Domain Analysis and Description

Formal Models of Processes and Prompts

Dines Bjørner

Fredsvej 11, DK–2840 Holte, Denmark

November 18, 2016: 14:15

1. Introduction

- The presentation of a calculus for analysing and describing manifest domains,
  - introduced in [*Manifest Domains: Analysis & Description*]
  - and summarised in Sect. 2.,

  was and is necessarily informal.
- In the present seminar
  - we shall provide a formal, operational semantics formalisation
  - of the analysis and description calculus.

- There are two aspects to the semantics of the analysis and description calculus.
  - There is the formal explanation of the process of applying the analysis and description prompts,
    - in particular the practical meaning[1]
    - of the results of applying the analysis prompts, and
  - there is the formal explanation of the meaning
  - of the results of applying the description prompts.

[1] in contrast to a formal mathematical meaning

- The former (i.e., the practical meaning of the results of applying the analysis prompts)
  - amounts to a model of the process whereby the domain analyser cum describer
    - navigates "across" the domain, alternating between
    - applying sequences of one or more analysis prompts and
    - applying description prompts.

- The latter (formal explanation of the meaning of the results of applying the description prompts) amounts to a model of
  - the domain (as it evolves in the mind of the analyser cum describer[2]),
  - the meaning of the evolving description, and thereby
  - the relation between the two.

[2]By 'domain analyser cum describer' we mean a group of one or more professionals, well-educated and trained in the domain analysis & description techniques outlined in, for example, [Bjø16b], and where these professionals work closely together. By 'working closely together' we mean that they, together, day-by-day work on each their sections of a common domain description document which they "buddy check", say every morning, then discuss, as a group, also every day, and then revise and further extend, likewise every day. By "buddy checking" we mean that group member $\mathcal{A}$ reviews group member $\mathcal{B}$'s most recent sections – and where this reviewing alternates regularly: $\mathcal{A}$ may first review $\mathcal{B}$'s work, then $\mathcal{C}$'s, etcetera.
We shall, occasionally refer to the 'domain analyser cum describer' as the 'domain engineer'.

1.1. The Triptych Approach to Software Development

- Before software can be designed and coded
- one must have firm understanding of its requirements.
- Before requirements can be prescribed
- one must have a clear grasp of the application domain.

Definition 1. The Triptych Approach to Software Development: *By a* **triptych software development**

- *we shall understand a development which, in principle,*
- *starts with*
  - *either studying an existing*
  - *or developing a new domain description,*
- *then proceeds to systematically deriving a requirements prescription from the domain description, and*
- *finally designs and codes the software from the requirements prescription* ■

1.2. Method and Methodology

Definition 2. **Method**: *By a* **method** *we shall understand*

- *a set of* **principles** *for selecting and applying*
- *a number of* **techniques** *and* **tools**
- *for* **analysing** *and* **synthesizing**
- *an artifact* ■

Definition 3. **Methodology**: *By* **methodology** *we shall understand*

- *the study and knowledge of one or more methods* ■

Definition 4. **Formal Method**: *By* **formal method** *we shall understand*

- *a method some or most of whose techniques and tools*
- *can be understood mathematically* ■

Definition 5. Formal Software Development: *By a* **formal software development method** *we shall understand*

- *a formal method where domain descriptions, requirements prescriptions and software designs*
- *are expressed in mathematically founded specification languages*
- *with the possibility of proving properties*
  - *of these specifications,*
  - *of steps and stages of development*
    - *(refinements within domain descriptions, requirements prescriptions, software designs and between these)*
    - *— properties such as correctness of software designs with respect to requirements,*
  - *and satisfaction of user expectations (from software) with respect to domains* ■

- This paper deals with some of the triptych method principles and techniques for developments of domain descriptions.
  - The paper puts forward a formal explanation of some of that method.

1.3. Related Work

- To this author's knowledge
  - there are not many papers, other than the author's own,
  - [Bjø16b], [Bjø17b], [Bjø17c], [Bjø16a] and the present paper,
  - which proposes a calculus of analysis and description prompts
  - for capturing a domain,
  - let alone, as this paper tries,
  - to formalise aspects of this calculus.

- There is, however a "school of software engineering", "anchored" in the 1987 publication: [Ost87, Leon Osterweil].
  - As the title of that paper reveals: *"Software Processes Are Software Too"*
  - the emphasis is on considering the software development process
  - as prescribable by a software program.
  - That is not what we are aiming at.
  - We are aiming at an abstract and formal description of a large class of domain analysis & description processes *in terms of possible development calculi*.
  - And in such a way that one can reason about such processes.

- The Osterweil paper suggests that any particular software development can be described by a program,
- and, if we wish to reason about the software development process
- we must reason over that program,
- but there is no requirement that the "software process programs" be expressed in a language with a proof system.[3]
- In contrast we can reason
  - over the properties of the development calculi as well as
  - over the resulting description.

[3] The **RAISE** **S**pecification **L**anguage [GHH+95] does have a proof system.

- There is another "school of programming", one that more closely adheres to the use of a calculus [BAvWS98, Mor90].
  - The calculus here is a set of refinement rules,
  - a *Refinement Calculus*[4],
  - that "drives" the developer
  - from a specification
  - to an executable program.

[4] Ralph–Johan Back appears to be the first to have proposed the idea of refinement calculi, cf. his 1978 PhD thesis *On the Correctness of Refinement Steps in Program Development*, http://users.abo.fi/backrj/index.php?page=Refinement calculus all.html&menu=3.

- Again, that is not what we are doing here.
  - The proposed calculi of analysis and of description prompts [Bjø16b]
  - "drives" the domain engineer in developing a domain description.
  - That description may then be 'refined' using a refinement calculus.

1.4. Structure of Seminar

- Section 2. provides a terse summary of the analysis & description of endurants.
  - It is without examples.
  - For such we refer to [Bjø16b].
- Section 3. is informal.
  - It discusses issues of syntax and semantics.
  - The reason we bring this short section is that the current paper turns "things upside/down":
    - from semantics we extract syntax!
    - From the real entities of actual domains we extract domain descriptions.

- Section 4. presents a
  - pseudo-formal operational semantics explication of the process of proceeding through iterated sequences of analysis prompts to description prompts.
  - The formal meaning of these prompts are given in Sect. 7.
- But first we must "prepare the ground":
  - The meaning of the analysis and description prompts
  - is given in terms of some formal "context" in which
  - the domain engineer works.

- Section 5.
  - discusses this notion of "image" —
  - an informal aspect of the 'context'.
  - It is a brief discussion.
- Section 6.
  - presents the formal aspect of the 'context':
  - perceived abstract syntaxes
  - of the ontology of domain endurants
  - and of endurant values.

- Section 7.
  - Discusses, in a sense, the mental processes
  - – *from syntax to semantics and back again !* –
  - that the domain engineer appears to undergo
  - while analysing (the semantic) domain entities
  - and synthesizing (the syntactic) domain descriptions.
- Section 8. presents the analysis and description prompts meanings.
  - It represents a high point of this paper.
  - It so-to-speak justifies the whole "exercise" !

- Section 9. concludes the paper.
  - We summarize what we have "achieved".
  - And we discuss whether this "achievement" is a valid one!
- Appendix A. details some formalisations of a "standard" nature.
- Appendix B. brings a "full" example of a domain description.
  - It is that of the essence of a credit card system.

2. Domain Analysis and Description

- In the rest of this seminar we shall consider entities in the context of their being manifest (i.e., spatio-temporal).
- The restrictions of what we cover with respect to [*Manifest Domains: Analysis & Description*] are:
  - we do not cover perdurants, only endurants,
  - and within endurants we do not cover
    - update mereology,
    - update attributes and
    - shared attributes.

- These omissions do not affect the main aim of this seminar,
  - namely that of presenting a plausible example of how one might wish to operationally formalise the notions of
  - the analysis & description process and of
  - the analysis & description prompts.
- The presentation is very terse.
- We refer to [Bjø16b] for details.
- Appendix (Pages 186–222) gives an "full" example of a "smallish" domain, including perdurants.

2.1. General

- In [*Manifest Domains: Analysis & Description*]
- we developed an ontology for structuring and a prompt calculus analysing and describing domains.
- Figure 1 on the following slide captures the ontology structure.[5]
- It is thus a slight simplification of the 'upper ontology' figure given in [*Manifest Domains: Analysis & Description*] in that it omits the **component** ontology.
- The rest of this section will summarise the calculus.
- We refer to [*Manifest Domains: Analysis & Description*] for examples.

[5] The differences, in Fig. 1, with respect to that of [Bjø16b], are: (i) we have "collapsed" the `is_continuous` and the `is_material` nodes of [Bjø16b] into one here, and (ii) we omit details on attribute categories.

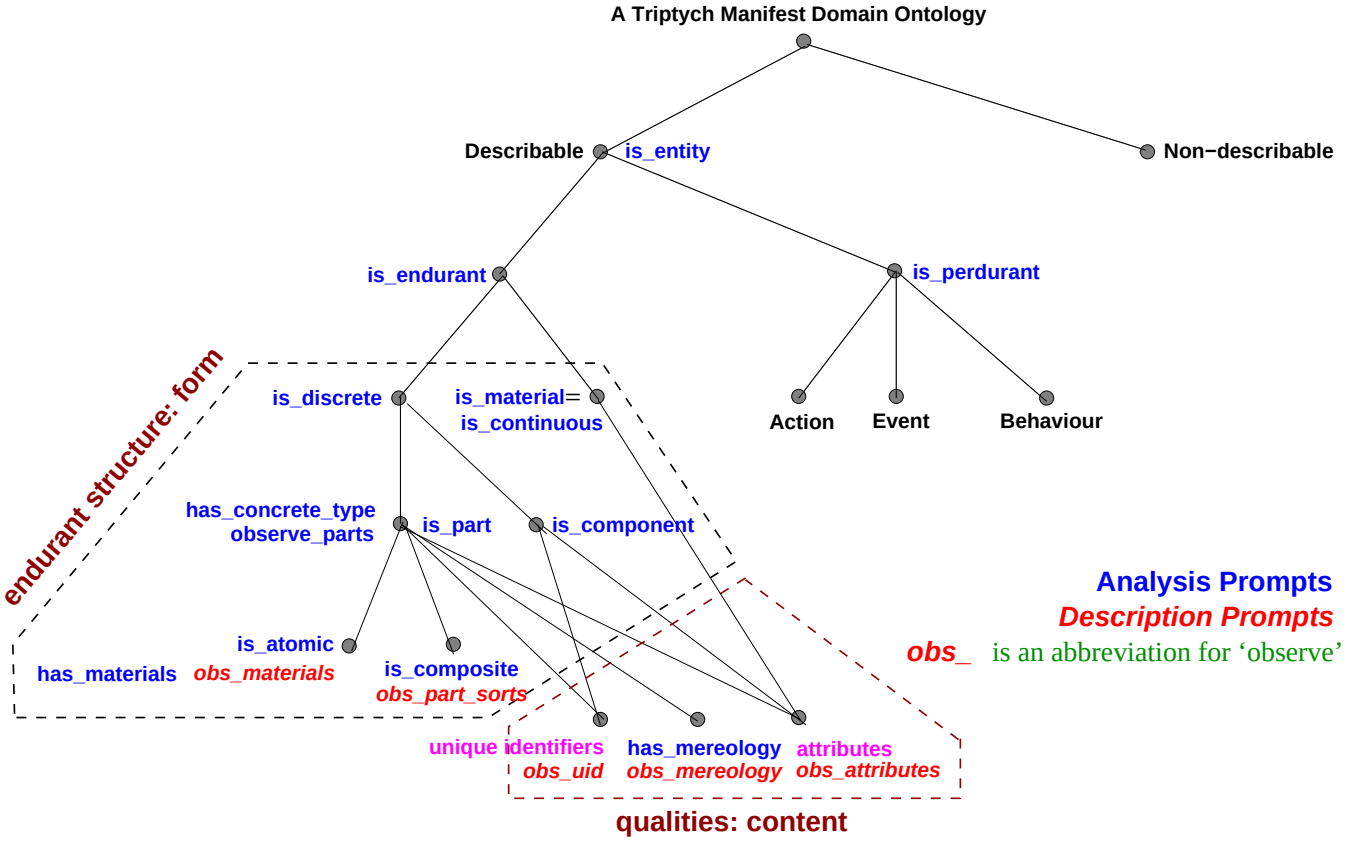

Figure 1: An Annotated Upper Ontology

- To the nodes of the upper ontology of Fig. 1 on the previous slide we have affixed some names.
  - Names beginning with a capital stand for sub-ontologies.
  - Names starting with a slanted *obs_* stand for description prompts.
  - Other names (starting with an `is_` or a `has_`, or other) stand for analysis prompts.[6]

[6] In a coloured version of this document the description prompts are coloured red and the analysis prompts are coloured blue.

2.2. Entities

Definition 6. **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* ■[7]

[7] **Definitions** and **examples** are delimited by ■ respectively ■

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* ■[8]

[8] **Analysis** prompt definitions and **description** prompt definitions and schemes are delimited by ■ respectively ■.

- Although "reasonably" precise, the definition of the concept of **entity** is still not precise enough for us to formalise it.
  - In Sect. we attempt a series of formalisations of the analysis prompts.
  - This is done on the background of some formalisation (Sect.) of the ontology being unfolded in this section (i.e., Sect.).
  - A formalisation that covers the notion of phenomena and entities is not offered.

2.3. Endurants and Perdurants

Definition 7. **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* ■

Definition 8. 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,
  - *where we to freeze time we would only see or touch*
    a fragment of the perdurant ■

Analysis Prompt 2 . *is_endurant:*

- *The domain analyser analyses an entity, ϕ, into an endurant as prompted by the* **domain analysis prompt**:
  - `is_endurant` — *e is an endurant if* `is_endurant(e)` *holds.*
- `is_entity` *is a* **prerequisite prompt** *for* `is_endurant` ■

Analysis Prompt 3 . *is_perdurant:*

- *The domain analyser analyses an entity ϕ into perdurants as prompted by the* **domain analysis prompt**:
  - `is_perdurant` — *e is a perdurant if* `is_perdurant(e)`[9] *holds.*
- `is_entity` *is a* **prerequisite prompt** *for* `is_perdurant` ■

[9] Since we do not cover perdurants in this paper we shall also refrain from trying to formalise this prompt.

2.4. Discrete and Continuous Endurants

Definition 9. Discrete Endurant:

- *By a* **discrete endurant** *we shall understand an endurant which is*
  - *separate,*
  - *individual or*
  - *distinct*

  in form or concept ■

Definition 10. Continuous Endurant:

- *By a* **continuous endurant** *we shall understand an endurant which is*
  - *prolonged, without interruption,*
  - *in an unbroken series or pattern* ■

Analysis Prompt 4 . *is_discrete:*

- *The domain analyser analyse 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 analyse endurants e into continuous entities as prompted by the* **domain analysis prompt**:
  - ***is_continuous*** — *e is continuous if* ***is_continuous(e)*** *holds* ■

2.5. Parts, Components and Materials

2.5.1. General

Definition 11. **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* ■

Definition 12. Component:

- *By a* **component** *we shall understand*
  - *a discrete endurant*
  - *which the domain engineer chooses*
  - *to* ***not*** *endow with* **internal qualities** *such as*
    - *unique identification,*
    - *mereology, and, even perhaps*
    - *no attributes* ■

Definition 13. Material:

- *By a* **material** *we shall understand a continuous endurant* ■

2.5.2. Part, Component and Material 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* ■

Analysis Prompt 7 . *is_component:*

- *The domain analyser analyse endurants e into part entities as prompted by the* **domain analysis prompt***:*
  - ⟴ `is_component` — *e is a component if* `is_component(e)` *holds* ■

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* ■
- There is no difference between `is_continuous` and `is_material`,
  - that is `is_continuous` ≡ `is_material`.
  - We shall henceforth use `is_material`.

2.6. Atomic and Composite Parts

Definition 14. Atomic Part:

- **Atomic part**s *are those which,*
  - *in a given context,*
  - *are deemed to not consist of meaningful, separately observable proper* ***sub-parts*** ■
- A **sub-part** is a part ■

Definition 15. Composite Part:

- **Composite part**s *are those which,*
  - *in a given context,*
  - *are deemed to indeed consist of meaningful, separately observable proper* ***sub-parts*** ■

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* ■

2.7. On Observing Part Sorts

2.7.1. Part Sort Observer Functions

Domain Description Prompt 1 . `observe_part_sorts`:

- *If* `is_composite`(p) *holds, then the analyser "applies" the description language observer prompt*

  $\circledast$ `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(p:P) schema

Narration:

[s] ... narrative text on sorts ...
[o] ... narrative text on sort observers ...
[p] ... narrative text on proof obligations ...

Formalisation:

type
[s] P_1, P_2, ..., P_n
value
[o] **obs_part**_P_i: $\mathsf{P} \to \mathsf{P}_i$ [$1 \leq i \leq m$]
proof obligation [Disjointness of part sorts]
[p] $\mathcal{D}$

2.7.2. On Discovering Concrete Part Types

Analysis Prompt 11 . *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* ■

- Many possibilities offer themselves to model a concrete type as:
  - either a set of abstract sorts,
  - or a list of abstract sorts,
  - or any compound of such sorts.
- Without loss of generality we suggest,
  - as concrete type, as set of sorts.
- We have modeled many domains.
- So far, only the set concrete type has been needed.

Domain Description Prompt 2 . `observe_concrete_type`:

- *Then the domain analyser applies the* **domain description prompt**:

  ◈ `observe_concrete_type`(p)

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

2. observe_concrete_type(p:P) schema

Narration:

[t_1] ... narrative text on types ...
[t_2] ... narrative text on types ...
[o] ... narrative text on type observers ...

Formalisation:

type
[t_1] Q
[t_2] T = Q-**set**
value
[o] **obs_part**_T: P → T

- *Q may be any part sort;*
- *has_concrete_type is a* **prerequisite prompt** *of observe_part_type* ■

2.7.3. 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 ids`,
  - `mereology` and
  - `attributes`.
- By **part qualities** we mean the sum total of
  - `external endurant` and
  - `internal endurant`

  qualities.

2.8. Unique Part Identifiers

- We assume that all parts and components have unique identifiers.
  - It may be, however, that we do not always need to define such a part or component identifier.

Domain Description Prompt 3 .
observe_unique_identifier:

- *We can, however, always apply the* **domain description prompt**:
  - *observe_unique_identifier(pk)*
- *to parts,* p:P, *or components,* k, *resulting in the analyser writing down the* unique identifier type and observer *domain description text according to the following schema:*

3. observe_unique_identifier(pk:(P|K)) schema

Narration:

[s] ... narrative text on unique identifier sort ...
[u] ... narrative text on unique identifier observer ...
[a] ... axiom on uniqueness of unique identifiers ...

Formalisation:

type
[s] PI, KI
value
[u] **uid**_P: P → PI
[u] **uid**_K: K → KI
axiom
[a] $\mathcal{U}$

2.9. Mereology

2.9.1. Part Mereology: Types and Functions

Analysis Prompt 12 . *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.*

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(p)`
  - *to parts of that type*
  - *and write down the* mereology types and observers *domain description text according to the following schema:*

4. observe_mereology(p:P) 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 = $\mathcal{E}$(PI1,PI2,...,PIm)
value
[m] **obs_mereo**_P: P → MT
axiom [Well−formedness of Domain Mereologies]
[a] $\mathcal{A}$

- *MT is a type expression over unique part identifiers.*
- $\mathcal{A}$ *is some predicate over unique part identifiers.*
- *The* PI_i *are unique part identifier types* ■

2.10. Part, Material and Component Attributes

Domain Description Prompt 5 . `observe_attributes`:

- *The domain analyser experiments, thinks and reflects about attributes of endurants (parts p:P, components, k:K, or materials, m:M).*
- *That process is initiated by the* **domain description prompt**:

  ◈ `observe_part_attributes`(e).

- *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_part_attributes(e:(P|K|M)) schema

Narration:

[t] ... narrative text on attribute sorts ...
[o] ... narrative text on attribute sort observers ...
[p] ... narrative text on attribute sort proof obligations ...

Formalisation:

type
[t] A_1, A_2, ..., A_n
value
[o] **attr**_A_i:(P|K|M)→A_i [$1 \leq i \leq n$]
proof obligation [Disjointness of Attribute Types]
[p] $\mathcal{A}$

- *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.*
- $\mathcal{A}$ *is a predicate over attribute types* A_1, A_2, ..., A_n.
- *It expresses their Disjointness* ■

2.11. Components

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

Analysis Prompt 13 . *has_components:*

- *The* **domain analysis prompt***:*
  - ⦻ *has_components*(p)
- *yields* **true** *if atomic part* p *potentially contains components otherwise false* ■

Domain Description Prompt 6 . `observe_component_sort`:

- *The* **domain description prompt**:

  ◈ `observe_component_sort`(p)

  yields the part component sorts and component observers *domain description text according to the following schema:*

6. observe_component_sort(p:P) schema

Narration:

[s] ... narrative text on component sort ...

[o] ... narrative text on component sort observer ...

Formalisation:

type

[s] K

value

[o] **obs_comps**: P → K-set

- *Components have unique identifiers and attributes, but no mereology* ■

2.12. Materials

- Only atomic parts may contain materials and
- materials may contain [atomic] parts.

2.12.1. Part Materials

- Let $p{:}P$ be some atomic part.

Analysis Prompt 14 . *has_material:*

- *The* **domain analysis prompt**:
  - ⦻ *has_material*(p)
- *yields* **true** *if the atomic part* $p{:}P$ *potentially contains a material otherwise false* ■

Domain Description Prompt 7 . *observe_material_sort*:

- *The* **domain description prompt**:

  ◈ *observe_material_sort(p)*

  yields the part material sort and material observer *domain description text according to the following schema:*

7. observe_material_sort(p:P) schema

Narration:

[s] ... narrative text on material sort ...

[o] ... narrative text on material sort observer ...

Formalisation:

type

[s] M

value

[o] **obs_mat**_M: P → M

2.12.2. Material Parts

- Materials may contain parts.
- We assume that such parts are always atomic and always of the same sort.
- **Example:**
  - Pipe parts usually contain oil material.
  - And that oil material may contain pigs which are parts whose purpose it is to clean and inspect (i.e., maintain) pipes ■

Analysis Prompt 15 . *has_parts:*

- *The* **domain analysis prompt***:*
  - ⟐ *has_parts*(*m*)
- *yields* **true** *if material m:M*
  potentially contains parts
  otherwise false ■

Domain Description Prompt 8 .

observe_material_part_sorts:

- *The* **domain description prompt**:
  - ⦚ *observe_material_part_sort*(*e*)

  yields the material part sorts and material part observers *domain description text according to the following schema:*

8. observe_material_part_sorts(m:M) schema

Narration:

[s] ... narrative text on material part sort ...

[o] ... narrative text on material part sort observer ...

Formalisation:

type

[s] mP

value

[o] **obs_mat**_mP: M → mP

2.13. Components and Materials

- Experimental evidence[10] appears to justify the following "limitations":
  - only atomic parts may contain
    - either at most one material, and always of the same sort,
    - or a set of zero, one or more components, all of the same sort;

    but not both;
  - materials need not be characterised by unique identifiers; and
  - components and materials need not be endowed with mereologies.

[10]— in the form of more than 20 medium-to-large scale domain models

2.14. Discussion

- We have covered the analysis and description calculi for endurants.
- We omit covering analysis and description techniques and tools for perdurants.

3. Syntax and Semantics
3.1. Form and Content

- Section appears to be expressed in the syntax of the **R**aise [GHH+95] **S**pecification **L**anguage, **RSL** [GHH+92].
  - But it only "appears" so.
  - When, in the "conventional" use of **RSL**, we apply meaning functions, we apply them to syntactic quantities.
  - In Sect. the "meaning" functions are the analysis, a.–o., and description, [1]–[8], prompts:

- The quantities that these prompts are "applied to"
  - are semantic ones,
  - in effect, they are the "ultimate" semantic quantities
  - that we deal with:
  - *the real, i.e., actual domain* entities!
- The quantities that these prompts "yield" are syntactic ones!

- That is, we have "turned matters inside/out".
- From semantics we "extract" syntax.
- The arguments of the above-listed 23 prompts are domain entities, i.e., in principle, in-formalisable things.
  - Their types, typically listed as P, denote possibly infinite classes, $\mathcal{P}$, of domain entities.
  - When we write P we thus mean $\mathcal{P}$.

3.2. Syntactic and Semantic Types

- When we, classically, define a programming language,
  - we first present its syntax, then it semantics.
  - The latter is presented as two – or three – possibly interwoven texts:
    - the static semantics, i.e., the well-formedness of programs,
    - the dynamic semantics, i.e., the mathematical meaning of programs —
    - with a corresponding proof system being the "third texts".
  - We shall briefly comment on the ideas of static and dynamic semantics.

- In designing a programming language, and therefore also in narrating and formalising it, one is well advised in
  - deciding first on the semantic types,
  - then on the syntactic ones.
- With describing [f.ex., manifest] domains, matters are the other way around:
  - The semantic domains are given in the form of the endurants and perdurants;
  - and the syntactic domains are given in the form that we, the humans of the domain, mention in our speech acts [Sea69, Aus76].

- That is, from a study of actual life domains,
  - we extract the essentials that speech acts deal with
  - when these speech acts are concerned with
  - performing or talking about
  - entities in some actual world.

3.3. Names and Denotations

- Above, we may have been somewhat cavalier with the use of names for sorts and names for their meaning.
  - Being so, i.e., "cavalier", is, unfortunately a "standard" practice.
  - And we shall, regrettably, continue to be cavalier,
    - i.e., "loose" in our use of names of syntactic "things"
    - and names for the denotation of these syntactic "things".
  - The context of these uses
  - usually makes it clear which use we refer to:
  - a syntactic use or a semantic one.
- As from Sect. 6. we shall be more careful distinguishing clearly between
  - the names of sorts and
  - the values of sorts,

  i.e., between syntax and semantics.

4. A Model of the Domain Analysis & Description Process

4.1. Introduction

4.1.1. A Summary of Prompts

- In Sect. 3.1 we listed the two classes of prompts:
  - the domain [endurant] analysis prompts:
  - and the domain [endurant] description prompts:

- These prompts are "imposed" upon the domain by 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,
  - detailed in [*Manifest Domains: Analysis & Description*], and
  - exemplified in Appendix .
- This process of application of prompts will be 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.
- We formalise these prompts in Sect. 7.

4.1.2. Preliminaries

- Let P be a sort, that is, a collection of endurants.
- By P we shall understand both a syntactic quantity: the name of P, and a semantic quantity, the type (of all endurant values of type) P.
- By ιp:P we shall understand a semantic quantity: an (arbitrarily selected) endurant in P.

- To guide our analysis & description process we decompose it into steps.
  - Each step "handles" a part sort p:P or a material sort m:M or a component sort k:K.
  - Steps handling discovery of composite part sorts generates a set of part sort names P_1, P_2, ..., P_n:PNm.
  - Steps handling discovery of atomic part sorts may generate a material sort name, m:MNm, or component sort name, k:KNm.

- The part, material and component sort names are put in a reservoir for *sorts to be inspected.*
- Once handled, the sort name is removed from that reservoir.
- Handling of material sorts besides discovering their attributes may involve the discovery of further part sorts — which we assume to be atomic.
- Each domain description prompt results in domain specification text (here we show only the formal texts, not the narrative texts) being deposited in the domain description reservoir, a global variable τ.
- We do not formalise this text.

- Clauses of the form
  - `observe_XXX(p)`,
- where `XXX` ranges over
  - `part_sorts`, `concrete_type`, `unique_identifier`, `mereology`, `part_attributes`, `part_component_sorts`, `part_material_sorts`, and `material_part_sorts`,
- stand for "text" generating functions.
- They are defined in Sect. .

4.1.3. Initialising the Domain Analysis & Description Process

- We remind the audience 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 sub-domains.
- So we need to initialise the domain analysis & description process by selecting (or choosing) the domain Δ.

- 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",
  - rambling, perhaps in an un-structured manner,
  - across its domain, Δ, and its sub-domains.
  - Informally jotting down more-or-less final sort names,
  - building, in the domain analyser & describer's mind
  - an image of that domain.

- After some time doing this
  - the domain analyser & describer is ready.
  - An image of the domain includes
  - the or a domain endurant, δ:Δ.
- Let Δ**nm** be the name of the sort Δ.
  - That name may be
    - either a part sort name,
    - or a material sort name,
    - or a component sort name.

4.2. A Model of the Analysis & Description Process

4.2.1. A Process State

1 Let Nm denote either a part or a material or a component sort name.

2 A global variable αps will accumulate all the sort names being discovered.

3 A global variable νps will hold names of sorts that have been "discovered", but have yet to be analysed & described.

type
1. Nm = PNm | MNm | KNm

variable
2. αps := [Δnm] **type** Nm-**set**
3. νps := [Δnm] **type** Nm-**set**

- We shall explain the use of [...]s and operations on the above variables in Slide 105.

- Each iteration of the "root" function, analyse_and_describe_endurant_sort(Nm,$n\iota$:nm),
  - as we shall call it,
  - involves the selection of a sort (value)
  - (which is that of either a part sort or a material sort)
  - with this sort (value) then being removed.

4 The selection occurs from the global state component νps (hence: ()) and changes that state (hence **Unit**).

value

4. sel_and_rem_Nm: **Unit** → Nm
4. sel_and_rem_Nm() ≡ **let** nm:Nm • nm ∈ νps **in** νps := νps \ {nm} ; nm **end**; **pre**: νps ≠ {}

4.2.2. A Technicality

5 The main analysis & description functions of the next sections, except the "root" function, are all expressed in terms of a pair, (nm,val):NmVAL, of a sort name and an endurant value of that sort.

type

5. NmVAL = (PNm×PVAL) | (MNm×MVAL) | (KNm×KVAL)

4.2.3. Analysis & Description of Endurants

6 To analyse and describe endurants means to first

7 examine those endurants which have yet to be so analysed and described

8 by selecting (and removing from ν**ps**) a yet un-examined sort **nm**;

9 then analyse and describe an endurant entity (ι:**nm**) of that sort — this analysis, when applied to composite parts, leads to the insertion of zero[11] or more sort names[12].

[11] If the sub-parts of ι:**nm** are all either atomic and have no materials or components or have already been analysed, then no new sort names are added to the repository ν**ps**).

[12] These new sort names are then "picked-up" for sort analysis &c. in a next iteration of the while loop.

- As was indicated in Sect. 2.,
  - ⟐ the mereology of a part, if it has one, 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,

10 then, if it has a mereology,

11 to analyse and describe the mereology of each part sort,

12 and finally to analyse and describe the attributes of each sort.

```
value
6.  analyse_and_describe_endurants: Unit → Unit
6.  analyse_and_describe_endurants() ≡
7.    while ∼is_empty(νps) do
8.      let nm = sel_and_rem_Nm() in
9.      analyse_and_describe_endurant_sort(nm,ι:nm) end end ;
10.   for all nm:PNm · nm ∈ αps do if has_mereology(nm,ι:nm)
11.     then observe_mereology(nm,ι:nm) end end
12.   for all nm:Nm · nm ∈ αps do observe_attributes(nm,ι:nm) end
```

- The ι:nm of Items 9, 10, 11 and 12 are crucial.
  - The domain analyser is focused on (part or material or component) sort nm
  - and is "directed" (by those items)
  - to choose (select) an endurant (a part or a material or component) ι:nm of that sort.

13 To analyse and describe an endurant

14 is to find out whether it is a part. If so then it is to analyse and describe it.

15 If it instead is a material, then to analyse and describe it as a material.

16 If it instead is a component, then to analyse and describe it as a component.

```
value
13.  analyse_and_describe_endurant_sort: NmVAL → Unit
13.  analyse_and_describe_endurant_sort(nm,val) ≡
14.    is_part(nm,val) → analyse_and_describe_part_sorts(nm,val),
15.    is_material(nm,val) → observe_material_part_sort(nm,val),
16.    is_component(nm,val) → observe_component_sort(nm,val)
```

[12] The conditional clause: $cond_1 \rightarrow clau_1, cond_2 \rightarrow clau_2, ..., cond_n \rightarrow clau_n$ is same as **if** $cond_1$ **then** $clau_1$ **else if** $cond_2$ **then** $clau_2$ **else** ... **if** $cond_n$ **then** $clau_n$ **end end** ... **end** .

17 The analysis and description of a part

18 first describe its unique identifier.

19 If the part is atomic it is analysed and described as such;

20 If composite it is analysed and described as such.

21 Part p must be discrete.

```
value
17.  analyse_and_describe_part_sorts: NmVAL → Unit
17.  analyse_and_describe_part_sorts(nm,val) ≡
18.      observe_unique_identifier(nm,val);
19.      is_atomic(nm,val)→ analyse_and_describe_atomic_part(nm,val),
20.      is_composite(nm,val)→ analyse_and_describe_composite_parts(nm,
21.      pre: is_discrete(nm,val)
```

22 To analyse and describe an atomic part is to inquire whether

a. it embodies materials, then we analyse and describe these;

b. and if it further has components, then we describe their sorts.

```
value
22.  analyse_and_describe_atomic_part: NmVAL → Unit
22.  analyse_and_describe_atomic_part(nm,val) ≡
22a..      if has_material(nm,val) then observe_part_material_sort(nm,val) end ;
22b..      if has_components(nm,val) then observe_part_component_sort(nm,val) end
```

23 To analyse and describe a composite endurant of sort nm (and value val)

24 is to analyse if the sort has a concrete type

25 then we analyse and describe that concrete sort type

26 else we analyse and describe the abstract sort.

```
value
23.  analyse_and_describe_composite_endurant: NmVAL → Unit
23.  analyse_and_describe_composite_endurant(nm,val) ≡
24.    if has_concrete_type(nm,val)
25.      then observe_concrete_type(nm,val)
26.      else observe_abstract_sorts(nm,val)
24.    end
23.    pre is_composite(nm,val)
```

- We do not associate materials or components with composite parts.

4.3. Discussion of The Process Model

- The above model
  - lacks a formal understanding of the individual prompts as listed in Sect. 3.1;
  - such an understanding is attempted in Sect. .

4.3.1. Termination

- The sort name reservoir ν**ps**
  - is "reduced" by one name in each iteration of the **while** loop of the analyse_and_describe_endurants, cf. Item 8 on Slide 95,
  - and is augmented by new part, material and component sort names in some iterations of that loop.
  - We assume that (manifest) domains are finite, hence there are only a finite number of domain sorts.
  - It remains to (formally) prove that the analysis & description process terminates.

4.3.2. Axioms and Proof Obligations

- We have omitted, from Sect. 2., treatment of
  - axioms concerning well-formedness of parts, materials and attributes and
  - proof obligations concerning disjointedness of observed part and material sorts and attribute types.
- [Bjø16b] exemplifies axioms and sketches some proof obligations.

4.3.3. Order of Analysis & Description: A Meaning of '⊕'

- The variables αps, νps and τ can be defined to hold either sets or lists.
- The operator $\oplus$ can be thought of
  - as either set union ($\cup$ and [...]≡{...}) — in which case the domain description text in τ is a set of domain description texts —
  - or as list concatenation ($\frown$ and [...]≡⟨...⟩) of domain description texts.
  - The list operator $\ell_1 \oplus \ell_2$ now has at least two interpretations:
    - either $\ell_1 \frown \ell_2$
    - or $\ell_2 \frown \ell_1$.
  - Thus, in the case of lists, the $\oplus$, i.e., $\frown$, does not (suffix or prefix) append ℓ_2 elements already in ℓ_1.

- The sel_and_rem_Nm function on Slide 93 applies to the set interpretation.
- A list interpretation is:

value
8. sel_and_rem_Nm: **Unit** → Nm
8. sel_and_rem_Nm() ≡ **let** nm = **hd** ν ps **in** ν ps := **tl** ν ps; nm **end**; **pre**: νps ≠<>

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

.

4.3.4. 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?
- If, for example, $\mathcal{D}_a$ 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 [Bjø11] 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 Sect. .
- But we do not prove theorems.

5. A Domain Analyser's & Describer's Domain Image

Assumptions:

- We assume that the domain analysers cum describers are
  - well educated and
  - well trained

  in the domain analysis & description techniques such as laid out in [Bjø16b].
- This assumption entails that the domain analysis & description development process
  - is structured in sequences of alternating (one or more)
  - analysis prompts and
  - description prompts.
- We refer to Footnote 2 (Slide 5) as well as to the discussion, *"Towards a methodology of manifest domain analysis & description"* of [Bjø16b, Sect. 1.6].

- We further assume that the domain analysers cum describers makes repeated attempts to analyse & describe a domain.
- We assume, further, that it is "the same domain" that is being analysed & described – two, three or more times, "all-over", before commitment is made to attempt a – hopefully – final analysis & description[10], from "scratch", that is, having "thrown away", previous drafts[11].

[10] – and if that otherwise planned, final analysis & description is not satisfactory, then yet one more iteration is taken.

[11] It may be useful, though, to keep a list of the names of all the endurant parts and their attribute names, should the group members accidentally forget such endurants and attributes: at least, if they do not appear in later document iterations, then it can be considered a deliberate omission.

- We then make the further assumption, as this iterative analysis & description process proceeds, from iteration i to $i+1$, that each and all members of the analysis & description group are forming, in their minds (i.e., brains) an "image" of the domain being analysed.
- As iterations proceed one can then say that what is being analysed & described increasingly becomes this 'image' as much as it is being the domain — which we assume is not changing across iterations.
- The iterated descriptions are now postulated to converge: a "final" iteration "differs" only "immaterially." from the description of the "previous" iteration.

The Domain Engineers's Image of Domains:

- In the opening ('Assumptions') of this section, i.e., above, we hinted at "an image", in the minds of the domain analysers & describers, of the domain being researched and for which a description document is being engineered.
- In this paragraph we shall analyse what we mean by such a image.
  - Since the analysis & description techniques are based on applying the analysis and description prompts (reviewed in Sect.) we can assume that the image somehow relates to the 'ontology' of the domain entities, whether endurants or perdurants, such as graphed in Fig. 1.
  - Rather than further investigating (i.e., analysing / arguing) the form of this, until now, vague notion, we simply conjecture that the image is that of an **'abstract syntax of domain types'.**

The Iterative Nature of The Description Process:

- Assume that the domain engineers
  - are analysing & describing a particular endurant;
  - that is, as we shall understand it,
  - are examining a given endurant node in the domain description tree!
- The **domain description tree** is defined by the facts that
  - composite parts have sub-parts
  - which may again be composite (tree branches),
  - ending with atomic parts (the leaves of the tree)
  - but not "circularly", i.e. recursively ■

- To make this claim:
  - *the domain analysers cum describers*
  - *are examining a given endurant node*
  - *in the domain description tree*
- amounts to saying that
  - *the domain engineers*
  - *have in their mind*
  - *a reasonably "stable"*
  - *"picture" of a domain*
  - *in terms of a domain description tree.*

- We need explain this assumption.
  - In this assumption there is "buried" an understanding
  - that the domain analysers cum describers
  - during the — what we can call "the final" —
  - domain analysis & description process,
  - that leads to a "deliverable" domain description,
  - are not investigating the domain to be described for the first time.
- That is,
  - we certainly assume that any "final" domain analysis & description process
  - has been preceded by a number of iterations
  - of "trial" domain analysis & description processes.

- Hopefully this iteration of
  - ⋈ experimental domain analysis & description processes converges.
- Each iteration leads to some domain description,
  - ⋈ that is, some domain description tree.
- A first iteration
  - ⋈ is thus based on a rather incomplete domain description tree
  - ⋈ which, however, "quickly" emerges into a less incomplete one
  - ⋈ in that first iteration.
- When the domain engineers decide
  - ⋈ that a "final" iteration seems possible

  then a "final" description emerges
- If acceptable, OK,
  otherwise yet an "final" iteration must be performed.

- Common to all iterations
  - is that the domain analysers cum describers
  - have in mind
  - some more-or-less "complete" domain description tree
  - and apply the prompts introduced in Sect. 4..

6. Domain Types

- There are two kinds of types associated with domains:
  - the syntactic types of endurant descriptions, and
  - the semantic types of endurant values.

6.1. Syntactic Types: Parts, Materials and Components

- In this section we outline an **'abstract syntax of domain types'.**
  - In Sect. we introduce the concept of sort names.
  - Then, in Sects. –, we describe the syntax of part, material and component types.
  - Finally, in Sects. –, we analyse this syntax with respect to a number of well-formedness criteria.

6.1.1. Syntax of Part, Material and Component Sort Names

27 There is a further undefined sort, N, of tokens (which we shall consider atomic and the basis for forming names).

28 From these we form three disjoint sets of sort names:

a. part sort names,

b. material sort names and

c. component sort names,

```
27    N
28a.  PNm  :: mkPNm(N)
28b.  MNm :: mkMNm(N)
28c.  KNm  :: mkKNm(N)
```

6.1.2. An Abstract Syntax of Domain Endurants

29 We think of the types of parts, materials and components to be a map from their type names to respective type expressions.

30 Thus part types map part sort names into part types;

31 material types map material sort names into material types; and

32 component types map components sort names into component types.

33 Thus we can speak of endurant types to be either part types or material types or component types.

34 A part type expression is either an atomic part type expression or is a composite part type expression or is a concrete composite part type expression.

35 An atomic part type expression consists of a type expression for the qualities of the atomic part and, optionally, a material type name or a component type name (cf. Sect.).

36 An abstract composite part type expression consists of a type expression for the qualities of the composite part and a finite set of one or more part type names.

37 A concrete composite part type expression consists of a type expression for the qualities of the part and a part sort name standing for a set of parts of that sort.

38 A material part type expression consists of of a type expression for the qualities of the material and an optional part type name.

39 We omit consideration of component types.

Endurants: Syntactic Types

```
29        TypDef  =  PTypes ∪ MTypes ∪ KTypes
30        PTypes  =  PNm -m-> PaTyp
31        MTypes  =  MNm -m-> MaTyp
32        KTypes  =  KNm -m-> KoTyp
33       ENDType  =  PaTyp | MaTyp | KoTyp
34         PaTyp  == AtPaTyp | AbsCoPaTyp | ConCoPaTyp
35       AtPaTyp  ::  mkAtPaTyp(s_qs:PQ,s_omkn:({|"nil"|}|MNn|KNm))
36  AbsCoPaTyp    ::  mkAbsCoPaTyp(s_qs:PQ,s_pns:PNm-set)
36                    axiom ∀ mkAbsCoPaTyp(pq,pns):AbsCoPaTyp · pns ≠ {}
37  ConCoPaTyp    ::  mkConCoPaTyp(s_qs:PQ,s_p:PNm)
38        MaTyp   ::  mkMaTyp(s_qs:MQ,s_opn:({|"nil"|}|PNm))
39        KoTyp   ::  mkKoTyp(s_qs:KQ)
```

6.1.3. Quality Types

40 There are three aspects to part qualities: the type of the part unique identifiers, the type of the part mereology, and the name and type of attributes.

41 The type unique part identifiers is a not further defined atomic quantity.

42 A part mereology is either `"nil"` or it is an expression over part unique identifiers, where such expressions are those of either simple unique identifier tokens, or of set, or otherwise over simple unique identifier tokens, or ..., etc.

43 The type of attributes pairs distinct attribute names with attribute types —

44 both of which we presently leave further undefined.

45 Material attributes is the only aspect to material qualities.

46 Components have unique identifiers. Component attribute types are left undefined.

Qualities: Syntactic Types

```
40          PQ  =  s_ui:UI×s_me:ME×s_atrs:ATRS}
41          UI
42          ME  ==  "nil"|mkUI(s_ui:UI)|mkUIset(s_uil:UI)|...
43        ATRS  =  ANm m→ ATyp
44  ANm, ATyp
45          MQ  =  s_atrs:ATRS
46          KQ  =  s_uid:UI × s_atrs:ATRS
```

- It is without loss of generality that we do not distinguish between part and material attribute names and types.
- Material and component attributes do not refer to any part or any other material and component attributes.

6.1.4. Well-formed Syntactic Types
6.1.4.1 Well-formed Definitions

47 We need define an auxiliary function, **names**, which, given an endurant type expression, yields the sort names that ar referenced immediately by that type.

a. If the endurant type expression is that of an atomic part type then the sort name is that of its optional component sort.
b. If an abstract composite part type then the sort names of its parts.
c. If a concrete composite part type then the sort name is that of the sort of its set of parts.
d. If a material type then sort name is that of the sort of its optional parts.
e. Component sorts have no references to other sorts.

```
value
47.  names: TypDef → (PNm|MNm|KNm) → (PNm|MNm|KNm)-set
47.  names(td)(n) ≡
47.      ∪ { ns | ns:(PNm|MNm|KNm)-set ·
47.                  case td(n) of
47a..                    mkAtPaTyp(_,n′) → ns={n′},
47b..                    mkAbsCoPaTyp(_,ns′) → ns=ns′,
47c..                    mkConCoPaTyp(_,pn) → ns={pn},
47d..                    mkMaTyp(_,n′) → ns={n′},
47e..                    mkKoTyp(_) → ns={}
47.                  end }
```

48 Endurant sort names being referenced in part types, PaTyp, in material types, MaTyp, and in component types, KoTyp, of the typdef:Typdef definition, *must be defined in* the defining set, **dom** typdef, of the typdef:Typdef definition.

```
value
48.  wf_TypDef_1: TypDef → Bool
48.  wf_TypDef_1(td) ≡ ∀ n:(PNm|MNm|CNm)·n ∈ dom td ⇒ names(td)(n)⊆dom td
```

Perhaps Item 48. should be sharpened:

49 from *"must be defined in"* [48.] to *"must be equal to"*:

49. $\wedge \forall$ n:(PNm|MNm|CNm) · n $\in$ **dom** td $\Rightarrow$ names(td)(n)=**dom** td

6.1.4.2 No Recursive Definitions

50 Type definitions must not define types recursively.

a. A type definition, typdef:TypDef, defines, typically composite part sorts, named, say, n, in terms of other part (material and component) types. This is captured in the
   - mncs (Item 35),
   - pns (Item 36),
   - p (Item 37) and
   - pns (Item 38),

   selectable elements of respective type definitions. These elements identify type names of materials and components, parts, a part, and parts, respectively. None of these names may be n.

b. The identified type names may further identify type definitions none of whose selected type names may be n.

c. And so forth.

```
value
50.  wf_TypDef_2: TypDef → Bool
50.  wf_TypDef_2(typdef) ≡ ∀ n:(PNm|MNm)· n ∈ dom typdef ⇒ n ∉ type_names(typdef)(n)

50a..  type_names: TypDef → (PNm|MNm) → (PNm|MNm)-set
50a..  type_names(typdef)(nm) ≡
50b..      let ns = names(typdef)(nm) ∪ { names(typdef)(n) | n:(PNm|MNm) · n ∈ ns } in
50c..      nm ∉ ns end
```

- ns is the least fix-point solution to the recursive definition of ns.

6.2. Semantic Types: Parts, Materials and Components

6.2.1. Part, Material and Component Values

- We define the values corresponding to the type definitions of Items 27.–46, structured as per type definition Item 33 on Slide 121.

51 An endurant value is either a part value, a material values or a component value.

52 A part value is either the value of an atomic part, or of an abstract composite part, or of a concrete composite part.

53 A atomic part value has a part quality value and, optionally, either a material or a possibly empty set of component values (cf. Sect.).

54 An abstract composite part value has a part quality value and of at least (hence the **axiom**) of

55 one or more (distinct part type) part values.

56 A concrete composite part value has a part quality value and a set of part values.

57 A material value has a material quality value (of material attributes) and a (usually empty) finite set of part values.

58 A component value has a component quality value (of a unique identifier and component attributes).

Endurant Values: Semantic Types

```
51     ENDVAL  =   PVAL | MVAL | KVAL
52        PVAL ==  AtPaVAL|AbsCoPVAL|ConCoPVAL
53     AtPaVAL  ::  mkAtPaVAL(s_qval:PQVAL,s_omkvals:({|"nil"|}|MVAL|KVAL-set))
54  AbsCoPVAL  ::  mkAbsCoPaVAL(s_qval:PQVAL,s_pvals:(PNm -m-> PVAL))
55                 axiom ∀ mkAbsCoPaVAL(pqs,ppm):AbsCoPVAL · ppm ≠ [ ]
56  ConCoPVAL  ::  mkConCoPaVAL(s_qval:PQVAL,s_pvals:PVAL-set)
57        MVAL  ::  mkMaVAL(s_qval:MQVAL,s_pvals:PVAL-set)
58        KVAL  ::  mkKoVAL(s_qval:KQVAL)
```

6.2.2. Quality Values

59 A part quality value consists of three qualities:

60 a unique identifier type name, resp. value, which are both further undefined (atomic value) tokens;

61 a mereology expression, resp. value, which is either a single unique identifier (type, resp.) value, or a set of such unique identifier (types, resp.) values, or ...; and

62 an aggregate of attribute values, modeled here as a map from attribute type names to attribute values.

63 In this paper we leave attribute type names and attribute values further undefined.

64 A material quality value consists just of an aggregate of attribute values, modeled here as a map from attribute type names to attribute values.

65 A component quality value consists of a pair: a unique identifier value and an aggregate of attribute values, modeled here as a map from attribute type names to attribute values.

Qualities: Semantic Types

```
59      PQVAL  =  UIVAL×MEVAL×ATTRVALS
60       UIVAL
61      MEVAL  == mkUIVAL(s_ui:UIVAL)|mkUIVALset(s_uis:UIVAL-set)|...
62  ATTRVALS   =  ANm →m AVAL
63  ANm, AVAL
64      MQVAL  =  ATTRVALS
65      KQVAL  =  UIVAL×ATTRVALS
```

- We have left to define the values of attributes.
  - For each part and material attribute value we assume a finite set of values.
  - And for each unique identifier type (i.e., for each UI) we likewise assume a finite set of unique identifiers of that type.
  - The value sets may be large.
  - These assumptions help secure that the set of part, material and component values are also finite.

6.2.3. Type Checking

- For part, material and component qualities we postulate an overloaded, simple type checking function, type_of, that applies to
  - unique identifier values, uiv:UIVAL, and yield their unique identifier type name, ui:UI, to
  - mereology values, mev:MEVAL, and yield their mereology expression, me:ME, and to
  - attribute values, AVAL and ATTRSVAL, and yield their types: ATyp, respectively (ANm $\overrightarrow{m}$AVAL)→(ANm $\overrightarrow{m}$ATyp).
  - Since we have let undefined both the syntactic type of attributes types, ATyp, and the semantic type of attribute values, AVAL, we shall leave type_of further unspecified.

value type_of: (UIVAL→UI)|(MEVAL→ME)|(AVAL→ATyp)|((ANm $\overrightarrow{m}$AVAL)→(ANm $\overrightarrow{m}$ATy

- The definition of the syntactic type of attributes types, ATyp, and the semantic type of attribute values, AVAL, is a simple exercise in a first-year programming language semantics course.

7. From Syntax to Semantics and Back Again !

- The two syntaxes of the previous section:
  - that of the **syntactic domains**, formula Items 27–46 (Slides 120–124), and
  - that of the **semantic domains**, formula Items 51–65 (Slides 132–134),

  are not the syntaxes of domain descriptions,
  but of some aspects common to all domain descriptions
  developed according to the calculi of this paper.

- The syntactic domain formulas
  underlie ("are common to", i.e., "abstracts") aspects of all domain descriptions.
- The semantic domain formulas
  underlay ("are common to", i.e., "abstracts") aspects of the meaning of all domain descriptions.
- These two syntaxes, hence, are, so-to-speak, in the minds
  of the domain engineer (i.e., the analyser cum describer)
  while analysing the domain.

7.1. The Analysis & Description Prompt Arguments

- The domain engineer analyse & describe endurants on the basis of
  - a sort name i.e., a piece of syntax, nm:Nm, and
  - an endurant value, i.e. a "piece" of semantics, val:VAL,
  - that is, the arguments, (nm,ι:nm), of the analysis and description prompts of Sect. .
- Those two quantities are what the domain engineer are "operating" with, i.e., are handling:
  - One is tangible, i.e. can be noted (i.e., "scribbled down"),
  - the other is "in the mind" of the analysers cum describers.

- We can relate the two in terms of the two syntaxes,
  - the syntactic types, and
  - the meaning of the semantic types.
- But first some "preliminaries".

7.2. Some Auxiliary Maps: Syntax to Semantics and Semantics to Syntax

- We define two kinds of map types:

66 Nm_to_ENDVALS are maps from endurant sort names to respective sets of all corresponding endurant values of, and

67 ENDVAL_to_Nm are maps from endurant values to respective sort names.

type
66. Nm_to_ENDVALS = (PNm $\overrightarrow{m}$PVAL-**set**)∪(MNm $\overrightarrow{m}$MVAL-**set**)∪(KNm $\overrightarrow{m}$KVAL-**set**)
67. ENDVAL_to_Nm = (PVAL $\overrightarrow{m}$PNm)∪(MVAL $\overrightarrow{m}$MNm)∪(KVAL $\overrightarrow{m}$KNm)

- We can derive values of these map types from type definitions:

68 a function, typval, from type definitions, typdef:TypDef to Nm_to_ENDVALS, and

69 a function valtyp, from Nm_to_ENDVALS, to ENDVAL_to_Nm.

value

68. typval: TypDef $\xrightarrow{\sim}$ Nm_to_ENDVALS
69. valtyp: Nm_to_ENDVALS $\xrightarrow{\sim}$ ENDVAL_to_Nm

70 The **typval** function is defined in terms of a meaning function M (let ρ:ENV abbreviate Nm_to_ENDVALS:

70. M: (PaTyp→ENV$\xrightarrow{\sim}$PVAL-**set**)|(MaTyp→ENV$\xrightarrow{\sim}$MVAL-**set**)|(KoTyp→ENV$\xrightarrow{\sim}$KVAL-se

68. typval(td) ≡ **let** ρ = [n↦M(td(n))(ρ)|n:(PNm|MNm|KNm)·n ∈ **dom** td] **in** ρ **end**
69. valtyp(ρ) ≡ [v↦n|n:(PNm|MNm|CNm),v:(PVAL|MVAL|KVAL)·n ∈ **dom** ρ∧v ∈ ρ(n)]

- The environment, ρ, of **typval**, Item 68, is the least fix point of the recursive equation

  ⟴ 68. **let** ρ = [n↦M(td(n))(ρ)|n:(PNm|MNm|CNm)·n ∈ **dom** td] **in** ...

7.3. The ι Description Function

- We can now define the meaning of the syntactic clause:

  ⟐ ιNm:Nm

71 ιNm:Nm "chooses" an arbitrary value from amongst the values of sort Nm:

value
71. ι nm:Nm ≡ iota(nm)
71. iota: Nm → TypDef → VAL
71. iota(nm)(td) ≡ **let** val:(PVAL|MVAL|KVAL)·val ∈ (typval(td))(nm) **in** val **end**

7.4. Discussion

- From the above two functions, **typval** and **valtyp**, and the type definition "table" td:TypDef and "argument value" val:PVAL|MVAL|KVAL, we can form some expressions.
  One can understand these expressions as, for example reflecting the following analysis situations:
  - **typval**(td): From the type definitions we form a map, by means of function typval, from sort names to the set of all values of respective sorts: Nm_to_ENDVALS.
    That is, whenever we, in the following, as part of some formula, write **typval**(td), then we mean to express that the domain engineer forms those associations, in her mind, from sort names to usually very large, non-trivial sets of endurant values.

- **valtyp(typval(td))**: The domain analyser cum describer "inverts", again in his mind, the **typval(td)** into a simple map, ENDVAL_to_Nm, from single endurant values to their sort names.
- **(valtyp(typval(td)))(val)**: The domain engineer now "applies", in her mind, the simple map (above) to an endurant value and obtains its sort name nm:Nm.
- td((**valtyp(typval(td))**)(val)): The domain analyser cum describer then applies the type definition "table" td:TypDef to the sort name nm:Nm and obtains, in his mind, the corresponding type definition, PaTyp|MaTyp|KoTyp.

• We leave it to the reader to otherwise get familiarised with these expressions.

8. A Formal Description of a Meaning of Prompts

8.1. On Function Overloading

- In Sect. the analysis and description prompt invocations
  - were expressed as
    - `is_XXX(e)`, `has_YYY(e)` and `observe_ZZZ(e)`
  - where `XXX`, `YYY`, and `ZZZ` were appropriate entity sorts
  - and `e` were appropriate endurants (parts, components and materials).

- The function invocations, `is_XXX(e)`, etcetera,
  - takes place in the context of a type definition, td:TypDef,
  - that is, instead of `is_XXX(e)`, etc. we get
    - `is_XXX(e)(td)`, `has_YYY(e)(td)` and `observe_ZZZ(e)(td)`.
  - We say that the functions `is_XXX`, etc., are "lifted".

8.2. The Analysis Prompts

- The analysis is expressed in terms of the analysis prompts:

a. `is_ entity`, 4
b. `is_ endurant`, 4
c. `is_ perdurant`, 4
d. `is_ discrete`, 5
e. `is_ continuous`, 5
f. `is_ part`, 5
g. `is_ component`, 5
h. `is_ material`, 5
i. `is_ atomic`, 6
j. `is_ composite`, 6
k. `has_ concrete_ type`, 6
l. `has_ mereology`, 8
m. `has_ components`, 9
n. `has_ material`, 10
o. `has_ parts`, 10

  - The analysis takes place in the context of
    - a type definition "image", td:TypDef,
    - in the minds of the domain engineers.

8.2.1. is_entity

- The **is_entity** predicate is meta-linguistic,
  - that is, we cannot model it on the basis of the type systems given in Sect. .
  - So we shall just have to accept that.

8.2.2. is_endurant

value

is_endurant: Nm×VAL → TypDef $\xrightarrow{\sim}$ **Bool**
is_endurant(_,val)(td) ≡ val ∈ **dom** valtyp(typval(td)); **pre**: VAL is any value type

8.2.3. is_discrete

value

is_discrete: NmVAL → TypDef $\xrightarrow{\sim}$ **Bool**
is_discrete(_,val)(td) ≡ (is_PaTyp|is_CoTyp)(td((valtyp(typval(td)))(val)))

8.2.4. is_part

value

is_part: NmVAL → TypDef $\xrightarrow{\sim}$ **Bool**
is_part(_,val)(td) ≡ is_PaTyp(td((valtyp(typval(td)))(val)))

8.2.5. is_material [≡ is_continuous]

- We remind the reader that
  - ◈ is_continuous≡is_material.

value

is_material: NmVAL → TypDef $\xrightarrow{\sim}$ **Bool**
is_material(_,val)(td) ≡ is_MaTyp(td((valtyp(typval(td)))(val)))

8.2.6. is_component

value

is_component: NmVAL → TypDef $\xrightarrow{\sim}$ **Bool**
is_component(_,val)(td) ≡ is_CoTyp(td((valtyp(typval(td)))(val)))

8.2.7. is_atomic

```
value
  is_atomic: NmVAL → TypDef ~→ Bool
  is_atomic(_val)(td) ≡ is_AtPaTyp(td((valtyp(typval(td)))())))
```

8.2.8. is_composite

```
value
  is_composite: NmVAL → TypDef ~→ Bool
  is_composite(_,val)(td) ≡ (is_AbsCoPaTyp|is_ConCoPaTyp)(td((valtyp(typval(td)))(val)))
```

8.2.9. has_concrete_type

value

has_concrete_type: NmVAL → TypDef $\xrightarrow{\sim}$ **Bool**
has_concrete_type(_,val)(td) ≡ is_ConCoPaTyp(td((valtyp(typval(td)))(val)))

8.2.10. has_mereology

value

has_mereology: NmVAL → TypDef $\xrightarrow{\sim}$ **Bool**
has_mereology(_,val)(td) ≡ s_me(td((valtyp(typval(td)))(val)))≠"nil"

8.2.11. has_materials

```
value
  has_material: NmVAL → TypDef ~→ Bool
  has_material(_,val)(td) ≡ is_MNm(s_omkn(td((valtyp(typval(td)))(val))))
    pre: is_AtPaTyp(td((valtyp(typval(td)))(val)))
```

8.2.12. has_components

```
value
  has_components: NmVAL → TypDef ~→ Bool
  has_components(_,val)(td) ≡ is_KNm(s_omkn(td((valtyp(typval(td)))(val))))
    pre: is_AtPaTyp(td((valtyp(typval(td)))(val)))
```

8.2.13. has_parts

value

has_parts: NmVAL → TypDef $\xrightarrow{\sim}$ **Bool**
has_parts(_,val)(td) ≡ is_PNm(s_opn(td((valtyp(typval(td)))(val))))
pre: is_MaTyp(td((valtyp(typval(td)))(val)))

8.3. The Description Prompts

- These are the domain description prompts to be defined:

8.3.1. A Description State

- In addition to the analysis state components αps and νps
- there is now an additional, the description text state component.

72 Thus a global variable τ will hold the (so far) generated (in this case only) formal domain description text.

variable
72. τ := [ ] **Text-set**

- We shall explain the use of [...]s and the operations of $\setminus$ and $\oplus$ on the above variables in Sect. 4.10.5 Slide 105.

8.3.2. observe_part_sorts

```
value
    observe_part_sorts: NmVAL → TypDef → Unit
    observe_part_sorts(nm,val)(td) ≡
        let mkAbsCoPaTyp(_,{P1,P2,...,Pn}) = td((valtyp(typval(td)))(val)) in
                        τ := τ ⊕ [" type P1,P2,...,Pn;
                                   value
                                      obs_part_P1nm→P1
                                      obs_part_P2:nm→P2
                                      ...,
                                      obs_part_Pn:nm→Pn;
                                   proof obligation
                                      D; " ]
                    ‖ νps := νps ⊕ ([P1,P2,...,Pn] \ αps)
                    ‖ αps := αps ⊕ [P1,P2,...,Pn]
        end
        pre: is_AbsCoPaTyp(td((valtyp(typval(td)))(val)))
```

- $\mathcal{D}$ is a predicate expressing the disjointness of part sorts $\mathsf{P}_1,\mathsf{P}_2,...,\mathsf{P}_n$

8.3.3. observe_concrete_type

```
value
    observe_concrete_type: NmVAL → TypDef → Unit
    observe_concrete_type(nm,val)(td) ≡
        let mkConCoPaTyp(_,P) = td((valtyp(typval(td)))(val)) in
                τ := τ ⊕ ["type T = P-set ; value obs_part_T: nm→T; " ]
              ‖ νps := νps ⊕ ([ P ] \ αps)
              ‖ αps := αps ⊕ [ P ]
        end
        pre: is_ConCoPaTyp(td((valtyp(typval(td)))(val)))
```

8.3.4. observe_unique_identifier

value

observe_unique_identifier: P → TypDef → **Unit**
observe_unique_identifier(nm,val)(td) ≡
τ := τ ⊕ [" **type** PI ; **value uid**_PI: nm → PI ; **axiom** $\mathcal{U}$; "]

- $\mathcal{U}$ is a predicate expression over unique identifiers.

8.3.5. observe_mereology

```
value
    observe_mereology: NmVAL → TypDef → Unit
    observe_mereology(nm,val)(td) ≡
        τ := τ ⊕ [ "type MT = M(PI1,PI2,...,PIn) ;
                   value obs_mereo_P: nm → MT ;
                   axiom ME; " ]
        pre: has_mereology(nm,val)(td)
```

- $\mathcal{M}$(PI1,PI2,...,PIn) is a type expression over unique part identifiers.
- $\mathcal{ME}$ is a predicate expression over unique part identifiers.

8.3.6. observe_part_attributes

```
value
    observe_part_attributes: NmVAL → TypDef → Unit
    observe_part_attributes(nm,val)(td) ≡
      let {A1,A2,...,Aa} = dom s_attrs(s_qs(val)) in
      τ := τ ⊕ [" type A1, A2, ..., Aa
                value attr_A1: nm→Ai
                    attr_A2: nm→A1
                    ...
                    attr_Aa: nm→Ai
                proof obligation [Disjointness of Attribute Types]
                    A ; " ]
      end
```

- $\mathcal{A}$ is a predicate over attribute types A_1, A_2, ..., A_a.

8.3.7. observe_part_material_sort

```
value
    observe_part_material_sort: NmVAL → TypDef → Unit
    observe_part_material_sort(nm,val)(td) ≡
        let M = s_pns(td((valtyp(typval(td)))(val))) in
        τ := τ ⊕ [" type M ; value obs_mat_M:nm→M " ]
        ‖ νps := νps ⊕ ([M]\αps)
        ‖ αps := αps ⊕ [M]
        end
        pre: is_AtPaVAL(val) ∧ is_MNm(s_pns(td((valtyp(typval(td)))(val))))
```

8.3.8. observe_component_sort

```
value
    observe_component_sort: NmVAL → TypDef → Unit
    observe_component_sort(nm,val)(td) ≡
        let K = s_omkn(td((valtyp(typval(td)))(val))) in
        τ := τ ⊕ [" type K ; value obs-comps: nm → K-set; " ]
        ‖ νps := νps ⊕ ([K]\αps)
        ‖ αps := αps ⊕ [K]
        end
        pre: is_AtPaTyp(td((valtyp(typval(td)))(val))) ∧ has_components(nm,val)
```

8.3.9. observe_material_part_sort

```
value
    observe_material_part_sort: NmVAL → TypDef → Unit
    observe_material_part_sort(nm,val)(td) ≡
        let P = s_pns(td((valtyp(typval(td)))(val))) in
        τ := τ ⊕ [" type P ; value obs_part_P: nm → P " ]
        ‖ νps := νps ⊕ ([P] \ αps)
        ‖ αps := αps ⊕ [P]
        end
        pre is_MaTyp(td((valtyp(typval(td)))(val))) ∧ is_PNm(s_pns(td((valtyp(typval(td)))(val))))
```

8.4. Discussion of The Prompt Model

- The prompt model of this section is formulated so as to reflect a "wavering", of the domain engineer, between syntactic and semantic reflections.
  - The syntactic reflections are represented by the syntactic arguments of the sort names, nm, and the type definitions, td.
  - The semantic reflections are represented by the semantic argument of values, val.

- When we, in the various prompt definitions,
  use the expression td((valtyp(typval(td)))(val))
  we mean to model that the domain analyser cum describer
  reflects semantically: "viewing", as it were, the endurant.
  - We could, as well, have written td(nm) —
  - reflecting a syntactic reference to the (emerging) type model
    in the mind of the domain engineer.

9. Conclusion

- It is time to summarise, conclude and look forward.

9.1. What Has Been Achieved

- [Bjø16b] proposed a set of domain analysis & description prompts –
  - and Sect. 2. summarised that language.
  - Sections 4. and 8.
    - proposed an operational semantics for the process of selecting and applying prompts,
    - respectively a more abstract meaning of of these prompts,
    - the latter based on some notions of an "image" of perceived abstract types of syntactic and of semantic structures of the perceived domain.
  - These notions were discussed in Sects. 5. and 6.

- To the best of our knowledge this is the first time
  - a reasonably precise notion of 'method'
    with a similarly reasonably precise notion of a calculi of tools
  - has been backed up formal definitions.

9.2. Are the Models Valid ?

- Are the formal descriptions of
  - the process of selecting and applying the analysis & description prompts, Sect. 4., and
  - the meaning of these prompts, Sect. 8.,
  - modeling this process and these meanings realistically ?
- To that we can only answer the following:
  - The process model is definitely modeling plausible processes.
    - We discuss interpretations of the analysis & description order
    - that this process model imposes in Sect. 3.3.3.
    - There might be other orders, but the ones suggested in Sect. 4. can be said to be "orderly" and reflects empirical observations.

- The model of the meaning of prompts, Sect. 8., is more of an hypothesis.
  - This model refers to "images" that the domain engineer is claimed to have in her mind.
  - It must necessarily be a valid model,
  - perhaps one of several valid models.
  - We have speculated, over many years, over the existence of other models.
  - But this is the most reasonable to us.
- We have hinted at possible 'laws of description prompts' in Sect. 4.3.3.
  - Whether the process and prompt models (Sects. 4. and 8.) are sufficient
  - to express, let alone prove such laws is an open question.
  - If the models are sufficient, then they certainly are valid.

10. Bibliography

10.1. Bibliographical Notes

- This paper, [Bjø17a], concludes a series of five papers by this author on domain engineering.
- The other papers are [Bjø16b, Bjø17b, Bjø16a, Bjø17c].

10.2. References

[Aus76] John Longshaw Austin. *How To Do Things With Words*. Oxford University Press, second edition, 1976.

[BAvWS98] Ralph-Johan Back, Abo Akademi, J. von Wright, and F. B. Schneider. *Refinement Calculus: A Systematic Introduction*. Springer–Verlag New York, Inc., 1998.

[Bjø11] 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.

[Bjø16a] Dines Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. *Perhaps to be submitted for consideration by Formal Aspects of Computing*, 2016. http://www.imm.dtu.dk/~dibj/2016/demos/faoc-demo.pdf.

[Bjø16b] Dines Bjørner. Manifest Domains: Analysis & Description. *Formal Aspects of Computing*, ...(...):1–51, 2016. DOI 10.1007/s00165-016-0385-z http://link.springer.com/article/10.1007/s00165-016-0385-z.

[Bjø17a] Dines Bjørner. Domain Analysis and Description – Formal Models of Processes and Prompts. *Perhaps to be submitted for consideration by Formal Aspects of Computing*, 2016–2017. http://www.imm.dtu.dk/~dibj/2016/process/process-p.pdf.

[Bjø17b] Dines Bjørner. Domain Facets: Analysis & Description. *Submitted for consideration by Formal Aspects of Computing*, 2016–2017. http://www.imm.dtu.dk/~dibj/2016/facets/faoc-facets.pdf.

[Bjø17c] Dines Bjørner. From Domain Descriptions to Requirements Prescriptions – A Different Approach to Requirements Engineering. *Submitted for consideration by Formal Aspects of Computing*, 2016–2017.

[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.

[Mor90] C. Carroll Morgan. *Programming from Specifications*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1990.

[Ost87] Leon Osterweil. Software processes are software too. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, pages 2–13, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[Sea69] John R. Searle. *Speech Act*. CUP, 1969.

1. M: A Meaning of Type Names
1.1. Preliminaries

- The typval function provides for a homomorphic image from TypDef to TypNm_to_VALS.
  - So, the narrative below, describes, item-by-item, this image.
  - We refer to formula Items 68 and 70 on Slide 144.
- The definition of M is decomposed into five sub-definitions, one for each kind of endurant type:
  - Atomic parts: mkAtPaTyp(s_qs:(UI×ME×ATRS),s_omkn:({|"nil"|}|MNn|KNm)), Sect. on the next slide, Items 73– 73(d.)iii on the following slide;
  - Abstract composite parts: mkAbsCoPaTyp(s_qs:PQ,s_pns:PNm-set), Sect. on Slide 180, Items 74– 74d. on Slide 180;
  - Concrete composite parts: mkConCoPaTyp(s_qs:PQ,s_p:PNm), Sect. on Slide 182, Items 75– 75d. on Slide 182;
  - Materials: mkMaTyp(s_qs:MQ,s_opn:({|"nil"|}|PNm)), Sect. on Slide 184, Items 76– 76b. on Slide 184; and
  - Components: mkKoTyp(s_qs:KQ), Sect. on Slide 185, Items 77– 77b. on Slide 185.
- We abbreviate, by ENV, the M function argument, ρ, of type: Nm_to_ENDVALS.

1.2. Atomic Parts

73 The meaning of an atomic part type expression,
Item 35. mkAtPaTyp((ui,me,attrs),omkn)
in mkAtPaTyp(s_qs:PQ,s_omkn:({|"nil"|}|MNn|KNm)),
is the set of all atomic part values,
Items 53., 59., 62. mkAtPaVAL((uiv,mev,attrvals),omkval)
in mkAtPaVAL(s_qval:(UIVAL×MEVAL×(ANm $\overrightarrow{m}$AVAL)),
s_omkvals:({|"nil"|}|MVAL|KVAL-**set**)).

a. uiv is a value in UIVAL of type ui,
b. mev is a value in MEVAL of type me,
c. attrvals is a value in (ANm $\overrightarrow{m}$AVAL) of type (ANm $\overrightarrow{m}$ATyp), and
d. omkvals is a value in ({|"nil"|}|MVAL|KVAL-**set**):
   i either `''nil''`,
   ii or one material value of type MNm,
   iii or a possibly empty set of component values, each of type KNm.

73. M: mkAtPaTyp((UI×ME×(ANm $\overrightarrow{m}$ATyp))×({|"nil"|}|MVAL|KVAL-**set**))→ENV$\xrightarrow{\sim}$PVAL-**set**
73. M(mkAtPaTyp((ui,me,attrs),omkn))(ρ) ≡
73. { mkATPaVAL((uiv,mev,attrval),omkvals) |
73a.. uiv:UIVAL•type_of(uiv)=ui,
73b.. mev:MEVAL•type_of(mev)=me,
73c.. attrval:(ANm $\overrightarrow{m}$AVAL)•type_of(attrval)=attrs,
73d.. omkvals: **case** omkn **of**
73(d.)i. "nil" → "nil",
73(d.)ii. mkMNn(_) → mval:MVAL•type_of(mval)=omkn,
73(d.)iii. mkKNm(_) → kvals:KVAL-**set**•kvals⊆{kv|kv:KVAL•type_of(kval)=omkn}
73d.. **end** }

- Formula terms 73a.–73(d.)iii express that any applicable uiv is combined with any applicable mev is combined with any applicable attrval is combined with any applicable omkvals.

1.3. Abstract Composite Parts

74 The meaning of an abstract composite part type expression,
Item 36. mkAbsCoPaTyp((ui,me,attrs),pns)
in mkAbsCoPaTyp(s_qs:PQ,s_pns:PNm-**set**),
is the set of all abstract, composite part values,
Items 54., 59., 62., mkAbsCoPaVAL((uiv,mev,attrvals),pvals)
in mkAbsCo-
PaVAL(s_qval:(UIVAL×MEVAL×(ANm $\overrightarrow{m}$AVAL)),s_pvals:(PNm $\overrightarrow{m}$PVAL)).

a. uiv is a value in UIVAL of type ui: UI,
b. mev is a value in MEVAL of type me: ME,
c. attrvals is a value in (ANm $\overrightarrow{m}$AVAL) of type (ANm $\overrightarrow{m}$ATyp), and
d. pvals is a map of part values in (PNm $\overrightarrow{m}$PVAL), one for each name, pn:PNm, in pns such that these part values are of the type defined for pn.

74. M: mkAbsCoPaTyp((UI×ME×(ANm $\overrightarrow{m}$ATyp)),PNm-**set**) → ENV $\xrightarrow{\sim}$ PVAL-**set**
74. M(mkAbsCoPaTyp((ui,me,attrs),pns))(ρ) ≡
74. { mkAbsCoPaVAL((uiv,mev,attrvals),pvals) |
74a.. uiv:UIVAL•type_of(uiv)=ui
74b.. mev:MEVAL•type_of(mev)=me,
74c.. attrvals:(ANm $\overrightarrow{m}$ATyp)•type_of(attrsval)=attrs,
74d.. pvals:(PNm $\overrightarrow{m}$PVAL)•pvals∈{[pn↦pval|pn:PNm,pval:PVAL•pn∈ pns∧pval∈ρ(pn)]} }

1.4. Concrete Composite Parts

75 The meaning of a concrete composite part type expression, Item 37.
mkConCoPaTyp((ui,me,attrs),pn)
in mkConCoPaTyp(s_qs:(UI×ME×(ANm $\overrightarrow{m}$ATyp)),s_pn:PNm),
is the set of all concrete, composite *set* part values,
Item 56. mkConCoPaVAL((uiv,mev,attrvals),pvals)
in mkConCoPaVAL(s_qval:(UIVAL×MEVAL×(ANm $\overrightarrow{m}$AVAL)),s_pvals:PVAL-**set**).

a. uiv is a value in UIVAL of type ui,
b. mev is a value in MEVAL of type me,
c. attrvals is a value in (ANm $\overrightarrow{m}$AVAL) of type attrs, and
d. pvals is a[ny] value in PVAL-**set** where each part value in pvals is of the type defined for pn.

75. M: mkConCoPaTyp((UI×ME×(ANm $\overrightarrow{m}$ ATyp))×PNm) → ENV $\xrightarrow{\sim}$ PVAL-**set**
75. M(mkConCoPaTyp((ui,me,attrs),pn))(ρ) ≡
75. { mkConCoPaVAL((uiv,mev,attrvals),pvals) |
75a.. uiv:UIVAL·type_of(uiv)=ui,
75b.. mev:MEVAL·type_of(mev)=me,
75c.. attrsval:(ANm $\overrightarrow{m}$ AVAL)·type_of(attrsval)=attrs,
75d.. pvals:PVAL-**set**·pvals⊆ρ(pn) }

1.5. Materials

76 The meaning of a material type, 38.,
expression mkMaTyp(mq,pn) in mkMaTyp(s_qs:MQ,s_pn:PNm)
is the set of values mkMaVAL(mqval,ps)
in mkMaVAL(s_qval:MQVAL,s_pvals:PVAL-**set**) such that

a. mqval in MQVAL is of type mq, and

b. ps is a set of part values all of type pn.

76. M: mkMaTyp(s_mq:(ANm $\overrightarrow{m}$ATyp),s_pn:PNm) → ENV $\xrightarrow{\sim}$ MVAL-**set**
76. M(mq,pn)(ρ) ≡
76. { mkMVAL(mqval,ps) |
76a.. mqval:MVAL·type_of(mqval)=mq,
76b.. ps:PVAL-**set**·ps⊆ρ(pn) }

1.6. Components

77 The meaning of a component type, 39., expression mkKoType(ui,atrs) in mkKoTyp(s_qs:(s_uid:UI×s_atrs:ATRS)) is the set of values, 38., mkKQVAL(uiv,attrsval) in, 58, mkKoVAL(s_qval:(uiv,attrsval)).

a. uiv is in UIVAL of type ui, and

b. attrsval is in ATTRSVAL of type atrs.

77. M: mkKoTyp(UI×ATRS) → ENV → KVAL-**set**
77. M(mkKoType(ui,atrs))(ρ) ≡
77. { mkKoVAL(uiv,attrsval) |
77a.. uiv:UIVAL·type_of(uiv)=ui,
77b.. attrsval:ATRSVAL·type_of(attrsval)=atrs }

2. A Domain Description Example: A Credit Card System

This appendix section presents a first attempt at a model of a credit card system. We present a domain description of an abstracted **credit card system**. The narrative part of the description is terse, perhaps a bit too terse. Credit cards are moving from simple plastic cards to smart phones. Uses of credit cards move from their mechanical insertion in credit card terminals to being swiped. Authentication (hence not modeled) moves from keying in security codes to eye iris "prints", and/or finger prints and/or voice prints or combinations thereof. The description of this section abstracts from all that in order to understand a bare, minimum essence of credit cards and their uses. Based on a model, such as presented here, the reader should be able to extend/refine the model into any future technology – for requirements purposes.

2.1. Endurants

2.1.1. Credit Card Systems

[Bjø16b, Sect.3.1.6, pg.11:]: observe_part_sorts

78 Credit card systems, $ccs{:}CCS$, consists of three kinds of parts:

79 an assembly, $cs{:}CS$, of credit cards[12],

80 an assembly, $bs{:}BS$, of banks, and

81 an assembly, $ss{:}SS$, of shops.

[12] We "equate" credit cards with their holders.

```
type
78   CCS
79   CS
80   BS
81   SS
value
79   obs_part_CS: CCS → CS
80   obs_part_BS: CCS → BS
81   obs_part_SS: CCS → SS
```

The composite part CS can be thought of as a credit card company, say VISA[13]. The composite part BS can be thought of as a bank society, say BBA: British Banking Association. The composite part SS can be thought of as the association of retailers, say bira: British Independent Retailers Association[14].

[13] Our simple model allows for only one credit card company. But that model can easily be extended to model a set of credit card companies, viz.: VISA, MasterCard, American Express, Diner's Club, etc..

[14] The model does not prevent "shops" from being airlines, or car rental agencies, or dentists, or consultancy firms. In this case SS would be some appropriate

[Bjø16b, Sect.3.1.7, pg.13]: observe_part_type

82 There are credit cards, $c{:}C$, banks $b{:}B$, and shops $s{:}S$.

83 The credit card part, $cs{:}CS$, abstracts a set, $soc{:}Cs$, of card.

84 The bank part, $bs{:}BS$, abstracts a set, $sob{:}Bs$, of banks.

85 The shop part, $ss{:}SS$, abstracts a set, $sos{:}Ss$, of shops.

type
82 C, B, S
83 Cs = C-**set**
84 Bs = B-**set**
85 Ss = S-**set**
value
83 **obs_part**_CS: CS → Cs, **obs_part**_Cs: CS → Cs
84 **obs_part**_BS: BS → Bs, **obs_part**_Bs: BS → Bs
85 **obs_part**_SS: SS → Ss, **obs_part**_Ss: SS → Ss

association.

[Bjø16b, Sect.3.2, pg.16]: observe_unique_identifier

86 Assembliers of credit cards, banks and shops have unique identifiers, $csi{:}CSI$, $bsi{:}BSI$, and $ssi{:}SSI$.

87 Credit cards, banks and shops have unique identifiers, $ci{:}CI$, $bi{:}BI$, and $si{:}SI$.

88 One can define functions which extract all the

89 unique credit card,

90 bank and

91 shop identifiers from a credit card system.

```
86    CSI, BSI, SSI
87    CI, BI, SI
value
86    uid_CS: CS→CSI, uid_BS: BS→BSI, uid_SS: SS→SSI,
87    uid_C: C→CI, uid_B: B→BI, uid_S: S→SI,
89    xtr_CIs: CCS → CI-set
89    xtr_CIs(ccs) ≡ {uid_C(c)|c:C·c ∈ obs_part_Cs(obs_part_CS(ccs))}
90    xtr_BIs: CCS → BI-set
90    xtr_BIs(ccs) ≡ {uid_B(s)|b:B·b ∈ obs_part_Bs(obs_part_BS(ccs))}
91    xtr_SIs: CCS → SI-set
91    xtr_SIs(ccs) ≡ {uid_S(s)|s:S·s ∈ obs_part_Ss(obs_part_SS(ccs))}
```

92 For all credit card systems it is the case that

93 all credit card identifiers are distinct from bank identifiers,

94 all credit card identifiers are distinct from shop identifiers,

95 all shop identifiers are distinct from bank identifiers,

axiom

92 ∀ ccs:CCS ·
92 **let** cis=xtr_CIs(ccs), bis=xtr_BIs(ccs), sis = xtr_SIs(ccs) **in**
93 cis ∩ bis = {}
94 ∧ cis ∩ sis = {}
95 ∧ sis ∩ bis = {} **end**

2.1.2. Credit Cards

[Bjø16b, Sect.3.3.2, pg.18]: observe_mereology

96 A credit card has a mereology which "connects" it to any of the shops of the system and to exactly one bank of the system,

97 and some attributes — which we shall presently disregard.

98 The wellformedness of a credit card system includes the wellformedness of credit card mereologies with respect to the system of banks and shops:

99 The unique shop identifiers of a credit card mereology must be those of the shops of the credit card system; and

100 the unique bank identifier of a credit card mereology must be of one of the banks of the credit card system.

```
type
96.     CM = SI-set × BI
value
96.     obs_mereo_CM: C → CM
98      wf_CM_of_C: CCS → Bool
98      wf_CM_of_C(ccs) ≡
96        let bis=xtr_BIs(ccs), sis=xtr_SIs(ccs) in
96         ∀ c:C·c ∈ obs_part_Cs(obs_part_CS(ccs)) ⇒
96           let (ccsis,bi)=obs_mereo_CM(c) in
99           ccsis ⊆ sis
100          ∧ bi ∈ bis
96        end end
```

2.1.3. Banks

[Bjø16b, Sect.3.3.2 pg.18]: observe_mereology
[Bjø16b, Sect.3.4.3 pg.20]: observe_attributes
Our model of banks is (also) very limited.

101 A bank has a mereology which "connects" it to a subset of all credit cards and a subset of all shops,

102 and, as attributes:

103 a cash register, and

104 a ledger.

105 The ledger records for every card, by unique credit card identifier,

106 the current balance: how much money, credit or debit, i.e., plus or minus, that customer is owed, respectively has borrowed from the bank,

107 the dates-of-issue and -expiry of the credit card, and

108 the name, address, and other information about the credit card holder.

109 The wellformedness of the credit card system includes the wellformedness of the banks with respect to the credit cards and shops:

110 the bank mereology's

111 must list a subset of the credit card identifiers and a subset of the shop identifiers.

```
type
101   BM = CI-set × SI-set
103   CR = Bal
104   LG = CI ~m~> (Bal×DoI×DoE×...)
106   Bal = Int
value
101   obs_mereo_B: B → BM
103   attr_CR: B → CR
104   attr_LG: B → LG
109       wf_BM_B: CCS → Bool
109       wf_BM_B(ccs) ≡
109           let allcis = xtr_CIs(ccs), allsis = xtr_SIs(ccs) in
109           ∀ b:B · b ∈ obs_part_Bs(obs_part_BS(ccs)) in
110               let (cis,sis) = obs_mereo_B(b) in
111               cis ⊆ ∀ cis ∧ sis ⊆ allsis end end
```

2.1.4. Shops

[Bjø16b, Sect.3.3.2 pg.18]: observe_mereology

112 The mereology of a shop is a pair: a unique bank identifiers, and a set of unique credit card identifiers.

113 The mereology of a shop

114 must list a bank of the credit card system,

115 band a subset (or all) of the unique credit identifiers.

We omit treatment of shop attributes.

```
type
112   SM = CI-set × BI
value
112   obs_mereo_S: S → SM
113   wf_SM_S: CCS → Bool
113   wf_SM_S(ccs) ≡
113     let allcis = xtr_CIs(ccs), allbis = xtr_BIs(ccs) in
113     ∀ s:S · s ∈ obs_part_Ss(obs_part_SS(ccs)) ⇒
113         let (cis,bi) obs_mereo_S(s) in
114         bi ∈ allbis
115       ∧ cis ⊆ allcis
113     end end
```

2.2. Perdurants

2.2.1. Behaviours

[Bjø16b, Sect.4.11.2, pg.36]: Process Schema I: Abstract
`is_composite(p)`
[Bjø16b, Sect.4.11.2, pg.37]: Process Schema II: Concrete
`is_concrete(p)`

116 We ignore the behaviours related to the CCS, CS, BS and SS parts.

117 We therefore only consider the behaviours related to the Cs, Bs and Ss parts.

118 And we therefore compile the credit card system into the parallel composition of the parallel compositions of all the credit card, crd, all the bank, bnk, and all the shop, shp, behaviours.

```
value
116  ccs:CCS
116  cs:CS = obs_part_CS(ccs),
116  uics:CSI =uid_CS(cs),
116  bs:BS = obs_part_BS(ccs),
116  uibs:BSI =uid_BS(bs),
116  ss:SS = obs_part_SS(ccs),
116  uiss:SSI =uid_SS(ss),
117  socs:Cs = obs_part_Cs(cs),
117  sobs:Bs = obs_part_Bs(bs),
117  soss:Ss = obs_part_Ss(ss),
```

```
value
118  sys: Unit → Unit,
116  sys() ≡
118      cards_uics(obs_mereo_CS(cs),...)
118    ‖ ‖ {crd_uid_C(c)(obs_mereo_C(c))|c:C·c ∈ socs}
118    ‖ banks_uibs(obs_mereo_BS(bs),...)
118    ‖ ‖ {bnk_uid_B(b)(obs_mereo_B(b))|b:B·b ∈ sobs}
118    ‖ shops_uiss(obs_mereo_SS(ss),...)
118    ‖ ‖ {shp_uid_S(s)(obs_mereo_S(s))|s:S·s ∈ soss},
116  cards_uics(...) ≡ skip,
116  banks_uibs(...) ≡ skip,
116  shops_uiss(...) ≡ skip

axiom  skip ‖ behaviour(...) ≡ behaviour(...)
```

2.2.2. Channels

[Bjø16b, Sect. 4.5.1, pg.31]: Channels and Communications
[Bjø16b, Sect. 4.5.2, pg.31]: Relations Between Attributes Sharing and Channels

119 Credit card behaviours interact with bank (each with one) and many shop behaviours.

120 Shop behaviours interact with bank (each with one) and many credit card behaviours.

121 Bank behaviours interact with many credit card and many shop behaviours.

The inter-behaviour interactions concern:

122 between credit cards and banks: withdrawal requests as to a sufficient, mk_Wdr(am), balance on the credit card account for buying am:AM amounts of goods or services, with the bank response of either is_OK() or is_NOK(), or the revoke of a card;

123 between credit cards and shops: the buying, for an amount, am:AM, of goods or services: mk_Buy(am), or the refund of an amount;

124 between shops and banks: the deposit of an amount, am:AM, in the shops' bank account: mk_Depost(ui,am) or the removal of an amount, am:AM, from the shops' bank account: mk_Removl(bi,si,am)

channel

```
119  {ch_cb[ ci,bi ]|ci:CI,bi:BI·ci ∈ cis ∧ bi ∈ bis}:CB_Msg
120  {ch_cs[ ci,si ]|ci:CI,si:SI·ci ∈ cis ∧ si ∈ sis}:CS_Msg
121  {ch_sb[ si,bi ]|si:SI,bi:BI·si ∈ sis ∧ bi ∈ bis}:SB_Msg
122  CB_Msg == mk_Wdrw(am:aM) | is_OK() | is_NOK() | ...
123  CS_Msg == mk_Buy(am:aM) | mk_Ref(am:aM) | ...
124  SB_Msg == Depost | Removl | ...
124  Depost == mk_Dep((ci:CI|si:SI),am:aM) |
124  Removl == mk_Rem(bi:BI,si:SI,am:aM)
```

2.2.3. Behaviour Interactions

125 The credit card initiates

a. **buy** transactions

i [1.Buy] by enquiring with its bank as to sufficient purchase funds (**am:aM**);

ii [2.Buy] if NOK then there are presently no further actions; if OK

iii [3.Buy] the credit card requests the purchase from the shop – handing it an appropriate amount;

iv [4.Buy] finally the shop requests its bank to deposit the purchase amount into its bank account.

b. **refund** transactions
   i [1.Refund] by requesting such refunds, in the amount of **am:aM**, from a[ny] shop; whereupon
   ii [2.Refund] the shop requests its bank to move the amount **am:aM** from the shop's bank account
   iii [3.Refund] to the credit card's account.

Thus the three sets of behaviours, crd, bnk and shp interact as sketched in Fig. 2.

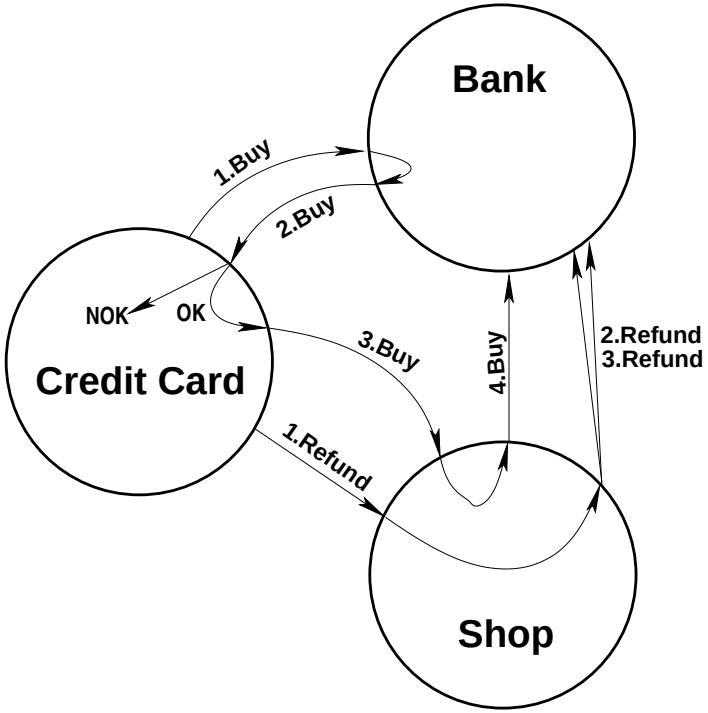

Figure 2: Credit Card, Bank and Shop Behaviours

[1.Buy]	Item 131, Pg.210	card	ch_cb[ci,bi]!mk_Wdrw(am) (shown as ... three lines down) and
	Item 140, Pg.215	bank	mk_Wdrw(ci,am)≠‖{ch_cb[bi,bi]?|ci:CI•ci ∈ cis}.
[2.Buy]	Items 133-134, Pg.210	bank	ch_cb[ci,bi]!is_[N]OK() and
	Item 131, Pg.210	shop	(...;ch_cb[ci,bi]?).
[3.Buy]	Item 133, Pg.210	card	ch_cs[ci,si]!mk_Buy(am) and
	Item 155, Pg.221	shop	mk_Buy(am)≠‖{ch_cs[ci,si]?|ci:CI•ci∈ cis}.
[4.Buy]	Item 156, Pg.221	shop	ch_sb[si,bi]!mk_Dep(si,am) and
	Item 145, Pg.217	bank	mk_Dep(si,am)≠‖{ch_cs[ci,si]?|si:SI•si∈sis}.
[1.Refund]	Item 137, Pg.212	card	ch_cs[ci,si]!mk_Ref((ci,si),am) and
	Item 156, Pg.221	shop	(si,mk_Ref(ci,am))≠‖{si′,ch_sb[si,bi]?|si,si′:SI•{si,si′}⊆sis∧si=si′}.
[2.Refund]	Item 160, Pg.221	shop	ch_sb[si,cbi]!mk_Ref(cbi,(ci,si),am and
	Item 149, Pg.219	bank	(si,mk_Ref(cbi,(ci,am)))≠‖{(si′,ch_sb[si,bi]?)|si,si′:SI•{si,si′}⊆sis∧si=si′
[3.Refund]	Item 161, Pg.221	shop	ch_sb[si,sbi]!mk_Wdr(si,am)) **end** and
	Item 150, Pg.219	bank	(si,mk_Wdr(ci,am))≠‖{(si′,ch_sb[si,bi]?)|si,si′:SI•{si,si′}⊆sis∧si=si′}

2.2.4. Credit Card

[Bjø16b, Sect. 4.11.2, pg. 37]: Processs Schema III: `is_atomic(p)`

126 The credit card behaviour, crd, takes the credit card unique identifier, the credit card mereology, and attribute arguments (omitted). The credit card behaviour, crd, accepts inputs from and offers outputs to the bank, bi, and any of the shops, si∈sis.

127 The credit card behaviour, crd, non-deterministically, internally "cycles" between buying and getting refunds.

value

126 $\mathsf{crd}_{ci:CI}$: (bi,sis):CM → **in,out** ch_cb[ci,bi],{ch_cs[ci,si]|si:SI·si ∈ sis}

126 crd_{ci}(bi,sis) ≡ (buy(ci,(bi,sis)) ⌈⌉ ref(ci,(bi,sis))) ; crd_{ci}(ci,(bi,sis))

[Bjø16b, Sect. 4.11.2, pg. 38]: Process Schemas IV–V: Core Processes (I–II)

128 By am:AM we mean an amount of money, and by si:SI we refer to a shop in which we have selected a number or goods or services (not detailed) costing am:AM.

129 The buyer action is simple.

130 The amount for which to buy and the shop from which to buy are selected (arbitrarily).

131 The credit card (holder) withdraws am:AM from the bank, if sufficient funds are available[15].

132 The response from the bank

133 is either OK and the credit card [holder] completes the purchase by buying the goods or services offered by the selected shop,

134 or the response is "not OK", and the transaction is skipped.

[15] First the credit card [holder] requests a withdrawal. If sufficient funds are available, then the withdrawal takes place, otherwise not – and the credit card holder is informed accordingly.

```
type
128  AM = Int
value
129  buy: ci:CI × (bi,sis):CM →
129     in,out ch_cb[ ci,bi ] out {ch_cs[ ci,si ]|si:SI·si ∈ sis}  Unit
129  buy(ci,(bi,sis)) ≡
130     let am:aM · am>0, si:SI · si ∈ sis in
131     let msg = (ch_cb[ ci,bi ]!mk_Wdrw(am);ch_cb[ ci,bi ]?) in
132     case msg of
133        is_OK() → ch_cs[ ci,si ]!mk_Buy(am),
134        is_NOK() → skip
129     end end end
```

135 The refund action is simple.

136 The credit card [handler] requests a refund am:AM

137 from shop si:SI.

This request is handled by the shop behaviour's sub-action ref, see lines 153.–162. page 221.

```
value
135  rfu: ci:CI × (bi,sis):CM → out {ch_cs[ ci,si ]|si:SI·si ∈ sis}  Unit
135  rfu(ci,(bi,sis)) ≡
136      let am:AM · am>0, si:SI · si ∈ sis in
137      ch_cs[ ci,si ]!mk_Ref(bi,(ci,si),am)
135      end
```

2.2.5. Banks

[Bjø16b, Sect. 4.11.2, pg. 37]: Processs Schema III: `is_atomic(p)`

138 The bank behaviour, **bnk**, takes the bank's unique identifier, the bank mereology, and the programmable attribute arguments: the ledger and the cash register. The bank behaviour, **bnk**, accepts inputs from and offers outputs to the any of the credit cards, $ci \in cis$, and any of the shops, $si \in sis$.

139 The bank behaviour non-deterministically externally chooses to accept either 'withdraw'al requests from credit cards or 'deposit' requests from shops or 'refund' requests from credit cards.

```
value
138  bnk_bi:BI: (cis,sis):BM → (LG×CR) →
138      in,out {ch_cb[ ci,bi ]|ci:CI·ci ∈ cis} {ch_sb[ si,bi ]|si:SI·si ∈ sis}  Unit
138  bnk_bi((cis,sis))(lg:(bal,doi,doe,...),cr) ≡
139      wdrw(bi,(cis,sis))(lg,cr)
139    ⌊⌋depo(bi,(cis,sis))(lg,cr)
139    ⌊⌋refu(bi,(cis,sis))(lg,cr)
```

140 The 'withdraw' request, wdrw, (an action) non-deterministically, externally offers to accept input from a credit card behaviour and marks the only possible form of input from credit cards, mk_Wdrw(ci,am), with the identity of the credit card.

141 If the requested amount (to be withdrawn) is not within balance on the account

142 then we, at present, refrain from defining an outcome (**chaos**), whereupon the bank behaviour is resumed with no changes to the ledger and cash register;

143 otherwise the bank behaviour informs the credit card behaviour that the amount can be withdrawn; whereupon the bank behaviour is resumed notifying a lower balance and 'withdraws' the monies from the cash register.

```
value
139  wdrw: bi:BI × (cis,sis):BM → (LG×CR) → in,out {ch_cb[ bi,ci ]|ci:CI·ci ∈ cis} Unit
139  wdrw(bi,(cis,sis))(lg,cr) ≡
140      let mk_Wdrw(ci,am) =⫿⫾{ch_cb[ ci,bi ]?|ci:CI·ci ∈ cis} in
139      let (bal,doi,doe) = lg(ci) in
141      if am>bal
142          then (ch_cb[ ci,bi ]!is_NOK(); bnk_bi(cis,sis)(lg,cr))
143          else (ch_cb[ ci,bi ]!is_OK(); bnk_bi(cis,sis)(lg†[ ci↦(bal−am,doi,doe) ],cr−am)) end
138      end end
```

- The ledger and cash register attributes, lg,cr, are programmable attributes.
  - ◈ Hence they are modeled as separate function arguments.

144 The deposit action is invoked, either by a shop behaviour, when a credit card [holder] buy's for a certain amount, am:AM, or requests a refund of that amount. The deposit is made by shop behaviours, either on behalf of themselves, hence am:AM, is to be inserted into the shops' bank account, si:SI, or on behalf of a credit card [i.e., a customer], hence am:AM, is to be inserted into the credit card holder's bank account, si:SI.

145 The message, ch_cs[ci,si]?, received from a credit card behaviour is either concerning a buy [in which case i is a ci:CI, hence sale, or a refund order [in which case i is a si:SI].

146 In either case, the respective bank account is "upped" by am:AM – and the bank behaviour is resumed.

```
value
144  deposit: bi:BI × (cis,sis):BM → (LG×CR) →
145        in,out {ch_sb[ bi,si ]|si:SI·si ∈ sis} Unit
144  deposit(bi,(cis,sis))(lg,cr) ≡
145        let mk_Dep(si,am) =⌊⌋⌈⌉{ch_cs[ ci,si ]?|si:SI·si ∈ sis} in
144        let (bal,doi,doe) = lg(si) in
146        bnk_bi(cis,sis)(lg†[ si↦(bal+am,doi,doe) ],cr+am)
144        end end
```

147 The refund action

148 non-deterministically externally offers to either

149 non-deterministically externally accept a mk_Ref(ci,am) request from a shop behaviour, si, or

150 non-deterministically externally accept a mk_Wdr(ci,am) request from a shop behaviour, si.

The bank behaviour is then resumed with the

151 credit card's bank balance and cash register incremented by am and the

152 shop' bank balance and cash register decremented by that same amount.

```
value
147  rfu: bi:BI × (cis,sis):BM → (LG×CR) → in,out {ch_sb[ bi,si ]|si:SI•si ∈ sis} Unit
147  rfu(bi,(cis,sis))(lg,cr) ≡
149      (let (si,mk_Ref(cbi,(ci,am))) =⨅{(si′,ch_sb[ si,bi ]?)|si,si′:SI•{si,si′}⊆sis∧si=si′} in
147       let (balc,doic,doec) = lg(ci) in
151       bnk_bi(cis,sis)(lg†[ ci↦(balc+am,doic,doec) ],cr+am)
147       end end)
148    ⨅
150      (let (si,mk_Wdr(ci,am)) =⨅{(si′,ch_sb[ si,bi ]?)|si,si′:SI•{si,si′}⊆sis∧si=si′} in
147       let (bals,dois,does) = lg(si) in
152       bnk_bi(cis,sis)(lg†[ si↦(bals−am,dois,does) ],cr−am)
147       end end)
```

2.2.6. Shops

[Bjø16b, Sect. 4.11.2, pg. 37]: Processs Schema III: `is_atomic(p)`

153 The shop behaviour, shp, takes the shop's unique identifier, the shop mereology, etcetera.

154 The shop behaviour non-deterministically, externally

either

155 offers to accept a Buy request from a credit card behaviour,

156 and instructs the shop's bank to deposit the purchase amount.

157 whereupon the shop behaviour resumes being a shop behaviour;

158 or

159 offers to accept a refund request in this amount, am, from a credit card [holder].

160 It then proceeds to inform the shop's bank to withdraw the refund from its ledger and cash register,

161 and the credit card's bank to deposit the refund into its ledger and cash register.

162 Whereupon the shop behaviour resumes being a shop behaviour.

```
value
153  shp_si:SI: (CI-set×BI)×...→in,out: {ch_cs[ci,si]|ci:CI•ci ∈ cis},{ch_sb[si,bi']|bi':BI•bi'isin bis} Unit
153  shp_si((cis,bi),...) ≡
155      (sal(si,(bi,cis),...)
158     ⌊⌋⌈
159      ref(si,(cis,bi),...)):

153  sal: SI×(CI-set×BI)×...→in,out: {cs[ci,si]|ci:CI•ci ∈ cis},sb[si,bi] Unit
153  sal(si,(cis,bi),...) ≡
155      let mk_Buy(am) =⌊⌋⌈{ch_cs[ci,si]?|ci:CI•ci ∈ cis} in
156      ch_sb[si,bi]!mk_Dep(si,am) end ;
157      shp_si((cis,bi),...)

153  ref: SI×(CI-set×BI)×...→in,out: {ch_cs[ci,si]|ci:CI•ci ∈ cis},{ch_sb[si,bi']|bi':BI•bi'isin bis} Unit
159  ref(si,(cis,sbi),...) ≡
159      let mk_Ref((ci,cbi,si),am) =⌊⌋⌈{ch_cs[ci,si]?|ci:CI•ci ∈ cis} in
160      (ch_sb[si,cbi]!mk_Ref(cbi,(ci,si),am)
161     ‖ ch_sb[si,sbi]!mk_Wdr(si,am)) end ;
162      shp_si((cis,sbi),...)
```