DADL: A Domain Analysis and Description Language*

Dines Bjørner†

February 14, 2026: 22:34

Abstract

We informally present a domain analysis and description language, DADL.

The domain analysis sub-language, DAL, consists of (i) a domain analysis ontology – in the form of a [visual] graph – and (ii) a number of prompts. The prompts are not computable functions – their result is calculated – in the head of the domain analyzer cum describer.

The domain description sub-language, DDL, consists of a dozen description units. Invocation of these description units is not computable nor is the text they "produce" – their result is again calculated – in the head of the domain analyzer cum describer.

Predicate DAL prompts "stear" the domain analyzer across the domain analysis ontology and some triggers [DAL] function prompts **and** the invocation of [the DDL] description units.

Whereas analysis consists of the domain analyzer mentally invoking various prompts, invocation of DDL units results in actual [fragments of] descriptions being "written down".

Domain analysis and description is based on a method – described in referenced literature. We shall here concentrate on a major method tool. Method principles, procedures and techniques are hinted at – but otherwise we refer to this literature.

So behind all this there is a method. Careful application of this method results in the domain analyzer cum describer actually "calculating" domain models from their "studied" observations; not computing, but "extracting" from the physical, actual domain, a suitable description.

Contents

1 Introduction **3**
1.1 The Emergence of DAL . . . 3
1.2 The Emergence of DDL . . . 4
1.3 On What Descriptions Describe . . . 4
1.3.1 The Analysis Prompts, I . . . 5
1.3.2 A Transcendental Deduction: From Endurants to Perdurants . . . 6
1.3.3 The Analysis Prompts, II . . . 6
1.4 Auxiliary Notions . . . 7
1.4.1 Sorts, Types and Values . . . 7
1.4.1.1 Sorts and Types: . . . 7
1.4.1.2 Values: . . . 7
1.4.2 Identifiers, Names and Patterns . . . 7
1.4.2.1 Identifiers: . . . 7
1.4.2.2 Names: . . . 7
1.4.2.3 Patterns: . . . 7
1.5 Taxonomy . . . 8
1.6 General Ontology and Taxonomy . . . 9
1.7 A Few Words on the Style of Presentation . . . 9

*INCOMPLETE DRAFT – WORK IN PROGRESS [FEBRUARY–MARCH 2026]

†Revised: 0 0 0 This document is developed as part of my quest to better understand the concept of *Domain Science & Engineering* such as it has evolved since 2009, notably since 2015: [7, 4, 8, 9, 5, 10, 12, 20, 22, 23, 32, 29, 30, 28, 31]. I AM STILL TRYING TO UNDERSTAND

2 The Domain Analysis Language, DAL 9

3 How to Create a Domain Description 10

4 The Domain Description Language, DDL 11
4.1 Universe of Discourse Definition . . . 12
4.2 Type Definitions . . . 13
4.2.1 Endurant Sorts and Types . . . 13
4.2.1.1 Cartesian Parts . . . 13
4.2.1.2 Part set Parts . . . 14
4.2.1.4 Unique Identifiers . . . 15
4.2.1.5 Mereology . . . 16
4.2.1.6 Attributes . . . 17
4.2.1.7 Intentional Pull . . . 19
4.2.2 Auxiliary Types . . . 19
4.3 Value Definitions . . . 21
4.3.1 Non-function Value Descriptions . . . 21
4.3.2 Function Value Descriptions . . . 22
4.4 Axiom Definitions . . . 23
4.5 Channel Declarations . . . 23
4.6 Variable Declarations . . . 24
4.7 Behaviour Definitions . . . 25
4.8 Endurants with Monitorable Attributes . . . 26
4.9 Behaviour Initialization . . . 26

5 Auxiliary Clauses 27
5.1 SPACE and TIME . . . 27
5.1.1 SPACE . . . 27
5.1.2 TIME . . . 28
5.2 Expressions . . . 29
5.2.1 The "let" ... "in" ... "end" Expressions . . . 29
5.2.1.1 Patterns: . . . 29
5.2.1.2 The Binding Expression: . . . 30
5.2.2 Arithmetic Expressions . . . 30
5.2.3 Boolean Expressions . . . 30
5.2.4 "chaos" . . . 30
5.2.5 Set, Cartesian, List and Map Enumerations . . . 31
5.2.6 Set Expressions . . . 31
5.2.7 Cartesian Expressions . . . 31
5.2.8 List Expressions . . . 31
5.2.9 Map Operator Types and Expressions . . . 32
5.2.10 Set, List and Map Comprehensions . . . 32
5.2.11 Conditional Expressions . . . 32
5.3 Statements . . . 34
5.4 CSP Clauses . . . 34

6 Closing 35
6.1 What Have We Achieved ? . . . 35
6.2 Engineering, Research, Science . . . 35
6.3 Further Domain Research – hopefully Domain Science . . . 36
6.4 Acknowledgments . . . 36

7 Bibliography 36

A Behaviour Patterns **41**
A.1 Behaviour Defintions . . . 41
A.2 Action Definitions . . . 43

B Example **43**
B.1 Universe of Discouse . . . 43
B.2 Compound Endurants . . . 44
B.3 Endurant Values . . . 44
B.4 Taxonomy . . . 45
B.5 Unique Identification . . . 45
B.6 Unique Identifier Values . . . 46
B.7 Uniqueness of Endurants . . . 46
B.8 Mereology . . . 47
B.9 Attributes . . . 47
B.10 Intentional Pull . . . 48
B.11 Auxiliary Types . . . 49
B.12 Simple Function Values . . . 49
B.13 Channel . . . 51
B.14 Variables . . . 51
B.15 Behaviours . . . 51
B.16 Initialization . . . 54

C SI Units **54**

The Triptych Dogma:

Before software can be designed we must understand "its" requirements.
Before requirements can be prescribed we must understand "the" domain.
So we must study, analyze and describe the domain.

1 Introduction

In [7, 4, 8, 9, 5, 10, 12, 20, 22, 23, 32, 29, 30, 28, 31] I, step-by-step, develop a concept of *Domain Science & Engineering*. The book [22] and the follow-up "book"[1] [32] reflects my 2020, respectively 2025 understanding. They present a method[2] for the analysis & description of domains[3].

We shall in this document focus on a major tool: the DADL.

We use the term DADL as a singular term for what really is two languages: an analysis language, DAL, and a description language, DDL.

1.1 The Emergence of DAL

DAL emerged as the result of the study of domain modeling [7, 4, 8, 9, 5, 10, 12, 20, 22, 23, 32, 29, 30, 28, 31]. DAL, at present, "consists" of a dozen or so predicate prompts (`is_...`) and two function prompts (`analyze_···_names`). DAL is not a specification language in that the above predicates and functions are written down. They are names of prompts where these prompts are meant to be "executed" by, i.e., in the minds of domain analyzers. That is, the DAL is a tool set. Deployment of each of its tools "guides" the analyzer from step to step.

[1] – to be submitted

[2] By a method we shall understand a set of principles, procedures, techniques and tools for studying and recording [cum creating] and artifact.

[3] By a domain we shall understand a *rationally describable segment of a discrete dynamics fragment of a human assisted reality: the world that we daily observe – in which we work and act, a reality made significant by human-created entities [or – describable by means of the tools provided by the method !].*

The concept of having a tool set like DAL is new. It is the result of reflections on what it takes to "create" domain descriptions. It is thus a contribution to software cum domain modeling methodology.

1.2 The Emergence of DDL

DDL is a derivative of RSL [40], the RAISE [41] Specification Language. It does not include the *scheme, class, object* structuring constructs nor statements. The former because the DDL concept of domain description units "replaces" structuring concepts. The latter because there is no need for statements!

First hints of a special, formal languages for describing domains appeared in [8, 9]. A background for "merging" of the concepts of mereology [as first proposed by the Polish mathematician *Stanisław Leśniewski*[4] and as propagated at IFIP WG 2.3 meetings in the 1980s and 1990s by Douglas T. Ross] and Tony Hoare's CSP [42, 45] was given in [6, 18]. First "formal" presentations of the special domain analysis & description method in which the language for analyzing and describing domains were published in [13, 21] followed up by the book [22]. Further papers ensued [17, 16, 27, 24, 29, 25] – as did a great number of technical reports outlining specific domains: [11, 14, 36, 15, 19, 26, 11, 14, 28, 30].The formal analysis & description language "derived" from the RAISE [41] specification language RSL [40] – which, in turn, "derived" from VDM [34], a first, formal method for software development – which emerged from the 1973–1974 IBM Vienna Labor PL/I project [2]. The PL/I definitions of the 1960s were developed in the context of John McCarthy's [52, 53], Peter Landin's [47, 48, 51, 50, 49, 37] and Christopher Strachey's [73] and was decisively predicated by Dana Scott's [63, 64]. We shall refer to the formal domain analysis & description language of this paper by DADL. DDL emerged over the years 2010–2025 as a result of, first of number of engineering case studies [23], then as the papers and book referred to above. It is time to rigorously outline DDL. A formal definition is in the waiting, i.e., to be next.

1.3 On What Descriptions Describe

[5] Programming languages are meant to specify computations. Domain analysis and specification languages, like DADL, are tools, meant to analyze and describe domains. They are not the domains [they describe]. They designate abstract, mathematical models of domains. Domain descriptions are not meant for execution! Domain descriptions are developed by people in order, for them and readers, to understand [a domain]. Domain descriptions may be subject, as an initial step, to software development. From domain descriptions one may be able to "derive" requirements prescriptions – as outlined in [22, *Chapter 9*] – and from requirements prescriptions one may be able to "derive" software designs – as outlined in [3].

So, what is it, in the domains, that we can and wish to describe?

That is, how do we characterize domains?

> By a *domain* we shall understand a *rationally describable* segment of a *discrete dynamics* fragment of a *human assisted* reality, i.e., of the world. It includes its *endurants*, i.e., *solid and fluid entities* of *parts* and *living species*, and *perdurants*. Endurants are those quantities of domains that we can observe (see and touch), in *space*, as "complete" entities at no matter which point in *time* – "material" entities that persist, endure. Perdurants are those quantities of domains for which only a fragment exists, in *space*, if we look at or touch them at any given snapshot in *time*. Perdurants, in the form of behaviours, derive, by transcendental deduction, from endurant parts.

So that is what we wish to describe in domains: endurants and perdurants. Endurants as ["static"] values. Perdurants as behaviours.

[4] https://plato.stanford.edu/entries/lesniewski/

[5] This section, especially pages 5–??, took me a lot of time to formulate. I was not quite in best of health. High blood sugar (diabetes 2). Did not sleep at night, really. Sat most of the day in my ear flap chair dozing off.

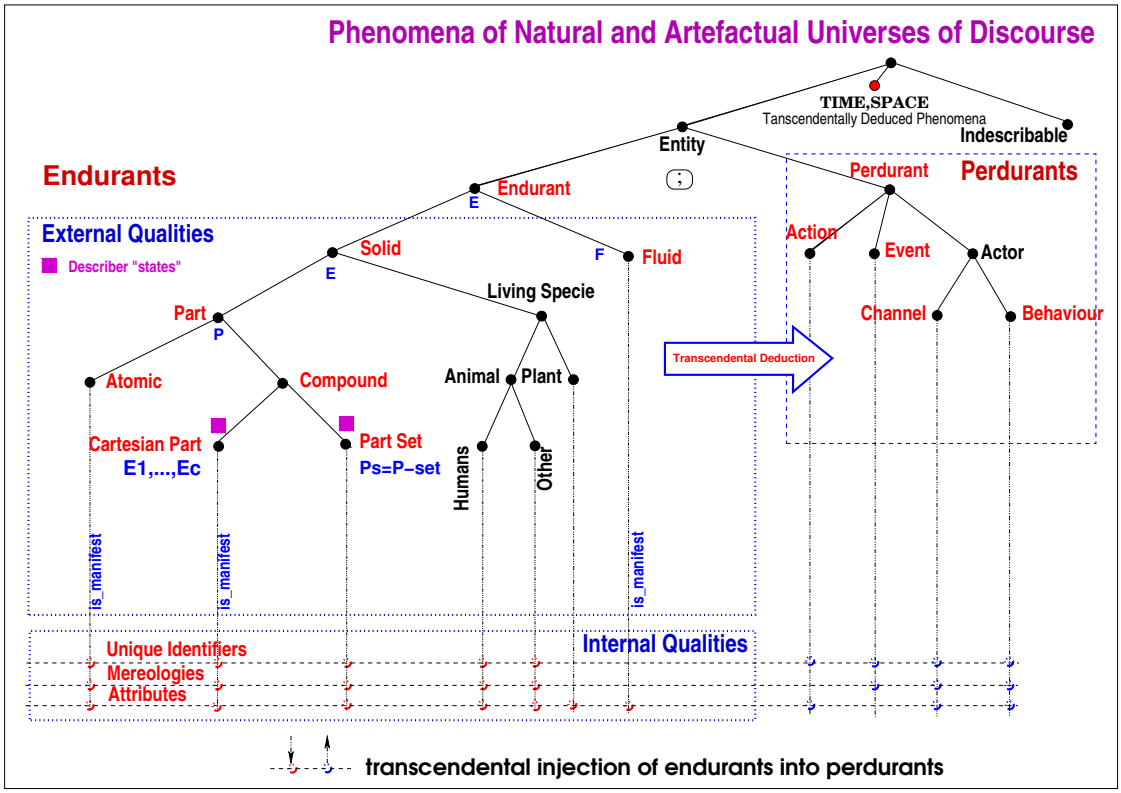

Figure 1: A Domain Analysis & Description Ontology

As outlined in the referenced papers and book of domain analysis & description, the study of domains is suggested performed in the context of endurants and perdurants such as summarized in Fig. 1.

Now, what comes next is 'a proposed theory'

It is postulated that the *domain analysis & description ontology* is a proposed theory, i.e., a possible domain analysis & description ontology. There could be others! Over the last almost 10 years, since [13, 21], I have questioned this ontology and I have tried envisage variants. I am obviously not 100% certain that this is "the ontology"!

The foundation for my doubts and search is my study of the Danish philosopher Kai Sørlanders books [65, 66, 67, 68, 69, 70] – the last one on which I translated, and Springer published [71].

1.3.1 The Analysis Prompts, I

So this is what I have so far come up with:

Phenomena are what humans observe in domains. Some phenomena are rationally describable – the entities, some are not. Entities are either endurants or perdurants. Endurants are either solid or fluid. Solids are wither parts or are living species. Parts are either atomic or compound. Compounds are either Cartesians or are part Sets[6].

Some endurants have internal qualities: *unique identities*, *mereologies* and *attributes*. Parts are uniquely identifiable. Mereologies express relations between solids and are here expressed in terms of unique [part] identifiers. Attributes are then what "gives life" to endurants.

[6]– for each of concepts: *solid, fluid, parts, living species, atomic, compound, Cartesians* and *part sets* – there is then given a reasonably precise characterization of that concept — enough for domain analyzers cum describers to make appropriate modeling choices.

1.3.2 A Transcendental Deduction: From Endurants to Perdurants

Perdurants emerge by transcendental deduction from manifest solids. In the transcendental deduction of the meaning of something, one goes from transcendence[7] to a [here] rational understanding.

We shall informally explain what us going on in this '*transcendental deduction*'. You have a part, i.e., a manifest, solid, observable endurant. It has unique identification, mereology and [one or more] attributes. You observe, see and touch, it, as an endurant: in *space*, as "complete" entities at no matter which point in *time* – "material" entities that persist, endure. "After" transcendentally deducing endurants into perdurants, we observe those quantities of domains for which only a fragment exists, in *space*, if we look at or touch them at any given snapshot in *time*. And what we claim to observe is a perdurant: the behaviour of the manifest, solid, observable endurant. That behaviour is modeled as a [perhaps forever cyclic, i.e., never-ending], function.

We associate the internal qualities of the underlying endurant as follows: the unique identifier, ι, uniquely identifies "that" behaviour. The endurant mereology specifies which other behaviours ι can [CSP] communicate (synchronize and exchange messages] with. The attributes are [the conventional function] arguments.

A Road Transport Example: A Narrative

> A road transport system, seen as an endurant, is a [Cartesian] composite of an aggregate of roads and an aggregate of automobiles. Road aggregates can be seen as a Cartesian of aggregates of hubs (street intersections) and aggregates on links (street segments [between adjacent hubs]). Hub aggregates are set of hubs, and hubs are seen as atomic parts. Link aggregates are set of links, and links are seen as atomic parts. Automobile aggregates are set of automobiles, and automobiles are seen as atomic parts. Hubs, links and automobiles have unique identification. The mereology of a hub is the set of pairs of unique link identifiers incident upon / emanating from the hub. The mereology of a link is a set of one or two hub identifiers [of the connecting hub[s]]. The mereology of an automobile is the set of identifiers of the hubs and links it may travel. Some attributes: link length, automobile velocity, automobile position (on a link, or at a hub), etc.
>
> An automobile, parked on a link, represents an endurant. Its position does not change, An automobile, moving along a link, represents a perdurant. Its position changes. Automobile-moves between hubs and links represent events.

So, as we analyze endurants into their external–, respectively internal qualities, we analyze perdurants, into behaviours – with these communicating, in the style of CSP, over channels. Behaviours are

- sets of sequences of expression evaluations (and name binding)[8],
- synchronization and communication between behaviours and
- behaviours.[9]

1.3.3 The Analysis Prompts, II

We summarize some of these derived concepts, mentioned above, and graphed in Fig. 1 on the previous page.

[7] Transcendence: the quality or state of exceeding ordinary limits, surpassing, or going beyond the physical world, often implying a spiritual or philosophical elevation to a higher level of existence, consciousness, or understanding, such as rising above the material and overcoming limitations to experience supreme excellence or connectedness [Webster]

[8] in so-called **let** clauses

[9] In earlier papers I characterized behaviours as [finite] sets of sequences of actions, events and behaviours. That is, of course, rubbish! I apologize.

- domain $\iota??\,\pi??$
- entity $\iota??\,\pi??$
- endurant $\iota??\,\pi??$
  - external quality $\iota??\,\pi??$
    - solid (entity) $\iota??\,\pi??$
    - fluid (entity) $\iota??\,\pi??$
    - part $\iota??\,\pi??$
    - living species $\iota??\,\pi??$
    - atomic part $\iota??\,\pi??$
    - compound part $\iota??\,\pi??$
    - Cartesian parts $\iota??\,\pi??$
    - part sets $\iota??\,\pi??$
  - internal quality $\iota??\,\pi??$
    - uniqueness $\iota??\,\pi??$
    - mereology $\iota??\,\pi??$
    - attributes $\iota??\,\pi??$
- perdurant $\iota??\,\pi??$
  - state
  - channel $\iota??\,\pi??$
  - action $\iota??\,\pi??$
  - event $\iota??\,\pi??$
  - behaviour $\iota??\,\pi??$

To describe a domain is to describe the "language spoken" by domain actors – whether they be humans or the behaviours of [manifest] parts. Programming language programs usually prescribe specific, orderly, "predictable" ([concurrent] sets of) sequences of behaviours. Domains usually imply (external, $[]$, or internal, $\sqcap$) non-deterministic[10] behaviours.

1.4 Auxiliary Notions

1.4.1 Sorts, Types and Values

1.4.1.1 Sorts and Types: By a *sort* we shall understand a further undefined "set of abstract values".

type Sort

By a *type* we shall understand a defined quantity: a concretized mathematical value: a number, a Boolean, a set, a Cartesian, a list, a map or a function (type), or combination of these.

type Type = Type−Expression

1.4.1.2 Values: By a *value* we shall understand, for example, some mathematical entity: a set (of values), a Boolean, a number, a Cartesian (value), a list (of values), a map (a discrete, enumerable function of values into values), or a function (of values into values).

1.4.2 Identifiers, Names and Patterns

1.4.2.1 Identifiers: By an *identifier* we shall understand a sequence of one or more, say alphabetic characters with, perhaps, a leading, intermediate and/or trailing sequence of one or two [indexed] digits: $abc, 1abc, a_1b_2c_3, abc_{123}$. The use of underscores may be allowed: _: *alpha_013*

1.4.2.2 Names: By a *name* we shall understand an identifier which is "assigned" to a sort, type or value, as well, as we shall see later, to an axiom, a variable or a channel!

1.4.2.3 Patterns: By a *pattern* we shall understand a *structured name:* A structured name is either an identifier, or is a *set pattern*, or is a *list pattern*, or is a *map pattern*.

A *set pattern* is of the schematic, i.e., suggestive, form $\{a,...\} \cup s$ where *a,...* are names of set elements and *set* the name of a set.

A *list pattern* is of the schematic, i.e., suggestive, form $\langle a,...\rangle\hat{}\ell$ where *a,...* are names of list elements and *set* the name of a list.

A *map pattern* is of the schematic, i.e., suggestive, form $[\,a{\mapsto}b,...\,] \cup map$ where *a,b,...* are names of map definition and range elements and *map* the name of a map.

[10] These are notions associated, for example, with CSP [43] behaviours.

1.5 Taxonomy

The Internet[11] describes taxonomy as follows: *Taxonomy is the science of defining, naming, and classifying organisms into hierarchical groups (taxa) based on shared characteristics and evolutionary relationships.* It uses a standardized system of ranks: *Domain, Kingdom, Phylum, Class, Order, Family, Genus, Species* to organize life, often utilizing binomial nomenclature for unique identification[12]

We shall instantiate this description into: A domain taxonomy is syntactically a tree structure – to reflect a conceived hierarchy. Tree nodes are here shown as square boxes. Its descendant nodes are of three kinds:

A: A single, suitably labeled, A, square box.

B: A square, suitably labeled square ("root") box, B, with m suitably labeled (B1, B2, ..., Bm) descendants shown as square boxes connected to the root box by simple lines.

C: A square, suitably labeled square ("root") box, C, with a subordinate labeled box, CS, which again has n suitably labeled (C1, C2, ..., Cn) square boxes. Appropriate boxes connected by simple lines.

Figure 2 illustrates these three cases:

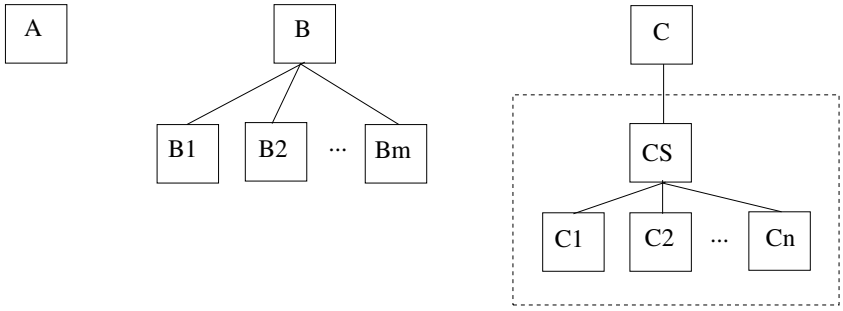

Figure 2: Taxonomy Elements

The dashed line around the CS, C1, C2, ..., Cn sub-tree are usually omitted in our taxonomy diagrams.

A [full] domain taxonomy now displays

- The universe of discourse as a root node.
- It is "just" a node depicting an endurant.
- If an endurant is atomic, A, then that endurant is just a square box.
- If the endurant is Cartesian, then that endurant is as the B tree with Bi being elements of the Cartesian.
- If the endurant is a part set, then that endurant is as the C tree with CS being a sort name for the set {C1,C2,...,Cn}.

Taxonomies are not ontologies.

[11] https://www.google.com/search?client=ubuntu-sn&channel=fs&q=Taxonomy

[12] This characterization, in terms of *Domain, Kingdom, Phylum, Class, Order, Family, Genus, Species* reveals the origin of the term 'taxonomy', namely the Swedish biologist Carl Linnaeus `https://en.wikipedia.org/wiki/Carl_Linnaeus`.

- Ontology is *the branch of metaphysics dealing with the nature of being – a set of concepts and categories in a subject area or domain that shows their properties and the* **relations** *between them.*

We refer to Appendix B.4 on page 45 for an example.

1.6 General Ontology and Taxonomy

The ontology of Fig. 1 on page 5 is one of possibly many for the study of domains. And the taxonomy of Fig. 5 on page 45 (Appendix) is, likewise, one of [definitely] many for the kind of domains that our approach to domain modeling can tackle.

Suppose we could indeed research/develop "alternative" ontologies [to that of Fig. 1 on page 5]. By thought-experiment we could then conceivable express their relationship in a taxonomy of domains! (?).

And, likewise, almost vice versa, suppose we could indeed research/develop ***

MORE TO COME

1.7 A Few Words on the Style of Presentation

The style of this paper may be unusual. Conventional presentations of specification cum programming languages usually presents their syntax, informally and formally, and then their semantics, likewise informally and formally. We do not do so here. Conventional programming languages are typically oriented towards prescribing *algorithms*. Usually that is, as bound by the computers and which their programs are interpreted, expressed by sequences of statements, oftentimes allowing for embedded blocks defining or redefining names. Domain descriptions as here suggested and practiced, instead, are expressed in terms of *properties* of the describable phenomena, i.e., entities, observed in the domain – one-by-one. That is, entities that are more-or-less manifestly observable. Some are endurants with internal qualities like unique identification, mereology and attributes. Some are perdurants: behaviours communicating over channels, etc.

2 The Domain Analysis Language, DAL

The domain analysis language was hinted at in Sects. 1.3.1 on page 5 and 1.3.3 on page 6.

There are two facets to the domain analysis language.

- A visual analysis ontology: Figure 1 on page 5;
- and the following, by now self-explanatory, analysis **prompts:**
  - the predicate prompts:
    - `is_entity`,
    - `is_endurant` and `is_perdurant`,
    - `is_solid` and `is_fluid`,
    - `is_part` and `is_living_species`,
    - `is_atomic` and `is_compound`, and
    - `is_Cartesian` and `is_part_set`
  - and the sort and type name "generating" prompts:

    `analyze_compound_names` and `analyze_attribute_names` functions.

Figure 1 on page 5 "sets the stage" for the domain analyzer. At *external* and *internal qualities* steps, the analyzer **examines** an endurant. In this examination that person, so-to-speak "mentally", moves down the ontology tree. For every `Cartesian` and `part set` step the domain analyzer "shifts" to become a domain describer – **describing**, as per the "instructions" of Sects. 4.1 on page 12, 4.2.1 on page 13– 4.9 on page 26, the external qualities of endurants. Then the analyzer "moves on" to analyze and describe *internal qualities*. Once these are described the analyzer cum describer "moves on" to examines the deducible perdurants!

3 How to Create a Domain Description

1. You are one or a few, say one or two, domain analyzers cum describers.
2. It is day 1.
3. You are given or you, yourself, chooses a domain.
4. You, so-to-speak, start/are at the root of what is meant by Fig. 1 on page 5.
5. Is the domain *describable*?
6. If so, you "proceed" to "stage: *Entity*.
7. Please note the [;]. It means to say that you first examine what you observe as an endurant; i.e., "move down left" !
8. Is what you observe *solid* or *fluid*?
9. "Move down", left or right, accordingly.
10. If left: is it is *part* or a *living species*?
11. "Move down", left or right, accordingly.
12. If left: is it *atomic* or a *compound*?
13. "Move down", left or right, accordingly.
14. Right: is it a *Cartesian* or a *part set*?
15. "Move down", left or right, accordingly.
16. Left: the part endurant is analyzed into its components, as outlined in Sect. 4.2.1.1, page 13.
17. The result of that analysis is `RSL` text – which is "added" to the emerging domain description.
18. Right: similarly, Sect. 4.2.1.2, to items 16–17.

And so on:

19. For each compound analyzed and described, items 16–17, the analyzed endurants, i.e., the elements of the Cartesian and the part set, are set aside.
20. Each of these are then analyzed, from the "root" of Fig. 1 on page 5.
21. Once all parts have been so analyzed and described you may wish to describe the taxonomy of the domain – as per Sect. 1.5 on page 8.
22. And: once all parts have been so analyzed and described you may wish to define a set of global endurant part values – as per Sect. 4.3 on page 21.
23. Then you "move on" to analyze internal qualities, first *unique identification*, then *mereology*, then *attributes*, then, possibly, if relevant, *intentional pull* – as per Sects.4.2.1.3–4.2.1.6.
24. Once you have described *unique identification* you may wish to define a set of global part identifier values – again as per Sect. 4.3 on page 21.
25. Once that is done you should define the uniqueness of parts, as an axiom, and as per Sect. 4.4 on page 23.

    Then you are ready for the "final stages", i.e., the transcendental deduction"
26. First you declare a global channel variable as per Sect. 4.5 on page 23.
27. Then you may wish to declare global, typical endurant part variables – for the case that extension or reduction of the number of domain parts is an issue. We refer to Sect.??? of [].
28. Now you are ready to define the behaviours, as per Sect. 4.7 on page 25 —
29. and, finally, the initialization of the domain, as per Sect. 4.9 on page 26.

What is a possible role, one amongst several, of the taxonomy?

It is, we suggest, to help You order the sequence in which You analyze and describe the external and internal qualities of domains.

Here it is suggested that You do so corresponding to a *depth-first, left-to-right* "traversal" of the taxonomy tree!

We find that doing so, and for the given domain analysis & description ontology, Fig. 1 on page 5, names of domain concepts, whether they be of parts, unique identifiers, mereologies and attributes, are defined before they are possibly used.

• • •

The above approach, with one, at most two, domain analyzers cum describers, applies, we suggest in situations where the aim is either research into [possibly specific or any] domains.

If the aim is to prepare for a subsequent, "real-life", "large-scale", development of, for example, software for a specific domain, then the above 1–2 person, experimental domain model development should be a first step. Based on the outcome, considering the complexity or "size" of the effort, one should then, we advice – since we have experience of this from several such projects – that a larger domain modeling effort succeed the experimental domain model development. Now, based on the size, structure and complexity (however one may assess these) of the initial model a proper, usually around 5–6 domain analyzer cum describer group is set up. Each member being "assigned" to model all aspects for a specific one, or a few endurants. Et cetera.

4 The Domain Description Language, DDL

So here we go!

Domain descriptions consists of

- one **universe of discourse** definition,
- one or more **type** definitions,
- one or more **value** definitions[13],
- one or more **axiom** definitions,
- zero or one **variable** declaration,
- one **channel** declaration,
- two or more **behaviour** definitions, and
- one **domain initialization**.

Value definitions are basically of three kinds:

- (i) **non-function values** Sects. 4.3.1, B.3 and B.6
- (ii) **simple function values** Sects. 4.3.2 and B.12
- (iii) **behaviours** Sects. 4.7 and B.15

We separate their description units into two: on-function and simple function values, Sect. 4.3 on page 21, and behaviours, Sect. 4.7 on page 25.

• • •

The purpose of description units is to introduce identifiers and define names for [external and internal qualities of] endurants and for perdurants. No two description units introduce the same identifiers. By 'definition' we mean that a [new, fresh, i.e., not elsewhere introduced] name has been introduced and associated with the right hand, i.e., defining side, of a/the definition. By 'declaration' we mean that a [new, fresh, i.e., not elsewhere introduced] name, has been introduced and associated with the right hand, ie., declaring side, of a/the declaration – ***and*** where the meaning of the declared name is a ***state*** component[14]. Any set of [thus well-formed] description units introduce a set of names. These description units define an *environment, ρ,* which "binds" these names to their meaning - whether they be sorts, types, values, axioms, channels or variables.

• • •

[13]– of non-function values as well as simple, auxiliary functions

[14]The concept of a ***state*** ...

A domain description is a set of description units.

There are two facets to this: the development of such sets and their presentation.

We refer to Fig. 1 on page 5.

Both the development and its written presentation can be said to emerge as the result of a *depth-first, left-to-right traversal* of the graph of Fig. 1.

Thus *endurants* are analyzed "before" *perdurants; solids* before *fluids*[15]; *parts* before *living species*[16], etc. *Atomic parts* emerge as the result of analyzing *compound parts*.

Once the *external qualities* of endurants have [all] been analyzed and described it is then suggested that their *internal qualities* be analyzed and described – say in the order the *external qualities* atomic and compound parts were developed.

And then the perdurants can be developed and presented.

• • •

In the **Informal Semantics** sub-paragraphs of each of the description unit presentations next there is [more or less explicit] mention of a **ubiquitos environment,** ρ.

Formal presentations of the semantics of programming languages, traditionally introduces a semantic quantity, an environment, ρ, that binds names to their meaning.

We imply the same kind of technique, i.e., this use of binding description unit names to their meaning, to explain, however informally, the meaning of domain descriptions.

When we state, in the various *Informal Semantics* sub-paragraphs below, that a description unit name is bound in this, the ubiquitos environment, we mean that all names[17] introduced are bound in this environment to their meaning. The semantics of the constructs, i.e., type expressions, value expressions, function definition bodies, etc., is then understood, i.e., evaluated, in this environment.

Here we must "warn" the reader: As You read the many next sections[18] You will read them sequentially, one after the other – but perhaps not in the order presented here. Their *'Informal Semantics'* paragraphs are not to imply that the ubiquitos environment is established in [some] sequential order. Oh No! The ρ is established in *one fell swoop!*[19] Sort of: "all in parallel".

The reader is [really] otherwise assumed to be reasonably learned wrt. formal semantics of programming languages.

4.1 Universe of Discourse Definition

There is [exactly] one *universe of discourse definition* in any domain description.

4.1.1 Purpose: The [one] universe of discourse definition serves to "open the ball!": to introduce, usually in an informal manner, what it is all about.

4.1.2 Syntax: The syntax will be expressed in a language "similar" to DADL, [however] in *slanted font*.

[15] We shall not cover 'fluids' in this document, but they are implicitly covered

[16] We shall not cover 'living species' in this document, but they are implicitly covered

[17]

- *sort names,*
- *type names,*
- *unique identifiers,*
- *unique identifiers observer names,*
- *mereology names,*
- *mereology observer names,*
- *attribute names,*
- *attribute observer names,*
- *value names,*
- *simple function names,*
- *the channel name,*
- *variable names,*
- *behaviour names*

[18] Those are: Sect. 4.1, Sect. 4.2 with its subsections 4.2.1 [with their many paragraphs: 4.2.1.1–.6] and subsection 4.2.2, and Sects. 4.3–4.9

[19] *one fell swoop* is an idiom meaning to do something suddenly, all at once, or in a single, swift action.

30. There is the keyword **Universe of Discourse**.
31. There is a type identifier,
32. And there is an informal **Narrative** text. This text "explains" *what the universe, i.e., the domain, "is all about"*.

30. **Universe of Discourse***:*
31. **type** *UoD*
32. **Narrative***: Text*

4.1.3 Semantics: The identifier, UoD[20] is introduced and become the name of the [type of] the domain being analyzed & described.

4.1.4 Pragmatics: The *Text* has no "meaning" ! The universe of discourse identifier, UoD, can be considered the sort name of the "root" endurant – from which further endurants may "flow". That is, can usually be considered the sort name of a solid endurant, and usually composite.

4.1.4 Example: We refer to appendix section B.1 on page 43.

4.2 Type Definitions

There are two kinds of type/sort definitions: (i) there are the definitions of the type of domain concepts: the external and internal qualities of endurants, and (ii) there are the definitions of what we might refer to as auxiliary types.

4.2.1 Endurant Sorts and Types

Each endurant type description unit is of kind *endurant, unique identifier, mereology* or *attribute*.

4.2.1.1 Cartesian Parts

There are usually several sort and observer definitions of parts. It starts with the universe of discourse sort – which is usually composite. We first deal with Cartesian compounds.

4.2.1.2.0 Analysis: The endurant, `e:E`, being analyzed satisfies the predicate prompt `is_Cartesian_part` and the functional analysis prompt `analyze_compound_names` yields a set of endurant names `E1, E2, ..., En`.

4.2.1.2.1 Purpose: To "unravel" the endurant sorts and elements of Cartesian compounds.

4.2.1.2.2 Syntax:

33. There is a keyword **Cartesian part sort[s] and observer[s].**
34. There is a keyword **Narrative.**
    (a) It refers to which endurant is being analyzed;
    (b) of which kind, here `Cartesian part`,
    (c) and that there is a distinct **obs_**erver function for each component.
35. And there is a keyword **Formalization.**
    (a) It starts with the literal **type**
    (b) and enumerates the names of the Cartesian elements as per the functional analysis prompt.
    (c) It then "continues" with the literal **value**

[20] – or any such as chosen by the domain analyzer cum describer

(d) and then enumerates the n **obs**_erver functions, one for each Cartesian element.

33. E **Cartesian part sort[s] and observer[s]:**
34. **Narrative:**
34a. The part being observed is E.
34b. It is analyzed to be of kind Cartesian
34c. and **obs**erved to ``consist'' of component endurants E1, E2, ..., En
35. **Formalization:**
35a. **type**
35b. E1, E2, ..., En
35c. **value**
35d. **obs**_E1: E→E1, **obs**_E2: E→E2, ..., **obs**_En: E→En,

4.2.1.2.3 Informal Semantics: The distinct endurant names of `analyze_compound_names(e)` of the analyzed Cartesian compound part, `E1, E2, ..., En,` are "bound" [in some environment, ρ] to their meaning: named sorts, and their corresponding **obs**_ervers to their meaning: named functions that applied to endurants `e:E` yield respective components.

Note: The **obs**_erver functions are not computable functions. They are "functions" that are "performed" by the domain analyzer cum describer.

4.2.1.2.4 Pragmatics: Analysis and description of Cartesian parts is a core aspect of any domain study.

4.2.1.2.5 Example: See Appendix section B.2 on page 44. Please note that we have "merged" several compound type description units into one.

4.2.1.2 Part set Parts

We then deal with part set compounds.

4.2.1.2.0 Analysis: The endurant, `e:E`, being analyzed satisfies the predicate prompt `is_Part_set_Parts` and the functional analysis prompt `analyze_compound_names` yields a pair of endurant names `PS, P`.

4.2.1.2.1 Purpose: To "unravel" the endurant sorts and elements of part set compounds.

4.2.1.2.2 Syntax:

36. There is a keyword **Part set part sort and observer.**
37. There is a keyword **Narrative.**
    (a) It refers to which endurant is being analyzed;
    (b) of which kind, here `part set part`,
    (c) and that there are two endurant names: one, `PS`, for a set of parts, and one, `P`, for these parts.
38. And there is a keyword **Formalization.**
    (a) It starts with the literal **type**
    (b) and enumerates the two new part names and defines one in terms of the other.
    (c) It then "continues" with the literal **value**
    (d) and lists the **obs**_erver function for part sets.

36. E: **Part set part sort and observer:**
37. **Narrative:**
37a. The part being observed is E.
37b. It is analyzed to be of kind *part set*

37c. and **obs**served to "consist" of sets, PS, of endurants P.
38. **Formalization:**
38a. **type**
38b. P, PS = P-**set**
38c. **value**
38d. **obs_**PS: E→PS

4.2.1.2.3 Informal Semantics: The distinct endurant names of `analyze_compound_names(e)` of the analyzed part set compound part, `P, PS` are "bound" [in some environment, ρ] to their meaning: named sorts, and the corresponding **obs_**erver to its meaning: a named function that applied to part set parts, `e:E`, yield a part set.

Note: The **obs_**erver function is not a computable function. It is a "function" that is "performed" by the domain analyzer cum describer.

4.2.1.2.4 Pragmatics: Analysis and description of part set parts is a core aspect of any domain study.

4.2.1.2.5 Example: See Appendix section B.2 on page 44. Please note that we have "merged" several compound type description units into one.

4.2.1.4 Unique Identifiers

4.2.1.3.0 Analysis: The endurant being analyzed has been determined to be either atomic or analyzed into a compound (Cartesian or part set part) – and "is ready" for analysis and description wrt. internal qualities.

4.2.1.3.1 Purpose: The domain analyzer cum describer has decided that certain manifest or conceptual parts need be ascribed internal qualities.

4.2.1.3.2 Syntax:

39. There is a keyword **Unique Identification.** It is prefixed by the name of the part being internal quality analyzed.
40. There is a keyword **Narrative.**
    (a) It refers to the name of the endurant being internal quality analyzed.
    (b) It is analyzed and its parts are defined as a (pair, a) unique identifier type, `EI`,
    (c) and a unique identifier observer, **uid_**`E`, that from parts `e:E` yield identifiers `ei:EI.`
41. There is a keyword **Formalization.**
    (a) It starts with the literal**type**
    (b) and then lists the name of the unique identifier being introduced.
    (c) It then continues with the literal **value**
    (d) and lists the signature of the unique identifier observer.

39. E: **Unique Identification:**
40. **Narrative:**
40a. The part being observed is of sort E
40b. We ascribe it the unique identifier sort name EI
40c. and a corresponding unique identifier observer signature.
41. **Formalization:**
41a. **type**
41b. EI
41c. **value**
41d. **uid_**E: E → EI

4.2.1.3.3 Informal Semantics: The unique identifier sort name EI and "its" unique identifier **obs_**observer function **uid_**E are bound [in some environment, ρ] to their meaning: named sorts, and the corresponding unique identifier **obs_**erver, **uid_**E, to their meaning: named functions that applied to endurants `e:E` yield respective unique identifiers.

Note: The **uid_**observer function is not a computable function. It is a "function" that is "performed" by the domain analyzer cum describer.

4.2.1.3.4 Pragmatics: Analysis and description of internal qualities is a core aspect of any domain study.

4.2.1.3.5 Example: See Appendix section B.5 on page 45.

4.2.1.5 Mereology

4.2.1.4.0 Analysis: The endurant being analyzed has been determined to be either atomic or analyzed into a compound (Cartesian or part set part), has been analyzed wrt. unique identification – and "is ready" for further analysis and description wrt. mereology.

4.2.1.4.1 Purpose: The domain analyzer cum describer has decided that certain manifest or conceptual parts need be ascribed internal mereology qualities, i.e., how those parts relate to other parts – that is, how they, as behaviours, [may] communicate with other [such] behaviours.

4.2.1.4.2 Syntax:

42. There is the keyword **Mereology.** It is prefixed by the name of the part being internal quality analyzed.
43. There is a keyword **Narrative.**
    (a) It refers to the name of the endurant, `E`, being internal quality analyzed.
    (b) Endurant `e:E` are analyzed and are described into having a mereology, defined as a (pair, a) type expression over unique identifiers, $\mathcal{E}(EI1, EI2, ..., EIn)$,
    (c) and a mereology observer **mereo_**`E`, that from parts `e:E` yield their mereology $\mathcal{EI}$..
44. There is a keyword **Formalization.**
    (a) It starts with the literal **type**
    (b) and defines the mereology, `ME`, of parts `E`.
    (c) It then continues with the literal **value**
    (d) and lists the signature of the mereology observer.

42. E: **Mereology:**
43. **Narrative:**
43a. The part being observed is of sort E.
43b. We ascribe it the mereology type name, ME,
43b. and define it in terms of a type expression, $\mathcal{EI}$, over part identifiers.
44. **Formalization:**
44a. **type**
44b. ME = $\mathcal{T}$(UI1,UI2,...,UIn)
44c. **value**
44d. **mereo_**E: E → ME

$\mathcal{T}$(UI1,UI2,...,UIn) is a type expression over unique identifier sorts.

4.2.1.4.3 Informal Semantics: The mereology sort name ME and "its" observer function **mereo_**E are bound [in some environment, ρ] to their meaning: named sorts, and the corresponding **mereo_**logy **obs_**erver to their meaning: named functions that applied to endurants `e:E` yield respective mereologies.

Note: The **mereo__**observer function is not a computable function. It is a "function" that is "performed" by the domain analyzer cum describer.

4.2.1.4.4 Pragmatics: Analysis and description of part mereologies is a core aspect of any domain study.

4.2.1.4.5 Example: See Appendix section B.8 on page 47.

4.2.1.6 Attributes Attributes are what gives *"blood & flesh, i.e., life"* to endurants. We suggest modeling four facets of endurant attributes:

- **Attribute type names:** The domain analyzer cum describer is the person who decides which attributes to model. Each is given a separate [type] name. `analyze_attribute_names(e)` yields a set of distinct attribute names for endurant `e`.
- **Attribute types:** For some attributes the domain analyzer cum describer may choose to model these attributes in terms of some auxiliary type.
- **Attribute categories:** For each attribute the domain analyzer cum describer decides whether its category: **sta**tic, **mon**itorable or **pr**o**g**rammable.
- **SI Units of physical attributes:** For some attributes that are physical: *length, mass, time, electric current, thermodynamic temperature, amount of substance and luminous intensity*, or derived from these, the domain analyzer cum describer may [additionally] ascribe an SI Unit[21] to that attribute. See Appendix C on page 54.

4.2.1.5.0 Analysis: The endurant being analyzed has been determined to be either atomic or analyzed into a compound (Cartesian or part set part), has been analyzed wrt. unique identification and mereology – and "is ready" for analysis and description wrt. attributes.

And, following M.A.Jackson, [46] – and as covered in Sect. 5.4.2.3 (pages 121–128) in [22] – these endurants have been analyzed in either of the attribute categories *static* or *dynamic*, and these latter into either *inert*, *reactive* or *active*, and this latter into either *autonomous*, *biddable* or *programmable* – all as indicated in Fig. 3. We shall only cover the *static*, *monitorable* and *programmable* categories.

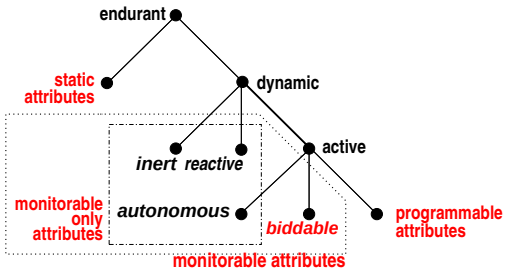

Figure 3: Attribute Value Ontology

[21] https://en.wikipedia.org/wiki/International_System_of_Units

4.2.1.5.1 Purpose: The domain analyzer cum describer has decided that certain manifest or conceptual parts need be ascribed internal qualities of attributes.

4.2.1.5.2 Syntax:

45. There is a keyword **Attributes.** It is prefixed by the name of the part being internal quality analyzed.
46. There is a keyword **Narrative.**
    (a) It refers to the name of the endurant, `E`, being internal quality analyzed.
    (b) Endurant `e:E` are analyzed and are described into having attributes of *type* `A1, A2, ..., Am` –
    (c) being [type] defined (`Attr1, Attr2, ..., Attrm`),
    (d) have attribute category `AttrCat` = **sta**tic, **mon**itorable or **pr**o**g**rammable and
    (e) having attribute observers.
47. There is a keyword **Formalization.**
    (a) It starts with the literal **type**
    (b) and then lists (enumerates) the type [names] of the attributes of `E`.
    (c) It then continues with the literal **value**
    (d) and then lists the signatures of the corresponding attribute observers.

45. E: **Attributes:**
46. **Narrative:**
46a. The part being observed is of sort E.
46b. We ascribe it the attribute type name, A1, A2, ..., Am,
46c. and corresponding attribute types (`Attri`),
46d. categories (`AttrCat`) and
46e. observer signatures (**attr_Ai** etc.).
47. **Formalization:**
47a. **type**
47b. A1 = Attr1 [AttrCat],
47b. A2 = Attr2 [AttrCat],
47b. ...,
47b. Am = Attrm [AttrCat]
47c. **value**
47d. **attr_**A1: A→A1, **attr_**A2: A→A2, ..., **attr_**A2: A→A2

4.2.1.5.3 Informal Semantics: The attribute sort names, `A1, A2, ..., Am`, their respective meaning (`Attr1, Attr2, ..., Attrm`), and "their" observer functions (**attr_**`A1`, **attr_**`A2`, ..., **attr_**`Am`) are bound [in some environment, ρ] to their meaning: named sorts, and the corresponding **attr_**ibute **obs_**erver to their meaning: named functions that applied to endurants `e:E` yield respective attribute (values).

Note: The **attr_**observer functions are not computable functions. They are "functions" that are "performed" by the domain analyzer cum describer.

4.2.1.5.4 Pragmatics: Analysis and description of endurant attributes is a core aspect of any domain study.

4.2.1.5.5 Example: See Appendix section B.9 on page 47.

4.2.1.7 Intentional Pull

Intentionality as a philosophical concept is defined by the Stanford Encyclopedia of Philosophy[22] as *"the power of minds to be about, to represent, or to stand for, things, properties and states of affairs."*

Two or more artefactual parts of different sorts, but with overlapping sets of intents may excert an *intentional "pull"* on one another. This *intentional "pull"* may take many forms. Let $p_x : X$ and $p_y : Y$ be two parts of *different sorts* (X, Y), and with *common intent*, ι, of the same universe of discourse. *Manifestations* of these, their common intent, must somehow be *subject to constraints*, and these must be *expressed predicatively*.

4.2.1.6.0 Analysis:

4.2.1.6.1 Purpose:

4.2.1.6.2 Syntax:

4.2.1.6.3 Informal Semantics:

4.2.1.6.4 Pragmatics:

4.2.1.6.5 Example: See Appendix section **??** on page ??.

4.2.2 Auxiliary Types

Section 4.1 on page 12 and paragraphs 4.2.1.1–4.2.1.6 (pages 13–18) introduced specific kinds of sorts. Each related strongly to a domain concept: Universe of discourse, Cartesians, part set parts, unique identifiers, mereology and attributes. [They are the "work horses", the base entities, of domains.]

In addition to these, and "in the tradition" of formal software specifications, domain descriptions make well use of additional, 'concrete' i.e., auxiliary composite types (sets, explicit Cartesians, lists, maps and functions) as base types (numbers, Booleans) and "token" types, that is, types whose [typically atomic] values are otherwise left undefined.

4.2.2.0 Analysis: In defining the type of certain domain concepts, typically part attributes, the domain engineer – "falling back" to common ways of formal specifications – finds it expedient to introduce 'concrete' types – such as sets, explicit Cartesians, lists, maps functions, numbers, Booleans and tokens.

There are two aspects to the introduction of auxiliary types: their **naming** and their **definition**.

Section 4.1 on page 12 and paragraphs 4.2.1.1–4.2.1.6 (pages 13–18) all introduced names of sorts and types. These names are distinct[23].

4.2.2.1 Purpose: The reason for introducing auxiliary types is to "make things easy" !

4.2.2.2 Syntax: In this section we shall make use of a different way of syntax definition.

Naming:

48. Let `T1, T2, ..., Tn, Ta, Tb` be [sort or type] names, `TN`. They are either the names of the auxiliary types to be defined or they are names of sorts or types introduced in Section 4.1 on page 12 and paragraphs 4.2.1.1–4.2.1.6 (pages 13–18).
49. An auxiliary type definition is then a map from the names of the auxiliary types to be defined to type expressions, `TE`.

Definition:

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

[23] This is a context condition.

50. A type expression, TE is either an atomic type or is a composite type or is "union" type of types.
51. Atomic types are wither natural numbers, integers, Booleans, or tokens – that is: not further defined atomic values.
52. Composite types are either [finite or possibly infinite] sets, are specific (n is fixed: 2, 3, 4 but usually not more[24]) Cartesians, [finite or possibly infinite] lists, maps, total or partial functions from definition set Ta to range set Tb elements.

type
48. TN
49. TD = TN $\overrightarrow{m}$ TE
50. TE = AT | CT | TN″|″TN″|″...″|″TN | **Unit**
51. AT = **Nat** | **Int** | **Real** | **Bool** | TOKEN
52. CT = TN-**set** | TN-**infset** | TN×TN×...×TN | TN* | TN$^\omega$ | TN $\overrightarrow{m}$ TN | TN→TN | TN$\xrightarrow{\sim}$TN

Here ″|″ are literals – as opposed to | which denotes type union.

- **Unit** the type consisting of the single value ()
- TN-**set** the type of finite sets
- TN-**infset** the type of finite and infinite sets
- TN×TN×...×TN the type of Cartesians
- TN* the type of finite lists
- TN$^\omega$ the type of finite and infinite lists
- TN $\overrightarrow{m}$ TN the type of finitely enumerable maps
- TN→TN the type of total functions
- TN$\xrightarrow{\sim}$TN the type of total and partial functions

Please note that sort and type names used in type expressions (items 52) must all be defined: either in some other auxiliary type definitions (item 49) or in Section 4.1 on page 12 or paragraphs 4.2.1.1–4.2.1.6 (pages 13–18).

Observe that we suggest, well mandate, that You use type names instead of type expressions[25].

53. To discriminate between alternative part set parts of type P we allow "discriminate union"
54. where right-hand sides have possibly different ms (larger than or equal to 1), with possibly "confused", i.e., equal **sel**_elector names.
55. Discriminated values can be composed from the implied **mk_** [make] constructors
56. and analyzed with the implied **sel**_ectors.

type
53. PA == P1 | P2 | ... | Pn
54. P1 :: **sel**_11:P11 × **sel**_12:P12 × ... × **sel**_1m:P1m
54. P2 :: **sel**_21:P21 × **sel**_22:P22 × ... × **sel**_2m:P2m
54. ...

[24] There is no theoretical reason for any lower bound n, but for pragmatic, i.e., readability/comprehensibility reasons it it advisable to keep n "small" !

[25] You may, of course, decide to use type expressions in defining composite types but then these definitions become long [unwieldy]. It means, then, that You must introduce additional type names. OK. Let these then be carefully constructed "mnemonics" !

54. Pn :: **sel**_n1:Pn1 × **sel**_n1:Pn2 × ... × **sel**_nm:Pnm

value for example:

55. pi = **mk**_P1(pi1,pi2,...,pim)
56. **sel**_i1(pi) ≡ pi1, **sel**_i2(pi)≡ pi2, ..., **sel**_im(pi) ≡ pim

57. We can define **sub-types**, `STE`, of type `TE` – namely such which satisfies the Boolean $\mathcal{B}$(v).

type

57. STE = {| v | c:TN • $\mathcal{B}$(v) |}

4.2.2.3 Informal Semantics: Names of auxiliary types are bound to their ["set-theoretical"] meaning in some, i.e., the environment, ρ.

4.2.2.4 Pragmatics: *** to come ***

4.2.2.5 Example: See Appendix section B.11 on page 49.

4.3 Value Definitions

One can, pragmatically, distinguish between two kinds of value definitions:

- **simple value descriptions** of parts, unique identifiers, mereologies, and attributes – individual ones or sets of these, all with value names being defined, given [sort or] type and, possibly given an expressed value; and
- **function value descriptions** of behaviours, actions and "simple" functions – all with signatures, i.e., names and types, and with function definitions.

4.3.1 Non-function Value Descriptions

4.3.1.0 Analysis: ...

4.3.1.1 Purpose: ...

4.3.1.2 Syntax:

58. There is a keyword **Simple Value Description[s].**
59. There is a keyword **Narrative.**
    (a) A [possibly enumerated] text "introduces" the value, its type and possible value.
60. There is a key word **Formalization:**
    (a) There is the literal **value**,
    (b) there is then the name and type [sort] of the value
        followed, optionally, by a value expression of the [value] type.

58. **Simple Value Description[s]:**
59. **Narrative:**
59a. Some text
60. **Formalization:**
60a. **value**
60b. v:T [= $\mathcal{E}$(...)]

4.3.1.3 Informal Semantics: The ubiquitos environment, ρ, reflects that a value name, v, is bound to some type and some value, val, of that type. If $\mathcal{E}$ is specified [and defined] then its value is the val.

4.3.1.4 Pragmatics: The description of simple values is a core stone of domain descriptions.

4.3.1.5 Example: Section B.3 on page 44 shows endurant [part] values; Sect. B.6 on page 46 shows unique identifier values.

4.3.2 Function Value Descriptions

4.3.2.0 Analysis: A core concept of domains is the concept of part behaviours, i.e., functions.

4.3.2.1 Purpose: Hence, to model, i.e., analyze and describe behaviours, we need the basic notion of functions, i.e., of *function signatures* and *function definitions*.

By a *function signature* we, syntactically, mean a triple of a function name, say f, a ':', and a function type, say $A \times B \times ... \times C \to R$ – where $A, B, ..., C, R$ are names of types described in other domain description units.

By a *function definition* we mean, syntactically, mean a triple of a function "invocation expression" $f(a, b, ..., c)$, an $\equiv$, and an expression $\mathcal{E}(a, b, ..., c, x, y, ..., z)$ – where $x, y, ..., z$ are names of values described in other domain description units.

4.3.2.2 Syntax:

61. There is a key word **Function Value Description[s]**.
62. There is a key word **Narrative**.
    (a) The narrative is an informal text which "explains" the function!
63. There is a key word **Formalization**.
    (a) There is the literal **value**.
    (b) There is a function signature: a function name and a function type [expression].
    (c) And there is the function definition.

61. **Function Value Description[s]**
62. **Narrative:**
62a. text
63. **Formalization:**
63a. **value**
63b. f: A → B
63c. f(a) ≡ $\mathcal{E}(a, ...)$

4.3.2.3 Informal Semantics: The function name f is bound in the ubiquitos environment ρ to its type and function value. The values [...] occurring in $\mathcal{E}(a, ...)$ are "already" bound in ρ.

4.3.2.4 Pragmatics: Auxiliary function values, hence definitions, are necessary in any domain description.

4.3.2.5 Example: Sect. B.12 (pages 49-51) shows several function values.

4.4 Axiom Definitions

4.4.0 Analysis:

4.4.1 Purpose:

4.4.2 Syntax:

64. There is a keyword **Axiom Definition.**
65. There is a keyword **Narrative.**
    (a) There then follows a text which informally expresses constraints on description unit descriptions.
66. There is a keyword **Formalization.**
    (a) There is the literal **axiom** followed by a suitably chosen axiom name.
    (b) And there is a predicate expression, $\mathcal{A}(x, y, ..., z)$ where $x, y, ..., z$ are names of types, values, channels or variables introduced in description units and thus "appearing" [i.e., defined] in the ubiquitos environment ρ.

We present only the formal text:

66a. **axiom** [Axiom Name:]
66b. $\mathcal{A}(x, y, ..., z)$

4.4.3 Informal Semantics: The axiom expresses that names defined in the ubiquitos environment ρ are bound to [range over] values satisfying the axiom.

4.4.4 Pragmatics: Since syntax definitions are usually context-free there is occasionally a need for expressing constraints.

4.4.5 Examples: See

- Appendix Sect. B.7 on page 46,
- Appendix Sect. B.9 on page 47 Item 161 on page 48, and
- Appendix Sect. B.11 on page 49 Items 166 on page 49 — 166e on page 49

4.5 Channel Declarations

4.5.0 Analysis: The core concept of domain behaviours, entail, for domains with two or more [manifest] parts, two or more such behaviours. For any such domain to be "interesting" these behaviours interact: synchronize and communicate. We shall use `CSP` [42, 45] to express such interactions.

These are envisaged, i.e., abstracted, as taking place over a medium that we shall refer to as **channel**s.

Any two different behaviours, u_i, u_j, may need to interact. We choose to express this a their interacting over one channel `ch[`$\{u_i, u_j\}$`]` of a channel array.

4.5.1 Purpose: Channels are introduced in order to express the interaction (synchronization and communication) between behaviours.

4.5.2 Syntax:

67. There is a keyword **Channel Declaration.**
68. There is a keyword **Narrative.**

(a) There then follows a text which informally expresses the interaction concept via channels.

69. There is a keyword **Formalization.**
    (a) There is the literal **channel**
    (b) followed by a channel clause of the form:
        - {ch[{ui,uj}]|ui,uj:UI•{ui,uj}⊆$\sigma_{parts_{uid}}$} M.

        The clause expresses that ch is the name of the channel array, that it is an array (ch[{ui,uj}) whose elements range over the [full] set of part identifiers – and whose "messages", the information being communicated, is of type M. Here M is a type defined in an appropriate [auxiliary] type description unit.

67. **Channel Declaration.**
68. **Narrative.**
68a. text
69. **Formalization.**
69a. **channel**
69b. { ch[{ui,uj}] | ui,uj:UI • {ui,uj}⊆$\sigma_{parts_{uid}}$ } M

4.5.3 Informal Semantics: The channel name, ch, is bound in the ubiquitous environment ρ to its meaning: that it stands for a channel, etc.!

4.5.4 Pragmatics: There is need for exactly one channel description unit. No need for behaviour interactions, hence for channels, "signals" an *un-interesting* domain!

4.5.5 Examples: See Appendix Sect. B.13 on page 51.

4.6 Variable Declarations

4.6.0 Analysis: In rare cases the domain analyzer cum describer may be compelled to model a domain phenomenon by means an updateable variable. An example where the compound sets of parts are changing: road nets are being extended-with/reduced-of hubs and links, and/or automobiles "added'. Now the ensemble of such parts, usually referred to by a [global] value [definition] hs, ls, as, need, for example, be "contained" in correspondingly declared variables (say of the same names).

4.5.1 Purpose: Variables are meant to "keep track" of a *dynamically evolving* (increasing/decreasing) number of manifest parts.

4.5.2 Syntax:

70. There is the keyword **Variable Declaration.**
71. There is the keyword **Narrative.**
    (a) And then there is some informal text which explains the role of the variable being declared and initialized.
72. There is the keyword **Formalization.**
    (a) There is the literal **variable**
    (b) followed by the name of the variable, v, its type, E, and an initializing expression.

70. **Variable Declaration.**
71. **Narrative.**
71a. text
72. **Formalization.**
72a. **variable**
72b. v:E := $\mathcal{E}$(...)

4.6.3 Informal Semantics: The ubiquitous environment ρ associates v with a *location* (ℓ) in a ubiquitous state, ξ, and, in ξ that location (ℓ) with a value of type E.

4.6.4 Pragmatics: Usually we may not model *dynamically evolving* domains – in which case there is no need for the introduction of variables.

4.6.5 Examples: See Appendix Sect. B.14 on page 51.

4.7 Behaviour Definitions

The laborious work on analyzing and describing external and internal qualifications (unique identification, mereology and all forms of attributes) now comes to fruition: the *realization and fulfillment* of our endeavor to understand domains.

4.7.0 Analysis: Some parts have been selected to [also] represent behaviours.

4.7.1 Purpose: To complete the essence of a domain model.

4.7.2 Syntax:

73. There is a keyword **Behaviour[s].**
74. There is a keyword **Narrative.**
    (a) Where we first describe/motivate the *signature* of the behaviour[s].
    (b) then describe the[ir] *function definition[s].*
75. There is a keyword **Formalization.**
    (a) There is a literal **value**.
    (b) There is then the signature. Here we shall recall that the following arguments are "called by value": *unique identifier*, *mereology*, *static attributes* and *programmable attributes*. The *monitorable attributes* are "called by name"[26].
    (c) Finally there is the formal function definition.

```
73. Behaviours:
74.     Narrative.
74a.        The arguments of the behaviour[ s ] are ... .
74b.        The behaviour evolves as follows ... .
75.     Formalization.
75a.        value
75b.            behaviour: UI → Mereo → Stat → Mon → Prgr → Unit
75c.            behaviour(ui)(mer)(sta)(mon)(prg) ≡ 𝒟(...)[27]
```

We have `Schönfinckel'ed/Curried`[28] the function type into five argument groups: the unique identifier, the **mer**eology, the **sta**tic attributes, the **mon**itorable attributes, and the **pr**o**g**rammable attributes. **Unit** stands for `()`, a "state-to-state function".

A superficial analysis of forms of $\mathcal{D}(...)$, the definition bodies, is presented in Appendix A on page 41.

4.7.3 Informal Semantics: The function name, `behaviour`, is bound in the ubiquitous environment ρ to its [mathematical] meaning. For the argument values, we refer to Sect. 4.9 on the following page.

4.7.4 Pragmatics: Behaviour definitions is, in a sense, what domain descriptions are all about.

4.7.5 Examples: See Appendix Sect. B.15 on page 51.

[26] Call-by-name functions recompute the passed-in expression's value every time it is accessed.
[27] The ... in $\mathcal{D}(...)$ contain references to ui, mer, sta, mon,and prg.
[28] `https://en.wikipedia.org/wiki/Moses_Schnfinkel#Further_reading` and `https://en.wikipedia.org/wiki/Haskell_Curry`

4.8 Endurants with Monitorable Attributes

- Monitorable attributes [may] change their value, as "experienced" by the behaviours of which they are attributes — more-or-less surreptitiously, "behind their back" so-to-speak.
- So, as arguments, they are "passed-by-name".
- The name is that of the attribute.
- If we [otherwise] refer to that attribute, as A, then we refer to the name as ηA.
- During evaluation of the behaviour definition [body], $\mathcal{D}(...)$ (item 75c on the preceding page), of behaviour ui, when encountering a monitorable argument, with name ηA, the following elaboration is suggested:
  - From the globally kept value of all parts, σ_{parts}[29], the part, p_{ui}, having identifier ui, is selected and its attribute, **attr_A**(p_{ui}) is selected.
  - Since the attribute in question is monitorable all we wish to know is its current value.
  - That current value changes, as we wrote above, without our knowledge.

We refer to Sect. 6.11, pages 111–117 of [32].

4.9 Behaviour Initialization

4.9.0 Analysis: We have reached the end of our analysis. There is defined, see Item **??** on page ??, as number of part values. Some, e.g., sp_i, are defined as one such values. Others, e.g., cp_i, as sets of parts. The sp_i and all elements, p_i, of respective cp_i, are to be transcendentally "morphed" into respective behaviours.

4.9.1 Purpose: To finalize the domain model.

4.9.2 Syntax:

76. There is a keyword **Initialization.**
77. There is a keyword **Narrative.**
    (a) The domain describer "assigns" some suitable name for the domain initialization behaviour as part of its signature.
    (b) Then is described which part behaviours are initialized. Some initializations are of just one part, others of a set of parts.
78. There is a keyword **Formalization.**
    (a) There is a literal **value**.
    (b) There is a domain analyzer cum describer chosen initialization behaviour.
    (c) There is then the parallel, ‖, composition of initialization of all "single" parts sp_i, and, in parallel,
    (d) the distributed parallel, ‖, composition initialization of each and all elements of each set of parts.

76. **Initalization**
77. **Narrative**
77a. There are the globally defined endurant part values sp_1, sp_2, ..., sp_m,
77a. and there are the likewise defined part set values $cp_1, cp_2, ..., cp_m$.
77b. For each of these we initialize their part behaviours.
78. **Formalization**
78a. **value**

[29] Cf. Example B.3 on page 44 item 127 on page 44.

78b. initialization: **Unit** → **Unit**
78b. initialization() ≡
78c. sp1(**uid**_P1(sp_1))(**mereo**_P1(sp_1))(**attr**_sA11(sp_1),...)(ηmA11(sp_1),...)(**attr**_pA11(sp_1),...)
78c. ‖ sp2(**uid**_P2(sp_2))(**mereo**_P2(sp_2))(**attr**_sA21(sp_2),...)(ηmA21(sp_2),...)(**attr**_pA21(sp_2),...)
78c. ‖ ...
78c. ‖ spm(**uid**_Pm(sp_m))(**mereo**_Pm(sp_m))(**attr**_sAm1(sp_m),...)(ηmAm1(sp_m),...)(**attr**_mAm1(sp_m),...)
78d. ‖ { p1(**uid**_CP1(p_1))(**mereo**_CP1(p_1))(**attr**_sA11(p_1),...)(ηmA11(p_1),...)(**attr**_pA11(p_1),...)
78d. | p_1 ∈ cp_1 }
78d. ‖ { p2(**uid**_CP2(p_2))(**mereo**_CP2(p_2))(**attr**_sA12(p_2),...)(ηmA12(p_2),...)(**attr**_pA12(p_2),...)
78d. | p_2 ∈ cp_2 }
78d. ‖ ...
78d. ‖ { pn(**uid**_CPn(p_n))(**mereo**_CPn(p_n))(**attr**_sA1n(p_n),...)(ηmA1n(p_n),...)(**attr**_pA1n(p_n),...)
78d. | p_n ∈ cp_n }

where Pi is the type of respective sp_i and CPi is the type of respective p_i. Please observe the careful naming of *s*tatic, *m*onitorable, and *p*rogrammable attributes types. Please also not the names of respective behaviours.

4.9.3 Informal Semantics: ...

4.9.4 Pragmatics: ...

4.9.5 Example: See Example B.16 on page 54.

5 Auxiliary Clauses

5.1 SPACE and TIME

The two concepts: **space** and **time** are not attributes of entities. In fact, they are not internal qualities of endurants. They are universal qualities of any world. 𝕊ℙ𝔸ℂ𝔼 and 𝕋𝕀𝕄𝔼 are unavoidable concepts of any world. But we can ascribe spatial attributes to any concrete, manifest endurant. And we can ascribe attributes to endurants that record temporal concepts.

5.1.1 SPACE

Space is just there. So we do not define an observer, observe_space. For us – bound to model mostly artefactual worlds on this earth – there is but one space. Although 𝕊ℙ𝔸ℂ𝔼, as a type, could be thought of as defining more than one space we shall consider these to be isomorphic! 𝕊ℙ𝔸ℂ𝔼 is considered to consist of (an infinite number of) ℙ𝕆𝕀ℕ𝕋s.

79. We can assume a point observer, observe_ℙ𝕆𝕀ℕ𝕋, is a function which applies to endurants, e, and yield a point, pt : ℙ𝕆𝕀ℕ𝕋

79. observe_ℙ𝕆𝕀ℕ𝕋: E → ℙ𝕆𝕀ℕ𝕋

At which "point" of an endurant, e, observe_ℙ𝕆𝕀ℕ𝕋(e), is applied, or which of the (infinitely) many points of an endurant $e : E$, observe_ℙ𝕆𝕀ℕ𝕋(e), yields we leave up to the domain modeler to decide!

We suggest, besides ℙ𝕆𝕀ℕ𝕋s, the following spatial attribute possibilities:

80. 𝔼𝕏𝕋𝔼ℕ𝕋 as a dense set of ℙ𝕆𝕀ℕ𝕋s;
81. Volume, of concrete type, for example, m^3, as the volume of an 𝔼𝕏𝕋𝔼ℕ𝕋 such that
82. ℙ𝕃𝔸ℕ𝔼s as dense sets of ℙ𝕆𝕀ℕ𝕋s have no volume, but an

83. Area, of concrete type, for example, m^2, as the "area" of a dense set of ℙOINTs;
84. 𝕃INE as dense set of ℙOINTs with no volume and no area, but
85. Length, of concrete type, for example, m.

For these we have that

86. the *intersection*, $\bigcap$, of two 𝔼XTENTs is an 𝔼XTENT of possibly nil Volume;
87. the intersection, $\bigcap$, of two 𝕃INEs may be either a possibly nil 𝕃INE or a ℙOINT.

Similarly we can define

88. the *union*, $\bigcup$, of two not-disjoint 𝔼XTENTs, two not-disjoint 𝕊URFACEs, and of two not-disjoint 𝕃INEs.

and:

89. the *[in]equality*, $\neq, =$, of pairs of 𝔼XTENT, pairs of 𝕊URFACEs, and pairs of 𝕃INEs.

We invite the reader to first express the signatures for these operations, then their pre-conditions, and finally, being courageous, appropriate fragments of axiom systems. We leave it up to the reader to introduce, and hence define, functions that add, subtract, compare, etc., 𝔼XTENTs, ℙLANEs, 𝕃INEs and 𝕊PACE.

5.1.2 TIME

A moving image of eternity.
The number of the movement in respect of the before and the after.
The life of the soul in movement as it passes from one stage of act or experience to another.
A present of things past: memory,
a present of things present: sight,
and a present of things future: expectations[30]
This thing all things devours:
Birds, beasts, trees, flowers;
Gnaws iron, bites steel,
Grinds hard stones to meal;
Slays king, ruins town,
And beats high mountain down.[31]

Concepts of time continue to fascinate philosophers and scientists

[74, 38, 54, 55, 61, 56, 57, 58, 59, 60, 62] and [39].

We shall not be concerned with any representation of time. That is, we leave it to the domain modeler to choose an own representation [39]. Similarly we shall not be concerned with any representation of time intervals.[32]

90. So there is an abstract type 𝕋ime,
91. and an abstract type 𝕋𝕀: 𝕋ime 𝕀nterval.
92. There is no 𝕋ime origin, but there is a "zero" 𝕋ime 𝕀nterval, **0**.
93. One can add (subtract) a time interval to (from) a time and obtain a time.

[30] Quoted from [1, Cambridge Dictionary of Philosophy]
[31] J.R.R. Tolkien, The Hobbit
[32] – but point out, that although a definite time interval may be referred to by number of years, number of days (less than 365), number of hours (less than 24), number of minutes (less than 60) number of seconds (less than 60), et cetera, this is not a time, but a time interval.

94. One can add and subtract two time intervals and obtain a time interval – with subtraction respecting that the subtrahend is smaller than or equal to the minuend.
95. One can subtract a time from another time obtaining a time interval respecting that the subtrahend is smaller than or equal to the minuend.
96. One can multiply a time interval with a real and obtain a time interval.
97. One can compare two times and two time intervals.

```
type
90  𝕋
91  𝕋𝕀
value
92  0:𝕋𝕀
93  +,−: 𝕋 × 𝕋𝕀 → 𝕋
94  +,−: 𝕋𝕀 × 𝕋𝕀 ~→ 𝕋𝕀
95  −: 𝕋 × 𝕋 → 𝕋𝕀
96  ∗: 𝕋𝕀 × Real → 𝕋𝕀
97  <,≤,=,≠,≥,>: 𝕋 × 𝕋 → Bool
97  <,≤,=,≠,≥,>: 𝕋𝕀 × 𝕋𝕀 → Bool
axiom
93 ∀ t:𝕋 • t+0 = t
```

98. We define the signature of the meta-physical time observer.

```
type
98  𝕋
value
98  record_𝕋𝕀𝕄𝔼: Unit → 𝕋
```

The time recorder applies to nothing and yields a time. **record**_𝕋𝕀𝕄𝔼 can only occur in behavioural descriptions.

5.2 Expressions

In various sections of this document we refer to [possibly sub-scripted] expressions: $\mathcal{E}(...)$ for any type of value, and $\mathcal{B}(...)$ or$\mathcal{A}(...)$ for Boolean values[33]. These clauses are usually simple RSL expressions.

The purpose of this section is to review these clauses.

5.2.1 The "let" ... "in" ... "end" Expressions

5.2.1.1 Patterns: A pattern is either

- a simple pattern: an identifier: id; or
- a Cartesian pattern: $(id_1, id_2, ..., id_m)$; or
- a set pattern: $\{id_e\} \cup id_s$ where id_s stands for a possibly empty set; or
- a list pattern $\langle id_f \rangle \hat{} id_l$ where id_l stands for a possibly empty list; or
- a map pattern: $[\, id_d \mapsto id_r \,] \cup id_m$, where id_m stands for a possibly empty map.

where id_i are identifiers.

[33] For examples of $\mathcal{E}(...)$, $\mathcal{B}(...)$, $\mathcal{D}(...)$ and $\mathcal{A}(...)$ we refer to Items 57 on page 21, 66b on page 23, 60b on page 21, 63c on page 22, 72b on page 24 and 75c on page 25.

5.2.1.2 The Binding Expression:

The binding expression has the form:

let pattern = $\mathcal{E}_d(...)$ **in** $\mathcal{E}_b(...)$ **end**

The purpose of binding expressions is, pragmatically, to simplify descriptions: MORE TO COME

The idea of patterns is as follows:

- If a simple pattern, id, then the value of $\mathcal{E}_d(...)$ is bound [in a "local" environment] to id; id is an argument in $\mathcal{E}_b(...)$.
- If a Cartesian pattern, $(id_1, id_2, ..., id_m)$, then the value of $\mathcal{E}_d(...)$ is expected to be a Cartesian of m elements: $(v_1, v_2, ..., v_m)$ – with id_i now bound to v_i [in the "local" environment]; $id_1, id_2, ..., id_m$ are arguments in $\mathcal{E}_b(...)$.
- If a set pattern, $\{id_e\} \cup id_s$, then the value of $\mathcal{E}_b(...)$ is expected to be a set of one or more elements. id_e is bound to one of these, and id_s to the remaining, possibly empty set.
- If a list pattern, $\langle id_f \rangle^\frown id_l$, then the value of $\mathcal{E}_b(...)$ is expected to be a list of one or more elements. id_e is bound to the first, the **h**ea**d**, and id_s to the remaining, possibly empty list.
- If a map pattern, $[\, id_d \mapsto id_r \,] \cup id_m$, then the value of $\mathcal{E}_b(...)$ is expected to be a map of one or more 'mappings'. id_d and $id_r id_r$ are bound to an arbitrary 'mapping' in $\mathcal{E}_b(...)$ and id_m to the 'remaining', possibly empty map.

5.2.2 Arithmetic Expressions

Let e_i be number-valued expressions. Then

$$0, 2, ..., 1.1, 1.2, ..., -1, -2, ..., -1.1, -1.2, ..., e_1 + e_2, e_1 - e_2, e_1 * e_2, e_1/e_2,$$

are likewise number-valued expressions – where $e_1/0$ yields **chaos**.

And

$$e_1 < e_2, e_1 \leq e_2, e_1 = e_2, e_1 \neq e_2, e_1 > e_2, e_1 \geq e_2$$

are **Boolean**-valued expressions.

5.2.3 Boolean Expressions

Besides the Boolean expressions treated in Sects. 5.2.2 and 5.2.6–5.2.10, there are the following Boolean expressions:

true, false, $a \wedge b, a \vee b, a \equiv b, a = b, a \neq b, a \supset b$

- $\wedge$ denotes conjunction ("and"),
- $\vee$ denotes dis-junction ("or"),
- $\equiv$ denotes identity ("if and only if"),
- $=$ denotes equal to,
- $\neq$ denotes not equal to,
- $\supset$ denotes implication.

5.2.4 "chaos"

The literal **chaos** denotes that. It is used to express that a an evaluation is meaningless.

5.2.5 Set, Cartesian, List and Map Enumerations

Let `e_i, r_i, d_i` be expressions [i.e., of any kind], then:

- **Set enumerations:** {}, {e_1, e_2, ..., e_n}
  stand for the empty set, respectively sets of up to n elements.
- **Cartesian enumerations:** (e_1, e_2, ..., e_n)
  stand for a Cartesian of n elements.
- **List enumerations:** ⟨⟩, ⟨e_1, e_2, ..., e_n⟩
  stand for the empty list, respectively lists of n elements.
- **Map enumerations:** [ ], [d_1↦r_1,d_2↦r_2,...,d_n↦r_n]
  stand for the empty map, respectively maps of up to n *d*efinitions set to *r*ange set elements.

5.2.6 Set Expressions

$$
\begin{array}{rcll}
e \in s & : & \textbf{true}|\textbf{false} & (1)\\
\textbf{card}\, s & : & \textbf{Nat} & (2)\\
s \cup s' & = & \{e \mid e \in s \vee e \in s'\} & (3)\\
s \cap s' & = & \{e \mid e \in s \wedge e \in s'\} & (4)\\
s \setminus s' & = & \{e \mid e \in s \wedge e \notin s'\} & (5)\\
s / s' & = & \{e \mid e \in s' \wedge e \notin s\} & (6)\\
s \subset s' & = & \forall e \bullet e \in s \Rightarrow e \in s' \wedge \exists e \bullet e \in s' \wedge e \notin s & (7)\\
s \subseteq s' & = & \forall e \bullet e \in s \Rightarrow e \in s' & (8)\\
s = s' & = & s \subseteq s' \wedge s' \subseteq s & (9)\\
s \neq s' & = & \neg(s = s') & (10)
\end{array}
$$

5.2.7 Cartesian Expressions

$$(a_1, a_2, \ldots, a_m) = (b_1, b_2, \ldots, b_n)$$

5.2.8 List Expressions

$$
\begin{array}{rcll}
\textbf{hd}\langle a1, a2, ..., am\rangle & = & a1 & (11)\\
\textbf{tl}\langle a1, a2, ..., am\rangle & = & \langle a2, ..., am\rangle & (12)\\
\textbf{len}\langle a1, a2, ..., am\rangle & = & m & (13)\\
\textbf{inds}\langle a1, a2, ..., am\rangle & = & \{1, 2, ..., m\} & (14)\\
\textbf{elems}\langle a1, a2, ..., am\rangle & = & \{a1, a2, ..., am\} & (15)\\
\langle a1, a2, ..., am\rangle(i)^{34} & = & ai & (16)\\
\langle a, b, c\rangle^\frown\langle a, b, d\rangle & = & \langle a, b, c, a, b, d\rangle & (17)\\
\langle a, b, c\rangle & = & \langle a, b, c\rangle & (18)\\
\langle a, b, c\rangle & \neq & \langle a, b, d\rangle & (19)
\end{array}
$$

[34] Alternatively we may write: ⟨a1,a2,...,am⟩[i]

5.2.9 Map Operator Types and Expressions

• (•): M → A $\xrightarrow{\sim}$ B
m(ai) = bi

dom: M → A**-infset** [domain of map]
dom [a1↦b1,a2↦b2,...,an↦bn] = {a1,a2,...,an}

rng: M → B**-infset** [range of map]
rng [a1↦b1,a2↦b2,...,an↦bn] = {b1,b2,...,bn}

†: M × M → M [override extension]
[a↦b,a′↦b′,a″↦b″] † [a′↦b″,a″↦b′] = [a↦b,a′↦b″,a″↦b′]

∪: M × M → M [merge ∪]
[a↦b,a′↦b′,a″↦b″] ∪ [a‴↦b‴] = [a↦b,a′↦b′,a″↦b″,a‴↦b‴]

\: M × A**-infset** → M [restriction by]
[a↦b,a′↦b′,a″↦b″]\{a} = [a′↦b′,a″↦b″]

/: M × A**-infset** → M [restriction to]
[a↦b,a′↦b′,a″↦b″]/{a′,a″} = [a′↦b′,a″↦b″]

=,≠: M × M → **Bool**

°: (A $\xrightarrow[m]{}$ B) × (B $\xrightarrow[m]{}$ C) → (A $\xrightarrow[m]{}$ C) [composition]
[a↦b,a′↦b′] ° [b↦c,b′↦c′,b″↦c″] = [a↦c,a′↦c′]

5.2.10 Set, List and Map Comprehensions

The expression of set, list and map values can be comprehended:

99. { e(id) | id:T • $\mathcal{E}(e(id))$ }
100. ⟨ e(i) | i:**Nat** • lower_bound≤i≤upper_bound ∧ $\mathcal{B}(e(i))$ ⟩
101. [d ↦ r | d:TD,r:TR • $\mathcal{B}(d,r)$]

99. The set of all those e(id) of type T for which the predicate $\mathcal{E}(e(id))$ holds.
100. The natural number ordered list of all those e(i) where i is of type **Nat** and have a value between lower_bound and upper_bound and where $\mathcal{B}(e(i))$ holds.
101. The map definition set values, d (of type TD) to range set values r (of type TR) and where $\mathcal{B}(d,r)$ holds.

5.2.11 Conditional Expressions

We treat three kinds of conditional expressions.

102. The **if** ... **then** ... **else** ... **end** Expression:

if boolean **then** expression_1 **else** expression_2 **end**

We take it for granted that he reader is familiar with this expression.

103. The McCarthy Expressions

```
boolean_expr_1 → expression_1,
boolean_expr_2 → expression_2,
...
boolean_expr_m → expression_m
```

The above is a short form of:

```
if boolean_expr_1
    then expression_1
    else (if boolean_expr_2
              then expression_2
              else (...
                  ...
                      (if boolean_expr_n
                          then expression_m
                          else chaos end ) ... ) end ) end
```

[35]

For the case that `boolean_expr_n` should evaluate to **false** the developer might wish to "add" a catch-all clause:

```
boolean_expr_1 → expression_1,
boolean_expr_2 → expression_2,
...
boolean_expr_m → expression_m,
_ → expression_catch_all
```

104. The **cases** ... **end** Expressions

```
case expression of
    pattern_1 → expression_1,
    pattern_2 → expression_2,

    pattern_m → expression_m
end
```

The above is a short form of

```
let pattern = expression in
if matches(pattern,pattern_1)
    then expression_1 [where pattern identifiers are substituted
                       with those of  pattern_1 identifiers]
    else
        if matches(pattern,pattern_2)
            then expression_2 [where pattern identifiers are substituted
                               with those of pattern_2 identifiers]
            else
                ...
```

[35] The above "explication" must be reformatted / improved!

```
                    if matches(pattern,pattern_m)
                        then expression_m [where pattern identifiers are substituted
                                           with those of pattern_m identifiers]
                        else chaos
                    end
                ...
            end
        end
    end
```

Here we leave it to the reader to define the *matches* and *substitute* functions.

5.3 Statements

There is only one kind of statement in DDL.

It is the simple assignment statement:

var := $\mathcal{E}$(...) ;

It presupposes that the variable `var` has been introduced in some domain description unit.

The assignment statement can only occur in the 'body' of a behaviour definition.

MORE TO COME

5.4 CSP Clauses

Behaviour definitions, in DDL, makes use of `CSP` [42, 43]. We have already introduced concepts of

- behaviour,
- *channel* `ch`,
- parallel, ||, composition of behaviours,
- deterministic choice, □, of/between behaviours, and
- non-deterministic choice, ⌈⌉, of/between behaviours.

Let the behaviour in which the next constructs, to be introduced, occur have identity `ui` and let `uj` be the identity of some other existing behaviour. These are the `CSP` constructs of `RSL`:

105. Behaviour `ui` offers to communicate a `value` to (!) behaviour `uj`.
106. Behaviour `ui` offers to accept (?) a value from behaviour `uj`.

```
105.  ch[ {ui,uj} ] ! value
106.  ch[ {ui,uj} ] ?
```

- `ch[ {ui,uj} ] ! value` is a "statement-like" clause.
  - It only occurs in behaviour definitions.

```
b(ui)(mereo)(sta)(mon)(prg) ≡
    ...
    ch[ {ui,uj} ] ! value ;
    ...
```

- `ch[{ui,uj}] ?` is an expression.
  - It only occurs in behaviour definitions.
  - Usually it will occur in a **let ... in ... end** clause:

    let v = ch[{ui,uj}] ? **in** ... **end**

6 Closing

6.1 What Have We Achieved ?

TO BE WRITTEN

6.2 Engineering, Research, Science

We have used the terms 'engineer', 'engineering', 'research', 'researcher', 'science' and 'scientist', intending a specific meaning of these. Here, then, is the Internet understanding of these terms:

- **Engineering:** Engineering is the practice of using natural science, mathematics, and the engineering design process to solve problems within technology.
- **Engineer:** A person who designs, builds, or maintains machines, structures, or systems.
- **Research:** Research is the systematic, creative, and documented investigation designed to discover new information, interpret existing data, or revise theories to increase human knowledge. It involves defining problems, formulating hypotheses, and collecting or analyzing data to reach conclusions. Key types include basic, applied, scientific, and social research.
- **Researcher:** A person who carries out academic or scientific research.
- **Science:** Science is a systematic discipline that builds and organizes knowledge in the form of testable hypotheses and predictions about the universe.
- **Scientist:** A scientist is a professional or researcher who systematically gathers and analyzes data to understand the natural world, often using the scientific method to test hypotheses.

We shall, for our use — i.e., not rejecting the above Internet characterizations, but re-interpreting them —

- **Domain Engineering:**
- **Domain Engineer:** A person with, in our case, a background in computer & computing science, who analyze & describe domains – not necessarily according to the approach of this document.
- **Domain Research:** Domain ...
- **Domain Researcher:** A person with, in our case, a background in computer & computing science and in, for example, the *Domain Science & Engineering* approach of this document, who studies domains with the aim of ...Domain
- **Domain Science:** Domain ...
- **Domain Scientist:** A person with, in our case, a background in computer & computing science, who ...

MORE TO COME

6.3 Further Domain Research – hopefully Domain Science

TO BE WRITTEN

6.4 Acknowledgments

TO BE WRITTEN

7 Bibliography

References

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

[2] H. Bekič, D. Bjørner, W. Henhapl, C. B. Jones, and P. Lucas. A Formal Definition of a PL/I Subset. Technical Report 25.139, IBM Laboratory, Vienna, December 1974.

[3] Dines Bjørner. *Software Engineering*, volume 1: Abstraction and Modelling, vol. 2: Specification of Systems and Languages, vol. 3: Domains, Requirements and Software Design of *Texts in Theoretical Computer Science, the EATCS Series*. Springer, 2006. Chapters 12–14 of vol.2 primarily authored by Christian Krog Madsen.

[4] Dines Bjørner. From Domains to Requirements www.imm.dtu.dk/ dibj/2008/ugo/ugo65.pdf. In *Montanari Festschrift*, volume 5065 of *Lecture Notes in Computer Science (eds. Pierpaolo Degano, Rocco De Nicola and José Meseguer)*, pages 1–30, Heidelberg, May 2008. Springer.

[5] Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering*. Research Monograph (# 4); JAIST Press, 1-1, Asahidai, Nomi, Ishikawa 923-1292 Japan, 2009. This Research Monograph contains the following main chapters:

1. *On Domains and On Domain Engineering – Prerequisites for Trustworthy Software – A Necessity for Believable Management*, pages 3–38.
2. *Possible Collaborative Domain Projects – A Management Brief*, pages 39–56.
3. *The Rôle of Domain Engineering in Software Development*, pages 57–72.
4. *Verified Software for Ubiquitous Computing – A VSTTE Ubiquitous Computing Project Proposal*, pages 73–106.
5. *The Triptych Process Model – Process Assessment and Improvement*, pages 107–138.
6. *Domains and Problem Frames – The Triptych Dogma and M.A.Jackson's PF Paradigm*, pages 139–175.
7. *Documents – A Rough Sketch Domain Analysis*, pages 179–200.
8. *Public Government – A Rough Sketch Domain Analysis*, pages 201–222.
9. *Towards a Model of IT Security — – The ISO Information Security Code of Practice – An Incomplete Rough Sketch Analysis*, pages 223–282.
10. *Towards a Family of Script Languages – – Licenses and Contracts – An Incomplete Sketch*, pages 283–328.

.

[6] Dines Bjørner. On Mereologies in Computing Science. In *Festschrift: Reflections on the Work of C.A.R. Hoare*, History of Computing (eds. Cliff B. Jones, A.W. Roscoe and Kenneth R. Wood), pages 47–70, London, UK, 2009. Springer. www.imm.dtu.dk/ dibj/bjorner-hoare75-p.pdf.

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

[8] Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics, Part I of II: The Engineering Part*. *Kibernetika i sistemny analiz*, 2(4):100–116, May 2010.

[9] Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics Part II of II: The Science Part*. *Kibernetika i sistemny analiz*, 2(3):100–120, June 2011.

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

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

[12] Dines Bjørner. Manifest Domains: Analysis & Description. *Formal Aspects of Computing*, 29(2):175–225, March 2017. First Online: 26 July 2016. DOI 10.1007/s00165-016-0385-z.

[13] Dines Bjørner. Manifest Domains: Analysis & Description. *Formal Aspects of Computing*, 29(2):175–225, March 2017. Online: 26 July 2016.

[14] Dines Bjørner. What are Documents ? www.imm.dtu.dk/ dibj/2017/docs/docs.pdf. Research Note, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, July 2017.

[15] Dines Bjørner. Container Terminals. www.imm.dtu.dk/ dibj/2018/yangshan/maersk-pa.pdf. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, September 2018. An incomplete draft report; currently 60+ pages.

[16] Dines Bjørner. Domain analysis & description - the implicit and explicit semantics problem www.imm.dtu.dk/ dibj/2017/bjorner-impex.pdf. In Régine Laleau, Dominique Méry, Shin Nakajima, and Elena Troubitsyna, editors, Proceedings Joint Workshop on *Handling IMPlicit and EXplicit knowledge in formal system development (IMPEX)* and *Formal and Model-Driven Techniques for Developing Trustworthy Systems (FM&MDD),* Xi'An, China, 16th November 2017, volume 271 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–23. Open Publishing Association, 2018.

[17] Dines Bjørner. Domain Science & Engineering – A Review of 10 Years Work and a Laudatio. www.imm.dtu.dk/ dibj/2017/zcc/bjorner-zcc-80.pdf. In NaiJun Zhan and Cliff B. Jones, editors, *Symposium on Real-Time and Hybrid Systems – A Festschrift Symposium in Honour of Zhou ChaoChen*, LNCS 11180, pp. 6184. Springer Nature Switzerland AG, June 2018.

[18] Dines Bjørner. To Every Manifest Domain a CSP Expression www.imm.dtu.dk/ dibj/2016/mereo/mereo.pdf. *Journal of Logical and Algebraic Methods in Programming*, 1(94):91–108, January 2018.

[19] Dines Bjørner. An Assembly Plant Domain – Analysis & Description. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, September 2019. www.imm.dtu.dk/ dibj/2021/assembly/assembly-line.pdf.

[20] Dines Bjørner. Domain Analysis & Description – Principles, Techniques and Modeling Languages. *ACM Trans. on Software Engineering and Methodology*, 28(2):66 pages, March 2019.

[21] Dines Bjørner. Domain Analysis & Description – Principles, Techniques and Modelling Languages. www.imm.dtu.dk/ dibj/2018/tosem/Bjorner-TOSEM.pdf. *ACM Trans. on Software Engineering and Methodology*, 28(2), April 2019. 68 pages.

[22] Dines Bjørner. *Domain Science & Engineering – A Foundation for Software Development*. EATCS Monographs in Theoretical Computer Science. Springer, Heidelberg, Germany, January 2020. xii+346 pages[36].

[23] Dines Bjørner. *Rigorous Domain Descriptions. A compendium of draft domain description sketches carried out over the years 1995–2024. Chapters cover:*

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

. url: www.imm.dtu.dk/~dibj/2021/dd/dd.pdf, Fredsvej 11, DK-2840 Holte, Denmark, November 15 2021.

[24] Dines Bjørner. Domain Modelling. In Jonathan Bowen et al., editor, *Theories of Programming and Formal Methods: Essays Dedicated to Jifeng He on the Occasion of His 80th Birthday*, Lecture Notes in Computer Science, Festschrift. Spriger, Heidelberg, Germany, August 2023. https://www.imm.dtu.dk/ dibj/2023/FEA/hjf.pdf and https://www.imm.dtu.dk/ dibj/2023/final/HeJiFeng.pdf.

[25] Dines Bjørner. Domain Modelling – A Primer. A short and significantly revised version of [22]. xii+202 pages, May 2023.

[26] Dines Bjørner. Double-entry Bookkeeping. Research, Institute of Mathematics and Computer Science. Technical University of Denmark, DK-2800 Kgs.Lyngby, Denmark, August 2023. `http://www.imm.dtu.dk/ dibj/2023/doubleentry/dblentrybook.pdf`.

[27] Dines Bjørner. Pipelines: A Domain Science & Engineering Description. In *FSEN 2023: Fundamentals of Software Engineering, Teheran, Iran, May 3–5*. `www.imm.dtu.dk/ dibj/2023/tehran/tehran.pdf`, 2023.

[28] Dines Bjørner. Banking – A Domain Description. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, 18 August 2025. https://www.imm.dtu.dk/~dibj/2025/banking/main.pdf.

[29] Dines Bjørner. Domain Analysis & Description. In *Theoretical Aspects of Computing, ICTAC 2025*, number 16237 in Lecture Notes in Computer Science, pages 39–66. Springer, November 24–28 2025. A Tutorial.

[30] Dines Bjørner. Transport – A Domain Description. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, 12 June 2025. https://www.imm.dtu.dk/~dibj/2025/transport/main.pdf.

[31] Dines Bjørner. DADL: A Domain Analysis and Description Language. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, February 2026. Incomplete draft. https://www.imm.dtu.dk/~dibj/2026/ddl/ddl.pdf.

[36] Due to copyright reasons no URL is given to this document's possible Internet location. A revised version of this book is [32]

[32] Dines Bjørner. Domain Science & Engineering, a Primer[37]. Technical University of Denmark. Significantly revised edition of [22]. xii+211 pages. To be submitted, February 2026.

[33] Dines Bjørner and Mikhail Chupilko. RUSSIAN: Domain Science & Engineering, a Primer. Translation of [32]. To be submitted., 2026.

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

[35] Dines Bjørner and Yang ShaoFa. CHINESE: Domain Science & Engineering, a Primer. Translation of [32]. To be submitted., 2026.

[36] Dines Bjørner. Urban Planning Processes. www.imm.dtu.dk/ dibj/2017/up/urban-planning.pdf. Research Note, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, July 2017.

[37] R.M. Burstall and Peter J. Landin. Programs and their proofs: An algebraic appraoch. *Machine Intelligence*, 4:17–43, 1969.

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

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

[40] 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.

[41] 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.

[42] Charles Anthony Richard Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8), Aug. 1978.

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

[44] Charles Anthony Richard Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985.

[45] Charles Anthony Richard Hoare. Communicating Sequential Processes. Published electronically: `usingcsp.com/cspbook.pdf`, 2004. Second edition of [44].

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

[47] Peter J. Landin. The Mechanical Evaluation of Expressions. *Computer Journal*, 6(4):308–320, 1964.

[48] Peter J. Landin. A Correspondence Between ALGOL 60 and Church's Lambda-Notation (in 2 parts). *Communications of the ACM*, 8(2-3):89–101 and 158–165, Feb.-March 1965.

[49] Peter J. Landin. A Formal Description of ALGOL 60. In *[72]*, pages 266–294, 1966.

[37] This book is currently being translated into Chinese, [35], by Dr. Yang ShaoFa, IoS/CAS (Institute of Software, Chinese Academy of Sciences), Beijing and into Russian, [33], by Dr. Mikhail Chupilko and his colleagues, ISP/RAS (Institute of Systems Programming, Russian Academy of Sciences), Moscow

[50] Peter J. Landin. A Lambda Calculus Approach. In L. Fox, editor, *Advances in Programming and Non-Numeric Computations*, pages 97–141. Pergamon Press, 1966.

[51] Peter J. Landin. The Next 700 Programming Languages. *Communications of the ACM*, 9(3):157–166, 1966.

[52] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machines, Part I. *Communications of the ACM*, 3(4):184–195, 1960.

[53] John McCarthy. Towards a Mathematical Science of Computation. In C.M. Popplewell, editor, *IFIP World Congress Proceedings*, pages 21–28, 1962.

[54] J. M. E. McTaggart. The Unreality of Time. *Mind*, 18(68):457–84, October 1908. New Series. See also: [55].

[55] Robin Le Poidevin and Murray MacBeath, editors. *The Philosophy of Time*. Oxford University Press, 1993.

[56] Arthur N. Prior. *Logic and the Basis of Ethics*. Clarendon Press, Oxford, UK, 1949.

[57] Arthur N. Prior. *Formal Logic*. Clarendon Press, Oxford, UK, 1955.

[58] Arthur N. Prior. *Time and Modality*. Oxford University Press, Oxford, UK, 1957.

[59] Arthur N. Prior. *Past, Present and Future*. Clarendon Press, Oxford, UK, 1967.

[60] Arthur N. Prior. *Papers on Time and Tense*. Clarendon Press, Oxford, UK, 1968.

[61] Arthur N. Prior. *Changes in Events and Changes in Things*, chapter in [55]. Oxford University Press, 1993.

[62] Gerald Rochelle. *Behind time: The incoherence of time and McTaggart's atemporal replacement*. Avebury series in philosophy. Ashgate, Brookfield, Vt., USA, 1998. vii + 221 pages.

[63] Dana S. Scott. Outline of a Mathematical Theory of Computation. In *Proc. 4th Ann. Princeton Conf. on Inf. Sci. and Sys.*, page 169, 1970.

[64] Dana S. Scott and Christopher Strachey. Towards a mathematical semantics for computer languages. In *Computers and Automata*, volume 21 of *Microwave Research Inst. Symposia*, pages 19–46, 1971.

[65] Kai Sørlander. *Under Evighedens Synsvinkel [Under the viewpoint of eternity]*. Munksgaard · Rosinante, Copenhagen, Denmark, 1997. 200 pages.

[66] Kai Sørlander. *Den Endegyldige Sandhed [The Final Truth]*. Rosinante, Copenhagen, Denmark, 2002. 187 pages.

[67] Kai Sørlander. *Forsvaret for Rationaliteten*. Informations Forlag, Copenhagen, Denmark, 2008. 232 pages.

[68] Kai Sørlander. *Fornuftens Skæbne – Tanker om Menneskets Vilkår*. Informations Forlag, Copenhagen, Denmark, 2014. 238 pages.

[69] Kai Sørlander. *Indføring i Filosofien [Introduction to The Philosophy]*. Informations Forlag, Copenhagen, Denmark, 2016. 233 pages.

[70] Kai Sørlander. *Den rene fornufts struktur [The Structure of Pure Reason]*. Ellekær, Slagelse, Denmark, 2022. See [71].

[71] Kai Sørlander. *The Structure of Pure Reason*. Springer, February 2025. This is an English translation of [70] – done by Dines Bjørner in collaboration with the author.

[72] T. B. Steel, editor. *Formal Language Description Languages for Computer Programming, IFIP TC-2 Working Conference, 2964, Baden.* North-Holland Publ.Co., Amsterdam, 1966.

[73] Christopher Strachey. Fundamental concepts in programming languages. Unpubl. Lecture Notes, NATO Summer School, Copenhagen, 1967, and Programming Research Group, Oxford Univ., 1968.

[74] Johan van Benthem. *The Logic of Time*, volume 156 of *Synthese Library: Studies in Epistemology, Logic, Methhodology, and Philosophy of Science (Editor: Jaakko Hintika).* Kluwer Academic Publishers, P.O.Box 17, NL 3300 AA Dordrecht, The Netherlands, second edition, 1983, 1991.

A Behaviour Patterns

A.1 Behaviour Defintions

A typical, informal rendition of abstracted behaviours, `BA, BC, BD, ...` is shown in Fig. A.1.

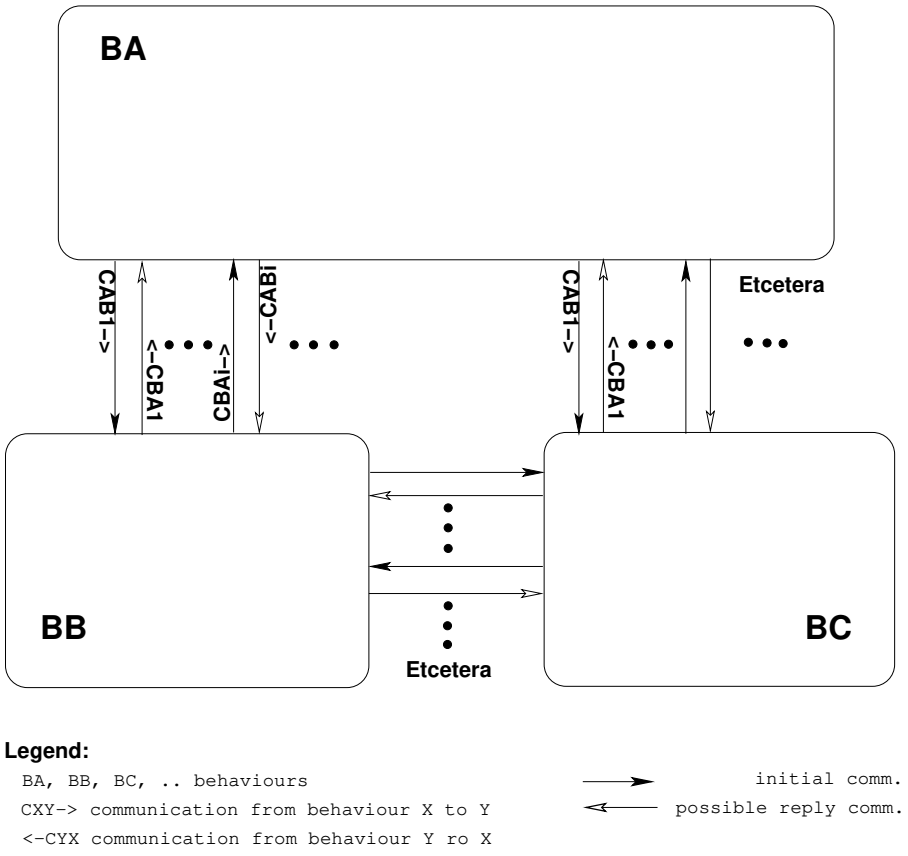

Figure 4: Communicating Behaviours

Figure A.1 should be understood as follows:[38] The **bold faced** labels **BA, BB, BC, ...** are meant to designate behaviours. The **black arrows**, from behaviour **Bx** to behaviour **By** are meant to designate CSP-like *communications* from **Bx** to **By**. The **open arrows** ("white"), from behaviour **Bx** to behaviour **By** are meant to designate possible CSP-like *communications* from **Bx** to **By**. These

[38] The explanation of Fig. A.1 is in now way an attempt to explain the semantics of behaviours. That is left to the RSL$^+$ formalization's.

latter communications, the "possible" ones, are then thought of as *in response* to the "earlier", in the figure: "immediately prior, next to" communication from **Bx** to **By**.

Figure A.1 on the previous page is now given a more precise "meaning" – with this "meaning" suggesting a general "pattern" for behaviour definitions:

107. There are behaviours `B, ...` with identities `bi, ...` .
    (a) These behaviours,typically, have the form of internal, ⌈⌉, non-deterministically "choosing" between
    (b) *pro-actively* initiating communications with other behaviors
    (c) and *re-actively* responding to such initiatives.

```
value
107a.  B(bi)(mereo)(stat)(mon)(prg) ≡
107b.      pro_active_B(bai)(mereo)(stat)(mon)(prg)
107c.  ⌈⌉ re_active_B(bai)(mereo)(stat)(mon)(prg)
```

108. ι107b π42 The pro-active behaviour (`B`) internal deterministically (⌈⌉) choosing between a number of initiating actions:
    (a) action 1,
    (b) action 2,
    (c) ...,
    (d) action n.

```
value
108.,ι107b π42.  pro_active_B(bi)(mereo)(stat)(mon)(prg) ≡
108a.                B_action_1(bi)(mereo)(stat)(mon)(prg)
108b.              ⌈⌉ B_action_2(bi)(mereo)(stat)(mon)(prg)
108c.              ⌈⌉ ...
108d.              ⌈⌉ B_action_m(bi)(mereo)(stat)(mon)(prg)
```

109. ι107b π42. The `responding` behaviour (`B`) reacts to a number of such initiating actions by
    (a) external non-deterministically ([]) offering to accept messages from responding behaviours,
    (b) and then performing corresponding actions.

```
value
109a.,ι107b π42.  respond_B(bi)(mereo)(stat)(mon)(prg) ≡
109a.                 let msg = [] { comm[{bj,bi}]? | bj:BI • bj ∈ bis } in
109b.                 react_behaviour_B(bi)(mereo)(stat)(mon)(prg)(msg) end
```

110. The `react_behaviour_B` inquires as to the type of the message, say, a command, received (**?**): if it is:
    (a) of type `Cmd_i` then it performs action `act_cmd_i`,
    (b) of type `Cmd_j` then it performs action `act_cmd_j`,
    (c) ..., or
    (d) of type `Cmd_k` then it performs action `act_cmd_k`.
    (e) If it is of neither of these types then it "skips" treatment of that response by resuming to be the behaviour `B`.

value
110. react_behaviour_B(bi)(mereo)(stat)(mon)(prg)(**msg**) ≡
110a. is_action_i(msg) → B_action_i(bi)(mereo)(stat)(mon)(prg)(**msg**),
110b. is_action_j(msg) → B_action_j(bi)(mereo)(stat)(mon)(prg)(**msg**),
110c. ...,
110d. is_action_k(msg) → B_action_k(bi)(mereo)(stat)(mon)(prg)(**msg**),
110e. _ → B(bi)(mereo)(stat)(mon)(prg)

A.2 Action Definitions

"Actions are what makes behaviours meaningful." We remind the reader that our function (incl. behaviour) definitions are expressed in a functional, "applicative", style. [that is, there are no assignable variables] The actions elaborate to **value**s. These values may be Booleans, numbers, sets, Cartesians, lists, maps and functions (over these), or the values by be `()`, of type **Unit**, as are the values (also of never-ending) behaviours.

Action signatures usually "follow that", i.e., are the same as "their" initiating behaviour signatures.

111. Actions, as semantic quantities,
   (a) evaluate some values,
   (b) typically change some programmable attributes,
   (c) and may communicate, "issue" or inform, to some other behaviours, some requests, respectively information –
   (d) whereupon the "revert", "tail-recursively" to the activating Behaviour.

111. action_i(bi)(mereo)(stat)(mon)(prg) ≡
111a. **let** v = evaluate_i(bi)(mereo)(stat)(mon)(prg) **in**
111b. **let** (bj,prg′) = elaborate_i(v)(bi)(mereo)(stat)(mon)(prg) **in**
111c. **comm**[{bi,bj}]!$\mathcal{E}$(prg′) ;
111d. behaviour(bi)(mereo)(stat)(mon)(prg′)
111. **end end**

Variants of Item ι111c π43 are also used:

{ **comm**[{bi,bj}]!$\mathcal{E}$(prg′) | bj ∈ bis } ;

where `bj` ranges over `bis`, a set of behaviour identities.

B Example

In this example we omit many of the keywords, etc., otherwise mentioned in Sects. 4.1–4.2.1, etc.

B.1 Universe of Discouse

112. The universe of discourse is that of
113. road traffic, RT –
114. whose narrative is then presented.

112. **universe of discourse**:
113. **type** RT
114. **narrative**: The road traffic domain consists of a road net aggregate and an automobile aggregate.
Road net aggregates consists of aggregates of hubs (street [link] intersections) and links.
The aggregates of hubs and links consists of sets of hubs and sets of links.
Hubs and links are here considered atomic.
Automobile aggregates ``are'' a set **of** automobiles −− that are here considered atomic.
Hubs, links and automobiles have unique identification, mereologies and attributes.
Autombile ride along hubs and links; leave hubs [links] to enter links [hubs], etc.

B.2 Compound Endurants

115. In the road traffic domain we can observe a road net aggregate and an automobile aggregate.
116. In a road net aggregate we can observer an aggregate of hubs and an aggregate of links.
117. In an aggregate of automobiles we can observe a set of [atomic] automobiles.
118. Aggregates of hubs ad links are sets of [atomic] hubs and links.

type
115. RNA, AA
116. AH, AL
117. AS = A**-set**
118. HS = H**-set**, LS = L**-set**

value
115. **obs_**RNA: RT → RNA, **obs_**AA: RT → AA
116. **obs_**AH: RNA → AH, **obs_**AL: RNA → AL
117. **obs_**AS: AA → AS
118. **obs_**HS: AH → RS, **obs_**LS: AL → LS

B.3 Endurant Values

119. There is the road transport [universe of discourse].
120. There is the road net aggregate.
121. There is the autombile aggregate.
122. There is the hub aggregate.
123. There is the link aggregate.
124. There is the set of autombiles.
125. There is the set of hubs.
126. There is the set of links.
127. And there is the entire set of all of these.
128. And there is the entire set of just all automobiles, hubs and links.

value
119. rt:RT
120. rna:RNA = **obs_**RNA(rt)
121. aa:AA = **obs_**AA(rt)
122. ah:AH = **obs_**AA(rna)
123. al:AL = **obs_**AL(rna)
124. as:AS = **obs_**AS(ah)
125. hs:HS = **obs_**HS(ah)
126. ls:LS = **obs_**LS(al)
127. σ_{parts} = {rt,rna,aa,ah,al} ∪ aa ∪ hs ∪ ls
128. σ_{atoms} = aa ∪ hs unon ls

B.4 Taxonomy

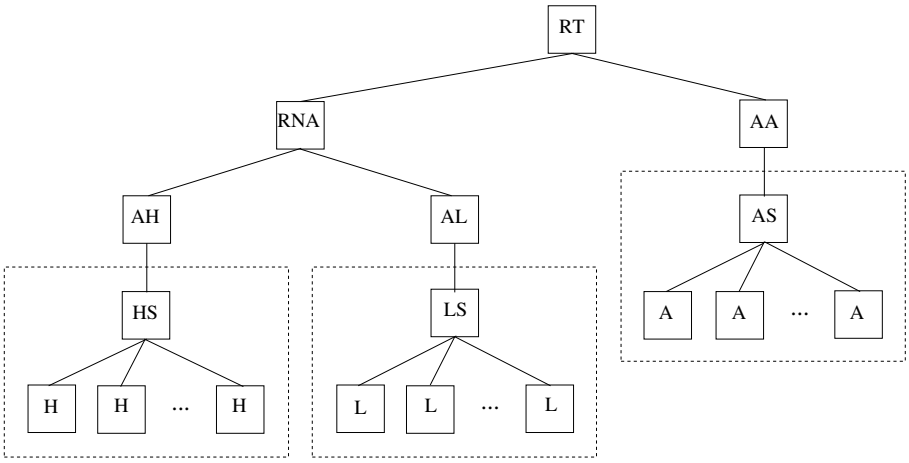

Figure 5: A Road Transport Taxonomy

B.5 Unique Identification

The unique identifierss of a road transport, rt:RT, is here limited to just hubs, links and automobiles:

129. The universe of discourse has a unique identifier.
130. The road net agregate has a unique identifier.
131. The automobile aggregate has a unique identifier.
132. The hub aggregate has a unique identifier.
133. The link aggregate has a unique identifier.
134. Each hub has a unique identifier.
135. Each link has a unique identifier.
136. Each automobile has a unique identifier.

type
129. RTI
130. RNAI
131. AAI
132. HAI
133. LAI
134. HI
135. LI
136. AI

value
129. **uid_**RT: RT → RTI
130. **uid_**RNA: RNA → RNAI
131. **uid_**AA: AA → AAI
132. **uid_**HA: HA → HAI
133. **uid_**LA: LA → LAI
134. **uid_**H: H → HI
135. **uid_**L: L → LI
136. **uid_**A: A → AI .

B.6 Unique Identifier Values

137. There is the unique identifier of the road transport.
138. There is the unique identifier of the road net aggregate.
139. There is the unique identifier of the autombile aggregate.
140. There is the unique identifier of the hub aggregate.
141. There is the unique identifier of the link aggregate.
142. There are the unique identifiers of the automobiles of the set of autombiles.
143. There are the unique identifiers of the hubs of the set of hubs.
144. There are the unique identifiers of the links of the set of links.
145. And there is the entire set of all of these.
146. And there is the entire set of all identifiers of the automobiles, hubs and links.

value
137. rt_{uid}RT: RT = **uid_**RT(rt)
138. rna_{uid}RNA: RNA = **uid_**RNA(rt)
139. aa_{uid}AA: AA = **uid_**AA(rt)
140. ah_{uid}AH: AH = **uid_**AA(rna)
141. al_{uid}AL: AL = **uid_**AL(rna)
142. as_{uid}AS: AS = {**uid_**A(a)|a:A•a∈as}
143. hs_{uid}:HS = {**uid_**H(h)|h:H•h∈hs}
144. ls_{uid}:LS = {**uid_**L(l)|l:L•a∈ls}
145. $\sigma_{parts_{uid}}$ = {rt,rna,aa,ah,al} ∪ aa ∪ hs unon ls
146. $\sigma_{atoms_{uids}}$ = aa ∪ hs ∪ ls

B.7 Uniqueness of Endurants

147. Parts are uniquely identfied.

axiom
147. **card** σ_{atoms} = **card** $\sigma_{atoms_{uids}}$

B.8 Mereology

We shall be concerned only with the mereology of some manifest parts.

148. The mereology of links is a 2 element set of hub identifiers.
149. The mereology of a hub is a possibly empty set of hub identifiers.
150. The mereology of an automobile is a set of hub and link identifiers

type
148. ML = LI**-set axiom** ∀ ml:MK • **card** ml = 2 ∧ ml ⊆ ls_{uis}
149. MH = HI**-set axiom** ∀ mh:MH • mh ⊆ hs_{uis}
150. MA = (HI|LI)**-set axiom** ∀ ma:MA • ma ⊆ as_{uis}

value
148. **mereo_**L: L → ML
149. **mereo_**H: H → MH
150. **mereo_**A: A → MA .

B.9 Attributes

Example attributes are:

- **Hubs:**
  151. Hubs have states, hσ:HΣ: the set of pairs of link identifiers, ($f$li,$t$li), of the links $f$rom and to which automobiles may enter, respectively leave the hub.
  152. Hubs have state spaces, hω:HΩ: the set of hub states "signaling" which states are open/closed, i.e., **green**/**red**.
- **Links:**
  153. Links that have lengths, LEN;
  154. Links have states, lσ:LΣ: the set of pairs of link identifiers, ($f$li,$t$li), of the links $f$rom and to which automobiles may enter, respectively leave the hub.
  155. Links have state spaces, lω:LΩ: the set of link states "signaling" which states are open/closed, i.e., **green**/**red**.
- **Automobiles:**
  156. Automobiles have road net positions, APos,
     (a) either *at a hub*, atH,
     (b) or *on a link*, onL, some fraction, f:**Real**, down a link, identified by li, from a hub, identified by fhi, towards a hub, identified by thi.
  157. Automobiles have velocity;
  158. Automobiles have acceleration;
  159. Etc.[39]
- **Hubs, Links, Automobiles:**

[39]

160. Hubs, links and automobiles have *histories:* time-stamped, chronologically ordered sequences of automobiles entering and leaving links and hubs, with automobile histories similarly recording hubs and links entered and left.
161. Link positions have well-defined identifiers and fractions.

type
151. HΣ = (LI×LI)**-set** [**prg**]
152. HΩ = HΣ**-set** [**sta**]
153. LEN = **Nat** [**sta**] [m]
154. LΣ = (HI×HI)**-set** [**prg**]
155. LΩ = LΣ**-set** [**sta**]
156. APos = atH | onL [**prg**]
156a. atH :: HI
156b. onL :: LI × (fhi:HI × f:**Real** × thi:HI)
157. Veloc = **Real** [**mon**] [km/h]
158. Accel = **Real** [**mon**] [m/sec]
159. ...
160. HHis,LHis = (𝕋𝕀𝕄𝔼×AI)* [**prg**]
160. AHis = (𝕋𝕀𝕄𝔼×(HI|LI))* [**prg**]

value
151. attr_HΣ: H → HΣ
152. attr_HΩ: H → HΩ
153. attr_LEN: L → LEN
154. attr_LΣ: L → LΣ
155. attr_LΩ: L → LΩ
156. attr_APos: A → APos
157. attr_Veloc: A → Veloc
158. attr_Accel: A → Accel
160. attr_HHis: H → HHis
160. attr_LHis: L → LHis
160. attr_AHis: A → AHis

axiom
161. ∀ mk_onL(li,(fhi,f,thi)):onL • 0<f<1 ∧ li∈ls_{uids} ∧ {fhi,thi}⊆hs_{uids} ∧ ....

B.10 Intentional Pull

162. An *intentional pull* of any road transport system, rts, is then:
    (a) if for any automobile, a, of rts, on a link, ℓ (hub, h), at time τ,
    (b) then that link, ℓ, (hub h) "records" automobile a at that time.
163. and:
    (c) if for any link, ℓ (hub, h) being visited by an automobile, a, at time τ,
    (d) then that automobile, a, is visiting that link, ℓ (hub, h), at that time.

axiom
162a. ∀ a:A • a ∈ as ⇒
162a. **let** ahist = attr_AHist(a) **in**
162a. ∀ ui:(LI|HI) • ui ∈ **dom** ahist ⇒
162b. ∀ τ:𝕋𝕀𝕄𝔼 • τ ∈ **elems** ahist(ui) ⇒
162b. **let** hist = is_LI(ui) → attr_LHist(retr_L(ui))(σ),
162b. _ → attr_HHist(retr_H(ui))(σ) **in**
162b. τ ∈ **elems** hist(uid_A(a)) **end end**
163. ∧
163c. ∀ u:(L|H) • u ∈ ls∪hs ⇒
163c. **let** uhist = attr(L|H)Hist(u) **in**
163d. ∀ ai:AI • ai ∈ **dom** uhist ⇒
163d. ∀ τ:𝕋𝕀𝕄𝔼 • τ ∈ **elems** uhist(ai) ⇒
163d. **let** ahist = attr_AHist(retr_A(ai))(σ) **in**
163d. τ ∈ **elems** uhist(ai) **end end**

B.11 Auxiliary Types

We introduce the concepts or *paths*, i.e., *routes*, through/across a road net.

164. A path element identifier is either a link identifier or a hub identifier.
165. A path (of a road net) is a finite[40] sequence of one or more alternating hub and link identifiers
166. such that
    (a) neighbouring link identifiers are those of the mereology of the "in-between" hubs, and such that neighbouring hub identifiers are/is those of the mereology of the "in-between" link;
    (b) and hub identifiers of a path are hub identifiers of the road net,
    (c) and its neighbouring link identifier(s) are in the mereology of the identified hub;
    (d) and link identifiers of a path are link identifiers of the road net,
    (e) and its neighbouring hub identifier(s) are/is in the mereology of the identified link.
167. Given a hub [a link] identifier we can retrieve the identified hub [link].

type
164. PEI = LI | HI
165. Path = PEI*
166. **axiom** [Well-formed Paths]
165. ∀ path:Path •
166a. ∀ {i,i+1}⊆**inds** path ⇒
166a. ((is_HI(path[i])∧is_LI(path[i+1]) ∨ is_LI(path[i])∧is_HI(path[i+1]))
166b. ∧ (path[i]∈hs_{uis}⇒path[i+1]∈ls_{uis}
166c. ∧ **uid**_H(retr_hub(path[i]))∈**mereo**_L(retr_hub(path[i])))
166d. ∧ (path[i]∈ls_{uis}⇒path[i+1]∈hs_{uis}
166e. ∧ **uid**_L(retr_link(path[i])∈**mereo**_H(retr_link(path[i]))))
value
167. retr_hub: HI → H, retr_link: LI → L, retr_unit: UI → U
167. retr_hub(hi) **as** h • h ∈ hs ∧ **uid**_H(h)=hi
167. retr_link(li) **as** l • l ∈ ls ∧ **uid**_L(l)=li
167. retr_unit(ui) **as** u • u ∈ hs∪ls ∧ **uid**_U(u)=ui
167. **uid**_U(u) ≡ is_L(u)→**uid**_L(u),is_H(u)→**uid**_H(u)

The above **pre/post** condition allows for circular paths, i.e., possibly infinite paths that may contain the same hub or link identifier more than once.

B.12 Simple Function Values

We define a function that given a road net calculates all its non-circular paths.

168. The paths[41] function takes a road net – represented here by its "global" sets of hubs and links – and yields a possibly infinite set of paths – satisfying the wellformedness criterion of Sect. B.11.

    We define the paths function in two ways.

169. Either axiomatically

[40] We shall only consider finite paths. The `paths` function, Item 168 below, can easily be modified to yield also infinite length paths!

[41] **Alarm!** Check that this function indeed generates only finite length paths!

170. in terms of an **as** predicate, with the result being the "largest" such set all of whose paths satisfy the wellformedness criterion;

171. or inductively[42]:

    (a) **basis clause:** every singleton path of either hub or link identifiers of the road net form a path.

    (b) **inductive clause:** If `pi` and `pj` are finite, respectively possibly infinite paths of the "result", `ps`, such that

        i. paths `pi`⌢⟨`ui`⟩ and ⟨uj⟩⌢`pj` are in `ps`, and
        ii. the resulting concatenated path is not circular, and
        iii. the mereology of the last element of `pi` identifies the first element of `pj`,
        iv. then their concatenation is a path in `ps`.

    (c) **extremal clause:** No path is an element of the desired set of paths unless it is obtained from the basis and the inductive clause by a finite number of uses.

value
168. paths: **Unit** → Path**-infset**
169. paths() **as** ps
170. **such that:** ∀ p:ps satisfy the wellformedness of Sect. B.11 on the previous page

We can also express the `paths` functions explicitly:

value
168. paths: **Unit** → Path**-infset**
171. paths() ≡
171a. **let** ps = {⟨ni⟩ | ni:NI ∈ hs_{uis}}∪{⟨ei⟩ | ei:EI ∈ ls_{uis}}
171(b)iv. ∪ { pi⌢⟨ui⟩⌢⟨uj⟩⌢pj | pi⌢⟨ui⟩:Path**-set**, ⟨uj⟩⌢pj:Path**-infset** •
171b. ∧ ({pi⌢⟨ui⟩,⟨uj⟩⌢pj}⊆ps)
171(b)i. ∧ (ui∼ ∈ **elems** pj ∧ uj∼ ∈ **elems** pi)
171(b)iii. ∧ (ui ∈ **mereo_**U(retr_unit(uj)))
171(b)ii. ∧ (uj ∈ **mereo_**U(retr_unit(ui)))} **in**
171c. ps **end**
type
168. U = H|L

Solution to the equation, lines 171a–171(b)i, is "'obtained' by a smallest set fix-point reasoning.

172. Given a "global" road net, g, we can calculate a "similarly global" *paths* value:

value
172. *paths*:Path**-set** = paths(g)

With the notion of paths of a road net one can now examine whether

- a road net is strongly connected, that is, whether any hub or link can be "reached" from any other hub or link; or
- a road net consists of two or more sub-graphs, i.e., there are no links between hubs in two such sub-graphs;
- etc.

173. We can formulate a *theorem:* for every road net we have that every path, `p`, in `g`, also contains its `rev`erse path, `rev(p)` in `g`.

[42] https://www.cs.odu.edu/ toida/nerzic/content/recursive_def/more_ex_rec_def.html

theorem: [All finite paths have finite reverse paths]
173. ∀ g:G,p:Path•p ∈ paths(g) ⇒ rev_path(p) ∈ paths(g)
value
173. rev_path: P → P
173. rev_path(p) ≡
173. **case** p **of**
173. ⟨⟩ → ⟨⟩,
173. ⟨ui⟩ → ⟨ui⟩,
173. ⟨ui⟩^p′^⟨uj⟩ → ⟨uj⟩^rev_path(p′)^⟨ui⟩
173. **end**

We can define further functions. For example:

174. `path_length`,

175. `shortest_path`, etc.

value
174. path_length: P → LEN
174. path_length(p) ≡
174. **case** p **of**
174. ⟨⟩ → 0
174. ⟨ui⟩ → retr_path_length(ui),
174. ⟨ui⟩^p′ → retr_length(ui)+path_path_length(p′)
174. **end**
174. retr_path_length: UI → LEN
174. retr_path_length(ui) ≡ (is_EI(ui)→**attr**_LEN(retr_link(ui)),is_NI(ui)→0)

175. shortest_path: G → P**-set**
175. shortest_path(g) ≡
175. **let** ps = paths(g) **in**
175. { p | p:P • retr_len(p) ∧ ∀ p′:P•p′∈ps ∧ retr_path_len(p)≤retr_path_len(p′) }
175. **end**

B.13 Channel

176. There is a set of channels between hubs, links and automobiles.

177. These channels communicate messages, `M`. `M` will "transpire" frm the behaviour definitions.

channel
176. { ch[{ui,uj}] | {ui,ij}:(HI|LI|AI)**-set** • ui≠uj∧{ui,uj}⊆σ_{uids} } M
type
176. M .

B.14 Variables

B.15 Behaviours

There are three behaviours:

178. `automobile`, corresponding to endurants `a:A`, and with appropriate argument types,

179. `hub`, corresponding to endurants `h:H`, and with appropriate argument types, and

180. `link`, corresponding to endurants `l:L`, and with appropriate argument types.

value
178. automobile: AI → AM → ... → (Apos × AHist) → **Unit**
179. hub: HI → HM → (HΩ × ...) → (HΣ × HHist) → **Unit**
180. link: LI → LM → (LEN × LΩ × ...) → (LΣ × LHist) → **Unit**

181. The **automobile** behaviour is either *at a hub* or *on a link* – and **communicates** with the *hub* and *link* behaviours as to its entering, leaving, or remaining at the hub, respectively on the link.

182. Any **hub** behaviour is "passively" awaiting **communication** from automobile behaviours as to their entering, leaving, or remaining at the hub.

183. Any **link** behaviour is "passively" awaiting **communication** from automobile behaviours as to their entering, leaving, or remaining on the link.

181. automobile(ai)(ris)(...)(atH(hi),ahis)
181. automobile(ai)(ris)(...)(onL(li,(fhi,f,thi)),ahis)
182. hub(hi)(mh)(hω,...)(hσ,hhist)
183. link(li)(ml)(len,lω,...)(lσ,lhist)

184. We abstract automobile behaviour at a Hub (hi).
    (a) Either the automobile remains at the hub,
    (b) or, **internally non-deterministically**,
    (c) leaves the hub entering a link,
    (d) or, **internally non-deterministically**,
    (e) stops.

184 automobile(ai)(ris)(...)(atH(hi),ahis) ≡
184a automobile_remain_at_hub(ai)(ris)(...)(atH(hi),ahis)
184b ⌈⌉
184c automobile_leaving_hub(ai)(ris)(...)(atH(hi),ahis)
184d ⌈⌉
184e automobile_stop(ai)(ris)(...)(atH(hi),ahis)

where we leave it to the reader to fill in the signature of these three behaviours.

185. [184a] The automobile **remains** at a hub:
    (a) time is recorded,
    (b) informing the hub behaviour, whereupon
    (c) the automobile remains at that hub, "idling",

185 automobile_remain_at_hub(ai)(ris)(...)(atH(hi),ahis) ≡
185a **let** τ = **record**_𝕋𝕀𝕄𝔼() **in**
185b ch[{ai,hi}] ! τ ;
185c automobile(ai)(ris)(...)(atH(hi),⟨(τ,hi)⟩^ahis) **end**

186. [184c] The automobile **leaves** the hub entering link li:

(a) time is recorded;
(b) hub is informed of automobile leaving and link that it is entering;
(c) "whereupon" the vehicle resumes (i.e., "while at the same time" resuming) the vehicle behaviour positioned at the very beginning (0) of that link.

```
186  automobile_leaving_b(ai)({li}∪ris)(...)(atH(hi),ahis) ≡
186a     let τ = record_𝕋𝕀𝕄𝔼() in
186b     (ch[ {ai,hi} ] ! τ ‖ ch[ {ai,li} ] ! τ) ;
186c     automobile(ai)(ris)(...)(onL(li,(hi,0,_)),⟨(τ,li)⟩^ahis) end
186      pre: [hub is not isolated]
```

187. [184e] Or the automobile **stop**s, "disappears — off the radar" !

```
187  automobile_stop(ai)(ris),(...)(atH(hi),ahis) ≡ stop .
```

188. We abstract automobile behaviour on a Link li:

(a) Either (*internally non-deterministically*) the automobile **remains** on the link, advancing a fraction or halts [temporarily],
(b) or, *internally non-deterministically*,
(c) **leaves** the link [when reaching its end] and enters the connected hub,
(d) or, *internally non-deterministically*,
(e) **stops** [leaves the road net altogether (!)].

```
188  automobile(ai)(ris)(...)(onL(li(fhi,f,thi)),lhis) ≡
188a     automobile_remains_on_link(ai)(ris)(...)(onL(li(fhi,f,thi)),lhis)
188b     ⌈⌉
188c     automobile_leaving_linl(ai)(ris)(...)(onL(li(fhi,f,thi)),lhis)
188d     ⌈⌉
188e     automobile_stops_on_link(ai)(ris)(...)(onL(li(fhi,f,thi)),lhis)
```

We leave it to the reader to complete the definitions of `automobile_remains_on_link, automobile_leaving_link` and `automobile_stops_on_link`.

189. Hubs

190. external non-deterministically receives time-stamped, t, messages,ai, from automobiles as to their entering, remaining or leaving the hub.

191. They update their hub history accordingly and resume being a hub.

```
value
189. hub(hi)(hm)(hω,...)(hσ,hhist) ≡
190.     let (t,ai) = [] { ch[ {hi,ai} ] ? | ai ∈ hm } in
191.     hub(hi)(hm)(hω,...)(hσ,⟨(t,ai)⟩^hhist) end
```

192. Links

193. external non-deterministically receives time-stamped, t, messages,ai, from automobiles as to their entering, remaining or leaving the link.

194. They update their link history accordingly and resume being a link.

```
value
192. link(li)(lm)(len,lω,...)(lσ,lhist) ≡
193.     let (t,ai) = []  { ch[ {li,ai} ] ? | ai ∈ lm } in
194.     link(li)(lm)(len,lω,...)(lσ,⟨(t,ai)⟩^lhist) end
```

B.16 Initialization

195. Let us refer to the system initialization as parallel composition of behaviours:
    (a) All hubs are initialized,
    (b) and
    (c) all links are initialized,
    (d) and
    (e) all automobiles are initialized.

value
195. initialisation: **Unit** → **Unit**
195. initialisation() ≡
195a. ‖ { hub(**uid_**H(h))(**mereo_**H(h))(**attr_**HΩ(h),...)(**attr_**HΣ(h),**attr_**HΣ(h),**attr_**HHist(h))| h:H • h ∈ *hs* }
195b. ‖
195c. ‖ { link(**uid_**L(l))(**mereo_**L(l))(**attr_**LEN(l),...)(**attr_**LΣ(l),**attr_**LHist(l))| l:L • l ∈ *ls* }
195d. ‖
195e. ‖ { automobile(**uid_**A(a))(**mereo_**A(a))(**attr_**APos(a),**attr_**AHist(a)) | a:A • a ∈ *as* }

C SI Units

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

Table 1: Base SI Units

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

Table 2: Derived SI Units

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

Table 3: Further SI Units

Prefix name		deca	hecto	kilo	mega	giga
Prefix symbol		da	h	k	M	G
Factor	10^0	10^1	10^2	10^3	10^6	10^9

Prefix name	tera	peta	exa	zetta	yotta
Prefix symbol	T	P	E	Z	Y
Factor	10^{12}	10^{15}	10^{18}	10^{21}	10^{24}

Table 4: Standard Prefixes for SI Units of Measure

Prefix name		deca	hecto	kilo	mega	giga
Prefix symbol		da	h	k	M	G
Factor	10^0	10^1	10^2	10^3	10^6	10^9

Prefix name	tera	peta	exa	zetta	yotta
Prefix symbol	T	P	E	Z	Y
Factor	10^{12}	10^{15}	10^{18}	10^{21}	10^{24}

Prefix name		deci	centi	milli	micro	nano
Prefix symbol		d	c	m	μ	n
Factor	10^0	10^{-1}	10^{-2}	10^{-3}	10^{-6}	10^{-9}

Prefix name	pico	femto	atto	zepto	yocto
Prefix symbol	p	f	a	z	y
Factor	10^{-12}	10^{-15}	10^{-18}	10^{-21}	10^{-24}

Table 5: SI Units of Measure and Fractions