sort observers ...
[i] ... narrative text on attribute sort recognisers ...
[p] ... narrative text on attribute sort proof obligations ...
Formalisation:
type
[t] A_i $[1 \leq i \leq n]$
value
[o] **attr**_A_i:P→A_i $[1 \leq i \leq n]$
[i] **is**_A_i:($A_1|A_2|...|A_n$)→**Bool** $[1 \leq i \leq n]$
proof obligation [Disjointness of Attribute Types]
[p] ∀ δ:Δ
[p] **let** P be any part sort **in** [the Δ domain description]
[p] **let** a:($A_1|A_2|...|A_n$) **in** **is**_A_i(a) ≠ **is**_A_j(a) **end end** $[i \neq j, 1 \leq i,j \leq n]$

The **type** (or rather sort) definitions: A_1, A_2, ..., A_n, inform us that the domain analyser has decided to focus on the distinctly named A_1, A_2, ..., A_n attributes.[22] And the **value** clauses **attr**_A_1:P→A_1, **attr**_A_2:P→A_2, ..., **attr**_A_n:P→A_n are then "automatically" given: if a part, p:P, has an attribute A_i then there is postulated, "by definition" [eureka] an attribute observer function **attr**_A_i:P→A_i etcetera △

The fact that, for example, A_1, A_2, ..., A_n, are attributes of p:P, means that the propositions

- `has_attribute_`A_1(p), `has_attribute_`A_2(p), ..., and `has_attribute_`A_n(p)

holds. Thus the observer functions **attr**_A_1, **attr**_A_2, ..., **attr**_A_n can be applied to p in P and yield attribute values a_1:A_1, a_2:A_2, ..., a_n:A_n respectively.

Example 33. Road Hub Attributes: After some analysis a domain analyser may arrive at some interesting hub attributes:

30 hub state: from which links (by reference) can one reach which links (by reference),
31 hub state space: the set of all potential hub states that a hub may attain,
32 such that
   a the links referred to in the state are links of the hub mereology
   b and the state is in the state space.
33 Etcetera — i.e., there are other attributes not mentioned here.

type
30 HΣ = (LI×LI)-**set**
31 HΩ = HΣ-**set**
value
30 **attr**_HΣ:H→HΣ
31 **attr**_HΩ:H→HΩ
axiom [Well−formedness of Hub States, HΣ]

[22] The attribute type names are not like type names of, for example, a programming language. Instead they are chosen by the domain analyser to reflect on domain phenomena. Cf. Example 31 on the facing page and Example 32.

32 ∀ h:H • **let** hσ = **attr_**HΣ(h) **in**
32.a {li,li′|li,li′:LI•(li,li′)∈ hσ}⊆**obs_mereo_**H(h)
32.b ∧ hσ ∈ **attr_**HΩ(h)
32 **end** □

3.4.4. ***Attribute Categories***

One can suggest a hierarchy of part attribute categories: static or dynamic values — and within the dynamic value category: inert values or reactive values or active values — and within the dynamic active value category: autonomous values or biddable values or programmable values. We now review these attribute value types. The review is based on [Jac95, M.A. Jackson]. **Part attributes** are either constant or varying, i.e., **static** or **dynamic** attributes. By a **static attribute**, a:A, `is_static_attribute`(a), we shall understand an attribute whose values are constants, i.e., cannot change. By a **dynamic attribute**, a:A, `is_dynamic_attribute`(a), we shall understand an attribute whose values are variable, i.e., can change. **Dynamic attributes** are either inert, reactive or active attributes. By an **inert attribute**, a:A, `is_inert_attribute`(a), we shall understand a dynamic attribute whose values only change as the result of external stimuli where these stimuli prescribe properties of these new values. By a **reactive attribute**, a:A, `is_reactive_attribute`(a), we shall understand a dynamic attribute whose values, if they vary, change value in response to the change of other attribute values. By an **active attribute**, a:A, `is_active_attribute`(a), we shall understand a dynamic attribute whose values change (also) of its own volition. **Active attributes** are either autonomous, biddable or programmable attributes. By an **autonomous attribute**, a:A, `is_autonomous_attribute`(a), we shall understand a dynamic active attribute whose values change value only "on their own volition". The values of an autonomous attributes are a "law onto themselves and their surroundings". By a **biddable attribute**, a:A, `is_biddable_attribute`(a), (of a part) we shall understand a dynamic active attribute whose values are prescribed but may fail to be observed as such. By a **programmable attribute**, a:A, `is_programmable_attribute`(a), we shall understand a dynamic active attribute whose values can be prescribed.

Example 34. Static and Dynamic Attributes: Link lengths can be considered **static.** Buses (i.e., vehicles) have a *timetable* attribute which is **inert**, i.e., can change, only when the bus company decides so. The weather can be considered **autonomous.** Pipeline valve units include the two attributes of *valve opening* (`open, close`) and *internal flow* (measured, say `gallons` per `second`). The valve opening attribute is of the **biddable** attribute category. The flow attribute is **reactive** (flow changes with valve opening/closing). Hub states (red, yellow, green) can be considered **biddable:** one can "try" set the signals but the electro-mechanics may fail. Bus companies **program** their own timetables, i.e., bus company timetables are **programmable** — are computers □

External Attributes: By an **external attribute** we shall understand a dynamic attribute which is not a reactive or a programmable attribute ⊙ Thus we can define the domain analysis prompt: `is_external_attribute`, as:

```
value
    is_external_attribute: P → Bool
    is_external_attribute(p) ≡
        is_dynamic_attribute(p) ∧ ∼is_reactive_attribute(p) ∧ ∼is_programmable_attribute(p)
        pre: is_endurant(p) ∧ is_discrete(p)
```

The idea of external attributes is this: They are the attributes whose (deterministic or non-deterministic) values are determined by factors "outside" the part of which they are an attribute. In contrast, the programmable attributes have their values determined by the part of which they are an attribute.

Figure 2 on the next page captures an attribute value ontology.

3.4.5. ***Access to Attribute Values***

In an action, event or a behaviour description (Sect. 4.9) static values of parts, p, (say of type A) can be "copied", **attr_**A(p), and still retain their (static) value. But, for action, event or behaviour descriptions, external dynamic values of parts, p, cannot be "copied", but **attr_**A(p) must be "performed" every time they are needed. That is: static values require at most one domain access, whereas external attribute values require repeated domain accesses. We shall return to the issue of attribute value access in Sect. 4.7.

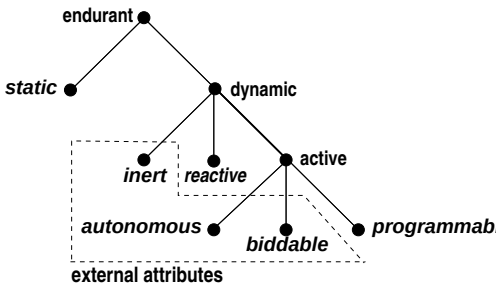

Fig. 2. Attribute Value Ontology

3.4.6. ***Event Values***

Among the external attribute values we observe a new kind of value: the **event values.** We may optionally ascribe ordinarily typed, say A, values, a:A, with event attributes. By an **event attribute** we shall understand an attribute whose values are either "nil" ([f]or "absent"), or are some more definite value (a:A) $\odot$ Event values *occur* instantaneously. They can be thought of as the raising of a signal followed immediately by the lowering of that signal.

Example 35. Event Attributes: (i) The passing of a vehicle past a tollgate is an event. It occurs at a usually unpredictable time. It otherwise "carries" no specific value. (ii) The identification of a vehicle by a tollgate sensor is an event. It occurs at a usually unpredictable time. It specifically "carries" a vehicle identifier value □

Event attributes are not to be confused with event perdurants. External attributes are either event attributes or are not. More on access to event attribute values in Sect. 4.7.4 on Page 33.

3.4.7. ***Shared Attributes***

Normally part attributes of different part sorts are distinctly named. If, however, observe_attributes(p_{ik}:P_i) and observe_attributes($\mathsf{p}_{j\ell}$:P_j), for any two distinct part sorts, P_i and P_j, of a domain, "discovers" identically named attributes, say A, then we say that parts p_i:P_i and p_j:P_j **share** attribute A. that is, that a:**attr**_A(p_i) (and a′:**attr**_A(p_j)) is a **shared attribute** (with a=a′ always (□) holding).

Attribute Naming: Thus the domain describer has to exert great care when naming attribute types. If P_i and P_j are two distinct types of a domain then if and only if an attribute of P_i is to be shared with an attribute of P_j must that attribute be identically named in the description of P_i and P_j and otherwise the attribute names of P_i and P_j must be distinct.

Example 36.. Shared Attributes. Examples of shared attributes: (i) Bus timetable attributes have the same value as the fleet timetable attribute – cf. Example 37 below. (ii) A link incident upon or emanating from a hub shares the connection between that link and the hub as an attribute. (iii) Two pipeline units[23], p_i with unique identifier π_i, and p_j with unique identifier π_j, that are connected, such that an outlet marked π_j of p_i "feeds into" inlet marked π_i of p_j, are said to share the connection (modeled by, e.g., $\{(\pi_i,\pi_j)\}$) □

Example 37. Shared Timetables: The fleet and vehicles of Example 17 on Page 13 and Example 18 on Page 14 is that of a bus company.

34 From the fleet and from the vehicles we observe unique identifiers.

35 Every bus mereology records the same one unique fleet identifier.

36 The fleet mereology records the set of all unique bus identifiers.

37 A bus timetable is a shared fleet and bus attribute.

type
34. FI, VI, BT

[23] See Example 29 on Page 19

value
34. **uid**_F: F → FI
34. **uid**_V: V → VI
35. **obs_mereo**_F: F → VI-**set** [cf. Sect. 3.3.2 on Page 17]
36. **obs_mereo**_V: V → FI
37. **attr**_BT: (F|V) → BT

axiom
□ ∀ f:F ⇒
∀ v:V • v ∈ **obs_part**_Vs(**obs_part**_VC(f)) • **attr**_BT(f) = **attr**_BT(v) □

3.5. Components

We refer to Sect. 3.1.1 on Page 10 for a first coverage of the concept of components: definition and examples. Components are discrete endurants which the domain analyser & describer has chosen to **not** endow with **internal qualities**.

Example 38. Parts and Components: We observe components as associated with atomic parts: The contents, that is, the collection of zero, one or more boxes, of a container are the components of the container part. Conveyor belts transport machine assembly units and these are thus considered the components of the conveyor belt □

We now complement the `observe_part_sorts` (of Sect. 3.1.6). We assume, without loss of generality, that only atomic parts may contain components. Let p:P be some atomic part.

Analysis Prompt 15. `has_components`: The ***domain analysis prompt***:

- `has_components`(p)

yields **true** if atomic part p may contain zero, one or more components otherwise false ◊

Let us assume that parts p:P embody components of sorts $\{K_1,K_2,\ldots,K_n\}$. Since we cannot automatically guarantee that our domain descriptions secure that each K_i ($[1\leq i\leq n]$) denotes disjoint sets of entities we must prove it.

Domain Description Prompt 6. `observe_component_sorts`: The ***domain description prompt***:

- `observe_component_sorts_P`(p)

yields the *component sorts and component sort observer* domain description text according to the following schema – whether or not the actual part p contains any components:

6. `observe_component_sorts_P` schema

Narration:
[s] ... narrative text on component sorts ...
[o] ... narrative text on component observers ...
[i] ... narrative text on component sort recognisers ...
[p] ... narrative text on component sort proof obligations ...
Formalisation:
type
[s] K1, K2, ..., Kn
[s] K = K1| K2 | ... | Kn
[s] KS = K-**set**
value
[o] **components**: P → KS
[i] **is**_K_i: (K_1|K_2|...|K_n) → **Bool** $[1\leq i\leq n]$
Proof Obligation: [Disjointness of Component Sorts]
[p] ∀ k_i:(K_1|K_2|...|K_n) •
[p] ⋀ {**is**_K_i(k_i) ≡ ⋀{∼**is**_K_j(k_j)|j ∈ {1..m}\{i}}|i ∈ {1..m}}

The K_i are all distinct △

Example 39. Container Components: We continue Example 19 on Page 15.

38 When we apply obs_component_sorts_C to any container c:C we obtain

a a type clause stating the sorts of the various components, ck:CK, of a container,
b a union type clause over these component sorts, and
c the component observer function signature.

type
38.a CK1, CK2, ..., CKn
38.b CKS = (CK1|CK2|...|CKn)-**set**
value
38.c **obs_comp**_CKS: C → CKS □

We have presented one way of tackling the issue of describing components. There are other ways. We leave those 'other ways' to the reader. We are not going to suggest techniques and tools for analysing, let alone ascribing qualities to components. We suggest that conventional abstract modeling techniques and tools be applied.

3.6. Materials

We refer to Sect. 3.1.1 on Page 10 for a first coverage of the concept of materials. Continuous endurants (i.e., **material**s) are entities, m, which satisfy:

- is_material(m) ≡ is_endurant(m)∧is_continuous(m)

Example 40. Parts and Materials: We observe materials as associated with atomic parts: Thus liquid or gaseous materials are observed in pipeline units □

We shall in this paper not cover the case of parts being immersed in materials[24]. We assume, without loss of generality, that only atomic parts may contain materials. Let p:P be some atomic part.

Analysis Prompt 16. has_materials: The ***domain analysis prompt***:

- has_materials(p)

yields **true** if the atomic part p:P potentially may contain materials otherwise false ◊

Let us assume that parts p:P embody materials of sorts $\{M_1,M_2,\ldots,M_n\}$. Since we cannot automatically guarantee that our domain descriptions secure that each M_i ($[1 \leq i \leq n]$) denotes disjoint sets of entities we must prove it.

Domain Description Prompt 7. observe_material_sorts_P: The ***domain description prompt***:

- observe_material_sorts_P(e)

yields the *material sorts and material sort observers* domain description text according to the following schema whether or not part p actually contains materials:

7. observe_material_sorts_P schema

Narration:
[s] ... narrative text on material sorts ...
[o] ... narrative text on material sort observers ...
[i] ... narrative text on material sort recognisers ...
[p] ... narrative text on material sort proof obligations ...
Formalisation:
type
[s] M1, M2, ..., Mn
[s] M = M1 | M2 | ... | Mn

[24] Most such cases have the material play a minor, an abstract rôle with respect to the immersed parts. That is, we presently leave it to hydro- and aerodynamics to domain analyse those cases.

[s] MS = M-**set**
value
[o] **obs_mat**_M_i: P → M [$1 \leq i \leq n$]
[o] **materials**: P → MS
[i] **is**_M_i: M → **Bool** [$1 \leq i \leq n$]
proof obligation [Disjointness of Material Sorts]
[p] $\forall\, m_i$:M • $\bigwedge$ {**is**_$M_i(m_i) \equiv \bigwedge\{\sim$**is**_$M_j(m_j)|j \in \{1..m\} \setminus \{i\}\}|i \in \{1..m\}\}$

The M_i are all distinct △

Example 41. Pipeline Material: We continue Example 24 on Page 15 and Example 29 on Page 19.

39 When we apply `obs_material_sorts_U` to any unit u:U we obtain

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

type
39.a LoG
value
39.b **obs_mat**_LoG: U → LoG

`has_materials`(u) is a prerequisite for **obs_mat**_LoG(u) □

3.6.1. ***Materials-related Part Attributes***

It seems that the "interplay" between parts and materials is an area where domain analysis in the sense of this paper is relevant.

Example 42. Pipeline Material Flow: We continue Examples 24, 29 and 41. Let us postulate a[n attribute] sort Flow. We now wish to examine the flow of liquid (or gaseous) material in pipeline units. We use two types

40 . **type** F, L.

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

41 current flow of material into a unit input connector,
42 maximum flow of material into a unit input connector while maintaining laminar flow,
43 current flow of material out of a unit output connector,
44 maximum flow of material out of a unit output connector while maintaining laminar flow,
45 current leak of material at a unit input connector,
46 maximum guaranteed leak of material at a unit input connector,
47 current leak of material at a unit input connector,
48 maximum guaranteed leak of material at a unit input connector,
49 current leak of material from "within" a unit, and
50 maximum guaranteed leak of material from "within" a unit.

type
40. F, L
value
41. **attr**_cur_iF: U → UI → F
42. **attr**_max_iF: U → UI → F
43. **attr**_cur_oF: U → UI → F
44. **attr**_max_oF: U → UI → F
45. **attr**_cur_iL: U → UI → L
46. **attr**_max_iL: U → UI → L
47. **attr**_cur_oL: U → UI → L
48. **attr**_max_oL: U → UI → L
49. **attr**_cur_L: U → L
50. **attr**_max_L: U → L

The maximum flow attributes are static attributes and are typically provided by the manufacturer as indicators of flows

below which laminar flow can be expected. The current flow attributes may be considered either reactive or biddable attributes □

3.6.2. ***Laws of Material Flows and Leaks***

It may be difficult or costly, or both, to ascertain flows and leaks in materials-based domains. But one can certainly speak of these concepts. This casts new light on *domain modeling*. That is in contrast to incorporating such notions of flows and leaks in *requirements modeling* where one has to show implement-ability. Modeling flows and leaks is important to the modeling of materials-based domains.

Example 43. Pipelines: Intra Unit Flow and Leak Law:

51 For every unit of a pipeline system, except the well and the sink units, the following law apply.
52 The flows into a unit equal
  a the leak at the inputs
  b plus the leak within the unit
  c plus the flows out of the unit
  d plus the leaks at the outputs.

axiom [Well−formedness of Pipeline Systems, PLS (1)]
51. ∀ pls:PLS,b:B\We\Si,u:U •
51. b ∈ **obs_part**_Bs(pls)∧u=**obs_part**_U(b)⇒
51. **let** (iuis,ouis) = **obs_mereo**_U(u) **in**
52. sum_cur_iF(u)(iuis) =
52.a. sum_cur_iL(u)(iuis)
52.b. ⊕ **attr**_cur_L(u)
52.c. ⊕ sum_cur_oF(u)(ouis)
52.d. ⊕ sum_cur_oL(u)(ouis)
51. **end**

53 The sum_cur_iF (cf. Item 52) sums current input flows over all input connectors.
54 The sum_cur_iL (cf. Item 52.a) sums current input leaks over all input connectors.
55 The sum_cur_oF (cf. Item 52.c) sums current output flows over all output connectors.
56 The sum_cur_oL (cf. Item 52.d) sums current output leaks over all output connectors.

53. sum_cur_iF: U → UI-**set** →F
53. sum_cur_iF(u)(iuis) ≡ ⊕ {**attr**_cur_iF(u)(ui)|ui:UI•ui ∈ iuis}
54. sum_cur_iL: U → UI-**set** → L
54. sum_cur_iL(u)(iuis) ≡ ⊕ {**attr**_cur_iL(u)(ui)|ui:UI•ui ∈ iuis}
55. sum_cur_oF: U → UI-**set** → F
55. sum_cur_oF(u)(ouis) ≡ ⊕ {**attr**_cur_iF(u)(ui)|ui:UI•ui ∈ ouis}
56. sum_cur_oL: U → UI-**set** → L
56. sum_cur_oL(u)(ouis) ≡ ⊕ {**attr**_cur_iL(u)(ui)|ui:UI•ui ∈ ouis}
⊕: (F|L) × (F|L) → F

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

Example 44. Pipelines: Inter Unit Flow and Leak Law:

57 For every pair of connected units of a pipeline system the following law apply:
  a the flow out of a unit directed at another unit minus the leak at that output connector
  b equals the flow into that other unit at the connector from the given unit plus the leak at that connector.

axiom [Well−formedness of Pipeline Systems, PLS (2)]
57. ∀ pls:PLS,b,b′:B,u,u′:U•
57. {b,b′}⊆**obs_part**_Bs(pls)∧b≠b′∧u′=**obs_part**_U(b′)

57. $\wedge$ **let** (iuis,ouis)=**obs_mereo_**U(u),(iuis′,ouis′)=**obs_mereo_**U(u′),
57. ui=**uid_**U(u),ui′=**uid_**U(u′) **in**
57. ui $\in$ iuis $\wedge$ ui′ $\in$ ouis′ $\Rightarrow$
57.a. **attr_**cur_oF(u′)(ui′) − **attr_**leak_oF(u′)(ui′)
57.b. = **attr_**cur_iF(u)(ui) + **attr_**leak_iF(u)(ui)
57. **end**
57. **comment:** b′ precedes b □

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

3.7. "No Junk, No Confusion"

Domain descriptions are, as we have already shown, formulated, both informally and formally, by means of abstract types, that is, by sorts for which no concrete models are usually given. Sorts are made to denote possibly empty, possibly infinite, rarely singleton, sets of entities on the basis of the qualities defined for these sorts, whether external or internal. By **junk** we shall understand that the domain description unintentionally denotes undesired entities. By **confusion** we shall understand that the domain description unintentionally have two or more identifications of the same entity or type. The question is *can we formulate a [formal] domain description such that it does not denote junk or confusion* ? The short answer to this is no ! So, since one naturally wishes "no junk, no confusion" what does one do ? The answer to that is *one proceeds with great care !* To avoid junk we have stated a number of sort well-formedness axioms, for example:[25]

- Page 17 for *wf* links and hubs,
- Page 18 for *wf* road net mereologies,
- Page 18 for *wf* pipeline mereologies,
- Page 21 for *wf* hub states,
- Page 27 for *wf* pipeline systems,
- Page 27 for *wf* pipeline systems,

To avoid confusion we have stated a number of proof obligations:

- Page 12 for *Disjointness of Part Sorts*,
- Page 21 for *Disjointness of Attribute Types* and
- Page 26 for *Disjointness of Material Sorts*.

3.8. Discussion of Endurants

In Sect. 3.1.5 on Page 11 a "depth-first" search for part sorts was hinted at. It essentially expressed that we discover domains epistemologically[26] but understand them ontologically.[27] The Danish philosopher Søren Kirkegaard (1813–1855) expressed it this way: *Life is lived forwards, but is understood backwards.* The presentation of the of the ***domain analysis prompt***s and the ***domain description prompt***s results in domain descriptions which are ontological. The "depth-first" search recognizes the epistemological nature of bringing about understanding. This "depth-first" search that ends with the analysis of atomic part sorts can be guided, i.e., hastened (shortened), by postulating composite sorts that "correspond" to vernacular nouns: everyday nouns that stand for classes of endurants.

We could have chosen our ***domain analysis prompt***s and ***domain description prompt***s to reflect a "bottom-up" epistemology, one that reflected how we composed composite understandings from initially atomic parts. We leave such a collection of ***domain analysis prompt***s and ***domain description prompt***s to the reader.

[25] Let *wf* abbreviate *well-formed*.

[26] **Epistemology**: the theory of knowledge, especially with regard to its methods, validity, and scope. Epistemology is the investigation of what distinguishes justified belief from opinion.

[27] **Ontology**: the branch of metaphysics dealing with the nature of being.

4. Perdurants

We shall not present a set of ***domain analysis prompt***s and a set of ***domain description prompt***s leading to description language, i.e., RSL texts describing perdurant entities. The reason for giving this albeit cursory overview of perdurants is that we can justify our detailed study of endurants, their part and sub parts, their unique identifiers, mereology and attributes. This justification is manifested (i) in expressing the types of signatures, (ii) in basing behaviours on parts, (iii) in basing the for need for CSP-oriented inter-behaviour communications on shared part attributes, (iv) in indexing behaviours as are parts, i.e., on unique identifiers, and (v) in directing inter-behaviour communications across channel arrays indexed as per the mereology of the part behaviours. These are all notions related to endurants and are now justified by their use in describing perdurants. Perdurants can perhaps best be explained in terms of a notion of state and a notion of time. We shall, in this paper, not detail notions of time, but refer to [Hei62, Far90, Bli90, van91].

4.1. States

Definition 11. State: By a **state** we shall understand any collection of **parts** each of which has at least one **dynamic attribute** or **has_components** or **has_material**s ⊙

Example 45. States: A road hub can be a state, cf. Hub State, HΣ, Example 33 on Page 21. A road net can be a state – since its hubs can be. Container stowage areas, CSA, Example 19 on Page 15, of container vessels and container terminal ports can be states as containers can be removed from and put on top of container stacks. Pipeline pipes can be states as they potentially carry material. Conveyor belts can be states as they may carry components □

4.2. Actions, Events and Behaviours

To us perdurants are further, pragmatically, analysed into actions, events, and behaviours. We shall define these terms below. Common to all of them is that they potentially change a state. Actions and events are here considered atomic perdurants. For behaviours we distinguish between discrete and continuous behaviours.

4.2.1. *Time Considerations*

We shall, without loss of generality, assume that actions and events are atomic and that behaviours are composite. Atomic perdurants may "occur" during some time interval, but we omit consideration of and concern for what actually goes on during such an interval. Composite perdurants can be analysed into "constituent" actions, events and "sub-behaviours". We shall also omit consideration of temporal properties of behaviours. Instead we shall refer to two seminal monographs: Specifying Systems [Lam02, Leslie Lamport] and Duration Calculus: A Formal Approach to Real-Time Systems [ZH04, Zhou ChaoChen and Michael Reichhardt Hansen] (and [Bjø06, Chapter 15]). For a seminal book on "time in computing" we refer to the eclectic [FMMR12, Mandrioli et al., 2012]. And for seminal book on time at the epistemology level we refer to [van91, J. van Benthem, 1991].

4.2.2. *Actors*

Definition 12. Actor: By an **actor** we shall understand something that is capable of initiating and/or carrying out actions, events or behaviours ⊙

We shall, in principle, associate an actor with each part. These actors will be described as behaviours. These behaviours evolve around a state. The state is the set of qualities, in particular the dynamic attributes, of the associated parts and/or any possible components or materials of the parts.

Example 46. Actors: We refer to the road transport and the pipeline systems examples of earlier. The fleet, each vehicle and the road management of the *Transportation System* of Examples 17 on Page 13 and 37 on Page 23 can be considered actors; so can the net and its links and hubs. The pipeline monitor and each pipeline unit of the *Pipeline System*, Example 24 on Page 15 and Examples 24 on Page 15 and 29 on Page 19 will be considered actors □

4.2.3. *Parts, Attributes and Behaviours*

Example 46 focused on what shall soon become a major relation within domains: that of parts being also considered actors, or more specifically, being also considered to be behaviours.

Example 47. Parts, Attributes and Behaviours: Consider the term 'train'[28]. It has several possible "meanings". (i) the train as a part, viz., as standing on a train station platform; (ii) the train as listed in a timetable (an attribute of a transport system part), (iii) the train as a behaviour: speeding down the rail track □

4.3. Discrete Actions

Definition 13. Discrete Action: By a **discrete action** [WS12, Wilson and Shpall] we shall understand a foreseeable thing which deliberately potentially changes a well-formed state, in one step, usually into another, still well-formed state, and for which an actor can be made responsible ⊙

An action is what happens when a function invocation changes, or potentially changes a state.

Example 48. Road Net Actions: Examples of *Road Net* actions initiated by the net actor are: insertion of hubs, insertion of links, removal of hubs, removal of links, setting of hub states. Examples of *Traffic System* actions initiated by vehicle actors are: moving a vehicle along a link, stopping a vehicle, starting a vehicle, moving a vehicle from a link to a hub and moving a vehicle from a hub to a link □

4.4. Discrete Events

Definition 14. Event: By an **event** we shall understand some unforeseen thing, that is, some 'not-planned-for' "action", one which surreptitiously, non-deterministically changes a well-formed state into another, but usually not a well-formed state, and for which no particular domain actor can be made responsible ⊙

Events can be characterised by a pair of (before and after) states, a predicate over these and, optionally, a *time* or *time interval*. The notion of event continues to puzzle philosophers [Dre67, Qui79, Mel80, Dav80, Hac82, Bad05, Kim93, CV96, Pi99, CV10]. We note, in particular, [Dav80, Bad05, Kim93].

Example 49. Road Net and Road Traffic Events: Some road net events are: "disappearance" of a hub or a link, failure of a hub state to change properly when so requested, and occurrence of a hub state leading traffic into "wrong-way" links. Some road traffic events are: the crashing of one or more vehicles (whatever 'crashing' means), a car moving in the wrong direction of a one-way link, and the clogging of a hub with too many vehicles □

4.5. Discrete Behaviours

Definition 15. Discrete Behaviour: By a **discrete behaviour** we shall understand a set of sequences of potentially interacting sets of discrete actions, events and behaviours ⊙

Example 50. Behaviours: (i) Road Nets: A sequence of hub and link insertions and removals, link disappearances, etc. (ii) Road Traffic: A sequence of movements of vehicles along links, entering, circling and leaving hubs, crashing of vehicles, etc. (iii) Pipelines: A sequence of pipeline pump and valve openings and closings, and failures to do so (events), etc. (iv) Container Vessels and Ports: Concurrent sequences of movements (by cranes) of containers from vessel to port (unloading), with sequences of movements (by cranes) from port to vessel (loading), with dropping of containers by cranes, etcetera □

4.5.1. *Channels and Communication*

Behaviours sometimes synchronise and usually communicate. We use the CSP [Hoa85] notation (adopted by RSL) to introduce and model behaviour communication. Communication is abstracted as the sending (ch ! m) and receipt (ch ?) of messages, m:M, over channels, ch.

type M
channel ch:M

[28] This example is due to Paul Lindgreen, a Danish computer scientist. It dates from the late 1970s.

Communication between (unique identifier) indexed behaviours have their channels modeled as similarly indexed channels:

out: ch[idx]!m
in: ch[idx]?
channel {ch[ide]|ide:IDE}:M

where IDE typically is some type expression over unique identitifer types.

4.5.2. ***Relations Between Attribute Sharing and Channels***

We shall now interpret the syntactic notion of attribute sharing with the semantic notion of channels. This is in line with the above-hinted interpretation of parts with behaviours, and, as we shall soon see, part attributes with behaviour states. Thus, for every pair of parts, p_{ik}:P_i and $p_{j\ell}$:P_j, of distinct sorts, P_i and P_j which share attribute values in A we are going to associate a channel. If there is only one pair of parts, p_{ik}:P_i and $p_{j\ell}$:P_j, of these sorts, then we associate just a simple channel, say $\mathsf{attr\_A\_ch}_{P_i,P_j}$, with the shared attribute.

channel $\mathsf{attr\_A\_ch}_{P_i,P_j}$:A.

If there is only one part, p_i:P_i, but a definite set of parts p_{jk}:P_j, with shared attributes, then we associate a *vector* of channels with the shared attribute. Let $\{p_{j1}, p_{j2}, ..., p_{jn}\}$ be all the parts of the domain sort P_j. Then *uids* : $\{\pi_{p_{j1}}, \pi_{p_{j2}}, ..., \pi_{p_{jn}}\}$ is the set of their unique identifiers. Now a schematic channel array declaration can be suggested:

channel {attr_A_ch[{π_i,π_j}]|π_i=**uid**_P_i(p_i)$\wedge\pi_j \in$ uids}:A.

The above can be extended in two ways: From channel matrices to channel tensors, etc., hence the term channel 'array'. And from simple shared attributes to "embedded sharing".

We say that P and Q enjoy **embedded attribute sharing** when the following is the case: Part sort Phas attribute type A, and part sort Q, different from P, has attribute type B where B is defined in terms of A ⊙For cases where P and Q enjoy embedded attribute sharing the mereology of parts p:P will include **uid**_Q(q) and the mereology of parts q:Q will include **uid**_P(p).

Example 51. Bus System Channels: We extend Examples 17 on Page 13 and 37 on Page 23. We consider the fleet and the vehicles to be behaviours.

58 We assume some transportation system, δ. From that system we observe
59 the fleet and
60 the vehicles.
61 The fleet to vehicle channel array is indexed by the 2-element sets of the unique fleet identifier and the unique vehicle identifiers. We consider bus timetables to be the only message communicated between the fleet and the vehicle behaviours.

```
value
58.   δ:Δ,
59.   f:F = obs_part_F(δ),
60.   vs:V-set = obs_part_Vs(obs_part_VC((obs_part_F(δ))))
      channel
61.   {attr_BT_ch[{uid_F(f),uid_V(v)}]|v:V•v ∈ vs}:BT □
```

4.6. Continuous Behaviours

By a **continuous behaviour** we shall understand a continuous time sequence of state changes. We shall not go into what may cause these state changes.

Example 52. Flow in Pipelines: We refer to Examples 29, 41, 42, 43 and 44. Let us assume that oil is the (only) material of the pipeline units. Let us assume that there is a sufficient volume of oil in the pipeline units leading up to a pump. Let us assume that the pipeline units leading from the pump (especially valves and pumps) are all open for

oil flow. Whether or not that oil is flowing, if the pump is pumping (with a sufficient head[29]) then there will be oil flowing from the pump outlet into adjacent pipeline units □

To describe the flow of material (say in pipelines) requires knowledge about a number of material attributes — not all of which have been covered in the above-mentioned examples. To express flows one resorts to the mathematics of fluid-dynamics using such second order differential equations as first derived by Bernoulli (1700–1782) and Navier–Stokes (1785–1836 and 1819–1903). There is, as yet, no notation that can serve to integrate formal descriptions (like those of `Alloy, B, The B Method,RSL,VDM` or `Z`) with first, let alone second order differential equations. But some progress has been made [LWZ13, ZWZ13] since [WYZ94].

4.7. Attribute Value Access

We refer to paragraph "Access to Attribute Values" in Section 3.4.5 Page 22. We distinguish between four kinds of attributes: the static attributes which are those whose values are fixed, i.e., does not change, the programmables which are those dynamic values are exclusively set by part processes, and the remaining dynamic attributes which here, technically speaking, are seen as separate external processes. The event attributes are those external attributes whose value occur for an instant of time.

4.7.1. *Access to Static Attribute Values*

The **static attributes** can be "copied", **attr**_A(p), and retain their values.

4.7.2. *Access to External Attribute Values*

By the **external attribute**s, to repeat, we shall understand the inert, the autonomous and the biddable attributes ⊙

62 Let ξA be the set of names, ηA, of all external attributes.

63 Each external attribute, A, is seen as an individual behaviour, each "accessible" by means of unique channel, attr_A_ch.

64 External attribute values are then accessed by the input, attr_A_ch ?.

65 The **type** of attr_A_ch is considered to be **Unit**$\xrightarrow{\sim}$A, abbreviated $\mathbb{U}$ A.

```
62. value
62.     ξA: {ηA|A is any external attribute name}
63. channel
63.     {attr_A_ch|ηA ∈ ξA}
64. value
64.     attr_A_ch ?
64. type
64.     attr_A_ch: Unit→~A [abbrev.:𝕌 A]
```

We shall omit the η prefix in actual descriptions. The choice of representing external attribute values as `CSP` processes[30] is a technical one.

4.7.3. *Access to Reactive and Programmable Attribute Values*

The **reactive attributes** and the **programmable attributes** are treated as function arguments. This is a technical choice. It is motivated as follows. We find that these values are a function of other part attribute values, including at least one programmable attribute value, and that the values are set (i.e., updated) by part behaviours. That is, to each part, whether atomic or composite, we associate a behaviour. That behaviour is (to be) described as we describe functions. These functions (normally) *"go on forever"*. Therefore these functions are described basically by a "tail" recursive definition:

[29] The **pump head** is the linear vertical measurement of the maximum height a specific pump can deliver a liquid to the pump outlet.

[30] — not to be confused with domain behaviours

value f: Arg → Arg; f(a) ≡ (... **let** a′ = $\mathscr{F}$(...)(a) **in** f(a′) **end**)

where $\mathscr{F}$ is some expression based on values defined within the function definition body of f and on f's "input" argument a, and where a can be seen as a reactive attribute or a programmable attribute.

4.7.4. ***Access to Event Values***

We refer to Sect. 3.4.6 on Page 23. Event values reflect a stage change in a part behaviour. We therefore model events as messages communicated over a channel, attr_A_ch, that is, attr_A_ch ! a, where A is the event attribute, i.e., message type. Thus fulfillment of attr_A_ch ? expresses both that the event has taken place and its value, if relevant. Example 57 on Page 38 illustrates the concept of event attributes and event values.

4.8. Perdurant Signatures and Definitions

We shall treat perdurants as function invocations. In our cursory overview of perdurants we shall focus on one perdurant quality: function signatures.

Definition 16. Function Signature: By a **function signature** we shall understand a function name and a function type expression ⊙

Definition 17. Function Type Expression: By a **function type expression** we shall understand a pair of type expressions. separated by a function type constructor either → (total function) or $\xrightarrow{\sim}$ (partial function) ⊙

The type expressions are part sort or type, or material sort or type, or component sort or type, or attribute type names, but may, occasionally be expressions over respective type names involving **-set**, ×, *, $\xrightarrow[m]{}$ and | type constructors.

4.9. Action Signatures and Definitions

Actors usually provide their initiated actions with arguments, say of type VAL. Hence the schematic function (action) signature and schematic definition:

action: VAL → Σ $\xrightarrow{\sim}$ Σ
action(v)(σ) **as** σ′
  pre: $\mathscr{P}$(v,σ)
  post: $\mathscr{Q}$(v,σ,σ′)

expresses that a selection of the domain, as provided by the Σ type expression, is acted upon and possibly changed. The partial function type operator $\xrightarrow{\sim}$ shall indicate that action(v)(σ) may not be defined for the argument, i.e., initial state σ and/or the argument v:VAL, hence the precondition $\mathscr{P}$(v,σ). The post condition $\mathscr{Q}$(v,σ,σ′) characterises the "after" state, σ′:Σ, with respect to the "before" state, σ:Σ, and possible arguments (v:VAL).

Example 53. Insert Hub Action Formalisation: We formalise aspects of the above-mentioned hub action:

66 Insertion of a hub requires
67 that no hub exists in the net with the unique identifier of the inserted hub,
68 and then results in an updated net with that hub.

value
66. insert_H: H → N $\xrightarrow{\sim}$ N
66. insert_H(h)(n) **as** n′
67. **pre**: ∼∃ h′:H•h′ ∈ **obs_part_**Hs(**obs_part_**HS(n))•**uid_**H(h)=**uid_**H(h′)
68. **post**: **obs_part_**Hs(**obs_part_**HS(n′))=**obs_part_**Hs(**obs_part_**HS(n))∪{h} □

Which could be the argument values, v:VAL, of actions ? Well, there can basically be only the following kinds of argument values: parts, components and materials, respectively unique part identifiers, mereologies and attribute values. It basically has to be so since there are no other kinds of values in domains. There can be exceptions to the above (Booleans, natural numbers), but they are rare !

Perdurant (action) analysis thus proceeds as follows: identifying relevant actions, assigning names to these, delineating the "smallest" relevant state[31], ascribing signatures to action functions, and determining action pre-conditions and action post-conditions. Of these, ascribing signatures is the most crucial: In the process of determining the action signature one oftentimes discovers that part or component or material attributes have been left ("so far") "undiscovered".

Example 53 showed example of a signature with only a part argument. Example 54 shows examples of signatures whose arguments are parts and unique identifiers, or parts, unique identifiers and attribute values.

Example 54. Some Function Signatures: Inserting a link between two identified hubs in a net:

value insert_L: L × (HI × HI) → N $\xrightarrow{\sim}$ N

Removing a hub and removing a link:

value remove_H: HI → N $\xrightarrow{\sim}$ N
remove_L: LI → N $\xrightarrow{\sim}$ N

Changing a hub state.

value change_HΣ: HI × HΣ → N $\xrightarrow{\sim}$ N □

4.10. Event Signatures and Definitions

Events are usually characterised by the absence of known actors and the absence of explicit "external" arguments. Hence the schematic function (event) signature:

value
event: Σ × Σ $\xrightarrow{\sim}$ **Bool**
event(σ,σ′) **as** tf
pre: $P(\sigma)$
post: tf = $Q(\sigma,\sigma')$

The event signature expresses that a selection of the domain as provided by the Σ type expression is "acted" upon, by unknown actors, and possibly changed. The partial function type operator $\xrightarrow{\sim}$ shall indicate that event(σ,σ') may not be defined for some states σ. The resulting state may, or may not, satisfy axioms and well-formedness conditions over Σ — as expressed by the post condition $Q(\sigma,\sigma')$. Events may thus cause well-formedness of states to fail. Subsequent actions, once actors discover such "disturbing events", are therefore expected to remedy that situation, that is, to restore well-formedness. We shall not illustrate this point.

Example 55. Link Disappearence Formalisation: We formalise aspects of the above-mentioned link disappearance event:

69 The result net is not well-formed.
70 For a link to disappear there must be at least one link in the net;
71 and such a link may disappear such that
72 it together with the resulting net makes up for the "original" net.

value
69. link_diss_event: N × N′ $\xrightarrow{\sim}$ **Bool**
69. link_diss_event(n,n′) **as** tf
70. **pre**: **obs_part**_Ls(**obs_part**_LS(n))≠{}
71. **post**: tf = ∃ l:L•l ∈ **obs_part**_Ls(**obs_part**_LS(n)) ⇒
72. l ∉ **obs_part**_Ls(**obs_part**_LS(n′))
72. ∧ n′ ∪ {l} = **obs_part**_Ls(**obs_part**_LS(n)) □

[31] By "smallest" we mean: containing the fewest number of parts. Experience shows that the domain analyser cum describer should strive for identifying the smallest state.

4.11. Discrete Behaviour Signatures and Definitions

4.11.1. ***Behaviour Signatures***

We shall only cover behaviour signatures when expressed in RSL/CSP [GHH+92]. The behaviour functions are now called processes. That a behaviour function is a never-ending function, i.e., a process, is "revealed" in the function signature by the "trailing" **Unit**:

behaviour: ... → ... **Unit**

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

behaviour: **Unit** → ...

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

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

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

behaviour: ... → **out** ch ...

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

behaviour: ARG → ...

where ARG can be any type expression:

T, T→T, T→T→T, etcetera

where T is any type expression.

Part Behaviours: We can, without loss of generality, associate with each part a behaviour; parts which share attributes (and are therefore referred to in some parts' mereology), can communicate (their "sharing") via channels. The process evolves around a state, or, rather, a set of values: its unique identity, $\pi : \Pi$,[32] its possibly changing mereology, mt:MT[33], the possible components and materials of the part, and the constant, the external and the programmable attributes of the part. A behaviour signature is therefore:

behaviour: π:Π × me:MT × sa:SA → rpa:RPA → **in** *ichns*(ea:EA) **in**,**out** *iochs*(me) **Unit**

where (i) π:Π is the unique identifier of part p, i.e., π=**uid_**P(p), (ii) me:ME is the mereology of part p, me = **obs_mereo_**P(p), (iii) sa:SA lists the static attribute values of the part, (iv) rpa:RPA lists the reactive and programmable attribute values of the part, (v) *ichns*(ea:EA) refer to the external attribute *i*nput *chan*nel*s*, and where (vi) *iochs*(me) are the input/output channels serving the attrributes shared between the part p and the parts designated in its mereology me, cf. Sect. 4.7.2.

We focus, for a little while, on the expression of sa:SA, ea:EA and rpa:RPA, that is, on the concrete types of SA, EA and RPA. $\mathscr{S}_{\mathscr{A}}(p)$: sa:SA lists the static value types, $(svT_1,...,svT_s)$, where s is the number of *s*tatic attributes of parts p:P. $\mathscr{E}_{\mathscr{A}}(p)$: ea:EA lists the external attribute value channels, ($\mathbb{U}A_1$, $\mathbb{U}$A$_2$, ..., $\mathbb{U}$A$_x$)[34], where x is the number, 0 or more, of external attributes, A$_1$, A$_2$, ..., A$_x$, of parts p:P. $\mathscr{R}\mathscr{P}_{\mathscr{A}}(p)$: rpa:RPA lists the reactive and programmable value expression types: $(pvT_1, pvT_2, ..., pvT_q)$ where q is the number of reactive and programmable attributes of parts p:P. A **reactive attribute value expression** is an expression involving one or more attribute value expressions of the type of the reactive attribute ⊙ A **programmable attribute value expression** is a simple expression of the type of the programmable value ⊙

[32] Unique identifiers of parts are like static attributes and hence (not really) contributing to the part state.

[33] For MT see footnote 17 on Page 18.

[34] See paragraph *Access to External Attribute Values* on Page 32.

4.11.2. ***Behaviour Definitions***

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

Process Schema I: Abstract `is_composite(p)`

```
value
    compile_process: P → RSL-Text
    compile_process(p) ≡
            M_cP_CORE(uid_P(p),obs_mereo_P(p),S_A(p))(R P_A(p))
          ‖ compile_process(obs_part_P1(p))
          ‖ compile_process(obs_part_P2(p))
          ‖ ...
          ‖ compile_process(obs_part_Pn(p))
```

The text macros: $\mathscr{S}_{\mathscr{A}}$ and $\mathscr{R}\mathscr{P}_{\mathscr{A}}$ were informally explained above. Part sorts P_1, P_2, ..., P_n are obtained from the `observe_part_sorts` prompt, Page 12.

Let P be a composite sort defined in terms of the concrete type Q-**set**. The process definition compiled from p:P, is composed from a process, $\mathscr{M}_{cP_{\text{CORE}}}$, relying on and handling the unique identifier, mereology and attributes of process p as defined by P operating in parallel with processes q:**obs_part**_Qs(p). The domain description "compilation" schematic below "formalises" the above.

Process Schema II: Concrete `is_composite(p)`

```
type
    Qs = Q-set
value
    qs:Q-set = obs_part_Qs(p)
    compile_process: P → RSL-Text
    compile_process(p) ≡
            M_cP_CORE(uid_P(p),obs_mereo_P(p),S_A(p))(R P_A(p))
          ‖ ‖{compile_process(q)|q:Q•q ∈ qs}
```

Process Schema III: `is_atomic(p)`

```
value
    compile_process: P → RSL-Text
    compile_process(p) ≡
            M_aP_CORE(uid_P(p),obs_mereo_P(p),S_A(p))(R P_A(p))
```

Example 56. Bus Timetable Coordination: We refer to Examples 17 on Page 13, 18 on Page 14, 37 on Page 23 and 51 on Page 31.

73 δ is the transportation system; f is the fleet part of that system; vs is the set of vehicles of the fleet; bt is the shared bus timetable of the fleet and the vehicles.

74 The fleet process is compiled as per Process Schema II (Page 36).

The definitions of the fleet and vehicle processes are simplified so as to emphasize the master/slave, programmable/inert relations between these processes.

```
type
    Δ, F, VS       [Example 17 on Page 13]
    V, Vs=V-set  [Example 18 on Page 14]
    FI, VI, BT     [Example 37 on Page 23]
value
73.  δ:Δ,
73.  f:F = obs_part_F(δ),
73.  vs:V-set = obs_part_Vs(obs_part_VS(f))
axiom
73.  ∀ v:V•v ∈ vs ⇒ □ attr_BT(f) = attr_BT(v) [Example 37 on Page 23]
value
74.  fleet: fi:FI → BT → out attr_BT_ch Unit
74.  fleet(fi)(bt) ≡ M_F(fi)(bt) || || {vehicle(uid_V(v))|v:V•v ∈ vs}

74.  vehicle: vi:VI → in attr_BT_ch Unit
74.  vehicle(vi) ≡ M_V(fi,attr_BT_ch)... ; vehicle(vi)
```

Fleet and vehicle processes $\mathscr{M}_F$ and $\mathscr{M}_V$ are both "never-ending" processes:

```
value
    M_F: fi:FI → BT → out attr_BT_ch Unit
    M_F(fi)(bt) ≡ let bt' = F(fi,bt) in M_F(fi)(bt') end
```

Function $\mathscr{F}$ is a simple action. The expression of actual synchronisation and communication between the fleet and the vehicle processes is contained in $\mathscr{F}$.

```
value
    F: fi:FI → bt:BT → out attr_BT_ch BT
    F(fi)(bt) ≡ (let bt' = f(bt)(...) in bt' end) ⌈⌉ (attr_BT_ch ! bt ; bt)
    f: BT → ... → BT
```

The auxiliary function f "embodies" the programmable nature of the timetable attribute □

Please note a master part's programmable attribute can be reflected in two ways: as a programmable attribute and as an output channel to the behaviour specification of slave parts. This is illustrated, in Example 56 where the fleet behaviour has programmable attribute BT and output channel attr_BT_ch to vehicle behaviours.

Process Schema IV: Core Process (I)

The core processes can be understood as never ending, "tail recursively defined" processes:

$\mathscr{M}_{cP_{\mathrm{CORE}}}$: π:Π×me:MT×sa:SA→pa:PA→**in in** *ichns*(ea:EA) **in**,**out** *iochs*(me) **Unit**
$\mathscr{M}_{cP_{\mathrm{CORE}}}(\pi,\mathrm{me},\mathrm{sa})(\mathrm{pa}) \equiv$ **let** $(\mathrm{me}',\mathrm{pa}') = \mathscr{F}(\pi,\mathrm{me},\mathrm{sa})(\mathrm{pa})$ **in** $\mathscr{M}_{cP_{\mathrm{CORE}}}(\pi,\mathrm{me}',\mathrm{sa})(\mathrm{pa}')$ **end**

$\mathscr{F}$: π:Π×me:MT×sa:SA → PA → **in** *ichns*(ea:EA) **in**,**out** *iochs*(me) → MT×PA

$\mathscr{F}$ potentially communicates with all those part processes (of the whole domain) with which it shares attributes, that is, has connectors. $\mathscr{F}$ is expected to contain input/output clauses referencing the channels of the in ... out ... part of their signatures. These clauses enable the sharing of attributes. $\mathscr{F}$ also contains expressions, attr_A_ch ?, to external attributes.

We present a rough sketch of $\mathscr{F}$. The $\mathscr{F}$ action non-deterministically internal choice chooses between

- either [1,2,3,4]
  - ∞ [1] accepting input from
  - ∞ [4] a suitable ("offering") part process,
  - ∞ [2] then optionally offering a reply to that other process, and
  - ∞ [3] finally delivering an updated state;

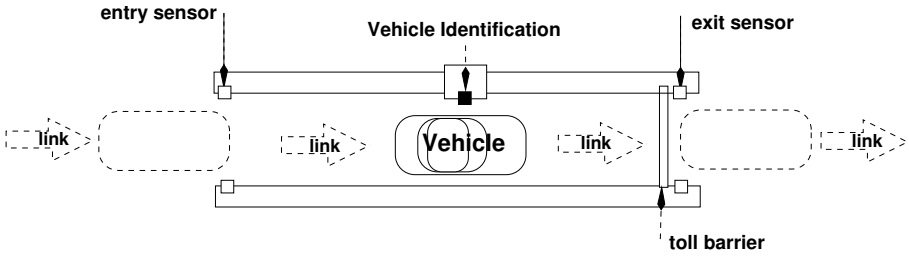

Fig. 3. A tollgate

- or [5,6,7,8]
  - ⋄ [5] finding a suitable "order" (val)
  - ⋄ [8] to a suitable ("inquiring") behaviour (π'),
  - ⋄ [6] offering that value (on channel ch[π']
  - ⋄ [7] and then delivering an updated state;
- or [9] doing own work resulting in an updated state.

Process Schema V: Core Process (II)

```
value
    𝒻: π:Π × me:MT × sa:SA → pa:PA → in ichns(ea:EA) in,out iochs(me)  MT × PA
    𝒻(π,me,sa,ea)(pa) ≡
[1]   [] { let val = ch[π'] ? in
[2]        ( ch[π'] ! in_reply(val)(me,sa,ea)(pa) ⌈⌉ skip ) ;
[3]        in_update(val)(me,sa,ea)(pa) end
[4]      | π': Π • π' ∈ ℰ(π,me)}
[5]   ⌈⌉ [] { let val = await_reply(π')(me,sa,ea)(pa) in
[6]        ch[π'] ! val ;
[7]        out_update(val)(me,sa,ea)(pa) end
[8]      | π': Π • π' ∈ ℰ(π,me)}
[9]   ⌈⌉   (me,own_work(sa,ea)(pa))

    channels ch[π'] are defined in in ichns(ea:EA) in,out iochs(me)

    in_reply:  VAL → SA×EA → PA → VAL
    in_update: VAL → MT×SA×EA → PA → MT×PA
    await_reply: Π → MT×SA×EA → PA → VAL
    out_update: VAL → MT×SA×EA → PA → MT×PA
    own_work: SA×EA → PA → PA
```

We leave these auxiliary functions and VAL undefined.

Example 57. Tollgates: Part and Behaviour: Our example is disconnected from that of a larger example of road pricing. Figure 3 abstracts essential features of a tollgate.

75 A tollgate is a composite part. It consists of

76 an entry sensor (ES), a vehicle identity sensor (IS), a barrier (B), and an exit sensor (XS).

77 The sensors function as follows:

a When a vehicle first starts passing the entry sensor then it sends an appropriate (event) message to the tollgate.

b When a vehicle's identity is recognised by the identity sensor then it sends an appropriate (event) message to the tollgate.

c When a vehicle ends passing the exit sensor then it sends an appropriate (event) message to the tollgate.

78 We therefore model these sensors as shared dynamic event attributes.

a For the sensors these are master attributes.
b For the tollgate they are slave attributes.
c In all three cases they are therefore modeled as channels.

79 A vehicle passing the gate

a first "triggers" the entry sensor ("Enter"),
b which results in the lowering ("Lower") of the barrier,
c then the vehicle identity sensor ("vi:VI"),
d with the tollgate "mysteriously"[35] handling that identity, and, simultaneously
e raising ("Raise") the barrier, and
f finally the output sensor ("Exit") is triggered as the vehicle leaves the tollgate,
g and the barrier is lowered.

80 whereupon the tollgate resumes being a tollgate.
81 TGI is the type unique tollgate identifiers.

Instead of one tollgate we may think of a number of tollgates: Each with their unique identifier — together with a finite set of two or more such identifiers, tgis:TGI-**set**.

```
type
75.    TG
76.    ES, IS, B, XS
79.a.  En = {|"Enter"|}
79.b.  Ba = {|"Lower","Raise"|}
79.c.  Id = VI
79.e.  Ex = {|"Exit"|}
81.    TGI
value
76.    obs_part_ES: TG → ES
76.    obs_part_IS: TG → IS
76.    obs_part_B: TG → B
76.    obs_part_XS: TG → XS
81.    uid_TGI: TG → TGI
79.a.  attr_Enter: TG|ES → {|"Enter"|} event
79.c.  attr_Identity: TG|IS → VI event
79.e.  attr_Exit: TG|XS → {|"Exit"|} event
channel
79.    {attr_En_ch[tgi]|tgi:TGI•tgi∈tgis} En
79.    {attr_Id_ch[tgi]|tgi:TGI•tgi∈tgis} VI
79.    {attr_Ba_ch[tgi]|tgi:TGI•tgi∈tgis} BA
79.    {attr_Ex_ch[tgi]|tgi:TGI•tgi∈tgis} Ex
value
79.  gate: tgi:TGI →
79.    in attr_En_ch[tgi],attr_Id_ch[tgi],attr_Ex_ch[tgi]
79.    out attr_Ba_ch[tgi] Unit
79.  gate(tgi,chs) ≡
79.a.    attr_En_ch[tgi] ? ;
79.b.    attr_Ba_ch[tgi] ! "Lower" ;
79.c.    let vi = attr_Id_ch[tgi] ? in
79.d.    ( handle(vi) ||
79.e.    attr_Ba_ch[tgi] ! "Raise" ) ;
79.f.    attr_Ex_ch[tgi] ? ;
79.g.    attr_Ba[tgi] ! "Lower" ;
80.      gate(tgi,chs) end
```

The enter, identity and exit events are slave attributes of the tollgate part and master attributes of respectively the entry sensor, the vehicle identity sensor, and the exit sensor sub-parts. We do not define the behaviours of these sub-parts. We only assume that they each issue appropriate attr_A_ch ! **out**put messages where A is either Enter, Identity, or Exit and where event values en:Enter and ex:Exit are ignored □

4.12. Concurrency: Communication and Synchronisation

Process Schemas I, II and IV (Pages 36, 36 and 37), reveal that two or more parts, which temporally coexist (i.e., at the same time), imply a notion of concurrency. Process Schema IV, through the RSL/CSP language expressions ch ! v and ch ?, indicates the notions of communication and synchronisation. Other than this we shall not cover these crucial notion related to parallelism.

[35] ... that is, passes vi on to the road pricing monitor — where we omit showing relevant channels.

4.13. Summary and Discussion of Perdurants

The most significant contribution of Sect. 4 has been to show that for every domain description there exists a normal form behaviour — here expressed in terms of a CSP process expression.

4.13.1. ***Summary***

We have proposed to analyse perdurant entities into actions, events and behaviours — all based on notions of state and time. We have suggested modeling and abstracting these notions in terms of functions with signatures and pre-/post-conditions. We have shown how to model behaviours in terms of CSP (communicating sequential processes). It is in modeling function signatures and behaviours that we justify the endurant entity notions of parts, unique identifiers, mereology and shared attributes.

4.13.2. ***Discussion***

The analysis of perdurants into actions, events and behaviours represents a choice. We suggest skeptical readers to come forward with other choices.

5. Closing

In Sect. 1.1 we emphasised that in order to develop software the designers *must have a reasonable grasp of the "underlying" domain.* That means that when we design software, its requirements, to us, must be based on such a "grasp", that is, that the domain description must cover that *"underlying" domain.* We are not claiming that the domain descriptions (for software development) must cover more than the *"underlying" domain.* But what that *"underlying" domain* then is, is an open question which we do not speculate on in this paper. Domain descriptions are not "cast in stone !" It is to be expected that domains are researched and their descriptions are developed as research projects — typically in universities. It is also to be expected that several domain descriptions coexist "simultaneously", that they may converge, that some whither away, are rejected, and that new descriptions are developed "on top of", that is, on the basis of existing ones, which they replace, descriptions that enlarge on, or restrict previous descriptions. It is finally to be expected that when requirements are to be "derived" from a domain description, see, for example, [Bjø16], that the requirements cum domain engineers redevelop a projected domain description having some existing domain descriptions "at hand".

5.1. Analysis & Description Calculi for Other Domains

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

5.2. On Domain Description Languages

We have in this paper expressed the domain descriptions in the RAISE [GHH+95] specification language RSL [GHH+92]. With what is thought of as minor changes, one can reformulate these domain description texts in either of Alloy [Jac06] or The B-Method [Abr09] or VDM [BJ78, BJ82, FL98] or Z [WD96]. One could also express domain descriptions algebraically, for example in CafeOBJ [FN97, FGO12]. The analysis and the description prompts remain the same. The description prompts now lead to Alloy, B-Method, VDM, Z or CafeOBJ texts. We did not go into much detail with respect to perdurants. For all the very many domain descriptions, covered elsewhere, RSL (with its CSP sub-language) suffices. But there are cases, not documented in this paper, where, [BGH+in], we have conjoined our RSL domain descriptions with descriptions in Petri Nets [Rei10] or MSC [IT99] (Message Sequence Charts) or StateCharts [Har87].

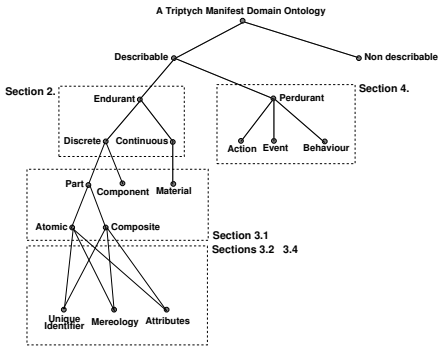

Fig. 4. The Upper Ontology of TripTych Manifest Domains

5.3. Comparison to Other Work

5.3.1. ***Background: The TripTych Domain Ontology***

We shall now compare the approach of this paper to a number of techniques and tools that are somehow related — if only by the term 'domain' ! Common to all the "other" approaches is that none of them presents a prompt calculus that help the domain analyser elicit a, or the, domain description. Figure 1 on Page 9 shows the tree-like structuring of what modern day AI researchers cum ontologists would call *an upper ontology*.

5.3.2. ***General***

Two related approaches to structuring domain understanding will be reviewed.

1: Ontology Science & Engineering: Ontologies are *"formal representations of a set of concepts within a domain and the relationships between those concepts"* — expressed usually in some logic. Ontology engineering [BF98] construct ontologies. Ontology science appears to mainly study structures of ontologies, especially so-called upper ontology structures, and these studies "waver" between philosophy and information science[36]. Internet published ontologies usually consists of thousands of logical expressions. These are represented in some, for example, low-level mechanisable form so that they can be interchanged between ontology research groups and processed by various tools. There does not seem to be a concern for "deriving" such ontologies into requirements for software. Usually ontology presentations either start with the presentation of, or makes reference to its reliance on, an *upper ontology*. The term 'ontology' has been much used in connection with automating the design of various aspects WWW applications [WDS06]. Description Logic [BCM$^+$03] has been proposed as a language for the Semantic Web [BHS05].

The interplay between endurants and perdurants is studied in [BDS04]. That study investigates axiom systems for two ontologies. One for endurants (SPAN), another for perdurants (SNAP). No examples of descriptions of specific domains are, however, given, and thus no specific techniques nor tools are given, method components which could help the engineer in constructing specific domain descriptions. [BDS04] is therefore only relevant to the current paper insofar as it justifies our emphasis on endurant versus perdurant entities.The interplay between endurant and perdurant entities and their qualities is studied in [Joh05]. In our study the term quality is made specific and covers the ideas of external and internal qualities, cf. Sect. 3.1.12 on Page 16. External qualities focus on whether endurant or perdurant, whether part, component or material, whether action, event or behaviour, whether atomic or composite part, etcetera. Internal qualities focus on unique identifiers (of parts), the mereology (of parts), and the attributes (of parts, components and materials), that is, of endurants. In [Joh05] the relationship between universals (types), particulars (values of types) and qualities is not "restricted" as in the TripTych domain analysis, but is axiomatically interwoven in an almost "recursive" manner. Values [of types ('quantities' [of 'qualities'])] are, for example, seen as sub-ordinated types; this is an ontological distinction that we do not make. The concern of [Joh05] is also the relations between

[36] We take the liberty of regarding information science as part of computer science, cf. Page 3.

qualities and both endurant and perdurant entities, where we have yet to focus on "qualities", other than signatures, of perdurants. [Joh05] investigates the quality/quantity issue wrt. endurance/perdurance and poses the questions: [b] are non-persisting quality instances enduring, perduring or neither ? and [c] are persisting quality instances enduring, perduring or neither ? and arrives, after some analysis of the endurance/perdurance concepts, at the answers: [b′] non-persisting quality instances are neither enduring nor perduring particulars (i.e., entities), and [c′] persisting quality instances are enduring particulars. Answer [b′] justifies our separating enduring and perduring entities into two disjoint, but jointly "exhaustive" ontologies. The more general study of [Joh05] is therefore really not relevant to our prompt calculi, in which we do not speculate on more abstract, conceptual qualities, but settle on external endurant qualities, on the unique identifier, mereology and attribute qualities of endurants, and the simple relations between endurants and perdurants, specifically in the relations between signatures of actions, events and behaviours and the endurant sorts , and especially the relation between parts and behaviours as outlined in Sect. 4.11. That is, the TripTych approach to ontology, i.e., its domain concept, is not only model-theoretic, but, we risk to say, radically different.

2: Knowledge Engineering: The concept of *knowledge* has occupied philosophers since Plato. No common agreement on what 'knowledge' is has been reached. From [LFCO87, Aud95, Mer04, Sta99] we may learn that *knowledge is a familiarity with someone or something; it can include facts, information, descriptions, or skills acquired through experience or education; it can refer to the theoretical or practical understanding of a subject; knowledge is produced by socio-cognitive aggregates (mainly humans) and is structured according to our understanding of how human reasoning and logic works.* The seminal reference here is [FHMV96]. The aim of *knowledge engineering* was formulated, in 1983, by an originator of the concept, Edward A. Feigenbaum [FM83]: *knowledge engineering* is an engineering discipline that involves integrating knowledge into computer systems in order to solve complex problems normally requiring a high level of human expertise. *Knowledge engineering* focus on continually building up (acquire) large, shared data bases (i.e., *knowledge bases*), their continued maintenance, testing the validity of the stored 'knowledge', continued experiments with respect to *knowledge representation*, etcetera. *Knowledge engineering* can, perhaps, best be understood in contrast to *algorithmic engineering*: In the latter we seek more-or-less conventional, usually *imperative programming language* expressions of algorithms *whose algorithmic structure embodies the knowledge required to solve the problem being solved by the algorithm.* The former seeks to solve problems based on an interpreter inferring possible solutions from logical data. This logical data has three parts: *a collection that "mimics" the semantics of, say, the imperative programming language, a collection that formulates the problem, and a collection that constitutes the knowledge particular to the problem.* We refer to [BN92]. Domain science & engineering is not aimed at letting the computer solve problems based on the knowledge it may have stored. Instead it builds models based on knowledge of the domain. Finally, the domains to which we are applying 'our form of' domain analysis are domains which focus on spatio-temporal phenomena. That is, domains which have concrete renditions: air traffic, banks, container lines, manufacturing, pipelines, railways, road transport, stock exchanges, etcetera. In contrast one may claim that the domains described in classical ontologies and knowledge representations are mostly conceptual: mathematics, physics, biology, etcetera.

5.3.3. ***Specific***

3: Database Analysis: There are different, however related "schools of database analysis". DSD: the Bachman (or data structure) diagram model [Bac69]; RDM: the relational data model [Cod70]; and ER: entity set relationshp model [Che76] "schools". DSD and ER aim at graphically specifying database structures. Codd's RDM simplifies the data models of DSD and ER while offering two kinds of languages with which to operate on RDM databases: SQL and Relational Algebra. All three "schools" are focused more on data modeling for databases than on domain modeling both endurant and perdurant entities.

4: Domain Analysis: Domain analysis, or *product line analysis* (see below), as it was then conceived in the early 1980s by James Neighbors [Nei84], is the analysis of related software systems in a domain to find their common and variable parts. This form of domain analysis turns matters "upside-down": it is the set of software "systems" (or packages) that is subject to some form of inquiry, albeit having some domain in mind, in order to find common features of the software that can be said to represent a named domain.

In this section we shall mainly be comparing the TripTych approach to domain analysis to that of Reubén Prieto-Dïaz's approach [PD87, PD90, PDA91]. Firstly, our understanding of *domain analysis* basically coincides with Prieto-Dïaz's. Secondly, in, for example, [PD87], Prieto-Dïaz's domain analysis is focused on the very important stages that precede the kind of *domain modeling* that we have described: major concerns are *selection of what appears to be similar, but specific entities, identification of common features, abstraction of entities and classification. Selection*

and *identification* is assumed in our approach, but we suggest to follow the ideas of Prieto-Dĩaz. *Abstraction* (from values to types and signatures) and *classification* into parts, materials, actions, events and behaviours is what we have focused on. All-in-all we find Prieto-Dĩaz's work very relevant to our work: relating to it by providing guidance to pre-modeling steps, thereby emphasising issues that are necessarily informal, yet difficult to get started on by most software engineers. Where we might differ is on the following: although Prieto-Dĩaz does mention a need for *domain specific language*s, he does not show examples of *domain description*s in such DSLs. We, of course, basically use mathematics as the DSL. In our approach we do not consider requirements, let alone software components, as do Prieto-Dĩaz, but we find that that is not an important issue.

5: Domain Specific Languages: Martin Fowler[37] defines a *Domain-specific language* (DSL) *as a computer programming language of limited expressiveness focused on a particular domain* [Fow20]. Other references are [MHS05, Spi01]. Common to [Spi01, MHS05, Fow20] is that they define a domain in terms of classes of software packages; that they never really "derive" the DSL from a description of the domain; and that they certainly do not describe the domain in terms of that DSL, for example, by formalising the DSL. In [HPK11] a domain specific language for railway tracks is the basis for verification of the monitoring and control of train traffic on these tracks. Specifications in that domain specific language, DSL, manifested by track layout drawings and signal interlocking tables, are then translated into RSL [GHH+92], and, from there into SystemC [GLMS02]. [HPK11] thus takes one very specific DSL and shows how to (informally) translate their "programs", which are not "directly executable", and hence does not satisfy Fowler's definition of DSLa, into executable programs. [HPK11] is a great paper, but it is not solving our problem, that of systematically describing any manifest domain. [HPK11] does, however, point a way to search for — say graphical — DSLs and the possible translation of their programs into executable ones.

6: Feature-oriented Domain Analysis (FODA): Feature oriented domain analysis (FODA) is a domain analysis method which introduced feature modeling to domain engineering. FODA was developed in 1990 following several U.S. Government research projects. Its concepts have been regarded as "critically advancing software engineering and software reuse." The US Government–supported report [KCH+90] states: *"FODA is a necessary first step"* for software reuse. To the extent that TripTych *domain engineering* with its subsequent *requirements engineering* indeed encourages reuse at all levels: *domain description*s and *requirements prescription*, we can only agree. Another source on FODA is [CE00]. Since FODA "leans" quite heavily on 'Software Product Line Engineering' our remarks in that section, next, apply equally well here.

7: Software Product Line Engineering: Software product line engineering, earlier known as domain engineering, is the entire process of *reusing domain knowledge* in the production of new software systems. Key concerns of software product line engineering are *reuse*, the building of repositories of *reusable software component*s, and *domain specific language*s with which to more-or-less automatically build software based on *reusable software component*s. These are not the primary concerns of TripTych *domain science & engineering*. But they do become concerns as we move from *domain description*s to *requirements prescription*s. But it strongly seems that *software product line engineering* is not really focused on the concerns of *domain description* — such as is TripTych *domain engineering*. It seems that *software product line engineering* is primarily based, as is, for example, FODA: Feature-oriented Domain Analysis, on analysing features of software systems. Our [Bjø11c] puts the ideas of *software product line*s and *model-oriented software development* in the context of the TripTych approach.

8: Problem Frames: The concept of *problem frames* is covered in [Jac01]. Jackson's prescription for software development focus on the "triple development" of descriptions of the *problem world*, the *requirements* and the *machine* (i.e., the *hardware* and *software*) to be built. Here *domain analysis* means the same as for us: the *problem world analysis*. In the *problem frame* approach the software developer plays three, that is, all the TripTych rôles: *domain engineer*, *requirements engineer* and *software engineer*, "all at the same time", iterating between these rôles repeatedly. So, perhaps belabouring the point, *domain engineering* is done only to the extent needed by the prescription of *requirements* and the *design* of *software*. These, really are minor points. But in "restricting" oneself to consider only those aspects of the domain which are mandated by the *requirements prescription* and *software design* one is considering a potentially smaller fragment [Jac10] of the domain than is suggested by the TripTych approach. At the same time one is, however, sure to consider aspects of the domain that might have been overlooked when pursuing *domain description development* in the "more general" TripTych approach.

[37] **http://martinfowler.com/dsl.html**

9: Domain Specific Software Architectures (DSSA): It seems that the concept of DSSA was formulated by a group of ARPA[38] project "seekers" who also performed a year long study (from around early-mid 1990s); key members of the DSSA project were Will Tracz, Bob Balzer, Rick Hayes-Roth and Richard Platek [Tra94]. The [Tra94] definition of *domain engineering* is *"the process of creating a DSSA: domain analysis and domain modeling followed by creating a software architecture and populating it with software components."* This definition is basically followed also by [MG92, SG96, MC04]. Defined and pursued this way, DSSA appears, notably in these latter references, to start with the analysis of software components, "per domain", to identify commonalities within application software, and to then base the idea of *software architecture* on these findings. Thus DSSA turns matter "upside-down" with respect to TripTych *requirements development* by starting with *software component*s, assuming that these satisfy some *requirements*, and then suggesting *domain specific software* built using these components. This is not what we are doing: we suggest, [Bjø08], that *requirements* can be "derived" systematically from, and formally related back to *domain description*s without, in principle, considering *software component*s, whether already existing, or being subsequently developed. Of course, given a *domain description* it is obvious that one can develop, from it, any number of *requirements prescription*s and that these may strongly hint at shared, (to be) implemented *software component*s; but it may also, as well, be the case that two or more *requirements prescription*s "derived" from the same *domain description* may share no *software component*s whatsoever ! It seems to this author that had the DSSA promoters based their studies and practice on also using formal specifications, at all levels of their study and practice, then some very interesting insights might have arisen.

10: Domain Driven Design (DDD): Domain-driven design (DDD)[39] *"is an approach to developing software for complex needs by deeply connecting the implementation to an evolving model of the core business concepts; the premise of domain-driven design is the following: placing the project's primary focus on the core domain and domain logic; basing complex designs on a model; initiating a creative collaboration between technical and domain experts to iteratively cut ever closer to the conceptual heart of the problem."*[40] We have studied some of the DDD literature, mostly only accessible on the Internet, but see also [Hay09], and find that it really does not contribute to new insight into *domain*s such as wee see them: it is just "plain, good old software engineering cooked up with a new jargon.

11: Unified Modeling Language (UML): Three books representative of UML are [BRJ98, RJB98, JBR99]. The term *domain analysis* appears numerous times in these books, yet there is no clear, definitive understanding of whether it, the *domain*, stands for entities in the domain such as we understand it, or whether it is wrought up, as in several of the 'approaches' treated in this section, to wit, in items [3–5, 7–9] with either *software design* (as it most often is), or *requirements prescription*. Certainly, in UML, in [BRJ98, RJB98, JBR99] as well as in most published papers claiming "adherence" to UML, that domain analysis usuallyis manifested in some UML text which "models" some *requirements* facet. Nothing is necessarily wrong with that, but it is therefore not really the TripTych form of *domain analysis* with its concepts of abstract representations of endurant and perdurants, with its distinctions between *domain* and *requirements*, and with its possibility of "deriving" *requirements prescription*s from *domain descriptions*. The UML notion of *class diagram*s is worth relating to our structuring of the domain. Class diagrams appear to be inspired by [Bac69, Bachman, 1969] and [Che76, Chen, 1976]. It seems that (i) each part sort — as well as other than part sorts — deserves a class diagram (box); and (ii) that (assignable) attributes — as well as other non-part types — are written into the diagram box. Class diagram boxes are line-connected with annotations where some annotations are as per the mereology of the part type and the connected part types and others are not part related. The class diagrams are said to be object-oriented but it is not clear how objects relate to parts as many are rather implementation-oriented quantities. All this needs looking into a bit more, for those who care.

12: Requirements Engineering: There are in-numerous books and published papers on *requirements engineering*. A seminal one is [van09]. I, myself, find [Lau02] full of very useful, non-trivial insight. [DT97] is seminal in that it brings a number or early contributions and views on *requirements engineering*. Conventional text books, notably [Pfl01, Pre01, Som06] all have their "mandatory", yet conventional coverage of *requirements engineering*. None of them "derive" requirements from domain descriptions, yes, OK, from domains, but since their description is not mandated it is unclear what "the domain" is. Most of them repeatedly refer to *domain analysis* but since a written record of that *domain analysis* is not mandated it is unclear what "domain analysis" really amounts to. Axel van Laamsweerde's

[38] ARPA: The US DoD Advanced Research Projects Agency
[39] Eric Evans: http://www.domaindrivendesign.org/
[40] http://en.wikipedia.org/wiki/Domain-driven_design

book [van09] is remarkable. Although also it does not mandate descriptions of domains it is quite precise as to the relationships between domains and requirements. Besides, it has a fine treatment of the distinction between *goal*s and *requirements*, also formally. Most of the advices given in [Lau02] can beneficially be followed also in `TripTych` *requirements development*. Neither [van09] nor [Lau02] preempts `TripTych` *requirements development*.

5.3.4. ***Summary of Comparisons***

We find that there are two kinds of relevant comparisons: the concept of ontology, its science more than its engineering, and the *Problem Frame* work of Michael A. Jackson. The ontology work, as commented upon in Item **[1]** (Pages 41–42), is partly relevant to our work: There are at least two issues: Different classes of domains may need distinct upper ontologies. Section 5.1 suggests that there may be different upper ontologies for non-manifest domains such as *financial systems*, etcetera. This seems to warrant at least a comparative study. We have assumed, cf. Sect. 3.4.1, that attributes cannot be separated from parts. [Joh05, Johansson 2005] develops the notion that *persisting quality instances are enduring particulars*. The issue need further clarification.

Of all the other "comparison" items (**[2]**–**[12]**) basically only Jackson's *problem frames* (Item **[8]**) and [HPK11] (Item **[5]**) really take the same view of *domain*s and, in essence, basically maintain similar relations between *requirements prescription* and *domain description*. So potential sources of, we should claim, mutual inspiration ought be found in one-another's work — with, for example, [GGJZ00, Jac10, HPK11], and the present document, being a good starting point.

But none of the referenced works make the distinction between discrete endurants (parts) and their qualities, with their further distinctions between *unique identifiers*, *mereology* and *attributes*. And none of them makes the distinction between *parts*, *components* and *materials*. Therefore our contribution can include the mapping of parts into behaviours interacting as per the part mereologies as highlighted in the process schemas of Sect. 4.11 Pages 36–38.

5.4. Open Problems

The present paper has outlined a great number of principles, techniques and tools of domain analysis & description. They give rise, now, to the investigation of further principles, techniques and tools as well as underlying theories. We list some of these "to do" items: *a mathematical model of prompts; a sharpened definition of "what is a domain"; laws of description prompts; a formal understanding of domain facets* [Bjø10a]; *a prompt calculus for perdurants; commensurate discrete and continuous models; a study of the interplay between parts, materials and components;* and *specific domain theories*.

5.5. Tony Hoare's Summary on 'Domain Modeling'

In a 2006 e-mail, in response, undoubtedly to my steadfast, perhaps conceived as stubborn insistence, on domain engineering, Tony Hoare summed up his reaction to domain engineering as follows, and I quote[41]:

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

1. *The models describe all aspects of the real world that are relevant for any good software design in the area. They describe possible places to define the system boundary for any particular project.*
2. *They make explicit the preconditions about the real world that have to be made in any embedded software design, especially one that is going to be formally proved.*
3. *They describe the whole range of possible designs for the software, and the whole range of technologies available for its realisation.*
4. *They provide a framework for a full analysis of requirements, which is wholly independent of the technology of implementation.*
5. *They enumerate and analyse the decisions that must be taken earlier or later in any design project, and identify those that are independent and those that conflict. Late discovery of feature interactions can be avoided.*"

All of these issues are covered, to some extent, in [Bjø06, Part IV]. Tony Hoare's list pertains to a wider range that just the Manifest Domains treated in this chapter.

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

5.6. Beauty Is Our Business

It's life that matters, nothing but life –
the process of discovering, the everlasting and perpetual process,
not the discovery itself, at all.[42]

I find that quote appropriate in the following, albeit rather mundane, sense: It is the process of analysing and describing a domain that exhilarates me: that causes me to feel very happy and excited. There is beauty [FvGGM90, E.W. Dijkstra Festschrift] not only in the result but also in the process.

5.7. Acknowledgements

This paper is dedicated to the memory of Peter Lucas (13 Jan. 1935 – 2 Feb. 2015). Peter brought me to the IBM Wiener Labor in 1973. The two years I spent with him became the most important years in my scientific life. Peter was a great scientist and a lovely person.

This paper has been many years underway. Earlier versions have been the basis for ("innumerable") PhD lectures and seminars around the world — after I retired from The Technical University of Denmark. I thank the many organisers of those events for their willingness to "hear me out": Jin Song Dong, NUS, Singapore; Kokichi Futatsugi and Kazuhiro Ogata, JAIST, Japan; Dominique Méry, Univ. of Nancy, France; Franz Wotawa and Bernhard K. Aichernig, Techn. Univ. of Graz, Austria; Wolfgang J. Paul, Univ. of Saarland, Germany; Alan Bundy, University of Edinburgh, Scotland; Tetsuo Tamai, then at Tokyo Univ., now at Hosei Univ., Tokyo, Japan; Jens Knoop, Techn. Univ. of Vienna, Austria; Dömölki Balint and Kozma László, Eötös Loránt Univ., Budapest, Hungary; Lars-Henrik Ericsson, Univ. of Uppsala, Sweden; Peter D. Mosses, Univ. of Swansea, Wales; Magne Haveraaen, Univ. of Bergen, Norway; Sun Meng, Peking Univ., China; He JiFeng and Zhu HuiBiao, East China Normal Univ., Shanghai, China; Zhou Chaochen, Lin Huimin and Zhan Naijun, Inst. of Softw., CAS, Beijing, China; and Victor P. Ivannikov, Inst. of Sys. Prgr., RAS, Moscow, Russia; Luís Soares Barbosa and Jose Nuno Oliveira, Univ. of Minho, Portugal. I finally thank the referees of this paper for their most helpful comments.

6. Bibliography

6.1. Bibliographical Notes

6.1.1. ***Published Papers***

Web page **www.imm.dtu.dk/~dibj/domains/** lists the published papers and reports mentioned below. I have thought about domain engineering for more than 25 years. But serious, focused writing only started to appear since [Bjø06, Part IV] — with [Bjø03, Bjø97] being exceptions: [Bjø07, 2007] suggests a number of domain science and engineering research topics; [Bjø10a, 2008] covers the concept of domain facets; [BE10, 2008] explores compositionality and Galois connections. [Bjø08, Bjø10c, 2008,2009] show how to systematically, but, of course, not automatically, "derive" requirements prescriptions from domain descriptions; [Bjø11a, 2008] takes the triptych software development as a basis for outlining principles for believable software management; [Bjø09, Bjø14a, 2009,2013] presents a model for Stanisław Leśniewski's [CV99] concept of mereology; [Bjø10b, Bjø11b] present an extensive example and is otherwise a precursor for the present paper; [Bjø11c, 2010] presents, based on the `TripTych` view of software development as ideally proceeding from domain description via requirements prescription to software design, concepts such as software demos and simulators; [Bjø13a, 2012] analyses the `TripTych`, especially its domain engineering approach, with respect to Maslow's [43] and Peterson's and Seligman's [44] notions of humanity: how can computing relate to notions of humanity; the first part of [Bjø14b, 2014] is a precursor for the present paper with its second part presenting a first formal model of the elicitation process of analysis and description based on the prompts more definitively presented in the current paper; and [Bjø14c, 2014] focus on domain safety criticality. The present paper

[42] `Fyodor Dostoyevsky, The Idiot, 1868, Part 3, Sect. V`

[43] *Theory of Human Motivation.* Psychological Review 50(4) (1943):370-96; and *Motivation and Personality,* Third Edition, Harper and Row Publishers, 1954.

[44] *Character strengths and virtues: A handbook and classification.* Oxford University Press, 2004

basically replaces the domain analysis and description section of all of the above reference — including [Bjø06, Part IV, 2006].

6.1.2. ***Reports***

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

1 *A Railway Systems Domain:* `http://euler.fd.cvut.cz/railwaydomain/` (2003)
2 *Models of IT Security. Security Rules & Regulations:* `it-security.pdf` (2006)
3 *A Container Line Industry Domain:* `container-paper.pdf` (2007)
4 *The "Market": Consumers, Retailers, Wholesalers, Producers:* `themarket.pdf` (2007)
5 *What is Logistics ?:* `logistics.pdf` (2009)
6 *A Domain Model of Oil Pipelines:* `pipeline.pdf` (2009)
7 *Transport Systems:* `comet/comet1.pdf` (2010)
8 *The Tokyo Stock Exchange:* `todai/tse-1.pdf` and `todai/tse-2.pdf` (2010)
9 *On Development of Web-based Software. A Divertimento:* `wfdftp.pdf` (2010)
10 *Documents (incomplete draft):* `doc-p.pdf` (2013)

6.2. References

[Abr09] Jean-Raymond Abrial. The B Book: Assigning Programs to Meanings *and* Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge, England, 1996 and 2009.

[Aud95] Rober Audi. *The Cambridge Dictionary of Philosophy*. Cambridge University Press, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, England, 1995.

[Bac69] C. Bachman. Data structure diagrams. *Data Base, Journal of ACM SIGBDP*, 1(2), 1969.

[Bad05] Alain Badiou. *Being and Event*. Continuum, 2005. (Lêtre et l'événements, Edition du Seuil, 1988).

[BCM+03] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. . Cambrige University Press, January 2003. 570 pages, 14 tables, 53 figures; ISBN: 0521781760.

[BDS04] Thomas Bittner, Maureen Donnelly, and Barry Smith. Endurants and Perdurants in Directly Depicting Ontologies. *AI Communications*, 17(4):247–258, December 2004. IOS Press, in [RG04].

[BE10] Dines Bjørner and Asger Eir. Compositionality: Ontology and Mereology of Domains. Some Clarifying Observations in the Context of Software Engineering in July 2008, eds. Martin Steffen, Dennis Dams and Ulrich Hannemann. In *Festschrift for Prof. Willem Paul de Roever Concurrency, Compositionality, and Correctness*, volume 5930 of *Lecture Notes in Computer Science*, pages 22–59, Heidelberg, July 2010. Springer.

[BF98] V. Richard Benjamins and Dieter Fensel. The Ontological Engineering Initiative (KA)2. Internet publication + Formal Ontology in Information Systems, University of Amsterdam, SWI, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands and University of Karlsruhe, AIFB, 76128 Karlsruhe, Germany, 1998. http://www.aifb.uni-karlsruhe.de/WBS/broker/KA2.htm.

[BGH+in] Dines Bjørner, Chris W. George, Anne Eliabeth Haxthausen, Christian Krog Madsen, Steffen Holmslykke, and Martin Pěnička. "UML"–ising Formal Techniques. In *INT 2004: Third International Workshop on Integration of Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 423–450. Springer–Verlag, 28 March 2004, ETAPS, Barcelona, Spain. Final Version.

[BHS05] Franz Baader, Ian Horrocks, and Ulrike Sattler. Description Logics as Ontology Languages for the Semantic Web. In Dieter Hutter and Werner Stephan, editors, *Mechanizing Mathematical Reasoning*, pages 228–248. Springer, Heidelberg, 2005.

[BJ78] Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.

[BJ82] Dines Bjørner and Cliff B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.

[Bjø97] Dines Bjørner. Michael Jackson's Problem Frames: Domains, Requirements and Design. In Li ShaoYang and Michael Hinchley, editors, *ICFEM'97: International Conference on Formal Engineering Methods*, Los Alamitos, November 12–14 1997. IEEE Computer Society. Final Version.

[Bjø03] Dines Bjørner. Domain Engineering: A "Radical Innovation" for Systems and Software Engineering ? In *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, Heidelberg, October 7–11 2003. Springer–Verlag. The Zohar Manna International Conference, Taormina, Sicily 29 June – 4 July 2003. .

[Bjø06] Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.

[Bjø07] Dines Bjørner. Domain Theory: Practice and Theories, Discussion of Possible Research Topics. In *ICTAC'2007*, volume 4701 of *Lecture Notes in Computer Science (eds. J.C.P. Woodcock et al.)*, pages 1–17, Heidelberg, September 2007. Springer.

[Bjø08] Dines Bjørner. From Domains to Requirements. In *Montanari Festschrift*, volume 5065 of *Lecture Notes in Computer Science (eds. Pierpaolo Degano, Rocco De Nicola and José Meseguer)*, pages 1–30, Heidelberg, May 2008. Springer.

[Bjø09] 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.

[Bjø10a] Dines Bjørner. Domain Engineering. In Paul Boca and Jonathan Bowen, editors, *Formal Methods: State of the Art and New Directions*, Eds. Paul Boca and Jonathan Bowen, pages 1–42, London, UK, 2010. Springer.

[Bjø10b] Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics, Part I of II: The Engineering Part*. *Kibernetika i sistemny analiz*, (4):100–116, May 2010.

[Bjø10c] Dines Bjørner. The Rôle of Domain Engineering in Software Development. Why Current Requirements Engineering Seems Flawed! In *Perspectives of Systems Informatics*, volume 5947 of *Lecture Notes in Computer Science*, pages 2–34, Heidelberg, Wednesday, January 27, 2010. Springer.

[Bjø11a] Dines Bjørner. Believable Software Management. *Encyclopedia of Software Engineering*, 1(1):1–32, 2011.

[Bjø11b] 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):100–120, May 2011.

[Bjø11c] Dines Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. In *Rainbow of Computer Science, Festschrift for Hermann Maurer on the Occasion of His 70th Anniversary.*, Festschrift (eds. C. Calude, G. Rozenberg and A. Saloma), pages 167–183. Springer, Heidelberg, Germany, January 2011.

[Bjø13a] Dines Bjørner. *Domain Science and Engineering as a Foundation for Computation for Humanity*, chapter 7, pages 159–177. Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC [Francis & Taylor], 2013. (eds.: Justyna Zander and Pieter J. Mosterman).

[Bjø13b] Dines Bjørner. Pipelines – a Domain Description[45]. Experimental Research Report 2013-2, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.

[Bjø14a] Dines Bjørner. *A Rôle for Mereology in Domain Science and Engineering*. Synthese Library (eds. Claudio Calosi and Pierluigi Graziani). Springer, Amsterdam, The Netherlands, October 2014.

[Bjø14b] Dines Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model. In Shusaku Iida, José Meseguer, and Kazuhiro Ogata, editors, *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi*. Springer, May 2014.

[Bjø14c] Dines Bjørner. Domain Engineering – A Basis for Safety Critical Software. Invited Keynote, ASSC2014: Australian System Safety Conference, Melbourne, 26–28 May, December 2014.

[Bjø16] Dines Bjørner. From Domain Descriptions to Requirements Prescriptions – A Different Approach to Requirements Engineering. *Submitted for consideration by Formal Aspects of Computing*, 2016.

[Bli90] Wayne D. Blizard. A Formal Theory of Objects, Space and Time. *The Journal of Symbolic Logic*, 55(1):74–89, March 1990.

[BN92] Dines Bjørner and Jørgen Fischer Nilsson. Algorithmic & Knowledge Based Methods — Do they "Unify" ? In *International Conference on Fifth Generation Computer Systems: FGCS'92*, pages 191–198. ICOT, June 1–5 1992.

[BRJ98] Grady Booch, Jim Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2000.

[Che76] Peter P. Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Trans. Database Syst*, 1(1):9–36, 1976.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[CV96] Roberto Casati and Achille C. Varzi, editors. *Events*. Ashgate Publishing Group – Dartmouth Publishing Co. Ltd., Wey Court East, Union Road, Farnham, Surrey, GU9 7PT, United Kingdom, 23 March 1996.

[CV99] R. Casati and A. Varzi. *Parts and Places: the structures of spatial representation*. MIT Press, 1999.

[CV10] Roberto Casati and Achille Varzi. Events. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2010 edition, 2010.

[Dav80] Donald Davidson. *Essays on Actions and Events*. Oxford University Press, 1980.

[Dre67] F. Dretske. Can Events Move? *Mind*, 76(479-492), 1967. Reprinted in [CV96, 1996], pp. 415-428.

[DT97] Merlin Dorfman and Richard H. Thayer, editors. *Software Requirements Engineering*. IEEE Computer Society Press, 1997.

[Far90] David John Farmer. *Being in time: The nature of time in light of McTaggart's paradox*. University Press of America, Lanham, Maryland, 1990. 223 pages.

[FGO12] Kokichi Futatsugi, Daniel Gâlinâ, and Kazuhiro Ogata. Principles of proof scores in CafeOBJ. *Theor. Comput. Science*, 464:90–112, 2012.

[FHMV96] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning about Knowledge*. The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts 02142, 1996. 2nd printing.

[FL98] 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.

[FM83] Edward A. Feigenbaum and Pamela McCorduck. *The fifth generation*. Addison-Wesley, Reading, MA, USA, 1st ed. edition, 1983.

[FMMR12] Carlo A. Furia, Dino Mandrioli, Angelo Morzenti, and Matteo Rossi. *Modeling Time in Computing*. Monographs in Theoretical Computer Science. Springer, 2012.

[FN97] Kokichi Futatsugi and Ataru Nakagawa. An overview of CAFE specification environment - an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In *Proc. of 1st International Conference on Formal Engineering Methods (ICFEM '97), November 12-14, 1997, Hiroshima, JAPAN*, pages 170–182. IEEE, 1997.

[Fow20] Martin Fowler. *Domain Specific Languages*. Signature Series. Addison Wesley, October 20120.

[FvGGM90] W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors. *Beauty is Our Business*, Texts and Monographs in Computer Science, New York, NY, USA, 1990. Springer. A Birthday Salute to Edsger W. Dijkstra.

[GGJZ00] Carl A. Gunter, Elsa L. Gunter, Michael A. Jackson, and Pamela Zave. A Reference Model for Requirements and Specifications. *IEEE Software*, 17(3):37–43, May–June 2000.

[45] http://www.imm.dtu.dk/~dibj/pipe-p.pdf

[GHH+92] 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.

[GHH+95] 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.

[GLMS02] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer, Dordrecht, 2002.

[GW99] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis — Mathematical Foundations*. Springer-Verlag, January 1999.

[Hac82] P.M.S. Hacker. Events and Objects in Space and Time. *Mind*, 91:1–19, 1982. reprinted in [CV96], pp. 429-447.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Hay09] Dan Haywood. *Domain-Driven Design Using Naked Objects*. The Pragmatic Bookshelf (an imprint of 'The Pragmatic Programmers, LLC.'), http://pragprog.com/, 2009.

[Hei62] Martin Heidegger. *Sein und Zeit (Being and Time)*. Oxford University Press, 1927, 1962.

[Hoa85] Charles Anthony Richard Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985. Published electronically: http://www.usingcsp.com/cspbook.pdf (2004).

[HPK11] A.E. Haxthausen, J. Peleska, and S. Kinder. A formal approach for the construction and and verification of railway control systems. *Formal Aspects of Computing*, 23:191–219, 2011.

[IT99] ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992, 1996, 1999.

[Jac95] Michael A. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press. Addison-Wesley, Reading, England, 1995.

[Jac01] Michael A. Jackson. *Problem Frames — Analyzing and Structuring Software Development Problems*. ACM Press, Pearson Education. Addison-Wesley, England, 2001.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.

[Jac10] 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.

[JBR99] Ivar Jacobson, Grady Booch, and Jim Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[JHJ07] Cliff B. Jones, Ian Hayes, and Michael A. Jackson. Deriving Specfications for Systems That Are Connected to the Physical World. In Cliff Jones, Zhiming Liu, and James Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems: Essays in Honour of Dines Bjørner and Zhou Chaochen on the Occasion of Their 70th Birthdays*, volume 4700 of *Lecture Notes in Computer Science*, pages 364–390. Springer, 2007.

[Joh05] Ingvar Johansson. Qualities, Quantities, and the Endurant-Perdurant Distinction in Top-Level Ontologies. In Dengel A. Bergmann R. Nick M. Roth-Berghofer Th. Althoff, K.-D., editor, *Professional Knowledge Management WM 2005*, volume 3782 of *Lecture Notes in Artificial Intelligence*, pages 543–550. Springer, 2005. 3rd Biennial Conference, Kaiserslautern, Germany, April 10-13, 2005, Revised Selected Papers.

[KCH+90] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S Peterson. FODA: Feature-Oriented Domain Analysis. Feasibility Study CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, November 1990. http://www.sei.cmu.edu/library/abstracts/reports/90tr021.cfm.

[Kim93] Jaegwon Kim. *Supervenience and Mind*. Cambridge University Press, 1993.

[Lam02] Leslie Lamport. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA, 2002.

[Lau02] Søren Lauesen. *Software Requirements - Styles and Techniques*. Addison-Wesley, UK, 2002.

[LFCO87] W. Little, H.W. Fowler, J. Coulson, and C.T. Onions. *The Shorter Oxford English Dictionary on Historical Principles*. Clarendon Press, Oxford, England, 1987.

[LWZ13] Zhiming Liu, J. C. P. Woodcock, and Huibiao Zhu, editors. *Unifying Theories of Programming and Formal Engineering Methods - International Training School on Software Engineering, Held at ICTAC 2013, Shanghai, China, August 26-30, 2013, Advanced Lectures*, volume 8050 of *Lecture Notes in Computer Science*. Springer, 2013.

[MC04] Neno Medvidovic and Edward Colbert. Domain-Specific Software Architectures (DSSA). Power Point Presentation, found on The Internet, Absolute Software Corp., Inc.: `Abs[S/W]`, 5 March 2004.

[Mel80] D.H. Mellor. Things and Causes in Spacetime. *British Journal for the Philosophy of Science*, 31:282–288, 1980.

[Mer04] Merriam Webster Staff. Online Dictionary: `http://www.m-w.com/home.htm`, 2004. Merriam–Webster, Inc., 47 Federal Street, P.O. Box 281, Springfield, MA 01102, USA.

[MG92] Erik Mettala and Marc H. Graham. The Domain Specific Software Architecture Program. Project Report CMU/SEI-92-SR-009, Software Engineering Institute Carnegie Mellon University Pittsburgh, Pennsylvania 15213, June 1992.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.

[Nei84] James M. Neighbors. The Draco Approach to Constructing Software from Reusable Components. *IEEE Transactions of Software Engineering*, SE-10(5), September 1984.

[PD87] Rubén Prieto-Díaz. Domain Analysis for Reusability. In *COMPSAC 87*. ACM Press, 1987.

[PD90] Rubén Prieto-Díaz. Domain analysis: an introduction. *Software Engineering Notes*, 15(2):47–54, 1990.

[PDA91] Rubén Prieto-Díaz and Guillermo Arrango. *Domain Analysis and Software Systems Modelling*. IEEE Computer Society Press, 1991.

[Pfl01] Shari Lawrence Pfleeger. *Software Engineering, Theory and Practice*. Prentice–Hall, 2nd edition, 2001.

[Pi99] Chia-Yi Tony Pi. *Mereology in Event Semantics*. Phd, McGill University, Montreal, Canada, August 1999.

[Pre01] Roger S. Pressman. *Software Engineering, A Practitioner's Approach*. International Edition, Computer Science Series. McGraw–Hill, 5th edition, 1981–2001.

[Qui79] A. Quinton. Objects and Events. *Mind*, 88:197–214, 1979.

[Rei10] Wolfang Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Leitfäden der Informatik. Vieweg+Teubner, 1st edition, 15 June 2010. 248 pages; ISBN 978-3-8348-1290-2.

[RG04] Jochen Renz and Hans W. Guesgen, editors. *Spatial and Temporal Reasoning*, volume 14, vol. 4, Journal: AI Communications, Amsterdam, The Netherlands, Special Issue. IOS Press, December 2004.
[RJB98] Jim Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
[Som06] Ian Sommerville. *Software Engineering*. Pearson, 8th edition, 2006.
[Sow99] John F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Brooks/Cole Thompson Learning, August 17, 1999.
[Spi01] Diomidis Spinellis. Notable design patterns for domain specific languages. *Journal of Systems and Software*, 56(1):91–99, February 2001.
[ST12] Donald Sannella and Andrzej Tarlecki. *Foundations of Algebraic Semantcs and Formal Software Development*. Monographs in Theoretical Computer Science. Springer, Heidelberg, 2012.
[Sta99] Staff of Encyclopœdia Brittanica. Encyclopœdia Brittanica. Merriam Webster/Brittanica: Access over the Web: http://www.eb.com:-180/, 1999.
[Tra94] Will Tracz. Domain-specific software architecture (DSSA) frequently asked questions (FAQ). *Software Engineering Notes*, 19(2):52–56, 1994.
[van91] Johan van Benthem. *The Logic of Time*, volume 156 of *Synthese Library: Studies in Epistemology, Logic, Methhodology, and Philosophy of Science (Editor: Jaakko Hintika)*. Kluwer Academic Publishers, P.O.Box 17, NL 3300 AA Dordrecht, The Netherlands, second edition, 1983, 1991.
[van09] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.
[WD96] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.
[WDS06] H. Wang, J. S. Dong, and J. Sun. Reasoning Support for Semantic Web Ontology Family Languages Using Alloy. *International Journal of Multiagent and Grid Systems, IOS Press*, 2(4):455–471, 2006.
[Whi20] A.N. Whitehead. *The Concept of Nature*. Cambridge University Press, Cambridge, 1920.
[WS12] George Wilson and Samuel Shpall. Action. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2012 edition, 2012.
[WYZ94] Ji Wang, XinYao Yu, and Chao Chen Zhou. Hybrid Refinement. Research Report 20, UNU/IIST, P.O.Box 3058, Macau, 1. April 1994.
[ZH04] 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.
[ZWZ13] Naijun Zhan, Shuling Wang, and Hengjun Zhao. Formal modelling, analysis and verification of hybrid systems. In *ICTAC Training School on Software Engineering*, pages 207–281, 2013.