Domain Endurants

An Analysis and Description Process Model

Dines Bjørner

Fredsvej 11, DK-2840 Holte, Danmark
DTU, DK-2800 Kgs. Lyngby, Denmark
E–Mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~dibj

Laudatio

Futatsugi says he's known me since the IFIP World Gongress in Tokyo, Japan, September 1980. I can certainly and clearly remember having met Kokichi in the late Joseph Goguen's SRI office in July 1984. He was there; so was José Meseguer and Jean-Pierre Jouannaud. They were **clear**ly onto something, **OBJ**ectively speaking, very exciting ! Nothing really "destined" us for one another. Kokichi was into algebraic specifications and I into model-oriented ones. Mathematicians versus engineers — some would say. Well, the RAISE, [25], specification language RSL, [24], does "mix" traditional model-oriented expressivity with sorts, observers and axioms — borrowed very specifically from early work on **OBJ** [27]. So maybe we were destined. At least I have enjoyed, tremendously, our acquaintance. Had we lived closer, geographically, I might even have been able to claim the kind of friendship that survives sitting together, not saying a word, for hours. That's not difficult in our case: Kokichi has his mother tongue, hopelessly isolated out here, in the Far East, and I have my mother tongue, hopelessly isolated back here ! Kokichi and his work has become an institution [26]. First ETL and then JAIST became firmly implanted in the universe of the communities of algebraic semantics and formal specification scientists. Not many Japanese computer scientists have become so well-known abroad as has Kokichi. One thing that has paved the way for this is Kokichi's personality. A Japanese at ease also in the Western World. Westerners being so very kindly accepted and welcome by Kokichi and his colleagues here in the beautiful, enigmatic Land of the Rising Sun. One thing I always do complain about when seeing Kokichi in my world is that he should bring his wife, charming Junko, there more often — well every time ! So, Kokichi, thanks for your scientific contributions; thanks for your being a fine Doctors Father; thanks for hosting one of my former students, Dr. Anne Elisabeth Haxthausen for half a year at ETL; thanks for hosting me here at JAIST for a whole year, 2006; and thanks for helping us "barbarian" Westerners getting to love Japan and all things Japanese.

Abstract: We present a summary, Sect. 2, of a structure of domain analysis and description concepts: techniques and tools. And we link, in Sect. 3, these concepts, embodied in *domain analysis prompt*s and *domain description prompt*s, in a model of how a diligent domain analyser cum describer would use them. We claim that both sections, Sects. 2–3, contribute to a methodology of software engineering.

1 Introduction

A Context for Domains: Before software can be designed we must have a reasonably good grasp of its requirements. Before requirements can be prescribed we must have a reasonably good grasp of the domain in which the software is to reside. So we turn to domain analysis & description as a means to obtain and record that 'grasp'. In this paper we summarise an approach to domain analysis & description recorded in more detail in [12]. Thus this paper is based on [12].

Related Papers: This paper is one in a series of papers on domain science & engineering. In [6] we present techniques related to the analysis and description of domain facets. In [4] we investigate some research issues of domain science. The paper [13] examines possible contributions of domain science & engineering to *computation for the humanities*. It is expected that the present paper may be followed by respective ("spin-off") papers on *Perdurants* [10], *A Formal Model of Prompts* [11], *Domain Facets* (cf. [6]) [9], and *On Deriving Requirements From Domain Descriptions* (cf. [5]) [14].

A `TripTych` of Software Engineering: The first 3+ lines above suggest an "idealised", the `TripTych`, approach to software development: first a phase of domain engineering in which is built a domain model; then a phase of requirements engineering in which is built a requirements model; and finally a phase of software design in which the code is developed. We show in [5] how to systematically "transform" domain descriptions into requirements prescriptions.

Structure of this Paper: The structure of this paper is as follows: First, in Sect. 2 we present a terse summary of a system of domain analysis & description concepts focused on endurants. This summary is rather terse, and is a *"tour de force"*. Section 2 is one of the two main sections of this paper. Section 3 suggests a formal-looking model of the structure of *domain analysis prompt*s and *domain description prompt*s introduced in Sect. 2. It is not a formalisation of domains, but of the domain analysis & description process. Domains are usually not computationally tractable. Less so is the domain analysis & description processes. Finally, Sect. 4 concludes this paper. An appendix, Appendix A, presents a domain description of a [class of] *pipeline systems*. Some seminars over the underlying paper may start by a brief presentation of this model. The reader is invited to browse this *pipeline system* model before, during and/or after reading Sects. 2–3.

2 The Domain Analysis Approach

2.1 Hierarchical versus Compositional Analysis & Description

In this paper we choose, what we shall call, a 'hierarchical analysis' approach which is based on decomposing an understanding of a domain from the "over-

all domain" into its components, and these, if not atomic, into their subcomponents • In contrast we could have chosen a 'compositional analysis' approach which starts with an understanding of a domain from its atomic endurants and composes these into composite ones, finally ending up with an "overall domain" description •

2.2 Domains

A 'domain' is characterised by its observable, i.e., manifest *entities* and their *qualities* • [1] *Example 1.* Domains: *a road net, a container line, a pipeline, a hospital* ■ [2]

2.3 Sorts, Types and Domain Analysis

By a 'sort' (or 'type' which we take to be the same) we shall understand the largest set of entities all of which have the same qualities[3] • *Example 2.* Sorts: Links of any road net constitute a sort. So does hubs. The largest set of (well-formed) collections of links constitute a sort. So does similar collections of hubs. The largest set of road nets (containing well-formed collections of hubs and links) form a sort ■

By 'domain analysis' we shall understand a process whereby a domain analyser groups entities of a domain into sorts (and types) • The rest of this paper will outline a class of domain analysis principles, techniques and tools.

2.4 Entities and Qualities

Entities: By an 'entity' we shall understand a phenomenon that can be observed, i.e., be seen or touched[4] by humans, or that can be conceived as an abstraction of an entity[5] • The method can thus be said to provide the *domain analysis prompt*: `is_entity` where `is_entity(`θ`)` holds if θ is an entity. *Example 3.* Entities: *(a) a road net, (b) a link*[6] *of a road net, (c) a hub*[7] *of a road net;* and *(d) insertion of a link in a road net, (e) disappearance of a link of a road net, and (f) the movement of a vehicle on a road net* ■

[1] Definitions start with a single quoted 'term' and conclude with a •
[2] Examples conclude with a ■
[3] Taking a sort (type) to be *the largest set of entities all of which have the same qualities* reflects Ganter & Wille's notion of a 'formal concept' [23].
[4] An entity which can be seen or touched is thus a physical phenomenon. If an entity has the quality the colour red, it is not the red that is an entity.
[5] There is no "infinite loop" here: a concept can be an abstraction of (another) concept, etc., which is finally an abstraction of a physical phenomenon.
[6] A link: a street segment between two adjacent hubs
[7] A hub: an intersection of street segments

Qualities: By a 'quality' of an entity we shall understand a property that can be given a name and precisely measured by physical instruments or otherwise identified • *Example 4.* Quality Names: *cadestral location of a hub, hub state*[8], *hub state space*[9], etcetera ■ *Example 5.* Quality Values: *the name of a road net, the ownership of a road net, the length of a link, the location of a hub,* etcetera ■

2.5 Endurants and Perdurants

Entities are either endurants or are perdurants.

Endurants: By an 'endurant entity' (or just, an endurant) we shall understand that can be observed or conceived, 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 • Thus the method provides a *domain analysis prompt*: `is_endurant` where `is_endurant`(e) holds if entity e is an endurant. *Example 6.* Endurants: Items (a–b–c) of Example 2.4 are endurants; so are the pipes, valves, and pumps of a pipeline.

Perdurants: By a 'perdurant entity' (or just, an 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 • Thus the method provides a *domain analysis prompt*: `is_perdurant` where `is_perdurant`(e) holds if entity e is a perdurant. *Example 7.* Perdurants: Items (d–e–f) of Example 2.4 are perdurants; so are the insertion of a hub, removal of a link, etcetera ■

2.6 Discrete and Continuous Endurants

Entities are either discrete or are continuous.

Discrete Endurants: By a 'discrete endurant' we shall understand something which is separate or distinct in form or concept, consisting of distinct or separate parts • We use the term 'part' for discrete endurants, that is: `is_part`$(p) \equiv$ `is_endurant`$(p) \wedge$`is_discrete`(p) • Thus the method provides a *domain analysis prompt*: `is_discrete` where `is_discrete`(e) holds if entity e is discrete. *Example 8.* Discrete Endurants: The examples of Example 2.5 are all discrete endurants ■

Continuous Endurants: By a 'continuous endurant' we shall understand something which is prolonged without interruption, in an unbroken series or pattern • We use the term 'material' for continuous endurants • Thus the method provides

[8] From which links can one reach which links at a given time.
[9] Set of all hub states over time.

a *domain analysis prompt*: `is_continuous` where `is_continuous`(e) holds if entity e is continuous. *Example 9.* Continuous Endurants: The pipes, valves, pumps, etc., of Example 2.5 may contain oil; water of a hydro electric power plant is also a material (i.e., a continuous endurant) ■

2.7 Discrete and Continuous Perdurants

We are not covering perdurants in this paper.

2.8 Atomic and Composite Discrete Endurants

Discrete endurants are either atomic or are composite.

Atomic Endurants: By an 'atomic endurant' we shall understand a discrete endurant which in a given context, is deemed to *not* consist of meaningful, separately observable proper sub-parts • The method can thus be said to provide the *domain analysis prompt*: `is_atomic` where `is_atomic`(p) holds if p is an atomic part. *Example 10.* Atomic Parts: Examples of atomic parts of the above mentioned domains are: aircraft (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) ■

Composite Endurants: By a 'composite endurant' we shall understand a discrete endurant which in a given context, is deemed to *indeed* consist of meaningful, separately observable proper sub-parts • The method can thus be said to provide the *domain analysis prompt*: `is_composite` where `is_composite`(p) holds if p is an a composite part. *Example 11.* 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) ■

It is the domain analysers who decide whether an endurant is atomic or composite. In the context of air traffic an aircraft might very well be described as an atomic entity; whereas in the context of an airline an aircraft might very well be described as a composite entity consisting of the aircraft 'body', the crew, the passengers, their luggage, the fuel, etc.

2.9 Part Observers

From atomic parts we cannot observe any sub-parts. But from composite parts we can.

Composite Sorts: For composite parts, p, the *domain description prompt*

`observe_part_sorts`(p)

yields some *formal description text* according to the following *schema*:

type P_1, P_2, ..., P_n;[10]
value **obs_**P_1: $\mathsf{P}\to\mathsf{P}_1$, **obs_**$\mathsf{P}_2$: $\mathsf{P}\to\mathsf{P}_2$,...,**obs_**$\mathsf{P}_n$: $\mathsf{P}\to\mathsf{P}_n$;[11]

where sorts P_1, P_2, ..., P_n must be disjoint. A proof obligation may need be discharged to secure disjointness.

Sort Models: A part sort is an abstract type. Some part sorts, P, may have a concrete type model, T. Here we consider only two such models: one model is as sets of parts of sort A: T = A-**set**; the other model has parts being of either of two or more alternative, disjoint sorts: T=P1|P2|...|PN. The *domain analysis prompt*: `has_concrete_type`(p) holds if part p has a concrete type. In this case the *domain description prompt*

`observe_concrete_type`(p)

yields some *formal description text* according to the following *schema*,

* either
  type P1, P2, ..., PN, T = $\mathcal{E}$(P1,P2,...,PN)[12]
  value **obs_**T: P → T[13]

  where $\mathcal{E}$(...) is some type expression over part sorts and where P1,P2,...,PN are either (new) part sorts or are auxiliary (abstract or concrete) types[14];
* or:
  type
    T = P1 | P2 | ... | PN[15]
    P_1, P_2, ..., P_n
    P1 :: mkP1(P_1), P2 :: mkP2(P_2), ..., PN :: mkPN(P) [16]
  value
    obs_T: P → T[17]

[10] This RSL **type** clause defines P_1, P_2, ..., P_n to be types.
[11] Thus RSL **value** clause defines n function values. All from type P into some type P_i.
[12] The concrete type definition T = $\mathcal{E}$(P1,P2,...,PN) define type T to be the set of elements of the type expressed by type expression $\mathcal{E}$(P1,P2,...,PN).
[13] **obs_**T is a function from any element of P to some element of T.
[14] The *domain analysis prompt*: `sorts_of`(t) yields a subset of {P1,P2,...,PN}.
[15] A|B is the union type of types A and B.
[16] Type definition A :: mkA(B) defines type A to be the set of elements mkA(b) where b is any element of type B
[17] **obs_**T is a function from any element of P to some element of T.

2.10 Material Observers

Some parts p of sort P may contain material. The *domain analysis prompt* `has_material`(p) holds if composite part p contains one or more materials. The *domain description prompt*

`observe_material_sorts`(p)

yields some *formal description text* according to the following *schema*:

type M_1, M_2, ..., M_m;
value obs_M_1: P → M_1, **obs_**M_2: P → M_2, ..., **obs_**M_m: P → M_m;

where values, m_i, of type M_i satisfy `is_material`(m) for all i; and where M_1, M_2, ..., M_m must be disjoint sorts. *Example 12.* Part Materials: The pipeline parts p pipes, valves, pumps, etc., contains some either liquid material, say crude oil. or gaseous material, say natural gas■

Some material m of sort M may contain parts. The *domain analysis prompt* `has_parts`(m) holds if material m contains one or more parts. The *domain description prompt*

`observe_part_sorts`(m)

yields some *formal description text* according to the following *schema*:

type P_1, P_2, ..., P_n;
value obs_P_1: M→P_1, **obs_**P_2: M→P_2,...,**obs_**P_m: M→P_m;

where values, p_i, of type P_i satisfy `is_part`(p_i) for all i; and where P_1, P_2, ..., P_n must be disjoint sorts. *Example 13.* Material and Part Relations: A global transport system can, for example, be described as primarily containing navigable waters, land areas and air — as three major collections of parts. Navigable waters contain a number of "neighbouring" oceans, channels, canals, rivers and lakes reachable by canals or rivers from other navigable waters (all of which are parts). The part sorts of navigable waters has water materials. All water materials has (zero or more) parts such as vessels and sea-ports. Land areas contain continents, some of which are neighbouring (parts), while some are isolated (that is, being islands not "border–"connected to other continents). Some land areas contain harbour. Harbours and seaports are overlapping parts sharing many attributes. And harbours and seaports are connected to road and rail nets. Etcetera, etcetera■ The above example, Example 2.10, help motivate the concept of mereology (see below).

2.11 Endurant Properties

External and Internal Qualities: We have already, above, treated the following properties of endurants: `is_discrete`, `is_continuous`, `is_atomic`, `is_composite` and `has_material`. We may think of those properties as external qualities. In contrast we may consider the following internal qualities: `has_unique_identifier` (parts), `has_mereology` (parts) and `has_attributes` (parts and materials).

2.12 Unique Identifiers

Without loss of generality we can assume that every part has a unique identifier[18]. A 'unique part identifier' (or just unique identifier) is a further undefined, abstract quantity. If two parts are claimed to have the same unique identifier then they are identical, that is, their possible mereology and attributes are (also) identical • The *domain description prompt*:

`observe_unique_identifier`(p)

yields some *formal description text* according to the following *schema*:

type PI;
value uid_P: P → PI;

Example 14. Unique Identifiers: A road net consists of a set of hubs and a set of links. Hubs and links have unique identifiers. That is: **type** HI, LI; **value uid_**H: H→HI, **uid_**L: L→LI; ■

2.13 Mereology

By 'mereology' [35] we shall understand the study, knowledge and practice of parts, their relations to other parts and "the whole" •

Part relations are such as: two or more parts being connected, one part being embedded within another part, and two or more parts sharing (other) attributes. *Example 15.* Mereology: The mereology of a link of a road net is the set of the two unique identifiers of exactly two hubs to which the link is connected. The mereology of a hub of a road net is the set of zero or more unique identifiers of the links to which the hub is connected ■ The *domain analysis prompt*: `has_mereology`(p) holds if the part p is related to some others parts $(p_a, p_b, \ldots, p_c)$. The *domain description prompt*:

`observe_mereology`(p)

can then be invoked and yields some *formal description text* according to the following *schema*:

type MT = $\mathcal{E}$(PI$_A$,PI$_B$,...,PI$_C$);
value mereo_P: P → MT;

where $\mathcal{E}$(...) is some type expression over unique identifier types of one or more part sorts. Mereologies are expressed in terms of structures of unique part identifiers. Usually mereologies are constrained. Constraints express that a mereology's unique part identifiers must indeed reference existing parts, but also that these mereology identifiers "define" a proper structuring of parts. *Example 16.* Mereology Constraints: We continue our line of examples of road net endurants, cf. Example 2.4 but now a bit more systematically: A road net, n:N, contains

[18] That is, `has_unique_identifier`(p) for all parts p.

a pair, (HS,LS), of sets Hs of hubs h:H and sets Ls of links. The mereology of links must identify exactly two hubs of the road net, the mereology of hubs must identify links of the road net, so connected hubs and links must have commensurate mereologies■ Two parts, p_i:P_i and p_j:P_j, of possibly the same sort (i.e., $P_i \equiv P_j$) are said to 'refer one to another' if the mereology of p_i contains the unique identifier of p_j and vice-versa• The parts p_i and p_j are then said to enjoy 'part overlap'• We refer to the concept of shared attributes covered at the very end of this section.

2.14 Attributes

Attributes are what really endows parts with qualities. The external properties[19] are far from enough to distinguish one sort of parts from another. Similarly with unique identifiers and the mereology of parts. We therefore assume, without loss of generality, that every part, whether discrete or continuous, whether, when discrete, atomic or composite, has at least one attributes.

By a 'part attribute', or just an 'attribute', we shall understand a property that is associated with a part p of sort P, and if removed from part p, that part would no longer be part p but may be a part of some other sort P'; and where that property itself has no physical extent (i.e., volume), as the part may have, but may be measurable by physical means• *Example 17.* Attributes: Some attributes of road net hubs are location, hub state[20], hub state space[21], and of road net links are location, length, link state[22], link state space[23], etcetera■ The *domain description prompt*

`observe_attributes`(p)

yields some *formal description text* according to the following *schema*:

type A_1, A_2, ..., A_n, ATTR;
value **attr_**A_1:P→A_1, **attr_**A_2:P→A_2, ..., **attr_**A_n:P→A_n,
attr_ATTR:P→ATTR;

where **for** ∀ p:P, **attr_**A_i(**attr_**ATTR(p)) ≡ **attr_**A_i(p).

Shared Attributes: A final quality of endurant entities is that they may share attributes. Two parts, p_i:P_i, p_j:P_j, of different sorts are said to enjoy 'shared attributes' if P_i and P_j have at least one attribute name in common• In such cases the mereologies of p_i and p_j are expected to refer to one another, i.e., be 'commensurable'.

[19] `is_discrete`,`is_continuous`,`is_atomic`,`is_compositehas_material`.
[20] Hub state: a set of pairs of unique identifiers of actually connected links.
[21] Hub state space: a set of hub states that a hub states may range over.
[22] Link state: a set of pairs of unique identifiers of actually connected hubs.
[23] Link state space: a set of link states that a link state may range over.

3 A Model of The Analysis & Description Process

3.1 A Summary of Prompts

In the previous section we outlined two classes of prompts: the domain [endurant] analysis prompts:[24]

```
a. is_entity
b. is_endurant
c. is_perdurant
d. is_part
e. is_discrete
f. is_continuous
g. is_atomic
h. is_composite
i. has_concrete_type
j. sorts_of
k. has_material
l. has_parts
m. has_unique_ identifier
n. has_mereology
o. has_attributes
```

and the domain [endurant] description prompts:

```
1. observe_part_sorts
2. observe_concrete_type
3. observe_material_sorts
4. observe_unique_identifier
5. observe_mereology
6. observe_attributes
```

These prompts are imposed upon the domain analyser cum describer. They are "figuratively" applied to the domain. Their orderly, sequenced application follows the method hinted at in the previous section and expressed in a pseudo-formal notation in this section. The notation looks formal but since we have not formalised these prompts it is only pseudo-formal. In [11] we shall formalise these prompts.

3.2 Preliminaries

Let P be a sort, that is, a collection of endurants. By ηP we shall understand a syntactic quantity: the name of P. By ιp:P we shall understand the semantic quantity: an (arbitrarily selected) endurant in P. And by $\eta^{-1}\eta$P we shall understand P. To guide the `TripTych` domain analysis & description process we decompose it into steps. Each step "handles" a sort p:P or a material m:M. Steps handling discovery of composite sorts generate a set of sort names ηP$_1$, ηP$_2$, ..., ηP$_n$ and ηM$_1$, ηM$_2$, ..., ηM$_n$. These are put in a reservoir for sorts to be inspected. The handled sort ηP or ηM is removed from that reservoir. Handling of material sorts concerns only their attributes. Each domain description prompt results in domain specification text (here we show only the formal texts) being deposited in the domain description reservoir, a global variable τ. The clause: `domain_description_prompt(p)` : $\tau := \tau \oplus$ ["text ; "] means that the formal

[24] The prompts are sorted in order of appearence. The one or two digits following the prompt names refer to page numbers minus the number of the first page of this paper + 1.

text "text ; " is joined to the global variable τ where that "text ; " is prompted by domain_description_prompt(p). The meaning of $\oplus$ will be discussed at the end of this section.

3.3 Initialising the Domain Analysis & Description Process

We remind the reader that we are dealing only with endurant domain entities. The domain analysis approach covered in Sect. 2 was based on decomposing an understanding of a domain from the "overall domain" into its components, and these, if not atomic, into their subcomponents. So we need to initialise the domain analysis & description by selecting (or choosing) the domain $\mathcal{A}$.

Here is how we think of that "initialisation" process. The domain analyser & describer spends some time focusing on the domain, maybe at the "white board"[25], rambling, perhaps in an un-structured manner, across its domain, $\mathcal{A}$, and its subdomains. Informally jotting down more-or-less final sort names, building, in the domain analysers' & describers' mind an image of that domain. After some time, doing this, the domain analyser & describer is ready. An image of the domain is in the form of "a domain" endurant, $\delta{:}\mathcal{A}$. Those are the quantities, $\eta\mathcal{A}$ (name of $\mathcal{A}$) [Item 1] and ιp:P (for ($\delta{:}\mathcal{A}$)) [Item 8], referred to below.

Thus this initialisation process is truly a creative one.

3.4 A Domain Analysis & Description State

1. A global variable αps will accumulate all the sort names being discovered.
2. A global variable νps will hold names of sorts yet to be analysed and described.
3. A global variable τ will hold the (so far) generated (in this case only) formal domain description text.

variable
1. αps := [$\eta\mathcal{A}$] ηP**-set** or ηP*
2. νps := [$\eta\mathcal{A}$] (ηP|ηM)**-set** or (ηP|ηM)*
3. τ := [ ] **Text-set** or **Text***

We shall explain the use of [...]s and the operations of \ and $\oplus$ on the above variables in Sect. 3.6.

3.5 Analysis & Description of Endurants

4. To analyse and describe endurants means to first
5. examine those endurant which have yet to be so analysed and described
6. by selecting and removing from νps (Item 11.) an as yet unexamined sort (by name);

[25] Here 'white board' is a conceptual notion. It could be physical, it could be yellow "post-it" stickers, or it could be an electronic conference "gadget".

7. then analyse and describe an endurant entity (ιp:P) of that sort — this analysis, when applied to composite parts, leads to the insertion of zero[26] or more sort names[27];
8. then to analyse and describe the mereology of each part sort,
9. and finally to analyse and describe the attributes of each sort.

value
4. analyse_and_describe_endurants: **Unit** → **Unit**
4. analyse_and_describe_endurants() ≡
5. **while** ∼is_empty(νps) **do**
6. **let** ηS = select_and_remove_ηS() **in**
7. analyse_and_describe_endurant_sort(ιs:S) **end end** ;
8. **for all** ηP • ηP ∈ αps **do** analyse_and_describe_mereology(ιp:P) **end**
9. **for all** ηP • ηP ∈ αps **do** analyse_and_describe_attributes(ιp:P) **end**

The ι of Items 7, 8 and 9 are crucial. The domain analyser is focused on sort S (and P) and is "directed" (by those items) to choose (select) an endurant ιs (ιp) of that sort. The ability of the domain analyser to find such an entity is a measure of that person's professional creativity.

As was indicated in Sect. 2, the mereology of a part may involve unique identifiers of any part sort, hence must be done after all such part sort unique identifiers have been identified. Similarly for attributes which also may involve unique identifiers. Each iteration of analyse_and_describe_endurant_sort(ιp:P) involves the selection of a sort (by name) (which is that of either a part sort or a material sort) with this sort name then being removed.

10. The selection occurs from the global state (hence: ()) and changes that (hence **Unit**).
11. The affected global state component is that of the reservoir, νps.

value
10. select_and_remove_ηS: **Unit** → ηP
10. select_and_remove_ηS() ≡
11. **let** ηS • ηS ∈ νps **in** νps := νps \ {ηS} ; ηS **end**

The analysis and description of all sorts also performs an analysis and description of their possible unique identifiers (if part sorts) and attributes. The analysis and description of sort mereologies potentially requires the unique identifiers of any set of sorts. Therefore the analysis and description of sort mereologies follows that of analysis and description of all sorts.

12. To analyse and describe an endurant

[26] If the sub-parts of p are all either atomic or already analysed, then no new sort names are added to the repository νps.
[27] These new sort names are then "picked-up" for sort analysis &c. in a next iteration of the while loop.

13. is to find out whether it is a part.
14. If so then it is to analyse and describe it as a part,
15. else it is to analyse and describe it as a material.

```
12.  analyse_and_describe_endurant_sort: (P|M) → Unit
12.  analyse_and_describe_endurant_sort(e:(P|M)) ≡
13.    if is_part(e)
13.         assert: is_part(e) ≡ is_endurant(e)∧is_discrete(e)
14.      then analyse_and_describe_part_sort(e:P)
15.      else analyse_and_describe_material_parts(e:M)
12.    end
```

Analysis & Description of Part Sorts:

16. The analysis and description of a part sort
17. is based on there being a set, ps, of parts[28] to analyse —
18. of which an archetypal one, p′, is arbitrarily selected.
19. analyse and describe part p′

```
16.  analyse_and_describe_part_sort: P → Unit
16.  analyse_and_describe_part_sort(p:P) ≡
17.    let ps = observe_parts(p) in
18.    let p′:P • p′ ∈ ps in
19.    analyse_and_describe_part(p′)
16.    end end
```

20. The analysis (&c.) of a part
21. first analyses and describes its unique identifiers.
22. If atomic
23. and
24. if the part embodies materials,
25. we analyse and describe these.
26. If not atomic then the part is composite
27. and is analysed and described as such.

```
20.  analyse_and_describe_part: P → Unit
20.  analyse_and_describe_part(p) ≡
21.    analyse_and_describe_unique_identifier(p) ;
22.    if is_atomic(p)
23.      then
```

[28] We can assume that there is at least one element of that set. For the case that the sort being analysed is a domain Δ, say *"The Transport Domain"*, p′ is some representative *"transport domain"* δ. Similarly for any other sort for which ps is now one of the sorts of δ.

```
24.              if has_materials(p)
25.                 then analyse_and_describe_part_materials(p) end
26.           else assert: is_composite(p)
27.              analyse_and_describe_composite_endurant(p) end
20.       pre: is_discrete(p)
```

We do not associate materials with composite parts.

Analysis & Description of Part Materials:

28. The analysis and description of the material part sorts, one or more, of atomic parts p of sort P containing such materials,
29. simply observes the material sorts of p,
30. that is, generates the one or more continuous endurants
31. and the corresponding observer function text.
32. The reservoir of sorts to be inspected is augmented by the material sorts — except if already previously entered (the $\setminus$ αps clause).

```
28.  analyse_and_describe_part_materials: P → Unit
28.  analyse_and_describe_part_materials(p) ≡
29.      observe_material_sorts(p) :
30.      τ := τ ⊕ [ "type M1,M2,...,Mm;
31.                  value obs_M1:P→M1,obs_M2:P→M2,...,obs_Mm:P→Mm;" ]
32.      νps := νps ⊕  ([ M1,M2,...,Mm ] \ αps)
28.      pre: has_materials(p)
```

Analysis & Description of Material Parts:

33. To analyse and describe materials, m, i.e., continuous endurants,
34. is only necessary if m has parts.
35. Then we observe the sorts of these parts.
36. The identified part sort names update both name reservoirs.

```
33.  analyse_and_describe_material_parts: M → Unit
33.  analyse_and_describe_material_parts(m:M) ≡
34.    if has_parts(m)
35.      then observe_part_sorts(m):
35.           τ := τ ⊕ [" type   P1,P2,...,PN ;
35.                      value  obs_Pi: M→Pi i:{1..N};" ]
36.        ‖ νps := νps ⊕ ([ ηP1,ηP2,...,ηPN ]\ αps)
36.        ‖ αps := αps ⊕ [ ηP1,ηP2,...,ηPN ]
33.      end
33.    assert: is_continuous(m)
```

Analysis & Description of Composite Endurants:

37. To analyse and describe a composite endurant of sort P
38. is to analyse and describe the unique identifier of that composite endurant,
39. then to analyse and describe the sort. If the sort has a concrete type
40. then we analyse and describe that concrete sort type
41. else we analyse and describe the abstract sort.

```
37.  analyse_and_describe_composite_endurant: P → Unit
37.  analyse_and_describe_composite_endurant(p) ≡
38.    analyse_and_describe_unique_identifier(p) ;
39.    if has_concrete_type(p)
40.      then analyse_and_describe_concrete_sort(p)
41.      else analyse_and_describe_abstract_sort(p)
39.    end
```

Analysis & Description of Concrete Sort Types:

42. The concrete sort type being analysed and described
43. is either
44. expressible by some compound type expression
43. or is
45. expressible by some alternative type expression.

```
42.  analyse_and_describe_concrete_sort: P → Unit
42.  analyse_and_describe_concrete_sort(p:P) ≡
44.    analyse_and_describe_concrete_compound_type(p)
43.    ⌈⌉
45.    analyse_and_describe_concrete_alternative_type(p)
42.    pre: has_concrete_type(p)
```

46. The concrete compound sort type
47. is expressible by some simple type expression, T=$\mathcal{E}$(Q,R,...,S) over either concrete types or existing or new sorts Q, R, ..., S.
48. The emerging sort types are identified
49. and assigned to both νps
50. and αps.

```
44.  analyse_and_describe_concrete_compound_type: P → Unit
44.  analyse_and_describe_concrete_compound_type(p:P) ≡
46.    observe_part_type(p):
46.      τ := τ ⊕ [ "type  Q,R,..,S, T = E(Q,R,...,S);
46.                 value obs_T: P → T ;" ] ;
47.    let {Pa,Pb,...,Pc} = sorts_of({Q,R,...,S})
48.          assert: {Pa,Pb,...,Pc} ⊆ {Q,R,...,S} in
49.    νps := νps ⊕ [ ηPa, ηPb, ..., ηPc ] ‖
50.    αps := αps ⊕ ([ ηPa, ηPb, ..., ηPc ] \ αps) end
44.    pre: has_concrete_type(p)
```

51. The concrete alternative sort type expression
52. is expressible by an alternative type expression T=P1|P2|...|PN where each of the alternative types is made disjoint wrt. existing types by means of the description language Pi::mkPi(s_u:P_i) construction.
53. The emerging sort types are identified and assigned
54. to both νps
55. and αps.

```
45.  analyse_and_describe_concrete_alternative_type: P → Unit
45.  analyse_and_describe_concrete_alternative_type(p:P) ≡
51.    observe_part_type(p):
52.      τ := τ ⊕ [ "type  T=P1 | P2 | ... | PN, Pi::mkPi(s_u:P_i) (1≤i≤N);
52.                 value obs_T: P→T ;" ] ;
53.    let {P_a,P_b,...,P_c} = sorts_of({P_i|1≤i≤n})
53.        assert: {P_a,P_b,...,P_c} ⊆ {P_i|1≤i≤n} in
54.    νps := νps ⊕ ([ ηP_a, ηP_b, ..., ηP_c ] \ αps) ‖
55.    αps := αps ⊕ [ ηP_a, ηP_b, ..., ηP_c ] end
42.    pre: has_concrete_type(p)
```

Analysis & Description of Abstract Sorts:

56. To analyse and describe an abstract sort
57. amounts to observe part sorts and to
58. update the sort name repositories.

```
56.  analyse_and_describe_abstract_sort: P → Unit
56.  analyse_and_describe_abstract_sort(p:P) ≡
57.    observe_part_sorts(p):
57.      τ := τ ⊕ [ "type P_1, P_2, ..., P_n;
57.                 value obs_P_i:P→P_i (0≤i≤n);" ]
58.   ‖ νps := νps ⊕ ([ ηP_1, ηP_2, ..., ηP_n ] \ αps)
58.   ‖ αps := αps ⊕ [ ηP_1, ηP_2, ..., ηP_n ]
```

Analysis & Description of Unique Identifiers:

59. To analyse and describe the unique identifier of parts of sort P is
60. to observe the unique identifier of parts of sort P
61. where we assume that all parts have unique identifiers.

```
59.  analyse_and_describe_unique_identifier: P → Unit
59.  analyse_and_describe_unique_identifier(p) ≡
60.    observe_unique_identifier(p):
60.      τ := τ ⊕ [ "type PI; value uid_P:P→PI;" ]
61.    assert: has_unique_identifier(p)
```

Analysis & Description of Mereologies:

62. To analyse and describe a part mereology
63. if it has one
64. amounts to observe that mereology
65. and otherwise do nothing.
66. The analysed quantity must be a part.

62. analyse_and_describe_mereology: P → **Unit**
62. analyse_and_describe_mereology(p) ≡
63. **if** has_mereology(p)
64. **then** observe_mereology(p) :
64. $\tau := \tau \oplus$ "**type** MT = $\mathcal{E}$(PI$_a$,PI$_b$,...,PI$_c$) ;
64. **value mereo_**P: P→MT ;"
65. **else skip end**
62. **pre**: is_part(p)

Analysis & Description of Part Attributes:

67. To analyse and describe the attributes of parts of sort P is
68. to observe the attributes of parts of sort P
69. where we assume that all parts have attributes.

67. analyse_and_describe_part_attributes: P → **Unit**
67. analyse_and_describe_part_attributes(p) ≡
68. observe_attributes(p):
68. $\tau := \tau \oplus$ ["**type** A$_1$, ..., A$_m$;
68. **value attr_**A$_1$:P→A$_1$,...,**attr_**A$_m$:P→A$_m$;"]
69. **assert:** has_attributes(p)

3.6 Discussion of The Model

The above model lacks a formal understanding of the individual prompts as listed in Sect. 3.1. Such an understanding is attempted in [11].

Termination: The sort name reservoir νps is "reduced" by one name in each iteration of the **while** loop of the analyse_and_describe_endurants, cf. Item 6, and is augmented, in each iteration of that loop, by sort names – if not already dispensed of iterations of in earlier itetrations, cf. formula Items 32, 36, 49, 54 and 49. We take it as a dogma that domains contain a finite number of differently typed parts and matyerials. This introduction and removal of sort names and the finiteness of sort names is then the basis for a proper proof of terminaton of the the analysis & description process.

Axioms and Proof Obligations: We have omitted from the above treatment of axioms concerning well-formedness of parts, materials and attributes and proof obligations concerning disjointness of observed part and material sorts and attribute types. A more proper treatment would entail adding a line of proof obligation text right after Item lines 65 and 68, and of axiom text right after Item lines 31, 35, 46, 48, 60, 68, No axiom is needed in connection with Item line 52.

[12] covers axioms and proof obligations in some detail.

Order of Analysis & Description: A Meaning of '$\oplus$': The variables αps, νps and τ are defined to hold either sets or lists. The operator $\oplus$ can be thought of as either set union ($\cup$ and [,]$\equiv${,}) — in which case the domain description text in τ is a set of domain description texts or as list concatenation ($^\frown$ and [,]$\equiv\langle,\rangle$) of domain description texts. The operator $\ell_1 \oplus \ell_2$ now has at least two interpretations: either $\ell_1{}^\frown\ell_2$ or $\ell_2{}^\frown\ell_1$. In the case of lists the $\oplus$ (i.e., $^\frown$) does not (suffix or prefix) append ℓ_2 elements already in ℓ_1. The select_and_remove_ηP function on Page 12 applies to the set interpretation. A list interpretation is:

```
value
6.  select_and_remove_ηP: Unit → ηP
6.  select_and_remove_ηP() ≡
6.      let ηP = hd νps in  νps := tl νps; ηP end
```

In the first case ($\ell_1{}^\frown\ell_2$) the analysis and description process proceeds from the root, breadth first, In the second case ($\ell_2{}^\frown\ell_1$) the analysis and description process proceeds from the root, depth first.

Laws of Description Prompts: The domain 'method' outlined in the previous section suggests that many different orders of analysis & description may be possible. But are they ? That is, will they all result in "similar" descriptions ? That is, if $\mathcal{D}_a$ and $\mathcal{D}_b$ are two domain description prompts where $\mathcal{D}_a$ and $\mathcal{D}_b$ can be pursued in any order will that yield the same description ? And what do we mean by 'can be pursued in any order', and 'same description' ? Let us assume that sort P decomposes into sorts P_a and P_b (etcetera). Let us assume that the domain description prompt $\mathcal{D}_a$ is related to the description of P_a and $\mathcal{D}_b$ to P_b. Here we would expect $\mathcal{D}_a$ and $\mathcal{D}_b$ to commute, that is $\mathcal{D}_a;\mathcal{D}_b$ yields same result as does $\mathcal{D}_b;\mathcal{D}_a$. In [7] we made an early exploration of such laws of domain description prompts.

To answer these questions we need a reasonably precise model of domain prompts. We attempt such a model in [11].

4 Conclusion

Domains can be studied, that is, analysed and described, without any thoughts of possible, subsequent phases of requirements and software development. To study

domains includes, for proper studies. the establishment of domain theories, that is, of theorems about what is being described. This paper does not, unfortunately, show even "top of the iceberg" domain theorems. Such theories are necessary in order to develop a trust in domain desxcriptions. Theorems can then be held up against the actual domain and it can then be checked whether that domain satisfy the theorems. We know that such domain theories can be established as a result of domain modelling. A domain description can be said to be the description of the language spoken by practitioners of the domain, that is, by its stake-holders, hence of a semantics of that language.

4.1 Comparison to Other Work

Domain Analysis: Section 2 outlined the `TripTych`] approach to the analysis & description of domain endurants. We shall now compare that approach to a number of techniques and tools that are somehow related — if only by the term 'domain'!

[1] Ontological and Knowledge Engineering: Ontological engineering [3] build ontologies. Ontologies are *"formal representations of a set of concepts within a domain and the relationships between those concepts"* — expressed usually in some logic. 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. Instead the ontology databases appear to be used for the computerised discovery and analysis of relations between ontologies.

The aim of knowledge engineering was formulated, in 1983, by an originator of the concept, Edward A. Feigenbaum [20]: 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. A seminal text is that of [19]. 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 [15].

The concerns of our form of domain science & engineering is based on that of algorithmic engineering. 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. Our form of domain science & engineering differs from conventional ontological engineering in the following, essential ways: Our domain descriptions rely essentially on a "built-in" upper ontology: types, abstract as well as model-oriented (i.e., concrete) and actions, events and behaviours. Domain science & engineering is not, to a first degree, concerned with modalities, and hence do not focus on the modelling of knowledge and belief, necessity and possibility, i.e., alethic modalities, epistemic modality (certainty), promise and obligation (deontic modalities), etcetera.

[2] Domain Analysis: Domain analysis, or product line analysis (see below) — as it was then conceived in the early 1980s by James Neighbors — is the analysis of related software systems in a domain to find their common and variable parts. It is a model of a wider business context for the system. 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 ([2]) we shall mainly be comparing the TripTych approach to domain analysis to that of Reubén Prieto-Dĩaz's approach [40–42]. Firstly, the two meanings of domain analysis basically coincide. Secondly, in, for example, [40], Prieto-Dĩaz's domain analysis is focused on the very important stages that precede the kind of domain modelling 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 the TripTych 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-modelling 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 languages, he does not show examples of domain descriptions in such DSLs. We, of course, basically use mathematics as the DSL. In the TripTych approach we do not consider requirements, let alone software components, as do Prieto-Dĩaz, but we find that that is not an important issue.

[3] Domain Specific Languages Martin Fowler[29] defines a *Domain-specific language* (DSL) *as a computer programming language of limited expressiveness focused on a particular domain* [21]. Other references are [38, 45]. Common to [45, 38, 21] 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

[29] http://martinfowler.com/dsl.h

they certainly do not describe the domain in terms of that DSL, for example, by formalising the DSL.

[4] Feature-oriented Domain Analysis (FODA): FODA is a domain analysis method which introduced feature modelling 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 [34] states: *"FODA is a necessary first step"* for software reuse. To the extent that domain engineering with its subsequent requirements engineering indeed encourages reuse at all levels: domain descriptions and requirements prescription, we can only agree. Another source on FODA is [18]. Since FODA "leans" quite heavily on 'Software Product Line Engineering' our remarks in that section, next, apply equally well here.

[5] Software Product Line Engineering [SPLE]: SPLE earlier known as domain engineering, is the entire process of reusing domain knowledge in the production of new software systems. Key concerns of SPLE are reuse, the building of repositories of reusable software components, and domain specific languages with which to more-or-less automatically build software based on reusable software components. These are not the primary concerns of our form of domain science & engineering. But they do become concerns as we move from domain descriptions to requirements prescriptions. But it strongly seems that software product line engineering is not really focused on the concerns of domain description — such as is our form of domain engineering. It seems that software product line engineering is primarily based, as is, for example, FODA, on analysing features of software systems. Our [8] puts the ideas of software product lines and model-oriented software development in the context of the TripTych approach.

[6] Problem Frames [PF]: The concept of PF is covered in [32]. 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 PF approach the software developer plays three, that is, all the 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 [31] 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" three stage development approach outlined above.

[7] Domain Specific Software Architectures (DSSA): It seems that the concept of DSSA was formulated by a group of ARPA[30] project "seekers" who also

[30] ARPA: The US DoD Advanced Research Projects Agency

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 [46]. The [46] definition of domain engineering is *"the process of creating a DSSA: domain analysis and domain modelling followed by creating a software architecture and populating it with software components."* This definition is basically followed also by [39, 44, 36]. 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 our requirements development by starting with software components, 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 that requirements can be "derived" systematically from, and related back, formally to domain descriptionss without, in principle, considering software components, 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 prescriptions and that these may strongly hint at shared, (to be) implemented software components; but it may also, as well, be the case two or more requirements prescriptions "derived" from the same domain description may share no software components 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.

[8] Domain Driven Design [DDD] DDD[31] *"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."*[32] We have studied some of the DDD literature, mostly only accessible on the Internet, but see also [29], and find that it really does not contribute to new insight into domains such as we see them: it is just "plain, good old software engineering cooked up with a new jargon.

[9] Unified Modelling Language (UML) Three books representative of UML are [16, 43, 33]. 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 most of the 'approaches' treated in this section, to wit, Items [3–8], with either software design (as it most often is), or requirements prescription. Certainly, in UML, in [16, 43, 33], as well as in most published papers claiming "adherence" to UML, domain analysis usually is manifested in some UML text which "models" some requirements facet. Nothing is necessarily wrong with that, but it is therefore not really our form of domain analysis with its concepts of abstract

[31] Eric Evans: http://www.domaindrivendesign.org/
[32] http://en.wikipedia.org/wiki/Domain-driven_design

representations of endurant and perdurants, and with its distinctions between domain and requirements, and with its possibility of "deriving" requirements prescriptions from domain descriptions. The UML notion of class diagrams is worth relating to our structuring of the domain. Class diagrams appear to be inspired by [2, Bachman, 1969] and [17, Chen, 1976]. It seems that each part sort — as well as other than part (or material) sorts — deserves a class diagram (box), that (assignable) attributes — as well as other non-part (or material) types — are written into the diagram box — as are action signatures — as well as other function signatures. 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.

• • •

Summary of Comparisons: It should now be clear from the above that basically only Jackson's *problem frames* really take the same view of domains 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, [28, 31], 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 and materials.

Domain Analysis and Philosophy: Many readers may have felt somewhat queasy about our definitions of, for example, the notions of domain, entity, endurant, perdurant, discrete, continuous, part and material. Perhaps they thought that these were not proper definitions. Well, the problem is that we are encroaching upon the disciplines of epistemology[33], in particular ontology[34]. Thus we have to thread carefully: On one hand we cannot and do not pretend to formalise philosophical notions. On the other hand we do wish to "get as close to such formalisations as possible" ! In the context of a philosophical inquiry our

[33] Epistemology is the branch of philosophy concerned with the nature and scope of knowledge and is also referred to as "theory of knowledge". It questions what knowledge is and how it can be acquired, and the extent to which any given subject or entity can be known. Much of the debate in this field has focused on analyzing the nature of knowledge and how it relates to connected notions such as truth, belief, and justification[1, 30].

[34] Ontology is the philosophical study of the nature of being, becoming, existence, or reality, as well as the basic categories of being and their relations. Traditionally listed as a part of the major branch of philosophy known as metaphysics, ontology deals with questions concerning what entities exist or can be said to exist, and how such entities can be grouped, related within a hierarchy, and subdivided according to similarities and differences [1, 30].

definitions are acceptable as witnessed by two work on which we draw [37, 22]. In the context of classical computer science they are not. In computer science we would expect precise, mathematical definitions. But that would defeat our purpose, namely to get "as close" to actual domains as possible! So we have opted for a compromise: To keep our 'philosophical-inquiry-acceptable' definitions, while, as in Sect. 3, beginning a journey of formalising such processes of 'philosophical-inquiry-processes'.

4.2 What Have We Achieved

Domain Analysis: In Sect. 2 we have presented a terse, seven+ page, summary of a novel approach to domain analysis. That this approach is different from other 'domain analysis' approaches is argued in [12, Sect. 6.2]. The new aspects are: the distinction between parts and materials, the distinction between external and internal properties (Sect. 2.11), the introduction of the concept of mereologies and the therefrom separate treatment of attributes. It seems to us that "conventional" domain analysis treated all endurant qualities as attributes. The many concepts, endurants and perdurants, discrete and continuous, hence parts and materials, atomic and composite, uniqueness of parts, mereology, and shared attributes, we claim, are forced upon the analysis by the nature of domains: existing in some not necessarily computable reality. In this way the proposed domain analysis & description approach is new.

Methodology: By a 'method' we shall understand a set of principles for selecting and applying techniques and tools in order to analyse and construct an artifact. Section 3 presents a partially instantiated framework for a formal model of a 'method' for domain analysis & description: Some principles are abstraction (sorts in preference for concrete types), separation of concerns (tackling endurants before perdurants), commensurate narratives and formalisations, tackling domain analysis either "top-down", hierarchically from composite endurants, or "bottom-up", compositionally, from atomic endurants, or in some orderly combination of these; etcetera. Some techniques are expressing axioms concerning well-formedness of mereologies and attribute values; stating (and discharging) proof obligations securing disjointness of sorts; etcetera. And some tools are the *domain analysis prompt*s, the *domain description prompt*s and the description language (here RSL [24]). We claim that we have sketched a formalisation of a method for domain analysis and description.

What is really new here is, as for domain analysis, that the analysis & description process is applied to a domain, that is, to our image of that domain, something not necessarily computable, and that our description therefore must not reduce the described domain to a computable artefact.

4.3 Future Work

There remains to conclude studies of, that is, to document and publish treatments of the following related topics: (i) domain analysis of perdurants (actions,

events and behaviours [12, Sect. 5]) — including related *domain analysis prompt*s and *domain description prompt*s[35], (ii) model(s) of prompts[36], (iii) domain facets, cf. [6][37], and (iv) derivation of requirements from domain descriptions, cf. [5][38]. . And there remains to actually establish theories of specific domains.

4.4 Acknowledgements

The author thanks three referees for their careful reading and comments. I think that I have dealt with all their remarks.

5 Bibliography

5.1 Bibliographical Notes

Concerning Sect. 3, A Model of The Analysis & Description Process: we could not find — and were therefore not influenced or inspired by — publications of formalised process models for software development.

5.2 References

1. R. Audi. *The Cambridge Dictionary of Philosophy*. Cambridge University Press, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, England, 1995.
2. C. Bachman. Data structure diagrams. *Data Base, Journal of ACM SIGBDP*, 1(2), 1969.
3. V. Benjamins and D. 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.
4. D. 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.
5. D. Bjørner. From Domains to Requirements. In *Montanari Festschrift*, volume 5065 of *Lecture Notes in Computer Science (eds. Pierpaolo Degano, Rocco De Nicola and José Meseguer)*, pages 1–30, Heidelberg, May 2008. Springer.
6. D. Bjørner. Domain Engineering. In P. Boca and J. Bowen, editors, *Formal Methods: State of the Art and New Directions*, Eds. Paul Boca and Jonathan Bowen, pages 1–42, London, UK, 2010. Springer.
7. D. 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.

[35] See forthcoming [10]
[36] See forthcoming [11]
[37] See forthcoming [9]
[38] See forthcoming [14]

8. D. Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. In *Rainbow of Computer Science, Festschrift for Hermann Maurer on the Occasion of His 70th Anniversary.*, Festschrift (eds. C. Calude, G. Rozenberg and A. Saloma), pages 167–183. Springer, Heidelberg, Germany, January 2011.
9. D. Bjørner. Domain Analysis & Description: Modelling Facets [Writing to begin Summer 2013] (paper[39], slides[40]). Research Report 2013-7, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Summer/Fall 2013. A first draft of this document might be written late summer of 2013.
10. D. Bjørner. Domain Analysis & Description: Perdurants [Writing to begin Summer 2013] (paper[41], slides[42]). Research Report 2013-7, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Summer/Fall 2013. A first draft of this document might be written late summer of 2013.
11. D. Bjørner. Domain Analysis: A Model of Prompts [Writing of crucial final section yet to begin] (paper[43], slides[44]). Research Report 2013-6, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Summer 2013. A first draft of this document will be written over the summer of 2013.
12. D. Bjørner. Domain Analysis (paper[45] slides[46]). Research Report 2013-1, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, April 2013.
13. D. 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).
14. D. Bjørner. On Deriving Requirements from Domain Specifications [Writing to begin Summer 2013] (paper[47], slides[48]). Research Report 2013-8, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Summer/Fall 2013. A first draft of this document might be written late summer of 2013.
15. D. Bjørner and J. F. 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.
16. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
17. P. P. Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Trans. Database Syst*, 1(1):9–36, 1976.
18. K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2000.
19. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts 02142, 1996. 2nd printing.

[39] http://www.imm.dtu.dk/~dibj/da-facets-p.pdf
[40] http://www.imm.dtu.dk/~dibj/da-facets-s.pdf
[41] http://www.imm.dtu.dk/~dibj/perd-p.pdf
[42] http://www.imm.dtu.dk/~dibj/perd-s.pdf
[43] http://www.imm.dtu.dk/~dibj/da-mod-p.pdf
[44] http://www.imm.dtu.dk/~dibj/da-mod-s.pdf
[45] http://www.imm.dtu.dk/~dibj/da-p.pdf
[46] http://www.imm.dtu.dk/~dibj/da-s.pdf
[47] http://www.imm.dtu.dk/~dibj/da-fac-p.pdf
[48] http://www.imm.dtu.dk/~dibj/da-fac-s.pdf

20. E. A. Feigenbaum and P. McCorduck. *The fifth generation*. Addison-Wesley, Reading, MA, USA, 1st ed. edition, 1983.
21. M. Fowler. *Domain Specific Languages*. Signature Series. Addison Wesley, October 20120.
22. C. Fox. *The Ontology of Language: Properties, Individuals and Discourse*. CSLI Publications, Center for the Study of Language and Information, Stanford University, California, ISA, 2000.
23. B. Ganter and R. Wille. *Formal Concept Analysis — Mathematical Foundations*. Springer-Verlag, January 1999. ISBN: 3540627715, 300 pages, Amazon price: US $ 44.95.
24. C. W. George, P. Haff, K. Havelund, A. E. Haxthausen, R. Milne, C. B. Nielsen, S. Prehn, and K. R. Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.
25. C. W. George, A. E. Haxthausen, S. Hughes, R. Milne, S. Prehn, and J. S. Pedersen. *The RAISE Development Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.
26. J. A. Goguen and R. Burstall. Introducing institutions. In E. Clarke and D. Kozen, editors, *Proceedings, Logics of Programming Workshop*, pages 221–256. Springer, 1984. Lecture Notes in Computer Science, Volume 164.
27. J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ . In J. A. Goguen and G. Malcolm, editors, *Software Engineering with OBJ : Algebraic Specification in Action*. Kluwer Press, 2000. Also Technical Report SRI-CSL-88-9, August 1988, SRI International.
28. C. A. Gunter, E. L. Gunter, M. A. Jackson, and P. Zave. A Reference Model for Requirements and Specifications. *IEEE Software*, 17(3):37–43, May–June 2000.
29. D. Haywood. *Domain-Driven Design Using Naked Objects*. The Pragmatic Bookshelf (an imprint of 'The Pragmatic Programmers, LLC.'), http://pragprog.com/, 2009.
30. T. Honderich. *The Oxford Companion to Philosophy*. Oxford University Press, Walton St., Oxford OX2 6DP, England, 1995.
31. M. Jackson. Program Verification and System Dependability. In P. Boca and J. Bowen, editors, *Formal Methods: State of the Art and New Directions*, pages 43–78, London, UK, 2010. Springer.
32. M. A. Jackson. *Problem Frames — Analyzing and Structuring Software Development Problems*. ACM Press, Pearson Education. Addison-Wesley, England, 2001.
33. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
34. 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.
35. E. Luschei. *The Logical Systems of Leśniewksi*. North Holland, Amsterdam, The Netherlands, 1962.
36. N. Medvidovic and E. Colbert. Domain-Specific Software Architectures (DSSA). Power Point Presentation, found on The Internet, Absolute Software Corp., Inc.: `Abs[S/W]`, 5 March 2004.
37. D. H. Mellor and A. Oliver, editors. *Properties*. Oxford Readings in Philosophy. Oxford Univ Press, May 1997. ISBN: 0198751761, 320 pages.
38. M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.

39. E. Mettala and M. 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.
40. R. Prieto-Díaz. Domain Analysis for Reusability. In *COMPSAC 87*. ACM Press, 1987.
41. R. Prieto-Díaz. Domain analysis: an introduction. *Software Engineering Notes*, 15(2):47–54, 1990.
42. R. Prieto-Díaz and G. Arrango. *Domain Analysis and Software Systems Modelling*. IEEE Computer Society Press, 1991.
43. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
44. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
45. D. Spinellis. Notable design patterns for domain specific languages. *Journal of Systems and Software*, 56(1):91–99, Feb. 2001.
46. W. Tracz. Domain-specific software architecture (DSSA) frequently asked questions (FAQ). *Software Engineering Notes*, 19(2):52–56, 1994.

A Pipeline Endurants

Our example is an abstraction of pipeline system endurants. The presentation of the example reflects a rigorous use of the domain analysis & description method outlined in Sect. 2, but is relaxed with respect to not showing all — one could say — intermediate analysis steps and description texts, but following stoichiometry ideas from chemistry makes a few short-cuts here and there. The use of the "stoichiometrical" reductions, usually skipping intermediate endurant sorts, ought properly be justified in each step — and such is adviced in proper, tool-supported industry-scale domain analyses & descriptions.

A.1 Parts

70. A pipeline system contains a set of pipeline units and a pipeline system monitor.
71. The well-formedness of a pipeline system depends on its mereology (cf. Sect. A.2) and the routing of its pipes (cf. Sect. A.3).
72. A pipeline unit is either a well, a pipe, a pump, a valve, a fork, a join, or a sink unit.
73. We consider all these units to be distinguishable, i.e., the set of wells, the set pipe, etc., the set of sinks, to be disjoint.

```
type
70.  PLS′, U, M
71.  PLS = {| pls:PLS′•wf_PLS(pls) |}
value
71.  wf_PLS: PLS → Bool
71.  wf_PLS(pls) ≡ wf_Mereology(pls) ∧ wf_Routes(pls)
70.  obs_Us: PLS → U-set
70.  obs_M: PLS → M
type
72.  U = We | Pi | Pu | Va | Fo | Jo | Si
73.  We :: Well
73.  Pi :: Pipe
```

73. Va :: Valv
73. Fo :: Fork
73. Jo :: Join
73. Si :: Sink

A.2 Part Identification and Mereology

Unique Identification:

74. Each pipeline unit is uniquely distinguished by its unique unit identifier.

type
74. UI
value
74. uid_UI: U → UI
axiom
74. ∀ pls:PLS,u,u′:U•{u,u′}⊆obs_Us(pls)⇒u≠u′⇒uid_UI(u)≠uid_UI(u′)

Unique Identifiers:

75. From a pipeline system one can observe the set of all unique unit identifiers.

value
75. xtr_UIs: PLS → UI-**set**
75. xtr_UIs(pls) ≡ {uid_UI(u)|u:U•u ∈ obs_Us(pls)}

76. We can prove that the number of unique unit identifiers of a pipeline system equals that of the units of that system.

theorem:
76. ∀ pls:PLS•**card** obs_Us(pl)=**card** xtr_UIs(pls)

Mereology:

77. Each unit is connected to zero, one or two other existing input units and zero, one or two other existing output units as follows:
    a. A well unit is connected to exactly one output unit (and, hence, has no "input").
    b. A pipe unit is connected to exactly one input unit and one output unit.
    c. A pump unit is connected to exactly one input unit and one output unit.
    d. A valve is connected to exactly one input unit and one output unit.
    e. A fork is connected to exactly one input unit and two distinct output units.
    f. A join is connected to exactly two distinct input units and one output unit.
    g. A sink is connected to exactly one input unit (and, hence, has no "output").

```
    type
77.   MER = UI-set × UI-set
    value
77.   mereo_U: U → MER
    axiom
77.    wf_Mereology: PLS → Bool
77.    wf_Mereology(pls) ≡
77.      ∀ u:U•u ∈ obs_Us(pls)⇒
77.        let (iuis,ouis) = mereo_U(u) in iuis ∪ ouis ⊆ xtr_UIs(pls) ∧
77.          case (u,(card uius,card ouis)) of
77a.               (mk_We(we),(0,1)) → true,
77b.               (mk_Pi(pi),(1,1)) → true,
77c.               (mk_Pu(pu),(1,1)) → true,
77d.               (mk_Va(va),(1,1)) → true,
77e.               (mk_Fo(fo),(1,1)) → true,
77f.               (mk_Jo(jo),(1,1)) → true,
77g.               (mk_Si(si),(1,1)) → true,
77.                _ → false end end
```

A.3 Part Concepts

An aspect of domain analysis & description that was not covered in Sect. 2 was that of derived concepts. Example pipeline concepts are routes, acyclic or cyclic, circular, etcetera. In expressing well-formedness of pipeline systems one often has to develop subsidiary concepts such as these by means of which well-formedness is then expressed.

Pipe Routes:

78. A route (of a pipeline system) is a sequence of connected units (of the pipeline system).
79. A route descriptor is a sequence of unit identifiers and the connected units of a route (of a pipeline system).

```
    type
78.   R′ = U^ω
78.   R = {| r:Route′•wf_Route(r) |}
79.   RD = UI^ω
    axiom
79.   ∀ rd:RD • ∃ r:R•rd=descriptor(r)
    value
79.   descriptor: R → RD
79.   descriptor(r) ≡ ⟨uid_UI(r[ i ])|i:Nat•1≤i≤len r⟩
```

80. Two units are adjacent if the output unit identifiers of one shares a unique unit identifier with the input identifiers of the other.

```
    value
80.   adjacent: U × U → Bool
80.   adjacent(u,u′) ≡
80.     let (,ouis)=mereo_U(u),(iuis,)=mereo_U(u′) in
80.     ouis ∩ iuis ≠ {} end
```

81. Given a pipeline system, pls, one can identify the (possibly infinite) set of (possibly infinite) routes of that pipeline system.
    a The empty sequence, $\langle\rangle$, is a route of pls.
    b Let u, u' be any units of pls, such that an output unit identifier of u is the same as an input unit identifier of u' then $\langle u, u'\rangle$ is a route of pls.
    c If r and r' are routes of pls such that the last element of r is the same as the first element of r', then $r\hat{}\mathbf{tl}r'$ is a route of pls.
    d No sequence of units is a route unless it follows from a finite (or an infinite) number of applications of the basis and induction clauses of Items 81a–81c.

```
value
81.   Routes: PLS → RD-infset
81.   Routes(pls) ≡
81a.      let rs = ⟨⟩ ∪
81b.            {⟨uid_UI(u),uid_UI(u′)⟩|u,u′:U•{u,u′}⊆obs_Us(pls) ∧ adjacent(u,u′)}
81c.          ∪ {r^tl r′|r,r′:R•{r,r′}⊆rs}
81d.      in rs end
```

Well-formed Routes:

82. A route is acyclic if no two route positions reveal the same unique unit identifier.

```
value
82.  acyclic_Route: R → Bool
82.  acyclic_Route(r) ≡ ∼∃ i,j:Nat•{i,j}⊆inds r ∧ i≠j ∧ r[i]=r[j]
```

83. A pipeline system is well-formed if none of its routes are circular (and all of its routes embedded in well-to-sink routes).

```
value
83.  wf_Routes: PLS → Bool
83.  wf_Routes(pls) ≡
83.      non_circular(pls) ∧ are_embedded_in_well_to_sink_Routes(pls)

83.  non_circular_PLS: PLS → Bool
83.  non_circular_PLS(pls) ≡
83.      ∀ r:R•r ∈ routes(p)∧acyclic_Route(r)
```

84. We define well-formedness in terms of well-to-sink routes, i.e., routes which start with a well unit and end with a sink unit.

```
value
84.  well_to_sink_Routes: PLS → R-set
84.  well_to_sink_Routes(pls) ≡
84.    let rs = Routes(pls) in
84.    {r|r:R•r ∈ rs ∧ is_We(r[1]) ∧ is_Si(r[len r])} end
```

85. A pipeline system is well-formed if all of its routes are embedded in well-to-sink routes.

```
85.  are_embedded_in_well_to_sink_Routes: PLS → Bool
85.  are_embedded_in_well_to_sink_Routes(pls) ≡
85.      let wsrs = well_to_sink_Routes(pls) in
85.      ∀ r:R • r ∈ Routes(pls) ⇒
85.          ∃ r′:R,i,j:Nat •
85.            r′ ∈ wsrs
85.          ∧ {i,j}⊆inds r′∧i≤j
85.          ∧ r = ⟨r′[ k ]|k:Nat•i≤k≤j⟩ end
```

Embedded Routes:

86. For every route we can define the set of all its embedded routes.

```
    value
86. embedded_Routes: R → R-set
86. embedded_Routes(r) ≡
86.     {⟨r[ k ]|k:Nat•i≤k≤j⟩ | i,j:Nat• i {i,j}⊆inds(r) ∧ i≤j}
```

A Theorem:

87. The following theorem is conjectured:
    a the set of all routes (of the pipeline system)
    b is the set of all well-to-sink routes (of a pipeline system) and
    c all their embedded routes

```
    theorem:
87.  ∀ pls:PLS •
87.  let rs = Routes(pls),
87.      wsrs = well_to_sink_Routes(pls) in
87a.  rs =
87b.      wsrs ∪
87c.      ∪ {{r′|r′:R • r′ ∈ embedded_Routes(r″)} | r″:R • r″ ∈ wsrs}
86.  end
```

A.4 Materials

88. The only material of concern to pipelines is the gas[49] or liquid[50] which the pipes transport[51].

```
    type
88.     GoL
    value
88.     obs_GoL: U → GoL
```

[49] Gaseous materials include: air, gas, etc.
[50] Liquid materials include water, oil, etc.
[51] The description of this document is relevant only to gas or oil pipelines.

A.5 Attributes

Part Attributes:

89. These are some attribute types:
    a estimated current well capacity (barrels of oil, etc.),
    b pipe length,
    c current pump height,
    d current valve open/close status and
    e flow (e.g., volume/second).

type
89a. WellCap
89b. LEN
89c. Height
89d. ValSta == open | close
89e. Flow

90. Flows can be added (also distributively) and subtracted, and
91. flows can be compared.

value
90. $\oplus, \ominus$: Flow×Flow → Flow
90. $\oplus$: Flow-**set** → Flow
91. $<,\leq,=,\neq,\geq,>$: Flow × Flow → **Bool**

92. Properties of pipeline units include
    a estimated current well capacity (barrels of oil, etc.),
    b pipe length,
    c current pump height,
    d current valve open/close status,
    e current $\mathcal{L}$aminar in-flow at unit input,
    f current $\mathcal{L}$aminar in-flow leak at unit input,
    g maximum $\mathcal{L}$aminar guaranteed in-flow leak at unit input,
    h current $\mathcal{L}$aminar leak unit interior,
    i current $\mathcal{L}$aminar flow in unit interior,
    j maximum $\mathcal{L}$aminar guaranteed flow in unit interior,
    k current $\mathcal{L}$aminar out-flow at unit output,
    l current $\mathcal{L}$aminar out-flow leak at unit output,
    m maximum guaranteed $\mathcal{L}$aminar out-flow leak at unit output.

value
92a. attr_WellCap: We → WellCap
92b. attr_LEN: Pi → LEN
92c. attr_Height: Pu → Height
92d. attr_ValSta: Va → VaSta
92e. attr_In_Flow$_{\mathcal{L}}$: U → UI → Flow
92f. attr_In_Leak$_{\mathcal{L}}$: U → UI → Flow
92g. attr_Max_In_Leak$_{\mathcal{L}}$: U → UI → Flow

92h. attr_body_Flow$_{\mathcal{L}}$: U → Flow
92i. attr_body_Leak$_{\mathcal{L}}$: U → Flow
92j. attr_Max_Flow$_{\mathcal{L}}$: U → Flow
92k. attr_Out_Flow$_{\mathcal{L}}$: U → UI → Flow
92l. attr_Out_Leak$_{\mathcal{L}}$: U → UI → Flow
92m. attr_Max_Out_Leak$_{\mathcal{L}}$: U → UI → Flow

A.6 Flow Laws

93. "What flows in, flows out !". For $\mathcal{L}$aminar flows: for any non-well and non-sink unit the sums of input leaks and in-flows equals the sums of unit and output leaks and out-flows.

Law:

```
93.  ∀ u:U\We\Si •
93.      sum_in_leaks(u) ⊕ sum_in_flows(u) =
93.      attr_body_Leak_L(u) ⊕
93.      sum_out_leaks(u) ⊕ sum_out_flows(u)
```

```
value
    sum_in_leaks: U → Flow
    sum_in_leaks(u) ≡
        let (iuis,) = mereo_U(u) in
        ⊕ {attr_In_Leak_L(u)(ui)|ui:UI•ui ∈ iuis} end
    sum_in_flows: U → Flow
    sum_in_flows(u) ≡
        let (iuis,) = mereo_U(u) in
        ⊕ {attr_In_Flow_L(u)(ui)|ui:UI•ui ∈ iuis} end
    sum_out_leaks: U → Flow
    sum_out_leaks(u) ≡
        let (,ouis) = mereo_U(u) in
        ⊕ {attr_Out_Leak_L(u)(ui)|ui:UI•ui ∈ ouis} end
    sum_out_flows: U → Flow
    sum_out_flows(u) ≡
        let (,ouis) = mereo_U(u) in
        ⊕ {attr_Out_Leak_L(u)(ui)|ui:UI•ui ∈ ouis} end
```

94. "What flows out, flows in !". For $\mathcal{L}$aminar flows: for any adjacent pairs of units the output flow at one unit connection equals the sum of adjacent unit leak and in-flow at that connection.

Law:

```
94.  ∀ u,u′:U•adjacent(u,u′) ⇒
94.      let (,ouis)=mereo_U(u), (iuis′,)=mereo_U(u′) in
94.      assert: uid_U(u′) ∈ ouis ∧ uid_U(u) ∈ iuis ′
94.      attr_Out_Flow_L(u)(uid_U(u′)) =
94.      attr_In_Leak_L(u)(uid_U(u))⊕attr_In_Flow_L(u′)(uid_U(u)) end
```