𝔻𝕆𝕄𝔸𝕀ℕ

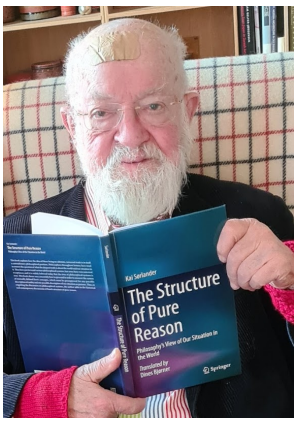
A Lexicon – The Method – An Example

*

Dines Bjørner

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

March 24, 2026: 09:56 am

Madeira November 2024 • [111] [ι 232,π 61] February 2025 • Bangkok October 2023

Abstract

A lexicon[1] of informatics terms is presented together with the 𝔻𝕆𝕄𝔸𝕀ℕ method and an example. The lexicon is personal. The 𝔻𝕆𝕄𝔸𝕀ℕ method is novel. The form of the example likewise.

[*] Around Friday, March 13, 2026, I had "more-or-less" completed a first draft of this document. I now find that I occsionally discover terms that I had so far omitted. Thus I am regularly extending the lexicon.

[1] – with due respect to [59]

Contents

Preface

- Over the years, and in my more than 120 publications, I have increasingly tried to be careful with consistent use of terms of my scientific field and its related fields: mathematics, philosophy and discrete, manifest and human-made and -assisted fields. i.e., domains.
- Hence this lexicon.
- The lexicon entries range from characterizations to definitions.
- Besides terms directly from the field of computing, this lexicon also lists terms from mathematics, philosophy and *domains* – where You are kindly asked to look up entries related to entry 63 on page 24.
- In composing this lexicon I often thought of
  - *Denis Diderot* [1713–1784] *Encyclopédie (1751–1772)* and
  - *Dr. Samuel Johnson* [1709–1784] *A Dictionary of the English Language (1755).*
- But, as noted on the cover of this lexicon, my real "predecessor" and master, is *Michael A. Jackson [59, 60]* [ι 123,π 36].
- There is `Wikipedia` – to which we all owe great thanks – and also `https://www.computer-dictionary-online.org/` to which I occasionally refer.
- In conceiving of the Domain Science and Engineering [17, 20, 22, 24] – between the years 2010–2025 – I had to use terms most of which were not [yet] in the conventional 'informatics' vocabulary; terms such as:

– domain	[ι 63,π 24][2]	– living species	[ι 134,π 39]	– mereology	[ι 149,π 42]
– entity	[ι 79,π 26]	– atomic part	[ι 13,π 13]	– attributes	[ι 14,π 14]
– endurant	[ι 74,π 25]	– compound part	[ι 38,π 20]	– perdurant	[ι 171,π 48]
– external quality	[ι 86,π 26]	– Cartesian part	[ι 33,π 20]	– channel	[ι 34,π 20]
– solid (entity)	[ι 222,π 58]	– part set	[ι 170,π 48]	– action	[ι 4,π 12]
– fluid (entity)	[ι 96,π 29]	– internal quality	[ι 119,π 34]	– event	[ι 81,π 26]
– part	[ι 169,π 48]	– unique identifier	[ι 249,π 70]	– behaviour	[ι 20,π 17]

  Hence this lexicon !
- **Where terms are not accredited, they are my, personal, characterisations !**
- There are enough of the latter kind of terms to justify, I sincerely believe, this separate lexicon.
- Writing this document started on October 4, 2025.
  I took a break from writing this document between late Oct. 2026 and March 7, 2026. It will now be updated and extended daily ! (?)
  Currently many entries have no text other than the term TO BE WRITTEN.
  As of today, March 24, 2026: 09:56 am, many terms are yet to be entered.
  There are currently 259 terms.

[2] [ι *i* π *p*] designates ιtem *i* πage *p*

1 Introduction

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. The domain embody *endurants* 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 persists, endures – capable of enduring adversity, severity, or hardship [Merriam Webster].

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* [Merriam Webster].

- **Domain engineering** is about the analysis & construction of descriptions of domains.
- **Domain science** is about the study of the underlying principles and properties of such domains and their analysis & description.

Domains – such as we see them: whole infrastructures, like the financial service industry, or the global, multi-mode (land, sea, air) transport, of retailers: their supply chains and customers, or the health sector: from family doctors, via clinics, pharmacies, etc., to hospitals, etc. – or components of these are not really complicated: they do not embody, as seen from their stakeholders, at least not from their customers, You and me, complex algorithms – in fact none.

Algorithms enter the picture when we implement efficient software for their optimised support.

Domain descriptions are characterised by properties. And these properties can usually be expressed, simply and elegantly, by predicates that You and I can easily fathom !

In school, from early on, we are taught courses in reckoning & [later] mathematics, in physics [including chemistry], in botanic & zoology, in geography and geology – i.e., the natural sciences.

But, till recently, at least, children and teenagers were/are not taught how our – human made – infrastructures function !

They and we have to learn about these through experience – through a kind of "osmosis" !

With clear, studied, available, public descriptions of human-made infrastructure domains, we can start developing course material, school text-books, for teaching and learning about the national, regional and world-wide banking "industry", similarly about the transport industry, the manufacturing industry, the retail industry, the health care industry, etc.

This document – together with [17, 19, 21, 24, 29] – aims at redressing this situation. Examples of domain descriptions are [6].

• • •

The lexicon basically contains two set of terms.

Terms that I are particularly inteested in making sure that the reader and I undertand in basically the same way. They may already be known to You, but, most likely they then have a meaning different from how I mean them !

And then there are the terms that are "already" well-defined: I typically use characterisations of these taken from the `Internet` !

Among the former terms, those that may be "special", are those of:

That is: 101 terms – approximately $\frac{1}{2}$ the terms of this vocabulary.

1.1 On Terminology

In beginning a software development project, the Danish computing scientist Peter Naur (25 October 1928 –3 January 2016), `https://en.wikipedia.org/wiki/Peter_Naur`) said:

- *"Establish first a terminology document,"*
- *"maintain it daily, throughout the project,"*
- *"adhering to it, diligently,"*
- *"and let it be an element of the software documentation."*

So I have taught my students – and it was a core mantra at DDC: Dansk Datamatik Center, an R&D centre (1979–1989) serving 10 Danish software developing and using companies and institutions. In all my own, experimental research projects [6], these *terminology documents* were manifested in the index to all narratives and formalisations: texts and formula.

In beginning this document, on October 4, 2025, I set out to just list and explain the terms of the area of computing science [ι 48,π 21] that I was interested in. Now, as from March 7, 2026, this *Informatics Lexicon* has taken a "new" turn: The terminology/lexicon is about the mathematical sciences of discrete mathematics [ι 59,π 23] and mathematical logic [ι 143,π 41], and about domains [ι 63,π 24] – their science and engineering. It therefore felt "natural" to let, in selected cases, the explication of terms be a "accompanied" by examples. These are shown a $\boxed{\text{framed}}$ texts and, usually, formula.

1.2 An Editorial Remark

On March 9, during the very early morning hours, 3 pm–4 pm, it then became clear to me that, really, the setting of this lexicon should reflect my last 15 years of work on *domain science & engineering* [13, 11, 14, 15, 12, 16, 17, 19, 21, 6, 29, 24, 25, 23, 27, 26, 28], i.e., the 𝔻𝕆𝕄𝔸𝕀ℕ concepts.

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. The domain embody *endurants* and *perdurants*.

By a **method** we shall understand a *set of* ***principles*** and ***procedures*** for *selecting* and *applying* a *set of* ***techniques*** *using* a *set of* ***tools*** in order to *construct* an *artefact* [DB].

Thus this document focus on terms related to 𝔻𝕆𝕄𝔸𝕀ℕ and *method, endurants* 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 persists, endures – capable of enduring adversity, severity, or hardship [Merriam Webster].

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* [Merriam Webster].

I shall, in particular, list and explain the terms related to these terms and I shall illustrate their "use" in developing domain descriptions, both the [method] *procedure* that I suggest analyser cum describers follow and the emerging domain description units.

1.3 The 𝔻𝕆𝕄𝔸𝕀ℕ Method

A [western glyph] lexicon naturally lists its entries, there are currently 259 such, in alphabetic order. The named *stages* and *steps* of the 𝔻𝕆𝕄𝔸𝕀ℕ method's *procedure*, however, "break" that order. So we now list these stages and steps in the order of their deployment.

- **Stage 0. Initialisation:** Cf. Entry [ι 115,π 34] and *Method* Sect. 3.3 on page 78
- **Stage 1. Universe of Discourse:** Cf. *Entry* [ι 247,π 70], *Method* Sect. 3.4 on page 79 and *Example* Sect. 4.1 on page 95.
- **Stage 2. Discover Domain:** Cf. *Method* Sect. 262 on page 79.
- **Stage 3. External Qualities:** Cf. *Entry* [ι 86,π 26], *Method* Sect. 3.6.1.1 on page 80 and *Example* Sect. 4.2.1.1 on page 95.
  - **Stage 4. Endurant Descriptions:** Cf. *Entry* [ι 75,π 25], *Method* Sect. 3.6.1.1 on page 80 and *Example* Sect. 4.2.1.1 on page 95.
  - **Stage 5. Endurant Taxonomy:** Cf. *Entry* [ι 78,π 26], *Method* Sect. 3.6.1.2 on page 83 and *Example* Sect. 4.2.1.3 on page 96.
  - **Stage 6. Endurant States:** Cf. *Entry* [ι 77,π 25], *Method* Sect. 3.6.1.3 on page 84 and *Example* Sect. 4.2 on page 95.
- **Stage 7. Internal Qualities:** Cf. *Entry* [ι 119,π 34], *Method* Sect. 3.7 on page 85 and *Example* Sect. 4.2.2 on page 96.
  - **Stage 8. Unique Identification:** Cf. Entry [ι 248,π 70] and *Method* Sect. 3.7.1 on page 85
    - **Stage 9. Unique Identifiers:** Cf. *Entry* [ι 249,π 70], *Method* Sect. 3.7.1.1 on page 86 and *Example* Sect. 4.2.2.1 on page 96.
    - **Stage 10. Unique Identifier State:** Cf. *Entry* [ι 250,π 70], *Method* Sect. 3.7.1.2 on page 87 and *Example* Sect. 4.2.2.1.1 on page 97.
    - **Stage 11. Uniqueness:** Cf. *Entry* [ι 251,π 70], *Method* Sect. 3.7.1.3 on page 87 and *Example* Sect. 4.2.2.1.2 on page 98.
  - **Stage 12. Mereology:** Cf. *Entry* [ι 149,π 42], *Method* Sect. 3.7.2 on page 87 and *Example* Sect. 4.2.2.2 on page 98.
  - **Stage 13. Attributes:** Cf. *Entry* [ι 14,π 14], *Method* Sect. 3.7.3 on page 88 and *Example* Sect. 4.2.2.3 on page 98.
  - **Stage 14. Intentional Pulls:** Cf. *Entry* [ι 117,π 34], *Method* Sect. 3.7.4 on page 89 and *Example* Sect. 4.2.2.4 on page 100.
- **Stage 15. Perdurants:** Cf. *Entry* [ι 171,π 48], *Method* Sect. 3.8 on page 90 and *Example* Sect. 4.5 on page 104.
  - **Stage 16. Channel:** Cf. *Entry* [ι 34,π 20], *Method* Sect. 3.8.1.1 on page 91 and *Example* Sect. 4.5.1 on page 104.
  - **Stage 17. Actions:** Cf. Entry [ι 4,π 12] and *Method* Sect. 3.8.1.2 on page 91
  - **Stage 18. Behavioral Taxonomy:** Cf. *Entry* [ι 24,π 17], *Method* Sect. 3.8.1.3 on page 91 and *Example* Sect. 4.5.3 on page 104.
  - **Stage 19. Behavioural Signatures:** Cf. *Entry* [ι 22,π 17], *Method* Sect. 3.8.1.4 on page 92 and *Example* Sect. 4.5.4 on page 105.
  - **Stage 20. Behaviour Definitions:** Cf. *Entry* [ι 21,π 17], *Method* Sect. 3.8.1.5 on page 92 and *Example* Sect. 4.5.4 on page 105.
  - **Stage 21. Domain Instantiation:** Cf. *Entry* [ι 72,π 24], *Method* Sect. 3.8.1.6 on page 93 and *Example* Sect. 4.5.5 on page 108.

1.4 Some Comments

Some job characterisations may be in order:

- Mathematician [mathematical scientists] study [the foundations of] mathematics.
- Mathematical (control) engineers practice, i.e., uses mathematics – say for automation: planning, monitoring and controlling [human or engine] processes – using mathematical theories.
- Computer scientists study [the foundations of] computing – using mathematics.
- Computing scientists study methods of constructing domain descriptions and requirements prescriptions and developing software – using mathematical theories.
- Domain, requirements and software engineers either use or do not use the results of computing scientists !

As a consequence:

- Civil, electrical, electronics and chemical engineers use the results of the scientists of their respective fields – and, really, most of them need not bother about the theories of their field !

  They usually do follow the physical laws of their field.
- Domain, requirements and software engineers – in contrast – must daily deal with [mathematical] abstractions [usually of mathematical nature].

  They are usually unaware of the laws of their field !

2 A Lexicon

The lexicon, besides "standard" terms of informatics [ι 114,π 34], contains terms with which I have "a special relationship" ! Some of these are listed in the beginning of Sect. 3.1, pages 76–76.

2.1 A...A...A...A...A...A...

1. **Abrial, Jean-Raymond:** (6 November 1938 – 26 May 2025) was a French computer scientist and inventor of the Z [ι 257,π 75] and the B [ι 18,π 17] formal methods [ι 97,π 29].

2. **Abstraction:**

```
Conception, my boy, fundamental brain-work,
    is what makes the difference in all art
```

D.G. Rossetti[3]*: letter to H. Caine*[4]

Abstraction is a tool, used by the human mind, and to be applied in the process of describing (understanding) complex phenomena.

Abstraction is the most powerful such tool available to the human intellect.

Science proceeds by simplifying reality. The first step in simplification is abstraction. Abstraction (in the context of science) means leaving out of account all those empirical data which do not fit the particular, conceptual framework within which science at the moment happens to be working.

Abstraction (in the process of specification) arises from a conscious decision to advocate certain desired objects, situations and processes as being fundamental; by exposing, in a first, or higher, level of description, their similarities and — at that level — ignoring possible differences.

[52, C.A.R. Hoare [ι 111,π 33] *Notes on Data Structuring*]

A Domain Example

Example: 1 *A road net of street segments and street intersections can be abstracted as a graph: segments as edges and intersections as vertices – and the actions of moving around that net can be abstraced as traversals of the graph.*

3. **Abstract Interpretation:** In computer science, abstract interpretation is a theory of sound approximation of the semantics of computer programs, based on monotonic functions over ordered sets, especially lattices. It can be viewed as a partial execution of a computer program which gains information about its semantics (e.g., control-flow, data-flow) without performing all the calculations.

   Its main concrete application is formal static analysis, the automatic extraction of information about the possible executions of computer programs; such analyses have two main usages:

   - inside compilers, to analyse programs to decide whether certain optimizations or transformations are applicable;

[3] Dante Gabrielli Rosetti, 1828–1882, English poet, illustrator, painter and translator
[4] T. Hall Caine, 1853–1931, British novelist, dramatist, short story writer, poet and critic.

- for debugging or even the certification of programs against classes of bugs.

Abstract interpretation was formalized by the French computer scientist working couple Patrick Cousot and Radhia Cousot in the late 1970s [41, 42] with [40] being a masterly monograph on the subject (i.e., a summary of 40 years of research).

To us, abstract interpretation is a form of transcendental deduction [ι 241,π 67].

4. **Action:** An action is 'something' that potentially changes the state [of a domain] [ι 229,π 60].

   [In analysing domain [ι 63,π 24] behaviours [ι 20,π 17] it seems natural to ask the questions: *which are its proactive actions* [ι 178,π 49], *which are its reactive actions* [ι 184,π 51], *which actions are external non-deterministic* [ι 85,π 26], *which actions are internal non-deterministic* [ι 118,π 34], etcetera.]

5. **Active:** We define the notion of 'active' as it applies to the behaviours [ι 20,π 17] of domains [ι 63,π 24]. A domain behaviour is said to be active if it either, at is own "free will", at some time, i.e., internal non-deterministically [ι 118,π 34] decides to perform an action [ι 4,π 12] – referred to as a pro-active action [ι 178,π 49], or if it, in reaction to an action performed, or being performed, by another behaviour, performs an action, that is an external non-deterministic [ι 85,π 26] re-active action [ι 184,π 51].

6. **Actor:** An actor is 'something' that causes and action [ι 4,π 12] or an event [ι 81,π 26] or a behaviour [ι 20,π 17].

7. **Algebra:** By an *algebra* we shall understand two sets

   - $A = \{a_1, a_2, ..., a_n, ...\}$ and
   - $B = \{b_1, b_2, ..., b_m, ...\}$, of values, the *carriers* of the algebra, and
   - a set, $\Omega = \{\omega_1, \omega_2, ..., \omega_m\}$, usually a finite set, of operations:
     - $\omega_i(a_{j_1}, a_{j_2}, ..., a_{j_n}) = b_k$,
   - where:
     - $\omega_i : \Omega$ and
     - $\{a_{j_1}, a_{j_2}, ..., a_{j_n}\} \subseteq A$ and $b_k \in B$

Everyday (!) Examples

Example: 2 *The example of item 2 on the previous page is an example of an algebra; and so is is the Boolean Algebra of item 27 on page 17.*

Others are:

- *A Queue Algebra: The queue algebra has, as carrier, the union of the set of all queue element values with the set of all queue values and, for example, create empty queue, enqueue, dequeue, first ("oldest"), last ("youngest"), and is_empty queue as operations.*
- *A Directory Algebra: The directory algebra has, as carrier, the union of the set of all directory entry values (i.e., of value triples of entry name, date and information values) with the set of all directory values and, for example, create empty directory, insert entry in directory, directory look-up, edit directory entry and remove*

directory entry as operations.

- *A Graph Algebra: A graph algebra has, as carrier, the union of the set of all nodes, and the set of all edges, where nodes and edges are all distinctly labeled, and, for example, create empty graph, insert_node in graph, insert_edge in graph [between one or two nodes], depth_first_ search in graph and breadth_first_ search in graph, as [some of its] operations.*

8. **Algebraic Semantics:** See Entry [ι 205,π 55].

- **Algol 60:** short for Algorithmic Language 1960) is a member of the ALGOL family of computer programming languages. It followed on from ALGOL 58 which had introduced code blocks and the begin and end pairs for delimiting them, representing a key advance in the rise of structured programming. ALGOL 60 was one of the first languages implementing function definitions (that could be invoked recursively). ALGOL 60 function definitions could be nested within one another (a feature introduced by ALGOL 60) with lexical scope. It gave rise to many other languages, including CPL, PL/I, Simula, BCPL, B, Pascal, and C. Practically every computer of the era had a systems programming language based on ALGOL 60 concepts [63].

9. **Algorithm:** In mathematics and computer science, an algorithm is a finite sequence of mathematically rigorous instructions, typically used to solve a class of specific problems or to perform a computation `https://en.wikipedia.org/wiki/Algorithm#Definition`.

10. **Alloy:** An abstract model-oriented specification language[57] `https://en.wikipedia.org/wiki/Alloy_(specification_language)`.

11. **Applicative Programming:** In the classification of programming languages, an applicative programming language is built out of functions applied to arguments. Applicative languages are functional, and applicative is often used as a synonym for functional. However, concatenative languages can be functional, while not being applicative.

    The semantics of applicative languages are based on beta reduction of terms, and Side effect such as mutation of state are not permitted.

    Lisp and ML [ι 153,π 42] are applicative programming languages.

12. **Artefact:** [We use the term 'artefact' with a meaning – perhaps – different from "normal" use.] An artefact to us is an object [ι 164,π 46], a human constructed product (a car, a house, a book, a piece of music, a painting).

13. **Atomic Part:** By an *atomic part* we shall understand a part which the domain analyser, cf. [ι 65,π 24], considers to be indivisible in the sense of not meaningfully divisible, for the purposes of the domain, cf. [ι 63,π 24], under consideration, that is, to not meaningfully consist of sub-parts.

14. **Attribute:** Property. We usually associate attributes, as an internal quality [ι 119,π 34], with endurants [ι 74,π 25].[5]

    We can roughly distinguish between two kinds of attributes: those which can be motivated by **physical** (including chemical) **concerns**, and those, which, although they embody some form of 'physics measures', appear to reflect on **event histories**: *"if 'something', ϕ, has 'happened' to an endurant, e_a, then some 'commensurate thing', ψ, has 'happened' to another (one or more) endurants, e_b."* where the *'something'* and *'commensurate thing'* usually involve some 'interaction' between the two (or more) endurants. It can take some reflection and analysis to properly identify endurants e_a and e_b and commensurate events ϕ and ψ.

An Informal Domain Example

Example: 3 *Example attributes of (i) road sections (links)* are (i.1) `length`, (i.2) `open-in-one-or-the-other-or-both directions`, etc., (ii) *banks* are (ii.1) `account number`, (ii.2) `balance`, etc.

Attributes

Example: 4 *We refer to Method Sect. 3.7.3 on page 88 and Example Sect. 4.2.2.3 on page 98.*

15. **Axiom:** An axiom, postulate, or assumption is a statement that is taken to be true, to serve as a premise or starting point for further reasoning and arguments.

An Axioms Example

Example: 5 *Narrative:* *The meaning of designations:*

1. *Stacks, s:S, have elements, e:E;*
2. *the empty_S operation takes no arguments and yields a result stack;*
3. *the is_empty_S operation takes an argument stack and yields a Boolean value result.*
4. *the stack operation takes two arguments: an element and a stack and yields a result stack.*
5. *the unstack operation takes an non-empty argument stack, s, and yields a stack result s' which is like s, only it omits the most recently stacked element.*
6. *the top operation takes an non-empty argument stack and yields the most recently stacked element.*

The consistency relations:

7. *an empty_S stack is_empty, and a stack with at least one element is not;*

[5] Parts [ι 169,π 48] and fluids [ι 96,π 29]are typically recognised because of their spatial form and are otherwise characterised by their intangible, but measurable attributes. That is, whereas endurants, whether solid (as are parts) or fluids, are physical, tangible, in the sense of being spatial [or being abstractions, i.e., concepts, of spatial endurants], attributes are intangible: cannot normally be touched[6], or seen[7], but can be objectively measured[8]. Thus, in our quest for describing domains where humans play an active rôle, we rule out subjective "attributes": feelings, sentiments, moods. Thus we shall abstain, in our domain science also from matters of aesthetics.

8. *unstacking an argument stack, stack(e,s), results in the stack s; and*
9. *inquiring the top of a non-empty argument stack, stack(e,s), yields e.*

Formalization:

type
1. E, S

value
2. empty_S: **Unit** → S
3. is_empty_S: S → **Bool**
4. stack: E × S → S
5. unstack: S $\xrightarrow{\sim}$ S
6. top: S $\xrightarrow{\sim}$ E

The consistency relations:

axiom
7. ∼ is_empty_S(empty_S())
7. ∼ is_empty_S(stack(e,s))
8. unstack(stack(e,s)) = s
9. top(stack(e,s)) = e .

16. **Automaton, Finite State:** A finite-state machine (FSM) or finite-state automaton (FSA, plural: automata), finite automaton, or simply a state machine, is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some inputs; the change from one state to another is called a transition. An FSM is defined by a list of its states, its initial state, and the inputs that trigger each transition. Finite-state machines are of two types—deterministic finite-state machines and non-deterministic finite-state machines. For any non-deterministic finite-state machine, an equivalent deterministic one can be constructed.

17. **Axiomatic Semantics:** Axiomatic semantics is a formal approach in computer science that defines a program's meaning by describing the logical assertions that hold true before (precondition) and after (postcondition) its execution. Based on Hoare logic, it proves program correctness by using axioms and inference rules.

    Key Aspects of Axiomatic Semantics:

    - **Hoare [ι 111,π 33] Triples:** The core structure is
      - {`Precondition`} COMMAND {`Postcondition`}

      It specifies that if the COMMAND is executed in a state where the `Precondition` is true, and the COMMAND [execution] terminates, the `Postcondition` will hold.
    - **Partial vs. Total Correctness:** Partial correctness focuses on what holds if the program terminates (safety), while total correctness also ensures the program will terminate.
    - **Invariants:** Crucial for loops, an invariant is a condition that remains true before and after each iteration. Components: It consists of a language for stating assertions (logical expressions about variables) and rules for inferring the truth of these assertions.
    - **Goal:** The primary goal is to provide a formal, mathematical framework for proving the correctness of programs, rather than just describing their behavior.

    Axiomatic semantics is used to formally verify algorithms, check for bugs, and define the semantics of programming languages, serving as a contract between developers and users.

See [ι 205,π 55].

2.2 B...B...B...B...B...B...

18. **B:** A formal program development and specification language due to J.R. Abrial [ι 1,π 11] [1]

19. **Backus, John Warner:** (December 3, 1924 –March 17, 2007) was an American computer scientist. He led the team that invented and implemented FORTRAN, the first widely used high-level programming language, and was the inventor of the Backus–Naur form (BNF) [ι 25,π 17], a widely used notation to define syntaxes of formal languages. He also contributed to the design of Algol 60 [ι 2.1,π 13], and later researched the function-level programming paradigm.

    `[https://en.wikipedia.org/wiki/John_Backus]`

20. **Behaviour:** By a behaviour we shall understand a set [ι 207,π 56] of sequences [ι 215,π 57] of actions [ι 4,π 12], events [ι 81,π 26] and behaviours. [This characterisation shall be seen in light of our focus on domains – where behaviours are the transcendental deduction [ι 241,π 67] of behavioural parts [ι 23,π 17].

21. **Behaviour Definition:** A rigorous [formal or semi-formal] definition, in some specification language, of the behaviour of a part type.

22. **Behaviour Signature:** The type of a behaviour: its "input" argument and its "result" value.

23. **Behavioural Part:** A [domain] part [ι 169,π 48] which, the analyser cum describer decides is of behavioural kind, that is, can be transcendentally deduced [ι 241,π 67] into a behaviour.

24. **Behavioural Taxonomy:** A graphic rendition, usually in the form of a graph [ι 107,π 31] whose nodes denote part behaviours and whose edges denote communications between behaviours.

25. **BNF:** [Backus Normal[9] Form] was first devised by John Warner Backus [ι 19,π 17] [4] in connection with the committee work on the `Algol 60` [ι 2.1,π 13] [63] programming language definition[10].

26. **Boole, George:** [2 November 1815 – 8 December 1864] Born English. Professor at Queen's College, Cork, Ireland. Known for Boolean Algebra [ι 27,π 17].

    `https://en.wikipedia.org/wiki/George_Boole`.

27. **Boolean Algebra:** An algebra ([ι 7,π 12]) over Boolean truth values: **false, true,chaos**.

A Mathematics Example

Example: 6 *We exemplify a Boolean Algebra with three values:* ***Bool:*** $\{$***true, false, chaos***$\}$ *and the following operations:*

$$\sim : \mathbf{Bool} \to \mathbf{Bool}, \qquad = : \mathbf{Bool}\times\mathbf{Bool}\to\mathbf{Bool}$$
$$\wedge : \mathbf{Bool}\times\mathbf{Bool}\to\mathbf{Bool} \qquad \neq : \mathbf{Bool}\times\mathbf{Bool}\to\mathbf{Bool}$$
$$\vee : \mathbf{Bool}\times\mathbf{Bool}\to\mathbf{Bool} \qquad \Rightarrow : \mathbf{Bool}\times\mathbf{Bool}\to\mathbf{Bool}$$

[9] As suggested by Donald E. Knuth, BNF was later to be understood as `Backus Naur Form!` `[https://en.wikipedia.org/wiki/Backus-Naur_form]`.

[10] That work was documented by Peter Naur [ι 159,π 44].

$\vee, \wedge,$ ***and*** $\Rightarrow$ ***Syntactic Truth Tables***

$\vee$	**true**	**false**	**chaos**
true	**true**	**true**	**true**
false	**true**	**false**	**chaos**
chaos	**chaos**	**chaos**	**chaos**

$\wedge$	**true**	**false**	**chaos**
true	**true**	**false**	**chaos**
false	**false**	**false**	**false**
chaos	**chaos**	**chaos**	**chaos**

$\Rightarrow$	**true**	**false**	**chaos**
true	**true**	**false**	**chaos**
false	**true**	**true**	**true**
chaos	**chaos**	**chaos**	**chaos**

2.3 C...C...C...C...C...C...

28. **Calculation:** A calculation is a deliberate process that transforms one or more inputs into one or more results. The term is used in a variety of senses, from the very definite arithmetical calculation of using an algorithm, to the vague heuristics of calculating a strategy in a competition, or calculating the chance of a successful relationship between two people [Wikipedia].

29. **Call-by-Value, Call-by-Reference, Call-by-Name, ...:** These are some *Call-by-...* concepts:

    - **Call-by-Value:** In call by value (or pass by value), the evaluated value of the argument expression is bound to the corresponding variable in the function (frequently by copying the value into a new memory region).
    - **Call-by-Reference:** Call by reference (or pass by reference) is an evaluation strategy where a parameter is bound to an implicit reference to the variable used as argument, rather than a copy of its value.
    - **Call-by-Name:** Call by name is an evaluation strategy where the arguments to a function are not evaluated before the function is called—rather, they are substituted directly into the function body (using capture-avoiding substitution) and then left to be evaluated whenever they appear in the function. If an argument is not used in the function body, the argument is never evaluated; if it is used several times, it is re-evaluated each time it appears.

    We refer to [https://en.wikipedia.org/wiki/Evaluation_strategy] for a more "complete story"!

30. **Cantor, Georg Ferdinand Ludwig Philipp:** German, (3 March 1845 –6 January 1918) was a mathematician who played a pivotal role in the creation of set theory, which has become a fundamental theory in mathematics. Cantor established the importance of one-to-one correspondence between the members of two sets, defined infinite and well-ordered sets, and proved that the real numbers are more numerous than the natural numbers. Cantor's method of proof of this theorem implies the existence of an infinity of infinities. He defined the cardinal and ordinal numbers and their arithmetic. Cantor's work is of great philosophical interest, a fact he was well aware of [https://en.wikipedia.org/wiki/Georg_Cantor].

31. **Cartesian:** In mathematics, specifically set theory, and computer & computing science, a Cartesian is the product of two or more sets, in our case designated by type (or sort) names: A×B×...×C, denoting the possibly infinite set {(a,b,...,c)|a:A,b:B,...,c:C}.

    [Named after René Descartes [ι 56,π 23].]

32. **Cartesian Enumeration:** A Cartesian part [ι 33,π 20] enumeration of n elements, expressed by expressions $e_1, e_2, ..., e_n$, for $n > 1$[11], is:

    - $(e_1, e_2, ..., e_n)$

    where, of course, the ellipses, ..., must be filled in with proper expressions. Cf. [ι 132,π 39], [ι 141,π 40] and [ι 209,π 56].

[11] – it makes no sense to express $()$, for $n = 0$, or (a) for $n = 1$

33. **Cartesian Part:** A Cartesian part is a compound part [ι 38,π 20] which consists of an "indefinite number" of two or more parts of distinctly named sorts or types .

34. **Channel:** A medium for communicating messages between behaviours, as expressed in CSP [54].

35. **Church, Alonzo:** (June 14, 1903 –August 11, 1995) was an American computer scientist, mathematician, logician, and philosopher who made major contributions to mathematical logic and the foundations of theoretical computer science. He is best known for the lambda calculus, the Church–Turing thesis, proving the unsolvability of the Entscheidungsproblem ("decision problem"), the Frege–Church ontology, and the Church–Rosser theorem. Alongside his doctoral student Alan Turing, Church is considered one of the founders of computer science [wikipedia.org/wiki/Alonzo_Church].

36. **Clause:** In the context of 𝔻𝕆𝕄𝔸𝕀ℕ, by a clause, we mean such formal specification (include program code) texts which is either an expression [ι 84,π 26] or a statement [ι 226,π 60] or a CSP [ι 52,π 22] input/output specification.

37. **Combinatorics:** is an area of mathematics primarily concerned with counting, both as a means and as an end to obtaining results, and certain properties of finite structures. It is closely related to many other areas of mathematics and has many applications ranging from logic to statistical physics and from evolutionary biology to computer science [https://-en.wikipedia.org/wiki/Combinatorics].

38. **Compound Part:** Compound parts are those which either are Cartesian- [ι 33,π 20] or are set-oriented parts [ι 211,π 56] .

39. **Computability** Computability is the classification of mathematical problems based on whether they can be solved by an algorithm or a computing device, such as a Turing machine. It identifies which problems are solvable (decidable) and which are not, with key concepts including Turing machines, the Halting Problem, and recursion theory.

Key Concepts in Computability [https://plato.stanford.edu/entries/computability/][12]

40. **Computable Functions:** Functions for which an effective method (algorithm) exists to find the output for any given input.
41. **Decidability:** A problem is decidable if a Turing [ι 242,π 68] machine [ι 243,π 68] can be constructed to halt in finite time with the correct answer ("yes" or "no") for any input.
42. **Uncomputability:** Not all problems are solvable. A famous example is the Halting Problem, which proves that it is impossible to create an algorithm that determines whether any arbitrary program will eventually halt or run forever.
43. **Primitive Recursive Functions:** A subset of computable functions that can be built from simple building blocks (zero, successor, projection) using composition and recursion.

[12] The next five terms appear indented, and out of lexical order in this lexicon.

44. **Relative Computability:** Defines whether a problem is solvable if a Turing machine is allowed to ask questions to an "oracle" that knows the answer to a specific non-computable problem.

45. **Computation:** A computation is any type of calculation that includes both arithmetical and non-arithmetical steps and which follows a well-defined model (e.g. an algorithm).

    Mechanical or electronic devices (or, historically, people) that perform computations are known as computers [ι 46,π 21]. An especially well-known discipline of the study of computation is computer science [ι 47,π 21] `[Wikipedia]`.

46. **Computer:** A computer is a collection of *hardware* and *software*, that is, is a machine that can be instructed to carry out sequences of arithmetic and logical operations automatically via computer programming `[Wikipedia]`.

47. **Computer Science:** is the study and knowledge of the abstract phenomena that "occur" within computers [DB].

    As such computer science includes *theory of computation, automata theory, formal language theory, algorithmic complexity theory, probabilistic computation, quantum computation, cryptography, machine learning and computational biology.*.

48. **Computing Science:** is the study and knowledge of how to construct "those things" that "occur" within computers [DB].

    As such computing science embodies *algorithm and data structure design*, *functional-, logic-, imperative-* and *parallel programming*; *code testing, model checking* and *specification proofs*. Much of this can be pursued using *formal methods* [ι 97,π 29].

49. **Conceptual Part:** A part [ι 169,π 48] is conceptual if it has no physical presence. That is: it is an abstract idea that serves as a foundation for more concrete principles, thoughts, and beliefs `[https://en.wikipedia.org/wiki/Concept]`.

    A conceptual part is not manifest [ι 138,π 40].

50. **Conservative Extension:** An extension of a logical theory is conservative, i.e., conserves, if every theorem expressible in the original theory is also derivable within the original theory [en.wiktionary.org/wiki/conservative_extension].

51. **Continuation:** In the context of programming language specification, a continuation is a [higher-order] function. It denotes the mathematical meaning, i.e., function, of a program text as from a label of that program.

    In `[https://en.wikipedia.org/wiki/Continuation]`:

    In computer science, a continuation is an abstract representation of the control state of a computer program. A continuation implements (reifies) the program control state, i.e. the continuation is a data structure that represents the computational process at a given point in the process's execution; the created data structure can be accessed by the programming language, instead of being hidden in the runtime environment. Continuations are useful for encoding other control mechanisms in programming languages such as exceptions, generators, coroutines, and so on.

We refer to: *Reynolds, John C. (1993). The discoveries of continuations" (PDF). LISP and Symbolic Computation. 6 (3/4): 233–248;* and *Christopher Strachey and Christopher P. Wadsworth. Continuations: a Mathematical semantics for handling full jumps Technical Monograph PRG-11. Oxford University Computing Laboratory. January 1974. Reprinted in Higher Order and Symbolic Computation, 13(1/2):135—152, 2000, with a foreword by Christopher P. Wadsworth.*

`[https://en.wikipedia.org/wiki/Continuation#Implementation]`

52. **CSP:** A conceptual program specification language [54]. The variants of RSL that we use "contains" CSP.

53. **CSP Output/Input:** In 𝔻𝕆𝕄𝔸𝕀ℕ communication between behaviours [ι 20,π 17], say u_i, u_j. is expressed by:

    - Output: ch[$\{u_i, u_j\}$] ! expr: The value of expression expr is offered, by the behaviour identified by u_i to the behaviour identified by u_j.

      This clause [ι 36,π 20] is neither a statement nor an expression.

      It may only occur in behaviour definitions.
    - Input: ch[$\{u_i, u_j\}$] ?: Any value is offered, i.e., accepted, by the behaviour identified by u_j from the behaviour identified by u_i.

      This clause is an expression.

      It may only occur in behaviour definitions.

    The enactment of any of these clauses occur only if both can be satisfied, offered and accepted, at the same time. This the CSP output/input clauses offer both synchronisation and communication.

2.4 D...D...D...D...D...D...

54. **Dahl, Ole-Johan:** (12 October 1931 –29 June 2002) was a Norwegian computer scientist. Dahl was a professor of computer science at the University of Oslo and is considered to be one of the fathers of Simula and object-oriented programming along with Kristen Nygaard. In 2001, Dahl and Nygaard won the ACM Turing Award.

    [https://en.wikipedia.org/wiki/Ole-Johan_Dahl]

55. **Dashboard:** A dashboard (also called dash, instrument panel or IP, or fascia) is a control panel set within the central console of a vehicle, boat, or cockpit of an aircraft or spacecraft. Usually located directly ahead of the driver (or pilot), it displays instrumentation and controls for the vehicle's operation. An electronic equivalent may be called an electronic instrument cluster, digital instrument panel, digital dash, digital speedometer or digital instrument cluster. By analogy, a succinct display of various types of related visual data in one place is also called a dashboard, such as a dashboard in computing.

    [https://en.wikipedia.org/wiki/Dashboard]

- **Decidability:** see [ι 41,π 20].

56. **Descartes, René** (French, 31 March 1596 –11 February 1650) was a French philosopher, scientist, logician, and mathematician, widely considered a seminal figure in the emergence of modern philosophy and science during Renaissance era. Mathematics was paramount to his method of inquiry, and he connected the previously separate fields of geometry and algebra into analytic geometry [https://en.wikipedia.org/wiki/René_Descartes].

57. **Description:** A specification – used here solely in the sense of *domain descriptions* (in contrast to *requirements prescriptions* [ι 199,π 54] and software specifications [ι 221,π 58]).

58. **Denotational Semantics:** See [ι 205,π 55].

59. **Discrete Mathematics:** is the study of mathematical structures that can be considered "discrete" (in a way analogous to discrete variables, having a one-to-one correspondence (bijection) with natural numbers), rather than "continuous" (analogously to continuous functions). Objects studied in discrete mathematics include integers, graphs, and statements in logic. By contrast, discrete mathematics excludes topics in "continuous mathematics" such as real numbers, calculus or Euclidean geometry. Discrete objects can often be enumerated by integers; more formally, discrete mathematics has been characterised as the branch of mathematics dealing with countable sets (finite sets or sets with the same cardinality as the natural numbers). However, there is no exact definition of the term "discrete mathematics" [https://en.wikipedia.org/wiki/Discrete_mathematics].

    To us, discrete mathematics embody *set theory* [ι 212,π 56], *mathematical logic* [ι 143,π 41], *number theory* [ι 163,π 45], *algebra* [ι 7,π 12], *graph theory* [ι 108,π 32], *combinatorics* [ι 37,π 20],

60. **Discrete Endurant:** By a *discrete* [or *solid*] endurant [ι 74,π 25] we shall understand an endurant which is separate, individual or distinct in form or concept, or, rephrasing: have 'body' [or magnitude] of three-dimensions: length, breadth and depth [74, Vol. II, pg. 2046]. Same as *solid endurant* [ι 228,π 60].

61. **Discover Domain:** To discover a domain one must first, stage 0, initialise a dashboard and then go through the 21 endurant and perdurant analysis & description stages !
62. **Divide and Conquer:** In computer science, divide and conquer is an algorithm design paradigm based on multi-branched recursion. A divide-and-conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly [Wikipedia].
63. **Domain:** By a *domain* we shall understand a *rationally describable* segment of a *discrete dynamics* fragment of a *human assisted* reality, i.e., of the world. It includes its *endurants*, i.e., *solid and fluid entities* of *parts* and *living species*, and *perdurants* .
64. **Domain Analysis:** Inquiry into a domain as to its endurants [ι 74,π 25] and perdurants [ι 171,π 48].
65. **Domain Analyser:** The one or more persons who analyses a domain, 63.
66. **Domain Analysis & Description:** The analysis & description of domains. We suggest that this analysis & description is according to the domain analysis & description ontology of Fig. 1 on the facing page.
67. **Domain Description:** The description of a domain – as to its endurants, [ι 74,π 25], and perdurants, [ι 171,π 48] – and according to the *Domain Analysis & Description Ontology* of Fig. 1 on the next page. We show a multi-stage domain analysis & description procedure. It "starts" with initializing the domain analysis & description – through initializing a so-called "dashboard", state 0, and then proceeds to a 21 stage (!) domain discovery.

Initialisation

Example: 7 *See Entry [ι 55,π 23] and Method Sect. 3.3 on page 78.*

Discover Domain

Example: 8 ***From:*** *Entry [ι 247,π 70], Method Sect. 3.4 on page 79 and Example Sect. 4.1 on page 95 and Entry [ι 72,π 24],* ***to:*** *Method Sect. 3.8.1.6 on page 93 and Example Sect. 4.5.5 on page 108.*

68. **Domain Analysis & Description Ontology:** An ontology, [ι 167,π 46], that "guides" the *domain analyzer cum describer* in systematically analysing and describing domains, such as we charaterise domains, cf. [ι 63,π 24], [ι 64,π 24] and [ι 67,π 24]. See Fig. 1.
69. **Domain Describer:** The one or more persons who describes a domain, [ι 63,π 24].
70. **Domain Engineering:** is the engineering of *domain descriptions* based on the engineering of *domain analyses* [DB].
71. **Domain Modeling:** Same as domain analysis & description – [ι 64,π 24] & [ι 67,π 24].
72. **Domain Instantiation:** By domain instantiation we mean the invocation of one or more, usually all, the behaviours of behavioural parts.
73. **Domain Science & Engineering:** The scientific study and knowledge of domains, [ι 63,π 24], and the engineering [ι 76,π 25] practice of analysing [ι 65,π 24] and describing [ι 69,π 24] domains.

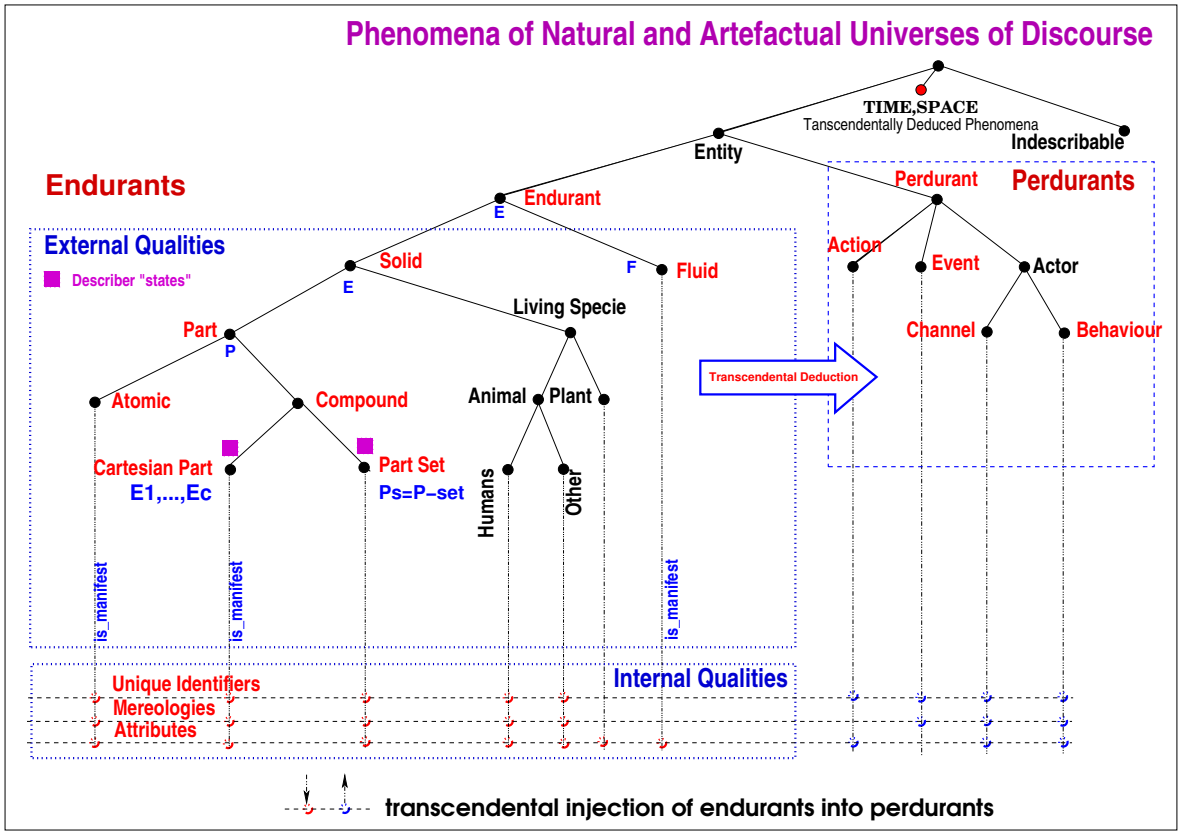


Figure 1: A Domain Analysis & Description Ontology

2.5 E...E...E...E...E...E...

74. **Endurant:** Endurants are those quantities of domains that we can observe (see and touch), in *space*, as "complete" entities at no matter which point in *time* – "material" entities that persist, endure .See `https://en.wikipedia.org/wiki/Endurance`.

75. **Endurant Description:** Endurants are either atoms, Cartesians, part sets, fluids or living species. To describe them is, for atoms, to describe their internal qualities [ι 119,π 34], for Cartesians and part sets to describe their components and internal qualities, and for fluids and living species to describes their internal qualities here omitted).

Endurant Description

Example: 9 *Cf. Method Sect. 3.5 on page 79 and Example Sect. 4.2 on page 95.*

76. **Engineering:** is the use of scientific principles to design and build machines, structures, and other items, including bridges, tunnels, roads, vehicles, and buildings `[Wikipedia]`. The engineer *walks the bridge between science and technology:* analysing man-made devices for their possible scientific properties and constructing technology based on scientific insight.

77. **Endurant States:** The values of [all] behavioral parts.

Endurant States

Example: 10 *Cf. Method Sect. 3.6.1.3 on page 84 and Example Sect. 4.2.1.2 on page 95.*

78. **Endurant Taxonomy:** A taxonomy [ι 233,π 62] of endurants. Usually in the form of an upside-down tree whose nodes designate endurants and whose branches designate "containment".

Endurant Taxonomy

Example: 11 *Cf. Method Sect. 3.6.1.2 on page 83 and Example Sect. 4.2.1.3 on page 96.*

79. **Entity:** By an entity we shall understand a phenomenon, i.e., something that can be observed, i.e., be seen or touched by humans, *or* that can be conceived as an abstraction of an entity; alternatively, a phenomenon is an entity, *if it exists, it is "being", it is that which makes a "thing" what it is: essence, essential nature*. If a phenomenon cannot be **rationally observed** or **thought about** and **rationally described** then it is not en entity.

80. **Epistemology:** is the branch of philosophy concerned with the theory of knowledge – and is the study of the nature of knowledge, justification, and the rationality of belief `[Wikipedia]`.

81. **Event:** A thing that happens or takes place, especially one of importance [Oxford Languages & Google]. Similar terms: *occurrence, happening, proceeding, episode, incident, affair*. To us, in domain science & engineering, [ι 73,π 24], an event is something that happens *surreptitiously,* causing a state, [ι 229,π 60], change.

82. **Event B:** A formal program specification language [2]. See also [ι 18,π 17].

    Event-B is a formal modeling language designed for rigorously modeling and verifying systems using abstraction and refinement. An Event-B model is a transition system where variables represent the state and events specify the transitions. Event-B uses a mathematical language that is based on set theory and predicate logic. Event-B is supported by the Rodin Platform, an extensible open source toolkit which includes facilities for modeling, verification (theorem proving and model checking) and validation (simulation).

    `https://eventb-soton.github.io/en-us/`

83. **Exception:** We refer to `[https://en.wikipedia.org/wiki/Exception_handling_(programming)#Termination_and_resumption_semantics]`.

84. **Expression:** In the context of 𝔻𝕆𝕄𝔸𝕀ℕ, by expression, we mean such formal specification (include program code) texts which designate a value. See also [ι 36,π 20] [clause] and [ι 226,π 60] [statement].

85. **External Nondeterminism:** External nondeterminism is here related only to domains . A nondeterministic domain [ι 63,π 24] is a domain which can behave, at certain points in it behaviour, between various alternatives of actions [ι 4,π 12]. The behaviour must decide between alternatives ["flip-the-coin", so to speak].

86. **External Qualities:** External qualities of endurants [ι 74,π 25] of a manifest [ι 138,π 40] domain [ι 63,π 24] are, in a simplifying sense, those we can see, touch and have spatial extent. They, so to speak, take form.

External Qualities

Example: 12 *Cf. Method Sect. 3.6 on page 80 and Example Sect. 4.2.1 on page 95.*

2.6 F...F...F...F...F...F...

87. **Facets [Domain]:**

By a domain facet we shall understand *one amongst a finite set of generic ways of analysing a domain: a view of the domain, such that the different facets cover conceptually different views, and such that these views together cover the domain*We shall cover the following facets:

- *intrinsics*,
- *support technologies*,
- *rules and regulations*,
- *scripts*,
- *license languages*,
- *management & organisation*,
- *human behaviour*.[13].

We refer to [21, *Chapter 8.*].

88. **Intrinsics [Domain Facet]:**
    - *By domain intrinsics we shall understand those phenomena and concepts of a domain which are basic to any of the other facets (listed earlier and treated, in some detail, below), with such domain intrinsics initially covering at least one specific, hence named, stakeholder view .*

    We refer to [21, Sect. 8.3].

89. **Support Technologies [Domain Facet]:**
    - *By a domain support technologywe shall understand ways and means of implementing certain observed phenomena or certain conceived concepts .*

    We refer to [21, Sect. 8.4].

90. **Rules & Regulations [Domain Facet]:**
    - *By a domain rule we shall understand some text (in the domain) which prescribes how people or equipment are expected to behave when dispatching their duties, respectively when performing their functions .*
    - *By a domain regulation we shall understand some text (in the domain) which prescribes what remedial actions are to be taken when it is decided that a rule has not been followed according to its intention .*

    We refer to [21, Sect. 8.4].

91. **Scripts [Domain Facet]:**
    - *By a domain script we shall understand the structured, almost, if not outright, formally expressed, wording of a procedure on how to proceed, one that has legally binding power, that is, which may be contested in a court of law .*

    We refer to [21, Sect. 8.5].

92. **License Languages [Domain Facet]:**

[13] By indenting the following entries we intend to show, in order of "execution",that the entries relate conceptually.

License: a right or permission
granted in accordance with law
by a competent authority
to engage in some business or occupation,
to do some act, or to engage in some transaction which but for such license
would be unlawful .

Merriam Webster Online [78]

We refer to [21, Sect. 8.6].

93. **Management & Organisation [Domain Facet]**
    - *By domain management we shall understand such people (such decisions) (i) who (which) determine, formulate and thus set standards concerning strategic, tactical and operational decisions; (ii) who ensure that these decisions are passed on to (lower) levels of management and to floor staff; (iii) who make sure that such orders, as they were, are indeed carried out; (iv) who handle undesirable deviations in the carrying out of these orders cum decisions; and (v) who "backstops" complaints from lower management levels and from "floor" staff .*
    - *By domain organisation we shall understand (vi) the structuring of management and non-management staff "oversee-able" into clusters with "tight" and "meaningful" relations; (vii) the allocation of strategic, tactical and operational concerns to within management and non-management staff clusters; and hence (viii) the "lines of command": who does what, and who reports to whom, administratively and functionally .*

    We refer to [21, Sect. 8.7].

94. **Human Behaviour [Domain Facet]:**
    - *By domain human behaviour we shall understand any of a quality spectrum of carrying out assigned work: from (i) careful, diligent and accurate, via (ii) sloppy dispatch, and (iii) delinquent work, to (iv) outright criminal pursuit .*

    We refer to [21, Sect. 8.8]

95. **Finite State Machine/Automaton:** See [ι 16,π 15].

96. **Fluid:** A substance that has no fixed shape and yields easily to external pressure; a gas or (especially) a liquid [Oxford Languages and Google]. We include plasmas as fluids.

97. **Formal Method:** By a *formal method* we shall here understand a *method* [ι 151,π 42] whose techniques and tools can be understood mathematically [ι 144,π 41].

    For formal *domain, requirements* and *software engineering* formality means the following:

    - There is a set, one or more, **specification languages** – say for domain descriptions, requirements prescriptions, software specifications, and software coding, i.e., programming languages.[14]

[14] Most formal specification languages are textual, but graphical languages like `Petri nets [95], Message Sequence Charts[56], Statecharts [50], Live Sequence Charts [51]`, etc., are also formal.

- These are all to be formal, that is, to have a formal syntax [ι 227,π 60], a formal semantics [ι 205,π 55], and a formal, typically *Mathematical Logic* [ι 143,π 41] proof system.
- Some of the *techniques* [ι 234,π 62] and *tools* [ι 238,π 67] must be supported by a mathematical understanding.

98. **FORTRAN:** A programming language [76].

99. **Fluid Endurant:** By a *fluid endurant* we shall understand an endurant which is prolonged, without interruption, in an unbroken series or pattern; or, rephrasing: a substance (liquid, gas or plasma) having the property of flowing, consisting of particles that move among themselves [74, Vol. I, pg. 774] .

100. **Fraenkel, Abraham:** (Hebrew: February, 1891 – 15 October, 1965) was a German-born Israeli mathematician. He was an early Zionist and the first Dean of Mathematics at the Hebrew University of Jerusalem. He is known for his contributions to axiomatic set theory, especially his additions to Ernst Zermelo's axioms, which resulted in the Zermelo–Fraenkel set theory.

    [https://en.wikipedia.org/wiki/Abraham_Fraenkel]

101. **Frege, Gottlob:** (1848-1925) A mathematician who put mathematics on a new and more solid foundation. He purged mathematics of mistaken, sloppy reasoning and the influence of Pythagoras. Mathematics was shown to be a subdivision of formal logic.

102. **Function:** A function is a mathematical object which, when *'applied'* to a [mathematical] value, called its **argument**, potentially yields a [mathematical] value, called its **result**. If it fails to yield a result, the function is said to be undefined for that argument, and is then called a **partial function**, otherwise, if it yields results for any argumenent it is said to be a **total function**.

    We can understand functions in terms of sets [ι 207,π 56]. A function can then be modeled as a set, f, of pairs: $(a, b) : (A \times B)$ such that no two pairs: $(a_i, b_i), (a_j, b_j)$ in f have $a_i = a_j$.

    We refer to the Lambda Calculus [ι 127,π 38].

103. **Functional Programming:** A programming paradigm where programs are constructed by applying and composing functions. It is a declarative programming paradigm in which function definitions are trees of expressions that map values to other values, rather than a sequence of imperative statements which update the running state of the program.

    [https://en.wikipedia.org/wiki/Functional_programming]

2.7 G...G...G...G...G...G...

104. **Gedanken:** We refer to John Reynolds, 1970. "GEDANKEN - A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept", J.C. Reynolds, CACM 13(5):308-319 (May 1970).

105. **Generic Programming:** A programming technique which aims to make programs more adaptable by making them more general. Generic programs often embody non-traditional kinds of polymorphism; ordinary programs are obtained from them by suitably instantiating their parameters. In contrast with normal programs, the parameters of a generic programs are often quite rich in structure. For example they may be other programs, types or type constructors or even programming paradigms [https://www.computer-dictionary-online.org/definitions-g/generic-programming].

106. **Grammar:** A formal definition of the syntactic structure (the syntax [ι 227,π 60]) of a language.

    A grammar is normally represented as a set of production rules which specify the order of constituents and their sub-constituents in a sentence (a well-formed string in the language). Each rule has a left-hand side symbol naming a syntactic category (e.g. "noun-phrase" for a natural language grammar) and a right-hand side which is a sequence of zero or more symbols. Each symbol may be either a terminal symbol or a non-terminal symbol. A terminal symbol corresponds to one "lexeme" - a part of the sentence with no internal syntactic structure (e.g. an identifier or an operator in a computer language). A non-terminal symbol is the left-hand side of some rule.

    One rule is normally designated as the top-level rule which gives the structure for a whole sentence.

    A parser (a kind of recogniser) uses a grammar to parse a sentence, assigning a terminal syntactic category to each input token and a non-terminal category to each appropriate group of tokens, up to the level of the whole sentence. Parsing is usually preceded by lexical analysis. The opposite, generation, starts from the top-level rule and chooses one alternative production wherever there is a choice.

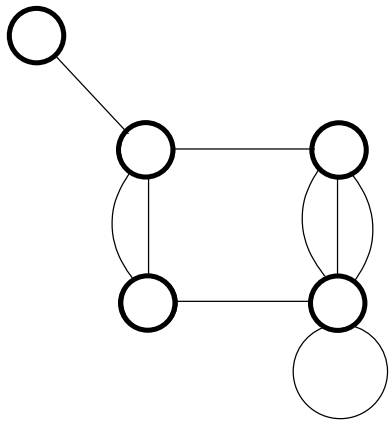
    In computing, a formal grammar, e.g. in BNF, can be used to parse a linear input stream, such as the source code of a program, into a data structure that expresses the (or a) meaning of the input in a form that is easier for the computer to work with. A compiler compiler like yacc might be used to convert a grammar into code for the parser of a compiler. A grammar might also be used by a transducer, a translator or a syntax directed editor. [`https://www.computer--dictionary-online.org/definitions-g/grammar`]

107. **Graph:** A graph can textually be understood as a pair of a set of nodes and a set of edges, (ns : N-**set**,es:E-**set**), such that for every edge e there is a pair of nodes (n_e, n_i) [where $\{n_e, n_i\} \subseteq ns$] such that e (in es) emanates from n_e and is incident upon n_i – where n_e=n_i means a loop !

A Mathematics Example

Example: 13 *Graphs may also be rendered "graph"ically (!):*

We refrain from illustrating the labeling of nodes and edges, as well as showing arrows on edges.



108. **Graph Theory:** In mathematics and computer science, graph theory is the study of graphs, which are mathematical structures used to model pairwise relations between objects. A graph in this context is made up of vertices (also called nodes or points) which are connected by edges (also called arcs, links, or lines). A distinction is made between undirected graphs, where edges link two vertices symmetrically, and directed graphs, where edges link two vertices asymmetrically. Graphs are one of the principal objects of study in discrete mathematics.

    [https://en.wikipedia.org/wiki/Graph_theory]

109. **Gödel, Kurt Friedrich:** (Born in Brno, Austria/Hunagrian empire, now Moravia, the Czech Republic: April 28, 1906 –January 14, 1978) was a logician, mathematician, and philosopher. Considered along with Aristotle and Gottlob Frege to be one of the most significant logicians in history, Gödel profoundly influenced scientific and philosophical thinking in the 20th century (at a time when Bertrand Russell, Alfred North Whitehead, and David Hilbert were using logic and set theory to investigate the foundations of mathematics), building on earlier work by Frege, Richard Dedekind, and Georg Cantor.

    [https://en.wikipedia.org/wiki/Kurt_Gödel]

2.8 H...H...H...H...H...H...

110. **Hardware:** The physical components of a computer: electronics, mechanics, optics, etc.

     [Wikipedia]

111. **Hoare, Charles Anthony Richard:** (11 January 1934 –5 March 2026), also known as C. A. R. Hoare, was a British computer scientist who made foundational contributions to programming languages, algorithms, operating systems, formal verification, and concurrent computing. His work earned him the 1980 ACM Turing Award, usually regarded as the highest distinction in computer science.

     Hoare developed the sorting algorithm quicksort in 1959–1960. He developed Hoare logic, an axiomatic basis for verifying program correctness. In the semantics of concurrency, he introduced the formal language communicating sequential processes (CSP) to specify the interactions of concurrent processes, and along with Edsger Dijkstra, formulated the dining philosophers problem. From 1977 on, he held positions at the University of Oxford as well as at Microsoft Research in Cambridge, UK.

     [https://en.wikipedia.org/wiki/Tony_Hoare]

2.9 I...I...I...I...I...I...

112. **Identity:** It is a fact, that is, an absolutely necessary condition for our description of any world, that its entities [ι 112,π 34] have unique identity. It is, however a problem in our domain analysis & description, to secure such identity; so we must, wherever necessary present **axioms** expressing so.

113. **Imperative Programming:** A programming paradigm of software that uses statements that change a process' state. In much the same way that the imperative mood in natural languages expresses commands, an imperative program consists of commands for the computer to perform. Imperative programming focuses on describing how a program operates step by step (with general order of the steps being determined in source code by the placement of statements one below the other), rather than on high-level descriptions of its expected results [`https://en.wikipedia.org/wiki/Imperative_programming`].

114. **Informatics:** The confluence of the computer and computing sciences with discrete mathematics, in particular mathematical logic and set theory, and with some aspects of philosophy [ι 175,π 49].

    To us Informatics, in contrast to IT [ι 122,π 35], is a universe of intellectual quality.

115. **Initialisation, Domain Analysis & Description:** The declaration and initialisation of a number of "dashboard" variables in preparation for domain analysis & description.

116. **Instantiation, Domain:** The invocation of one or more [part [ι 169,π 48]] behaviours [ι 20,π 17].

117. **Intentional Pull:** The concept of intentional pull is a wider notion than that of invariant. Here we are not concerned with pre-/post-conditions on operations. Intentional pull is exerted between two or more phenomena of a domain when their relation can be asserted to **always** (□) hold.

Intentional Pulls

Example: 14 *Cf. Method Sect. 3.7.4 on page 89 and Example Sect. 4.2.2.4 on page 100.*

118. **Internal Non-determinism:** Internal non-determinism refers to computational processes [that is: also domain [ι 20,π 17] behaviours] where the path of execution [i.e., domain actions [ι 4,π 12]] is not fully determined (by the initial specification), allowing the same input [state] to produce different outcomes, with the source of this unpredictability arising from within the system itself (such as scheduling order, memory management, or internal state changes). This contrasts with external non-determinism, which stems from external factors like unpredictable user input or network delays.

    [AI]

119. **Internal Qualities:** In the context of 𝔻𝕆𝕄𝔸𝕀ℕ internal qualities are the properties of endurants – such as unique identifiers [ι 249,π 70], mereology [ι 149,π 42], and attributes [ι 14,π 14] [DB].

Internal Qualities

Example: 15 *Cf. Method Sect. 3.7 on page 85 and Example Sect. 4.2.2 on page 96.*

120. **Invariants:** The concept of invariants in the context of computing science is most clearly illustrated in connection with the well-formedness of data structures. Invariants then express properties that must hold, i.e., as a **pre-condition**, before any application of an operation to those data structures and shall hold, i.e. as a **post-condition** after any application of an operation to those data structures.

121. **ISWIM:** [If you See What I Mean] is an abstract computer programming language (or a family of languages) devised by Peter Landin [ι 128,π 38] and first described in his article "The Next 700 Programming Languages", published in the Communications of the ACM in 1966 [73].

    Although not implemented, it has proved very influential in the development of programming languages, especially functional programming languages such as ... ML, Haskell and their successors, and data-flow programming languages like Lucid. Design

    ISWIM is an imperative programming language with a functional core, consisting of a syntactic sugaring of lambda calculus to which are added mutable variables and assignment and a powerful control mechanism: the program point operator. Being based on lambda calculus, ISWIM has higher-order functions and lexically scoped variables.

    The operational semantics of ISWIM are defined using Landin's SECD [ι 204,π 55] machine and use call-by-value, that is eager evaluation. A goal of ISWIM was to look more like mathematical notation, so Landin abandoned ALGOL's semicolons between statements and begin ... end blocks and replaced them with the off-side rule and scoping based on indentation.

122. **IT:** Information Technology is the computer hardware – the electronics, printers, data communication equipment, display screens, etc.: tablets, cell phones and more.

    It is a universe of material quantity: smaller size, faster speed, larger memories, lower cost, ... – in contrast to Informatics [ι 114,π 34].

2.10 J...J...J...J...J...J...

123. **Jackson, Michael A.:** (born 16 February 1936) is a British computer scientist, and independent computing consultant in London, England. He is also a visiting research professor at the Open University in the UK.

     [https://en.wikipedia.org/wiki/Michael_A._Jackson_(computer_scientist)]

     Michael A. Jackson's work, "epitomised" in [59, 60], has had a significant impact: His style of presenting software development [58, 62, 61] and his lexicon– rigorously, but not using formalisms, is remarkable.

2.11 K...K...K...K...K...K...

124. **Kant, Immanuel** (1724–1804) is the central figure in modern philosophy. He synthesized early modern rationalism and empiricism, set the terms for much of nineteenth and twentieth century philosophy, and continues to exercise a significant influence today in metaphysics, epistemology, ethics, political philosophy, aesthetics, and other fields. The fundamental idea of Kant's *critical philosophy* – especially in his three Critiques: the Critique of Pure Reason (1781, 1787), the Critique of Practical Reason (1788), and the Critique of the Power of Judgment (1790) –is human autonomy. He argues that the human understanding is the source of the general laws of nature that structure all our experience; and that human reason gives itself the moral law, which is our basis for belief in God, freedom, and immortality. Therefore, scientific knowledge, morality, and religious belief are mutually consistent and secure because they all rest on the same foundation of human autonomy, which is also the final end of nature according to the teleological worldview of reflecting judgment that Kant introduces to unify the theoretical and practical parts of his philosophical system.

    [https://plato.stanford.edu/entries/kant/]

125. **Kleene, Stephen Cole:** An American mathematician and logician [https://en.wikipedia.org/wiki/Stephen_Cole_Kleene] who, with Alonzo Church [ι 35,π 20], established the fields of mathematical logic, meta-mathematics [ι 147,π 41], including recursive function theory [ι 185,π 51] and computablility [ι 39,π 20] [64, 65].

2.12 L...L...L...L...L...L...

126. **Language:** By *language* we shall, with [Wikipedia], mean a [possibly *structured*] *system* of communication between humans (sometimes via an intermediate medium: text, graphics, music, art, etc.).

The 'structured system' that we refer to has come to be known as *Syntax* [ι 227,π 60], *Semantics* [ι 205,π 55] and *Pragmatics* [ι 176,π 49].

There are many "kinds" of languages[15]:

- textual:
  - human spoken and written
  - theatre plays
  - operas
  - movies
  - etc.
  - formal languages [mathematics]
- graphic:
  - Computer Languages:
    - LSC [ι 133,π 39]
    - MSC [ι 150,π 42]
    - Petri Nets [ι 173,π 48]
    - State Charts [ι 224,π 60]
    - etc.
  - Ballet Notation[16]
- Music
  - as played and heard
  - as a notation[17]
- Art
  - painting
  - sculpture
  - etc.
- etc., etc. !

127. **Lambda Calculus:** In mathematical logic, the lambda calculus (also written as λ-calculus) is a formal system for expressing computation based on function abstraction and application using variable binding and substitution.

128. **Landin, Peter Johan:** (5 June 1930 —–3 June 2009) was a British computer scientist. He was one of the first to realise that the lambda calculus [ι 127,π 38] could be used to model a programming language, an insight that is essential to the development of both functional programming and denotational semantics.

[67, 66, 67, 70, 68, 69, 73, 72, 71]

[https://en.wikipedia.org/wiki/Peter_Landin]

129. **Linguistics:** By *linguistics* we shall mean the scientific study of human spoken and written [text] languages: semiotics [ι 206,π 55], syntax [ι 227,π 60], semantics [ι 205,π 55], and pragmatics [ι 176,π 49].

[15] The above enumeration is not quite well-structured. Need be edited.

[16] Dance notation is the symbolic representation of human dance movement and form, using methods such as graphic symbols and figures, path mapping, numerical systems, and letter and word notations. Several dance notation systems have been invented, many of which are designed to document specific types of dance while others have been developed with capturing the broader spectrum of human movement potential. A dance score is a recorded dance notation that describes a particular dance.

One such is *Laban Notation*: [https://en.wikipedia.org/wiki/Labanotation], another is *Benesh Movement Notation*: [https://en.wikipedia.org/wiki/Benesh_Movement_Notation].

[17] https://en.wikipedia.org/wiki/Musical_notation

130. **List:** By *list* we shall mean a finite, or possibly infinite sequence, i.e., an ordered set of elements[18].

131. **List Comprehension:** Let `TE` be a type name or type expression, `i, li, ui` be natural numbers and $\mathcal{B}$`(e(i))` be a predicate expression then

    $\langle$ e(i) | e:TE,i:**Nat** • li≤i≤ui ∧ $\mathcal{B}$(e(i)) $\rangle$

    is a list comprehension: the natural numer index ordered sequence of all `e(i)` in `TE` that satisfies predicate $\mathcal{B}$`(e(i))`.

132. **List Enumeration:** A List [ι 130,π 39] enumeration of n elements, expressed by expressions $e_1, e_2, ..., e_n$, for $n > 0$, is:

    $\langle e_1, e_2, ..., e_n \rangle$

    where, of course, the elipses, ..., must be filled in with proper expressions.Cf. [ι 32,π 19], -[ι 141,π 40] and [ι 209,π 56].

133. **Live Sequence Charts:** David Harel suggested that MSC [ι 150,π 42] had shortcomings such as:

    (i) MSC propose a weak partial ordering semantics that makes it impossible to capture some behavioral requirements,

    (ii) The relationship between the MSC requirements and the executable specification is not clear,

    and proposed Live Sequence Charts (LSC) as an extension on the MSC standard.

    [113]

    `[https://link.springer.com/chapter/10.1007/978-3-540-27863-4_21]`.

134. **Living Species:** – are such as *plants* and *animals* – including *humans*.

[18] In mathematics, a sequence is an enumerated collection of objects in which repetitions are allowed and order matters. Like a set, it contains members (also called elements, or terms). The number of elements (possibly infinite) is called the length of the sequence. Unlike a set, the same elements can appear multiple times at different positions in a sequence, and unlike a set, the order does matter. Formally, a sequence can be defined as a function from natural numbers (the positions of elements in the sequence) to the elements at each position. The notion of a sequence can be generalized to an indexed family, defined as a function from an arbitrary index set. `[Wikipedia]`

2.13 M...M...M...M...M...M...

135. **Machine, Finite State:** See [ι 16,π 15].

136. **Machine:** By a, or rather, the *machine* we shall understand a combination of software and hardware.

137. **Machine Requirements:** are those requirements which can be expressed sôlely in terms of machine concepts.

138. **Manifest Part:** By a manifest part we shall understand a part which 'manifests' itself in a physical, visible manner, "occupying" an AREA or a VOLUME and a POSITION in SPACE ! A manifest part is not conceptual [ι 49,π 21].

139. **Map:** By a map we shall understand a effectively enumerable function:

$$[a_1 \mapsto b_1, a_2 \mapsto b_2, ..., a_n \mapsto b_n, ...],$$

where a_i, b_i are map domain[19], **dom**, definition set, respectvely map range, **rng**, set value (expression)s.

140. **Map Comprehension:** Let `d(i), r(i)` be expresions of **type** `D`, respectiely **type** `R`, and $\mathcal{B}$`(d(i),r(i))`, then:

$$[\ \mathsf{d(i)} \mapsto \mathsf{r(i)} \mid \mathsf{d(i)}{:}\mathsf{D},\ \mathsf{r(i)}{:}\mathsf{R} \bullet \mathcal{B}(\mathsf{d(i)},\mathsf{r(i)})\ ]$$

is a map comprehension expression which denotes the map from definition set values, `d(i)`, to map range set value, `r(i)`, for which $\mathcal{B}$`(d(i),r(i))` holds.

141. **Map Enumeration:** A map [ι 139,π 40] enumeration of n map elements, expressed by expressions $a_1 \mapsto b_1, a_2 \mapsto b_2, ..., a_n \mapsto b_n$, for $n \geq 0$, is:

$$[\ a_1 \mapsto b_1, a_2 \mapsto b_2, ..., a_n \mapsto b_n\ ]$$

where, of course, the elipses, ..., must be filled in with proper expressions. Cf. [ι 32,π 19], [ι 132,π 39] and [ι 209,π 56].

142. **Map Operation:** These are the map operations

- `m_a` ∪ `m_b` map union[20],
- `m_a` † `m_b` map overide[21],
- `m`\`s` map restriction[22],
- `m/s` map restriction[23],
- **dom** `m` map definition set[24],
- **rng** `m` map range set[25], and
- `m(x)` application[26].

[19] Beware of the possible confusion of names: Domain as in [ι 63,π 24] and **dom**ain - as here !
[20] **dom** `m_a` and **dom** `m_b` do not overlap.
[21] **dom** `m_a` and **dom** `m_b` may overlap: resulting map "obeys" the overlap mapping.
[22]
[23]
[24]
[25]
[26]

143. **Mathematical Logic:** Mathematical logic is the study of formal logic within mathematics. Major subareas include model theory, proof theory, set theory, and recursion theory (also known as computability theory). Research in mathematical logic commonly addresses the mathematical properties of formal systems of logic such as their expressive or deductive power. However, it can also include uses of logic to characterize correct mathematical reasoning or to establish foundations of mathematics.

    [https://en.wikipedia.org/wiki/Mathematical_logic]

144. **Mathematics:** By *mathematics* we shall here understand a such human endeavours that makes precise certain facets of language, whether natural or 'constructed' (as for mathematical notation), and out of those endeavours, i.e., mathematical constructions, also called theories, build further abstractions.

145. **Matter:** Space, in the sense of $\mathbb{SPACE}$, is "inhabited" ! The inhabitants are entities. They possess properties (attributes [ι 14,π 14]) about which we reason We shall take the view that these entities are of $\mathbb{MATTER}$. *Matter is anything that has mass and takes up space.*

146. **Mathematical Proof:** A mathematical proof is a deductive argument for a mathematical statement, showing that the stated assumptions logically guarantee the conclusion. The argument may use other previously established statements, such as theorems; but every proof can, in principle, be constructed using only certain basic or original assumptions known as axioms,] along with the accepted rules of inference. Proofs are examples of exhaustive deductive reasoning that establish logical certainty, to be distinguished from empirical arguments or non-exhaustive inductive reasoning that establish "reasonable expectation". Presenting many cases in which the statement holds is not enough for a proof, which must demonstrate that the statement is true in all possible cases. A proposition that has not been proved but is believed to be true is known as a conjecture, or a hypothesis if frequently used as an assumption for further mathematical work.

    Proofs employ logic expressed in mathematical symbols, along with natural language that usually admits some ambiguity. In most mathematical literature, proofs are written in terms of rigorous informal logic. Purely formal proofs, written fully in symbolic language without the involvement of natural language, are considered in proof theory. The distinction between formal and informal proofs has led to much examination of current and historical mathematical practice, quasi-empiricism in mathematics, and so-called folk mathematics, oral traditions in the mainstream mathematical community or in other cultures. The philosophy of mathematics is concerned with the role of language and logic in proofs, and mathematics as a language.

    [https://en.wikipedia.org/wiki/Mathematical_proof]

    See also [ι 237,π 62].

147. **Metamathematics:** is the study of mathematics itself using mathematical methods. This study produces metatheories, which are mathematical theories about other mathematical theories.

148. **Metaphysics:** is the branch of philosophy that examines the fundamental nature of reality, including the relationship between mind and matter, between substance and attribute, and between potentiality and actuality [75] [Wikipedia].

    One may claim that this paper is [also] about metaphysics.

149. **Mereology:** is the theory of part-hood relations: of the relations of part to whole and the relations of part to part within a whole [116, 103, 39].

    The concept of 'mereology' and its study is accredited to the Polish mathematician, philosopher and logician Stanisław Leśniewski (1886–1939).

Mereology

Example: 16 *Cf. Method Sect. 3.7.2 on page 87 and Example Sect. 4.2.2.2 on page 98.*

150. **Message Sequence Charts:** A message sequence chart (or MSC) is an interaction diagram from the SDL[27] family standardized by the International Telecommunication Union[28].

    The purpose of recommending MSC (Message Sequence Chart) is to provide a trace language for the specification and description of the communication behaviour of system components and their environment by means of message interchange. ...In MSCs communication behaviour is presented in a very intuitive and transparent manner, particularly in the graphical representation. The MSC language is easy to learn, use and interpret. In connection with other languages it can be used to support methodologies for system specification, design, simulation, testing, and documentation.

    `[https://en.wikipedia.org/wiki/Message_sequence_chart]`

151. **Method:** By a method we shall understand a set of **principles** [ι 177,π 49] and **procedures** [ι 179,π 49] for *selecting* and *applying* a set of **techniques** [ι 234,π 62] *using* a set of **tools** [ι 238,π 67] in order to *construct* an *artefact* [ι 12,π 13] [DB].

152. **Methodology:** is the comparative study and knowledge of methods [DB].

    [The two terms: 'method' and 'methodology' are often confused, including used interchangeably.]

153. **ML:** ML (Meta Language) is the metalanguage developed for the Edinburgh LCF theorem prover in the 1970s. It is an early statically typed, functional language with polymorphic type inference in the Hindley–Milner style, and other features like exceptions and mutable variables. ML's design in LCF directly inspired the later ML family (notably Standard ML, Caml, and their derivatives) and influenced subsequent functional language development.

    `[https://en.wikipedia.org/wiki/ML_(programming_language)]`

154. **Model:** A mathematical model is a description of a system using mathematical concepts and language. We shall include descriptions[29], prescriptions[30] and specifications[31] using formal languages in presenting models.

155. **Model Checking:** In computer science, model checking or property checking is a method for checking whether a finite-state model of a system meets a given specification (also known

[27] https://en.wikipedia.org/wiki/Specification_and_Description_Language
[28] https://en.wikipedia.org/wiki/ITU-T
[29] – as for domains
[30] – as for requirements
[31] – as for software

as correctness). This is typically associated with hardware or software systems, where the specification contains liveness requirements (such as avoidance of livelock) as well as safety requirements (such as avoidance of states representing a system crash).

[https://en.wikipedia.org/wiki/Model_checking]

156. **Modeling:** Modelling is the act of creating models, which include discrete mathematical structures (sets, Cartesians, lists, maps, etc.), and are logical theories represented as algebras. That is, any given RSL text denotes a set of models, and each model is an algebra, i.e., a set of named values and a set of named operations on these. Modelling is the engineering activity of establishing, analyzing and using such structures and theories. Our models are established with the intention that they "model" "something else" other than just being the mathematical structure or theory itself. That "something else" is, in our case, some part of a reality[32], or of a construed such reality, or of requirements to the, or a reality[33], or of actual software[34].

157. **Model Theory:** In mathematical logic, model theory is the study of the relationship between formal theories (a collection of sentences in a formal language expressing statements about a mathematical structure), and their models (those structures in which the statements of the theory hold). The aspects investigated include the number and size of models of a theory, the relationship of different models to each other, and their interaction with the formal language itself. In particular, model theorists also investigate the sets that can be defined in a model of a theory, and the relationship of such definable sets to each other. As a separate discipline, model theory goes back to Alfred Tarski, who first used the term "Theory of Models" in publication in 1954.[Since the 1970s, the subject has been shaped decisively by Saharon Shelah's stability theory.

[https://en.wikipedia.org/wiki/Model_theory]

[32] — as in domain modelling
[33] — as in requirements modelling
[34] — as in software design

2.14 N...N...N...N...N...N...

158. **Narration & Formalisation:** To communicate what a domain "is", one must be able to narrate of what it consists. To understand a domain one must give a formal description of that domain. When we put an ampersand, &, between the two terms we mean to say that they form a whole: not one without the other, either way around! In our domain descriptions we enumerate narrative sentences and ascribe this enumeration to formal expressions.

159. **Naur, Peter:** (25 October 1928 –3 January 2016) was a Danish computer science pioneer and 2005 Turing Award winner. He is best remembered as a contributor, with John Backus, to the Backus–Naur form (BNF [ι 25,π 17]) notation used in describing the syntax for most programming languages. He also contributed to creating the language ALGOL 60 [ι 2.1,π 13].

160. **Nondeterminism:** is a property of behaviours [ι 20,π 17] that evolve in time, in which complete information about the internal state of the system at some point in time admits multiple future trajectories.

161. **Nondeterministic Algorithm:** In computer science and computer programming, a nondeterministic algorithm is an algorithm that, even for the same input, can exhibit different behaviors on different runs, as opposed to a deterministic algorithm.

162. **Number:** A number is a mathematical object used to count, measure, and label. The most basic examples are the natural numbers: $1, 2, 3, 4, 5$, and so forth. Individual numbers can be represented in language with number words or by dedicated symbols called numerals; for example, "five" is a number word and "5" is the corresponding numeral. As only a limited list of symbols can be memorized, a numeral system is used to represent any number in an organized way. The most common representation is the Hindu–Arabic numeral system, which can display any non-negative integer using a combination of ten symbols, called numerical digits. Numerals can be used for counting (as with cardinal number of a collection or set), labels (as with telephone numbers), for ordering (as with serial numbers), and for codes (as with ISBNs). In common usage, a numeral is not clearly distinguished from the number that it represents.

    In mathematics, the notion of number has been extended over the centuries to include zero (0), negative numbers, rational numbers such as one half ($\frac{1}{2}$), real numbers such as the square root of 2, i.e., $\sqrt{2}$, and complex numbers which extend the real numbers with a square root of $\sqrt{-1}$, and its combinations with real numbers by adding or subtracting its multiples. Calculations with numbers are done with arithmetical operations, the most familiar being addition, subtraction, multiplication, division, and exponentiation. Their study or usage is called arithmetic, a term which may also refer to number theory, the study of the properties of numbers.

    Viewing the concept of zero as a number required a fundamental shift in philosophy, identifying nothingness with a value. During the 19th century, mathematicians began to develop the various systems now called algebraic structures, which share certain properties of numbers, and may be seen as extending the concept. Some algebraic structures are explicitly referred to as numbers (such as the p-adic numbers and hypercomplex numbers) while others are not, but this is more a matter of convention than a mathematical distinction.

    `[https://en.wikipedia.org/wiki/Number]`

163. **Number Theory:** is a branch of pure mathematics devoted primarily to the study of the integers and arithmetic functions. Number theorists study prime numbers as well as the properties of mathematical objects constructed from integers (for example, rational numbers), or defined as generalizations of the integers (for example, algebraic integers).

    Integers can be considered either in themselves or as solutions to equations (Diophantine geometry). Questions in number theory can often be understood through the study of analytical objects, such as the Riemann zeta function, that encode properties of the integers, primes or other number-theoretic objects in some fashion (analytic number theory). One may also study real numbers in relation to rational numbers, as for instance how irrational numbers can be approximated by fractions (Diophantine approximation).

    Number theory is one of the oldest branches of mathematics alongside geometry. One quirk of number theory is that it deals with statements that are simple to understand but are very difficult to solve. Examples of this are Fermat's Last Theorem, which was proved 358 years after the original formulation, and Goldbach's conjecture, which remains unsolved since the 18th century. German mathematician Carl Friedrich Gauss (1777–1855) once remarked, "Mathematics is the queen of the sciences – and number theory is the queen of mathematics." It was regarded as the epitome of pure mathematics, with no applications outside mathematics, until the 1970s, when prime numbers became the basis for the creation of public-key cryptography algorithms.

    [49] and [https://en.wikipedia.org/wiki/Number_theory].

2.15 O...O...O...O...O...O...

164. **Object:** [see next two entries] To us, in the context of 𝔻𝕆𝕄𝔸𝕀ℕ, an object, although we do not use that term, is the pair of a behavioural part (with its unique identifier, mereology and attributes) and its corresponding behaviour [DB].

165. **Object Oriented:** [see previous and next entries] To us, in the context of 𝔻𝕆𝕄𝔸𝕀ℕ, object orientedness is expessed in our insistance of modelling entities [ι 79,π 26] as objects in the sense that we do in [ι 164,π 46] [DB].

166. **Object Oriented Programming:** (OOP) is a programming paradigm based on objects – software entities that encapsulate data and function(s). An OOP computer program consists of objects that interact with one another. An OOP language is one that provides object-oriented programming features, but as the set of features that contribute to OOP is contested, classifying a language as OOP –and the degree to which it supports OOP – is debatable. As paradigms are not mutually exclusive, a language can be multi-paradigm (i.e. categorized as more than only OOP).

    To us `Simula 67` was the first OOP language [5] – masterminded by Ole-Johan Dahl [ι 54,π 23].

    And, to us, 𝔻𝕆𝕄𝔸𝕀ℕ is object oriented [DB] !

    `[https://en.wikipedia.org/wiki/Object-oriented_programming]`

167. **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 [36, 37] `[Wikipedia]`. An ontology identifies and distinguishes concepts and their relationships; it describes content and relationships [Bob Bater]. Ontologies *specify*.

    The two terms, taxonomy and ontology relate. We refer to the term 'taxonomy', Item 233 on page 62.

168. **Operational Semantics:** [See [ι 205,π 55].] is a category of formal programming language semantics in which certain desired properties of a program, such as correctness, safety or security, are verified by constructing proofs from logical statements about its execution and procedures, rather than by attaching mathematical meanings to its terms (denotational semantics). Operational semantics are classified in two categories: structural operational semantics (or small-step semantics) formally describe how the individual steps of a computation take place in a computer-based system; by opposition natural semantics (or big-step semantics) describe how the overall results of the executions are obtained. Other approaches to providing a formal semantics of programming languages include axiomatic semantics and denotational semantics.

    The operational semantics for a programming language describes how a valid program is interpreted as sequences of computational steps. These sequences then are the meaning of the program. In the context of functional programming, the final step in a terminating sequence returns the value of the program. (In general there can be many return values for a single program, because the program could be nondeterministic, and even for a deterministic program there can be many computation sequences since the semantics may not specify exactly what sequence of operations arrives at that value.)

Perhaps the first formal incarnation of operational semantics was the use of the lambda calculus to define the semantics of Lisp. Abstract machines in the tradition of the SECD [ι 204,π 55] machine are also closely related.

[https://en.wikipedia.org/wiki/Operational_semantics]

2.16 P...P...P...P...P...P...

169. **Part:** By a *part* we shall understand a solid endurant existing in time and subject to laws of physics, including the *causality principle* and *gravitational pull*[35].

Road Net Parts

Example: 17 *Cf. Method Sect. 3.6.1.1 on page 80 and Example Sect. 4.2.1.1 on page 95.*

170. **Part Set:** A compound part [ι 38,π 20] is a part set, same as a set part, [ι 211,π 56], consists of a [finite or infinite] set of parts.

171. **Perdurant:** 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*.

172. **Petri, Carl Adam:** (12 July 1926 in Leipzig –2 July 2010 in Siegburg)] was a German mathematician and computer scientist.

    Petri created his major scientific contribution, the concept of the Petri net, in 1939 at the age of 13, for the purpose of describing chemical processes. In 1941, his father told him about Konrad Zuse's work on computing machines and Carl Adam started building his own analog computer.

    After earning his Abitur at Thomasschule in 1944, he was drafted into the Wehrmacht. He was taken into British captivity until 1949, when he departed England.

    Petri started studying mathematics at the Technische Hochschule Hannover (today, the Leibniz University Hannover) in 1950. He documented Petri nets in 1962 as part of his dissertation, *Kommunikation mit Automaten (Communication with automata).* From 1959 until 1962 he worked at the University of Bonn and received his PhD degree in 1962 from the Technische Universität Darmstadt. From 1963 to 1968 he established and directed the computing centre of Bonn University. In 1968, he became head of Forschungsinstitut für Informationssysteme of the newly founded Gesellschaft für Mathematik und Datenverarbeitung (GMD). He retired in 1991.

173. **Petri Net:** A Petri net, also known as a place/transition net, is one of several mathematical modeling languages for the description of distributed systems. It is a class of discrete event dynamic system. A Petri net is a directed bipartite graph that has two types of elements: places and transitions. Place elements can be depicted as white circles and transition elements can be depicted as rectangles. A place can contain any number of tokens, depicted, say, as black circles. A transition is enabled if all places connected to it as inputs contain at least one token. ... Petri nets offer a graphical notation for stepwise processes that include choice, iteration, and concurrent execution. Unlike these standards, Petri nets have an exact mathematical definition of their execution semantics, with a well-developed mathematical theory for process analysis.

    `[https://en.wikipedia.org/wiki/Petri_net]`

    Seminal textbooks/monographs are [92, 93, 94, 95].

174. **Phenomenon:** By a phenomenon we shall understand a fact that is observed to exist or happen .

    Some phenomena are rationally describable – to a large or full degree – others are not.

175. **Philosophy:** is the study of general and fundamental questions about existence, knowledge, values, reason, mind, and language. Such questions are often posed as problems to be studied or resolved [Wikipedia].

    We refer to [ι 232,π 61].

176. **Pragmatics:** studies the ways in which context contributes to meaning.

    Pragmatics encompasses speech act theory, conversational implicature, talk in interaction and other approaches to language behaviour in philosophy, sociology, linguistics and anthropology [80, 79] [Wikipedia].

- **Primitive Recursive Functions:** see [ι 43,π 20].

177. **Principle:** By a *principle* we shall, loosely, understand *(i) elemental aspect of a craft or discipline, (ii) foundation, (iii) general law of nature, etc* [www.etymonline.com].

    The 𝔻𝕆𝕄𝔸𝕀ℕ method evolves around ***principles***, *procedures*, *techniques* and *tools*.

Principle

Example: 18 *The principles of the 𝔻𝕆𝕄𝔸𝕀ℕ method include abstraction, mathematics, formal specification, narrative & formalisation, enumeration and indexing.*

178. **Proactive Action:** A proactive action is an action [ι 4,π 12] performed by – occurring in – a behaviour [ι 20,π 17] at is "own free will"! That is: it is not in response to [caused by, the result of] a prior action.

179. **Procedure:** A procedure instructs on how to execute one or more activities of a sequential (stages and step) process (task). It describes the sequence of these stages and steps, and specifies for each stage and step what needs to be done, often including when the procedure should be executed and by whom.

    [https://en.wikipedia.org/wiki/Procedure_(business)]

    The 𝔻𝕆𝕄𝔸𝕀ℕ method evolves around *principles*, ***procedures***, *techniques* and *tools*.

Procedure

Example: 19 *There is a man procedure of the 𝔻𝕆𝕄𝔸𝕀ℕ method. It is elaborated in Sect. 3 and centers around the 𝔻𝕆𝕄𝔸𝕀ℕ procedure Sects. 3.2–3.8 (pages 76–94).*

180. **Program:** a series of coded software instructions to control the operation of a computer or other machine.

181. **Proof:** evidence or argument establishing a fact or the truth of a statement.

182. **Proof Systems:** are built to prove statements. They can be thought as an inference machine with special statements, called provable statements, or sometimes theorems being its final products. The starting points are called axioms [ι 15,π 14] of the system.

2.17 Q...Q...Q...Q...Q...Q...

2.18 R...R...R...R...R...R...

183. **RAISE:** An abstract, model-oriented **R**igorous **A**pproach to **I**ndustrial **S**oftware **E**ngineering [47]. See [ι 201,π 54].

     The RAISE method ("Rigorous Approach to Industrial Software Engineering") is a formal method for software development, comprising the RAISE Specification Language (RSL), a development method, and supporting tools. It enables the step-wise, formal development of complex sequential and concurrent software from abstract specifications to implementations.

     Key Aspects of the RAISE Method:

     - **Purpose:** It is designed for industrial use, allowing developers to manage complex, large-scale systems with high rigor rather than just small, theoretical examples.
     - **RSL** [ι 201,π 54] (RAISE Specification Language): A wide-spectrum language used to write specifications at various levels of abstraction, combining model-oriented and algebraic approaches with concurrency.
     - **Step-wise Development:** The process starts with a high-level, abstract specification of the requirements. Through a series of development steps, this is refined into more concrete applicative modules and finally into imperative, implemented software.
     - **Components:** The RAISE method consists of three key parts: the formal language (RSL), a specific development method for structuring the process, and tools to support, verify, and translate the specifications.
     - **Origins:** RAISE was developed through the European ESPRIT projects (1985-1990) and refined in the LaCoS project (1990-1995), often cited for its industrial application in European software engineering.

     `[https://www2.imm.dtu.dk/courses/02263/E20/Files/methodnotes99.pdf]`

184. **Reactive Action:** A reactive action is an action [ι 4,π 12] performed by – occurring in – a behaviour [ι 20,π 17] – in response to [caused by, the result of] a prior action.

- **Relative Computability:** see [ι 44,π 21].

185. **Recursive Function Theory:** `[https://plato.stanford.edu/entries/recursive--functions/]` [97]

186. **Resumption Semantics:** We refer to `[https://en.wikipedia.org/wiki/Exception_-handling_(programming)#Termination_and_resumption_semantics]`.

187. **Requirements:** By a requirements we understand (cf., [55, IEEE Standard 610.12]): *"A condition or capability needed by a user to solve a problem or achieve an objective"*

     In *software development* the requirements explain what properties the desired software should have, not how these properties might be attained. In our, the *triptych* approach, requirements are to be "derived" from domain descriptions.[36]

[36] By indenting the following entries we intend to show, in order of "execution",that the entries relate conceptually.

188. **Requirements, Domain:** Domain requirements primarily express the assumptions that a design must rely upon in order that that design can be verified. Although domain requirements firstly express assumptions it appears that the software designer is well-advised in also implementing, as data structures and procedures, the endurants, respectively perdurants expressed in the domain requirements prescriptions. Whereas domain endurants are "real-life" phenomena they are now, in domain requirements prescriptions, abstract concepts (to be represented by a machine).

    Domain requirements can be handled in five consecutive steps:

    (a) projection [ι 191,π 52],
    (b) instantiation [ι 189,π 52],
    (c) determination [ι 190,π 52],
    (d) extension [ι 192,π 52], and
    (e) fitting [ι 193,π 53].

    We refer to [21, Chapter *9*].

189. **Requirements, Instantiation:** [ι 188,π 52] By domain instantiation we mean *a **refinement** of the partial domain requirements prescription (resulting from the projection step) in which the refinements aim at rendering more concrete, more specific the endurants: parts and fluids, as well as the perdurants: actions, events and behaviours of the domain requirements prescription .*

190. **Requirements, Determination:** [ι 188,π 52] By domain determination we mean *a refinement of the partial domain requirements prescription, resulting from the instantiation step, in which the refinements aim at rendering less non-determinate, more determinate the endurants: parts and fluids, as well as the perdurants: functions, events and behaviours of the partial domain requirements prescription .* Determinations usually render these concepts less general. That is, the value space of endurants that are made more determinate is "smaller", contains fewer values, as compared to the endurants before determination has been "applied".
    We refer to [21, *Sect. 9.4.3*].

191. **Requirements, Projection:** [ι 188,π 52] By a domain projection is meant *a subset of the domain description, one which projects out all those endurants: parts and fluids, as well as perdurants: actions, events and behaviours that the stake-holders do not wish represented or relied upon by the machine .* We refer to [21, Sect. *9.4.1*].

192. **Requirements, Extension:** [ι 188,π 52] By domain extension we understand *the introduction of endurants and perdurants that were not feasible in the original domain, but for which, with computing and communication, and with new, emerging technologies, for example, sensors, actuators and satellites, there is the possibility of feasible implementations, hence the requirements, that what is introduced becomes part of the unfolding requirements prescription .*

Instantiations usually render these concepts less general.Properties that hold of the projected domain shall also hold of the (therefrom) instantiated domain.
Refinement of endurants can be expressed either in the form of concrete types, or of further "delineating" axioms over sorts, or of a combination of concretisation and axioms.
We refer to [21, Sect. *9.4.2*].

193. **Requirements, Fitting:** [ι 188,π 52] Often a domain being described "fits" onto, is "adjacent" to, "interacts" in some areas with, another domain: *transportation* with *logistics*, *health-care* with *insurance*, *banking* with *securities trading* and/or *insurance*, and so on. The issue of requirements fitting arises when two or more software development projects are based on what appears to be the same domain. The problem then is to harmonise the two or more software development projects by harmonising, if not too late, their requirements developments. We thus assume that there are n domain requirements developments, d_{r_1}, d_{r_2}, ..., d_{r_n}, being considered, and that these pertain to the same domain — and can hence be assumed covered by a same domain description.
By requirements fitting we mean a *harmonisation* of $n > 1$ domain requirements that have overlapping (shared) not always consistent parts and which results in n partial domain requirements, $p_{d_{r_1}}$, $p_{d_{r_2}}$, ..., $p_{d_{r_n}}$, and m shared domain requirements, $s_{d_{r_1}}$, $s_{d_{r_2}}$, ..., $s_{d_{r_m}}$, that "fit into" two or more of the partial domain requirements . The above characterisation pertains to the result of 'fitting'.
We refer to [21, Sect. *9.4.5*].

194. **Requirements, Interface:** Interface requirements can be expressed only using terms from both the domain and the machine . Users are not part of the machine. So no reference can be made to users, such as *"the system must be user friendly"*, and the like ! By interface requirements we [also] mean *requirements prescriptions which refines and extends the domain requirements by considering those requirements of the domain requirements whose endurants (parts, fluids) and perdurants (actions, events and behaviours) are **"shared"** between the domain and the machine (being requirements prescribed).* The two interface requirements definitions above go hand–in–hand, i.e., complement one-another.
We refer to [21, Sect. *9.5.1*].

195. **Requirements, Machine:** To express machine requirements one must consider many facets:
(A) There are the technology requirements of (1) performance and (2) dependability. Within dependability requirements there are (a) accessibility, (b) availability, (c) integrity, (d) reliability, (e) safety, (f) security and (g) robustness requirements. A proper treatment of dependability requirements need a careful definition of such terms as *failure, error, fault,* and, from these *dependability.*
(B) And there are the development requirements of (i) process, (ii) maintenance, (iii) platform, (iv) management and (v) documentation requirements. Within maintenance requirements there are (ii.1) adaptive, (ii.2) corrective, (ii.3) perfective,

(ii.4) preventive, and (ii.5) extensional requirements. Within platform requirements there are (iii.1) development, (iii.2) iexecution, (iii.3) maintenance, and (iii.4) demonstration platform requirements.

We refer to [10, Sect. 9.6].

196. **Requirements, Derived:**

By derived requirements we mean *requirements prescriptions* which are expressed in terms of the machine concepts and facilities introduced by the emerging requirements.

We refer to [21, *Sect. 9.5.2*].

197. **Requirements, Derived Perdurant:** By a derived perdurant we shall understand a perdurant which is not shared with the domain, but which focus on exploiting facilities of the software or hardware of the machine .

"Exploiting facilities of the software", to us, means that requirements, imply the presence, in the machine, of concepts (i.e., hardware and/or software), and that it is these concepts that the derived requirements "rely" on. We illustrate all three forms of perdurant extensions: derived actions, derived events and derived behaviours.

We refer to [21, *Sect. 9.5.2*].

198. **Requirements Engineering:** is the engineering of constructing requirements [DB].

The aim of requirements engineering is to **design the machine** [ι 136,π 40].

199. **Requirements Prescription:** By a requirements prescription we mean a document which outlines the requirements that some software is expected to fulfill.

200. **Requirements Specification:** By a requirements specification we mean the sames as a requirements prescription.

201. **RSL:** the **RAISE S**pecification **L**anguage [46]. See [ι 183,π 51]. We use three variants of RSL:

- $\text{RSL}^{\div}$ This is RSL (of [46]) without its `class, scheme` and `object` constructs.
- RSL^{+} This is $\text{RSL}^{\div}$ extended with type, T, names ηT.
  We use RSL^{+} as a meta-language for describing modeling.
- RSL^{+}`-text` RSL^{+} extended with `RSL-texts`.
  We use RSL^{+}`-text` as part of the meta-language for describing modeling.

202. **Russell, Bertrand Arthur William [3rd Earl Russell]:** (18 May 1872 –2 February 1970), was an English philosopher, logician, mathematician, and public intellectual. He influenced mathematics, logic, set theory, and various areas of analytic philosophy.

He was one of the early 20th century's prominent logicians and a founder of analytic philosophy, along with his predecessor Gottlob Frege, his friend and colleague G. E. Moore, and his student and protégé Ludwig Wittgenstein.

[`https://en.wikipedia.org/wiki/Bertrand_Russell`]

2.19 S...S...S...S...S...S...

203. **Scott, Dana S.:** (born October 11, 1932) is an American logician who is the Hillman University Professor emeritus of Computer Science, Philosophy, and Mathematical Logic at Carnegie Mellon University. He is now retired and lives in Berkeley, California. He and Michael O. Rabin won the 1976 ACM Turing Award for their work on automata theory, while his collaborative work with Christopher Strachey in the 1970s laid the foundations of modern approaches to the semantics of programming languages. His work on type [i.e., 'domain'] theory [98, 99, 100, 101, 102, 48] is a foundation for 𝔻𝕆𝕄𝔸𝕀ℕ !

    `[https://en.wikipedia.org/wiki/Dana_Scott]`

204. **SECD:** The ***S****tack,* ***C****ontrol,* ***E****nvironment* and ***D****ump* structure of Peter Landin's [ι 128,π 38] operational semantics specification of a simple, applicative language.

    See also [ι 121,π 35], [ι 256,π 74].

205. **Semantics:** is the linguistic and philosophical study of meaning in language, programming languages, formal logics, and semiotics. It is concerned with the relationship between signifiers — like words, phrases, signs, and symbols — and what they stand for in reality, their denotation [38] `[Wikipedia]`.

    There are basically four kinds of semantics, expressed somewhat simplistically:

    - **Algebraic Semantics** is a form of axiomatic semantics based on algebraic laws for describing and reasoning about program semantics in a formal manner.
    - **Axiomatic Semantics** or **Mathematical Logic Proof Systems** is an approach based on mathematical logic – and can be used for proving the correctness of properties of specifications.
    - **Operational Semantics** is a category of formal programming language semantics in which certain desired properties of a program, such as correctness, safety or security, are verified by constructing proofs from logical statements about its execution and procedures, rather than by attaching mathematical meanings to its terms (denotational semantics). Operational semantics are classified in two categories: **structural operational semantics** [ι 231,π 61] (or small-step semantics) formally describe how the individual steps of a computation take place in a computer-based system; by opposition **natural semantics** (or big-step semantics) describe how the overall results of the executions are obtained.

      `[https://en.wikipedia.org/wiki/Operational_semantics]`
    - **Denotational Semantics** model-theoretically assigns a *meaning*, a *denotation*, to each *phrase structure*, i.e., *syntactic category* – usually in the form of functions..

206. **Semiotics:** is the study and knowledge of sign process (semiosis), which is any form of activity, conduct, or any process that involves signs, including the production of meaning `[Wikipedia]`. See [ι 230,π 60].

    A sign is anything that communicates a meaning, that is not the sign itself, to the interpreter of the sign. The meaning can be intentional – such as a word uttered with a specific meaning, or unintentional – such as a symptom being a sign of a particular medical condition. Signs can communicate through any of the senses, visual, auditory, tactile, olfactory, or gustatory

[Wikipedia]. The study and knowledge of semiotics is often "broken down" into the studies, etc., of *syntax*, *semantics* and *pragmatics*.

207. **Set:** In mathematics [ι 144,π 41], a set is a collection of different things; the things are elements [members] of the set and are either domain objects [ι 63,π 24], or are mathematical objects: numbers, symbols, points in space, lines, other geometric shapes, variables, or other sets. A set may be finite or infinite. There is a unique set with no elements, called the empty set; a set with a single element is a singleton.

    "Examples" of sets:

    - a band of musicians
    - a swarm of flies
    - a bunch of crooks
    - a crew of sailors
    - a gang of outlaws
    - a group of people
    - a herd of cattle
    - a a mob of hair
    - a pack of dogs
    - a flock of geese
    - a pride of lions
    - a school of dolphins

208. **Set Comprehension:** Let TE be a type name or type expression, and $\mathcal{B}(e)$ be a predicate expression then

    $$\{ e \mid e : TE \bullet \mathcal{B}(e) \}$$

    is a set comprehension: the set of all e in $e : TE$ that satisfies predicate $\mathcal{B}(e)$.

209. **Set Enumeration:** A set [ι 207,π 56] enumeration of n elements, expressed by expressions $e_1, e_2, ..., e_n$, for $n > 0$, is:

    $$\{ e_1, e_2, ..., e_n \}$$

    where, of course, the ellipses, ..., must be filled in with proper expressions.Cf. [ι 32,π 19], [ι 132,π 39] and [ι 141,π 40].

210. **Set Operation:** These are the operations on sets, s:

    - $s_1 \cup s_2$: union
    - $s_1 \cap s_2$: intersection
    - $s_1 = s_2$: equal
    - $s_1 \neq s_2$: inequality
    - $s_1 \subset s_2$: proper subset
    - $s_1 \subseteq s_2$: subset
    - **card** s: cardinality
    - $e \in s$: membership

211. **Set Part:** A compound part [ι 38,π 20] is a set part, same as a part set, [ι 170,π 48], consists of a [finite or infinite] set of parts.

212. **Set Theory:** studies sets, which can be informally described as collections of objects. Although objects of any kind can be collected into a set, set theory – as a branch of mathematics – is mostly concerned with those that are relevant to mathematics as a whole.

    The modern study of set theory was initiated by the German mathematicians Richard Dedekind and Georg Cantor in the 1870s. In particular, Georg Cantor is commonly considered the founder of set theory. The non-formalized systems investigated during this early stage go

under the name of naive set theory. After the discovery of paradoxes within naive set theory (such as Russell's paradox, Cantor's paradox and the Burali-Forti paradox), various axiomatic systems were proposed in the early twentieth century, of which Zermelo-Fraenkel set theory (with or without the axiom of choice) is still the best-known and most studied.

Set theory is commonly employed as a foundational system for the whole of mathematics, particularly in the form of Zermelo-Fraenkel set theory with the axiom of choice. Besides its foundational role, set theory also provides the framework to develop a mathematical theory of infinity, and has various applications in computer science (such as in the theory of relational algebra), philosophy, formal semantics, and evolutionary dynamics. Its foundational appeal, together with its paradoxes, and its implications for the concept of infinity and its multiple applications have made set theory an area of major interest for logicians and philosophers of mathematics. Contemporary research into set theory covers a vast array of topics, ranging from the structure of the real number line to the study of the consistency of large cardinals.

`[https://en.wikipedia.org/wiki/Set_theory]`

213. **Set Type:** If T is a type name or type expression, then T-**set** is a type expression and denotes the set of all subsets of T.

214. **Science:** is a systematic activity that builds and organizes knowledge in the form of testable explanations and predictions about the universe `[Wikipedia]`.

    Science is the intellectual and practical activity encompassing the systematic study of the structure and behaviour of the physical and natural world through observation and experiment.

215. **Sequence:** By a *sequence* we shall mean the same as a list $[\iota\, 130, \pi\, 39]$.

216. **SMT: Satsfiability Modulo Theories** In computer science and mathematical logic, satisfiability modulo theories is the problem of determining whether a mathematical formula is satisfiable. It generalizes the Boolean satisfiability problem (SAT[37]) complex formulas involving real numbers, integers, and/or various data structures such as lists, arrays, bit vectors, and strings. The name is derived from the fact that these expressions are interpreted within ("modulo") a certain formal theory in first-order logic with equality (often disallowing quantifiers). SMT solvers are tools that aim to solve the SMT problem for a practical subset of inputs. SMT solvers such as Z3[38] ... have been used as a building block for a wide range of applications across computer science, including in automated theorem proving, program analysis, program verification, and software testing.

    See `[https://en.wikipedia.org/wiki/Satisfiability_modulo_theories]`.

217. **Software:** is the is the set of all the documents that have resulted from a completed *software development: domain analysis & description, requirements analysis & prescription, software: software code, software installation manuals, software maintenance manuals, software users guides, development project plans, budget, etc.*

[37] `[https://en.wikipedia.org/wiki/Boolean_satisfiability_problem]`

[38] `[https://en.wikipedia.org/wiki/Z3_Theorem_Prover]` and `[https://github.com/Z3Prover]` and Leonardo De Moura; Nikolaj Bjørner (2008). "Z3: an efficient SMT solver". Tools and Algorithms for the Construction and Analysis of Systems. 4963: 337–340., and `[https://www.microsoft.com/en-us/research/-blog/the-inner-magic-behind-the-z3-theorem-pror]`.

218. **Software Design:** is the engineering of constructing software [DB].

    Whereas software requirements engineering focus on the logical properties that desired software should attain, software design, besides focusing on achieving these properties *correctly*, also focus on the properties being achieved *efficiently*.

219. **Software Engineering:** to us, is then the combination of domain and requirements engineering with software design [DB].

    This is my characterisation of software engineering.

220. **Software Development:** is then the combination of the development of domain description, requirements prescription and software design [DB].

    This is my characterisation of software engineering.

- **Software Requirements:** We refer to [ι 187,π 51]

221. **Software Specification:** A specification of software – ideally it includes the following documents:

    - Informative documents:
        - name, place, dates;
        - partners, customers;
        - scope, span, synopsis;
        - assumptions, dependencies;
        - implicit/derivative goals;
        - standards;
        - contracts, design briefs;
        - logbook; and
        - discussion.
    - Descriptive documents:
        - rough sketches;
        - terminlogies;
        - narratives;
        - formal specifications;
        - code; and
        - discussion.
    - Analytic documents:
        - concepts;
        - validation;
        - verification:
            * theorems & proofs;
            * model checking;
            * tests.

222. **Solid:** A solid element is a pure substance composed of only one type of atom, characterized by a fixed, orderly, and tightly packed crystalline or amorphous structure. Under standard conditions, most elements are solid, including metals (iron, gold, copper) and nonmetals (carbon, sulfur). They maintain a rigid, definite shape and volume. [From the net: AI.]

    See also `[https://en.wikipedia.org/wiki/Solid]` and `[https://www.merriam-webster.com/dictionary/solid]`.

    We shall, however, use the term in a wider sense – allowing for more than one type of atoms. Thus we shall extend the meaning of the term 'solid' to include manifest [ι 138,π 40] as well as conceptual [ι 49,π 21] entities: those that take form: shape and volume, and those that embody an idea: that serves as a foundation for more concrete principles, thoughts, and beliefs.

223. **Space:** Space is not an attribute of entities. Space can be understood as a logic consequence.

We motivate the concept of space as follows: [109, pp 154] The two relations *asymmetric* and *symmetric*, by a transcendental deduction, can be given an interpretation: The relation (spatial) *direction* is asymmetric; and the relation (spatial) *distance* is symmetric. Direction and distance can be understood as spatial relations. From these relations are derived the relation *in-between*. Hence we must conclude that *primary entities exist in space. Space* is therefore an unavoidable characteristic of any possible world .

Mathematicians and physicists model space in, for example, the form of Hausdorf (or topological) space[39]; or a metric space which is a set for which distances between all members of the set are defined; Those distances, taken together, are called a metric on the set; a metric on a space induces topological properties like open and closed sets, which lead to the study of more abstract topological spaces; or Euclidean space, due to Euclid of Alexandria.

(a) There is an abstract notion of (definite) $\mathbb{SPACE}$(s) of further unanalysable points; and

(b) there is a notion of $\mathbb{POINT}$ in $\mathbb{SPACE}$.

type
223a $\mathbb{SPACE}$
223b $\mathbb{POINT}$

(c) A point observer, **observe**_$\mathbb{POINT}$, is a function which applies to endurants, e, and yield a point, $\ell : \mathbb{POINT}$. As the endurant have extent, i.e., occupy space, we need say something about where "in" that endurant space the point "is taken. We do not ! ! !

value
223c **observe**_$\mathbb{POINT}$: E $\to$ $\mathbb{POINT}$

We suggest, besides $\mathbb{POINT}$s, the following spatial attribute possibilities:

(c) $\mathbb{EXTENT}$ as a dense set of $\mathbb{POINT}$s;

(d) Volume, of concrete type, for example, m^3, as the "volume" of an $\mathbb{EXTENT}$ such that

(e) $\mathbb{SURFACE}$s as dense sets of $\mathbb{POINT}$s have no volume, but an

(f) Area, of concrete type, for example, m^2, as the "area" of a dense set of $\mathbb{POINT}$s;

(g) $\mathbb{LINE}$ as dense set of $\mathbb{POINT}$s with no volume and no area, but

(h) Length, of concrete type, for example, m.

For these we have that

(i) the *intersection*, $\bigcap$, of two $\mathbb{EXTENT}$s is an $\mathbb{EXTENT}$ of possibly nil Volume,

(j) the intersection, $\bigcap$, of two $\mathbb{SURFACE}$s may be either a possibly nil $\mathbb{SURFACE}$ or a possibly nil $\mathbb{LINE}$, or a combination of these.

(k) the intersection, $\bigcap$, of two $\mathbb{LINE}$s may be either a $\mathbb{LINE}$ or a $\mathbb{POINT}$ or "nothing" !

Similarly we can define

[39] Armstrong, M. A. (1983) [1979]. Basic Topology. Undergraduate Texts in Mathematics. Springer. ISBN 0-387-90839-0.

(l) the *union*, $\bigcup$, of two not-disjoint $\mathbb{EXTENT}$s,
(m) the *union*, $\bigcup$, of two not-disjoint $\mathbb{SURFACE}$s,
(n) the *union*, $\bigcup$, and of two not-disjoint $\mathbb{LINE}$s.

and:

(a) the *[in]equality*, $\neq$, $=$, of pairs of $\mathbb{EXTENT}$, pairs of $\mathbb{SURFACE}$s, and pairs of $\mathbb{LINE}$s.

224. **Statecharts:** invented by computer scientist David Harel, are gaining widespread usage since a variant has become part of the Unified Modeling Language (UML). The diagram type allows the modeling of superstates, orthogonal regions, and activities as part of a state.

Classic state diagrams require the creation of distinct nodes for every valid combination of parameters that define the state. For all but the simplest of systems, this can lead to a very large number of nodes and transitions between nodes (state and transition explosion), which reduces the readability of the state diagram. With Harel statecharts it is possible to model multiple cross-functional state diagrams within the statechart. Each of these cross-functional state machines can transition internally without affecting the other state machines. The current state of each cross-functional state machine defines the state of the system. The Harel statechart is equivalent to a state diagram but improves its readability.

`[https://en.wikipedia.org/wiki/State_diagram]`

225. **State Machine:** `[https://en.wikipedia.org/wiki/State_diagram]`

226. **Statement:** In the context of $\mathbb{DOMAIN}$ by statement, we mean such formal specification (include program code) texts which designate state changes. See also [ι 36,π 20] [clause] and [ι 84,π 26] [expression].

227. **Syntax:** is the set of rules, principles, and processes that govern the structure of sentences (sentence structure) in a given language, usually including word order `[Wikipedia]`. See also [ι 106,π 31].

We assume, as an absolute minimum of knowledge, that the reader of this document is well aware of the concepts of `BNF` (*Backus Normal Form*) `Grammars` and `CFGs` (*Context Free Grammars*).

228. **Solid Endurant:** By a *solid* [or *discrete*] endurant [ι 74,π 25] we shall understand an endurant which is separate, individual or distinct in form or concept, or, rephrasing: have 'body' [or magnitude] of three-dimensions: length, breadth and depth [74, Vol. II, pg. 2046].

Same as *discrete endurant* [ι 60,π 23].

229. **State:** A state is here sen as a non-empty collection of endurants [of a domain] [ι 74,π 25].

230. **Syntax, Semantics and Pragmatics:** With the advent of computing and their attendant programming languages these concepts of semiotics has taken on a somewhat additional meaning. When, in computer & computing science and in software engineering, we speak of syntax, we mean a quite definite and (mathematically) precise thing. With the advent of our ability to mathematically precise describe the semantics of programming languages, we similarly mean quite definite and (mathematically) precise things. For natural, i.e., human languages, this is

not so. As for pragmatics there is this to say. Computers have not pragmatics. Humans have. When, in this paper we bring the term 'pragmatics' into play we are referring not to the computer "being pragmatic", but to our pragmatics, as scientists, as engineers.

231. **Structural Operational Semantics:** see [ι 205,π 55] and [83, *Gordon D. Plotkin*].

232. **Sørlander, Kai:** Danish philosopher. In his books: [104, 105, 106, 107, 108, 109, 110, 111] Sørlander argues for the view that there is a *definitive and universal truth about our situation in the world.* He asks *what is thus necessary that it could under any conceivable circumstances be different. Is there anything at all which is thus necessary that it could not be otherwise ?* Through transcendental reasoning [ι 241,π 67] he then shows that time and space follows, logically from associative, symmetric and transitive relation ans the principle of contradiction. He builds upon the notion of the interdependence between the meaning of designations and relation of implication between propositions. (The latter is exemplified in the example given in entry [ι 15,π 14].) My work on 𝔻𝕆𝕄𝔸𝕀ℕ owes much to Sørlander's philosophy.

2.20 T...T...T...T...T...T...

233. **Taxonomy:** is the practice and science of classification of things or concepts, including the principles that underlie such classification [Wikipedia].

A taxonomy formalizes the hierarchical relationships among concepts and specifies the term to be used to refer to each; it prescribes structure and terminology.

Taxonomies *classify.*

In 𝔻𝕆𝕄𝔸𝕀ℕ we distinguish between *Endurant Taxonomy* [ι 78,π 26] and *Behavioural Taxonomy* [ι 24,π 17].

Taxonomy

Example: 20 *Cf. Method Sect. 3.6.1.2 on page 83 and Example Sects. 4.2.1.3 on page 96 and 4.5.3 on page 104..*

234. **Technique:** By a *technique* we shall, loosely, understand *(i) formal practical details in artistic, etc., expression, (ii) art, skill, craft in work"* [www.etymonline.com].

The 𝔻𝕆𝕄𝔸𝕀ℕ method evolves around *principles, procedures,* ***techniques*** and *tools.*

Technique

Example: 21 *One of the many techniques of the 𝔻𝕆𝕄𝔸𝕀ℕ method is that of transcendentally deducing behaviours from the internal qualities [ι 119,π 34] of behavioural parts [ι 23,π 17].*

235. **Technology:** is the sum of techniques, skills, methods, and processes used in the production of goods or services or in the accomplishment of objectives, such as scientific investigation [Wikipedia]. Technology can be the knowledge of techniques, processes, and the like, or it can be embedded in machines to allow for operation without detailed knowledge of their workings. Systems (e.g. machines) applying technology by taking an input, changing it according to the system's use, and then producing an outcome are referred to as technology systems or technological systems [Wikipedia].

236. **Testing:** is the act of checking whether software meets its intended objectives and satisfies expectations.

Software testing can provide objective, independent information about the quality of software and the risk of its failure to a user or sponsor or any other stakeholder.

Software testing can determine the correctness of software for specific scenarios but cannot determine correctness for all scenarios. It cannot find all bugs.

[https://en.wikipedia.org/wiki/Software_testing]

237. **Theorem Proofs:** When this term is used it usually refer to proofs of theorems about computer programs [ι 180,π 49]. Basically these proofs come in three forms: (i) in oral presentations, (ii) in engineering & scientic journal papers, and (iii) as the result of – usuallly interactive – computer supported proofs.

See also [ι 146,π 41].

238. **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[40]

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.[41]

[114, 44, 77, 85, 91, 86, 87, 88, 89, 90, 96] and [45].

We motivate the abstract notion of time as follows. [109, pp 159] Two different states must necessarily be ascribed different incompatible predicates. But how can we ensure so ? Only if states stand in an asymmetric relation to one another. This state relation is also transitive. So that is an indispensable property of any world. By a transcendental deduction we say that *primary entities exist in time*. *So every possible world must exist in time* .

We shall not be concerned with any representation of time. That is, we leave it to the domain analyser cum describer to choose an own representation [45]. Similarly we shall not be concerned with any representation of time intervals.[42]

(b) So there is an abstract type $\mathbb{T}ime$,

(c) and an abstract type $\mathbb{TI}$: $\mathbb{T}ime\mathbb{I}nterval$.

(d) There is no $\mathbb{T}ime$ origin, but there is a "zero" $\mathbb{TI}$me interval.

(e) One can add (subtract) a time interval to (from) a time and obtain a time.

(f) 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.

(g) One can subtract a time from another time obtaining a time interval respecting that the subtrahend is smaller than or equal to the minuend.

(h) One can multiply a time interval with a real and obtain a time interval.

(i) One can compare two times and two time intervals.

[40] Quoted from [3, Cambridge Dictionary of Philosophy]
[41] J.R.R. Tolkien, The Hobbit
[42] – 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.

```
type
238b  𝕋
238c  𝕋𝕀
value
238d  0:𝕋𝕀
238e  +,−: 𝕋 × 𝕋𝕀 → 𝕋
238f  +,−: 𝕋𝕀 × 𝕋𝕀 ~→ 𝕋𝕀
238g  −: 𝕋 × 𝕋 → 𝕋𝕀
238h  ∗: 𝕋𝕀 × Real → 𝕋𝕀
238i  <,≤,=,≠,≥,>: 𝕋 × 𝕋 → Bool
238i  <,≤,=,≠,≥,>: 𝕋𝕀 × 𝕋𝕀 → Bool
axiom
238e ∀ t:𝕋 • t+0 = t
```

(a) We define the signature of the meta-physical time observer.

```
type
237a  𝕋
value
237a  record_𝕋𝕀𝕄𝔼(): Unit → 𝕋
```

The time recorder applies to nothing and yields a time. **record_**$\mathbb{TIME}$() can only occur in action, event and behavioural descriptions.

Modern models of time, by mathematicians and physicists evolve around spacetime[43] We shall not be concerned with this notion of time.

Models of time related to computing differs from those of mathematicians and physicists in focusing on divergence and convergence, zero (Zenon) time and interleaving time [126] are relevant in studies of real-time, typically distributed computing systems. We shall also not be concerned with this notion of time.

van Benthem: A Continuum Theory of Time:

The following is taken from Johan van Benthem [114]: Let P be a point structure (for example, a set). Think of time as a continuum; the following axioms characterise ordering ($<$, $=$, $>$) relations between (i.e., aspects of) time points. The axioms listed below are not thought of as an axiom system, that is, as a set of independent axioms all claimed to hold for the time concept, which we are encircling. Instead van Benthem offers the individual axioms as possible "blocks" from which we can then "build" our own time system — one that suits the application at hand, while also fitting our intuition. Time is transitive: If $p<p'$ and $p'<p''$ then $p<p''$. Time may not loop, that is, is not reflexive: $p \not< p$. Linear time can be defined: Either one time comes before, or is equal to, or comes after another time. Time can be left-linear, i.e., linear "to the left" of a given time. One could designate a time axis as beginning at some time, that is, having no predecessor times. And one can designate a time axis as ending at some time, that

[43] The concept of **Spacetime** was first "announced" by Hermann Minkowski, 1907–08 – based on work by Henri Poincaré, 1905–06, `https://en.wikisource.org/wiki/Translation:_The_Fundamental_Equations_for_Electromagnetic_Processes_in_Moving_Bodies`

is, having no successor times. General, past and future successors (predecessors, respectively successors in daily talk) can be defined. Time can be dense: Given any two times one can always find a time between them. Discrete time can be defined.

[TRANS: Transitivity] $\forall$ p,p′,p″:P • p < p′ < p″ $\Rightarrow$ p < p″
[IRREF: Irreflexitivity] $\forall$ p:P • p $\not<$ p
[LIN: Linearity] $\forall$ p,p′:P • (p=p′ $\vee$ p<p′ $\vee$ p>p′)
[L−LIN: Left Linearity] $\forall$ p,p′,p″:P • (p′<p $\wedge$ p″<p) $\Rightarrow$ (p′<p″ $\vee$ p′=p″ $\vee$ p″<p′)
[BEG: Beginning] $\exists$ p:P • $\sim\exists$ p′:P • p′<p
[END: Ending] $\exists$ p:P • $\sim\exists$ p′:P • p<p′
[SUCC: Successor]
  [PAST: Predecessors] $\forall$ p:P,$\exists$ p′:P • p′<p
  [FUTURE: Successor] $\forall$ p:P,$\exists$ p′:P • p<p′
[DENS: Dense] $\forall$ p,p′:P (p<p′ $\Rightarrow$ $\exists$ p″:P • p<p″<p′)
[CDENS: Converse Dense] $\equiv$ [TRANS: Transitivity] $\forall$ p,p′:P ($\exists$ p″:P • p<p″<p′ $\Rightarrow$ p<p′)
[DISC: Discrete]
  $\forall$ p,p′:P • (p<p′ $\Rightarrow$ $\exists$ p″:P • (p<p″ $\wedge$ $\sim\exists$ p‴:P • (p<p‴<p″))) $\wedge$
  $\forall$ p,p′:P • (p<p′ $\Rightarrow$ $\exists$ p″:P • (p″<p′ $\wedge$ $\sim\exists$ p‴:P • (p″<p‴<p′)))
[TRANS: Transitivity] $\forall$ p,p′,p″:P • p < p′ < p″ $\Rightarrow$ p < p″
[IRREF: Irreflexitivity] $\forall$ p:P • p $\not<$ p
[LIN: Linearity] $\forall$ p,p′:P • (p=p′ $\vee$ p<p′ $\vee$ p>p′)
[L−LIN: Left Linearity]
  $\forall$ p,p′,p″:P • (p′<p $\wedge$ p″<p) $\Rightarrow$ (p′<p″ $\vee$ p′=p″ $\vee$ p″<p′)
[BEG: Beginning] $\exists$ p:P • $\sim\exists$ p′:P • p′<p
[END: Ending] $\exists$ p:P • $\sim\exists$ p′:P • p<p′
[SUCC: Successor]
  [PAST: Predecessors] $\forall$ p:P,$\exists$ p′:P • p′<p
  [FUTURE: Successor] $\forall$ p:P,$\exists$ p′:P • p<p′
[DENS: Dense] $\forall$ p,p′:P (p<p′ $\Rightarrow$ $\exists$ p″:P • p<p″<p′)
[CDENS: Converse Dense] $\equiv$ [TRANS: Transitivity] $\forall$ p,p′:P ($\exists$ p″:P • p<p″<p′ $\Rightarrow$ p<p′)
[DISC: Discrete]
  $\forall$ p,p′:P • (p<p′ $\Rightarrow$ $\exists$ p″:P • (p<p″ $\wedge$ $\sim\exists$ p‴:P • (p<p‴<p″))) $\wedge$
  $\forall$ p,p′:P • (p<p′ $\Rightarrow$ $\exists$ p″:P • (p″<p′ $\wedge$ $\sim\exists$ p‴:P • (p″<p‴<p′)))

A strict partial order, SPO, is a point structure satisfying TRANS and IRREF. TRANS, IRREF and SUCC imply infinite models. TRANS and SUCC may have finite, "looping time" models.

Wayne D. Blizard: A Theory of Time–Space

We shall present an axiom system [35, Wayne D. Blizard, 1980] which relate abstracted entities to spatial points and time. Let $A, B, \ldots$ stand for entitites, $p, q, \ldots$ for spatial points, and t, τ for times. 0 designates a first, a begin time. Let t' stand for the discrete time successor of time t. Let $N(p, q)$ express that p and q are spatial neighbours. Let $=$ be an overloaded equality operator applicable, pairwise to entities, spatial locations and times, respectively. A_p^t expresses that entity A is at location p at time t. The axioms — where we omit (obvious) typings (of A, B, P, Q, and T): $'$ designates the time successsor function: t'.

$$
\begin{array}{rrrcll}
(I) & & \forall A \forall t \exists p & : & A_p^t & \\
(II) & & (A_p^t \wedge A_q^t) & \supset & p = q & \\
(III) & & (A_p^t \wedge B_p^t) & \supset & A = B & \\
(IV)(?) & & (A_p^t \wedge A_p^{t'}) & \supset & t = t' & \\
(V & i) & \forall p, q & : & N(p,q) \supset p \neq q & \text{Irreflexivity} \\
(V & ii) & \forall p, q & : & N(p,q) = N(q,p) & \text{Symmetry} \\
(V & iii) & \forall p \exists q, r & : & N(p,q) \wedge N(p,r) \wedge q \neq r & \text{No isolated locations} \\
(VI & i) & \forall t & : & t \neq t' & \\
(VI & ii) & \forall t & : & t' \neq 0 & \\
(VI & iii) & \forall t & : & t \neq 0 \supset \exists \tau : t = \tau' & \\
(VI & iv) & \forall t, \tau & : & \tau' = t' \supset \tau = t & \\
(VII) & & A_p^t \wedge A_q^{t'} & \supset & N(p,q) & \\
(VIII) & & A_p^t \wedge B_q^t \wedge N(p,q) & \supset & \sim (A_q^{t'} \wedge B_p^{t'}) &
\end{array}
$$

(II–IV,VII–VIII): The axioms are universally 'closed'.

That is: we have omitted the usual $\forall A, B, p, q, t$s.

(I): For every entity, A, and every time, t, there is a location, p, at which A is located at time t.

(II): An entity cannot be in two locations at the same time.

(III): Two distinct entities cannot be at the same location at the same time.

(IV): Entities always move: An entity cannot be at the same location at different times. *This is more like a conjecture: Could be questioned.*

(V): These three axioms define N.

(V i): Same as $\forall p :\sim N(p,p)$. "Being a neighbour of", is the same as "being distinct from".

(V ii): If p is a neighbour of q, then q is a neighbour of p.

(V iii): Every location has at least two distinct neighbours.

(VI): The next four axioms determine the time successor function $'$.

(VI i): A time is always distinct from its successor: time cannot rest. There are no time fix points.

(VI ii): Any time successor is distinct from the begin time. Time 0 has no predecessor.

(VI iii): Every non–begin time has an immediate predecessor.

(VI iv): The time successor function $'$ is a one–to–one (i.e., a bijection) function.

(VII): The *continuous path axiom:* If entity A is at location p at time t, and it is at location q in the immediate next time (t'), then p and q are neighbours.

(VIII): *No "switching":* If entities A and B occupy neighbouring locations at time t them it is not possible for A and B to have switched locations at the next time (t').

Except for Axiom (IV) the system applies both to systems of entities that "sometimes" rests, i.e., do not move. These entities are spatial and occupy at least a point in space. If some entities "occupy more" space volume than others, then we interpret, in a suitable manner, the notion of the point space P (etc.). We do not show so here.

238. **Tool:** By a *tool* we shall, loosely, understand *(i) instrument, implement used by a craftsman or laborer, weapon, (ii) that with which one prepares something, etc.* [www.etymonline.com].

    The 𝔻𝕆𝕄𝔸𝕀ℕ method evolves around *principles, procedures, techniques* and **tools**.

    In 𝔻𝕆𝕄𝔸𝕀ℕ we consider the domain analysis language, i.e., the **is_** predicate prompts, and the domain description language, RSL$^+$-text (with its **obs_** s, **uid_** s, **mereo_** s, and **attr_** s) as tools [DB].

239. **The Triptych Dogma:**

Then Triptych Dogma

In order to *specify* 𝕊*oftware*, we must understand its ℝ*equirements*.
In order to *prescribe* ℝ*equirements* we must understand the 𝔻*omain*.
So we must study, analyze and describe 𝔻*omains*.

$$\mathbb{D},\mathbb{S} \models \mathbb{R}$$

In proofs of 𝕊*oftware* correctness,
with respect to ℝ*equirements*,
assumptions are made with respect to the 𝔻*omain*.

240. **Transcendense** In philosophy, transcendence is the basic ground concept from the word's literal meaning (from Latin), of climbing or going beyond, albeit with varying connotations in its different historical and cultural stages. It includes philosophies, systems, and approaches that describe the fundamental structures of being, not as an ontology (theory of being), but as the framework of emergence and validation of knowledge of being. These definitions are generally grounded in reason and empirical observation and seek to provide a framework for understanding the world that is not reliant on religious beliefs or supernatural forces."Transcendental" is a word derived from the scholastic, designating the extra-categorical attributes of beings. https://en.wikipedia.org/wiki/Transcendence_(philosophy)

241. **Transcendental Deduction** Transcendental Deduction is a central argument in *Immanuel Kant*s [ι 124,π 37] *Critique of Pure Reason* that justifies how subjective, a priori concepts (categories like causality or substance) can objectively apply to experienced objects. It argues that for coherent experience to exist, the mind must synthesize sensory data using these innate concepts, making them necessary conditions for objective knowledge.

    It is a core element of 𝔻𝕆𝕄𝔸𝕀ℕ that [behavioural] parts can be transcendentally deduced, i.e., "morphed", [in]to behaviours[44].

    Key Aspects of the Transcendental Deduction[45]:

    - Objective and Subjective Sides: The deduction has a subjective side (how the mind makes judgments) and an objective side (how concepts apply to objects).

[44] I owe to Kai Sørlander's philosophy [104, 105, 106, 107, 108, 109, 110, 111] the concept of transcendental-deducing parts to behaviours.

[45] https://plato.stanford.edu/entries/kant-transcendental/ and https://medium.com/philosophy-today/what-is-the-transcendental-deduction-of-the-categories-0ec2fb823b98

- Categories as Necessary: Kant argues that without these a priori concepts, experience would be a chaotic, disconnected, and meaningless "manifold".
- Synthesis of Understanding: The mind does not passively receive information; it actively synthesizes, or combines, sensory data into a unified, coherent experience.
- Response to Hume: It serves as a direct, and often debated, rebuttal to David Hume's skepticism regarding the objective validity of concepts like causality.
- Limitations: The deduction restricts the use of these concepts to "appearances" (things as they appear to us) rather than "things-in-themselves".

242. **Turing, Alan Mathison:** (23 June 1912 –7 June 1954) was an English mathematician, computer scientist, logician, cryptanalyst, philosopher and theoretical biologist. He was highly influential in the development of theoretical computer science, providing a formalisation of the concepts of algorithm and computation with the Turing machine, which can be considered a model of a general-purpose computer. Turing is widely considered to be the father of theoretical computer science `https://en.wikipedia.org/wiki/Alan_Turing`.

243. **Turing Machine:** A Turing machine is a mathematical model of computation describing an abstract machine that manipulates symbols on a strip of tape according to a table of rules. Despite the model's simplicity, it is capable of implementing any computer algorithm [ι 9,π 13] `https://en.wikipedia.org/wiki/Turing_machine.`

244. **Type:** By a *type* we shall, loosely, understand a possibly infinite set of values.

Types:

Example: 22 *Examples of types, T, are:*

- *natural numbers,* ***Nat****,*
- *integers,* ***Int****,*
- *reals,* ***Real****,*
- *Booleans,* ***Bool****,*
- *finite sets, T-****set****,*
- *possibly infinite sets, T-****infset****,*
- *Cartesians, $T1\times T2\times ...\times Tn$,*
- *finite lists, T^+,*
- *possibly infinite lists, T^ω,*
- *maps, $Td\overset{m}{\rightarrow}Tr$,*
- *total functions, $Td\rightarrow Tr$,*
- *partial functions, $Td\overset{\sim}{\rightarrow}Tr$.*

245. **Type Definition:** Let T be a type names, i.e., identifier, and TE a type expression, cf. [ι 246,π 68], then these are type definitions:

type
T [further unspecified sort],
T = TE ["concretized" type, see [ι 246,π 68] next:]

246. **Type Expression:** Let `T1, T2, ..., Ti, ..., Tn` be type, not sort, names, cf. Example [ι 244,π 68], then these are type expressions TE:

- `Ti`,
- `TEi`-**set** [46],
- `TEi`-**infset** [47],
- `TE1`×`TE2`×...×`TEm` [48],
- `TE`* [49],
- `TE`$^{\omega}$ [50],
- `TEi`$\xrightarrow[m]{}$`TEj` [51],
- `TEi`→`TEj` [52],
- `TEi`$\xrightarrow{\sim}$`TEj`, [53] and
- `TE1`|`TE2`|...|`TEm` [54]

[46] the set of all finite sets of type `TEi`
[47] the set of all finite and infinite sets of type `TEi`
[48] the set of all Cartesians over `TE1, TE2, ..., TEm`
[49] the set of all finite lists over `TE`
[50] the set of all finite and infinite lists over `TE`
[51] the set of all maps from `TEi` to `TEj`
[52] the set of all total functions from `TEi` to `TEj`
[53] he set of all partial and total functions from `TEi` to `TEj`
[54] the union set of types `TE1, TE2,...,TEm`

2.21 U...U...U...U...U...U...

- **Uncomputability:** see [ι 42,π 20]: uncomputable refers to problems, functions, or numbers that cannot be solved or generated by any algorithm or Turing machine, even with infinite time and memory. These mathematical limitations exist because the set of possible algorithms is countable, while the set of potential problems is uncountable.

247. **Universe of Discourse:** We use the term 'universe of discourse' as the general name for any arbitrarily selected domain.

Universe of Discourse

Example: 23 *Cf. Method Sect. 3.4 on page 79 and Example Sect. 4.1 on page 95.*

248. **Unique Identification:** Solid endurants [ι 74,π 25], i.e., parts [ι 169,π 48], are distinguishable by, what we shall call, their identifiers – which are therefore unique. See next entry !

Unique Identification

Example: 24 *Cf. Method Sect. 3.7.1 on page 85 and Example Sect. 4.2.2.1 on page 96.*

249. **Unique Identifiers:** [See the previous entry.] We do not have to bother about how unique identifiers are represented. Just that they are !

Unique Identifiers

Example: 25 *Cf. Method Sect. 3.7.1.1 on page 86 and Example Sect. 4.2.2.1 on page 96.*

250. **Unique Identifier State:** Any selection of parts, form a state, and so do the set[s] of their unique identifiers – see next entry !

Unique Identifier States

Example: 26 *Cf. Method Sect. 3.7.1.2 on page 87 and Example Sect. 4.2.2.1.1 on page 97.*

251. **Uniqueness:** [See the previous entry.] Any set of behavioural parts [ι 23,π 17] and their unique identifiers have the same cardinality, i.e., the parts are unique.

Uniqueness

Example: 27 *Cf. Method Sect. 3.7.1.3 on page 87 and Example Sect. 4.2.2.1.2 on page 98.*

2.22 V...V...V...V...V...V...

252. **VDM:** The Vienna Development Method. VDM emerged as a result of the IBM Vienna Laboratory's research into and development of a formal definition of the PL/I programming language (1973–1974).

253. **VDM SL:** The VDM [ι 252,π 71] Specification Language. ["Originally" this language was referred to as `Meta IV`.] The name `VDM SL` emeregd as the result of an ISO standadisation. `VDM SL` is basically a discrete mathematics [ι 59,π 23] notation in the style of Landin's [ι 128,π 38] `ISWIM` [ι 121,π 35].

    First monographs/textbooks were [31, 33]

254. **Verification:** The process of establishing the truth, accuracy, or validity of something.

2.23 W...W...W...W...W...W...

255. **Wirth, Niklaus:** (15 February 1934 –1 January 2024) was a Swiss computer scientist. He designed several programming languages, including Pascal, and pioneered several classic topics in software engineering. In 1984, he won the Turing Award, generally recognized as the highest distinction in computer science, "for developing a sequence of innovative computer languages" [117, 124, 118, 119, 120, 121, 122, 123, *Algol W, Euler, Pascal, Modula 2*, Oberon, etc.].

    [https://en.wikipedia.org/wiki/Niklaus_Wirth]

2.24 X...X...X...X...X...X...

2.25 Y...Y...Y...Y...Y...

256. **Y:** Landin's [ι 128,π 38] name for the fix-point function [67, 66, 67, 70, 68, 69, 73, 72, 71] !

2.26 Z...Z...Z...Z...Z...Z...

257. **Z:** An abstract model-oriented specification language [125].

258. **Z3:** A leading *SMT: Satisfiability Modulo Theories* tool. See Enty [ι 216,π 57].

259. **Zermelo, Ernst Friedrich Ferdinand:** (27 July 1871 –21 May 1953) was a German logician and mathematician, whose work has major implications for the foundations of mathematics. He is known for his role in developing Zermelo–Fraenkel axiomatic set theory and his proof of the well-ordering theorem.

    [https://en.wikipedia.org/wiki/Ernst_Zermelo]

3 The DOMAIN Method

3.1 Introductory Remarks

> This section presents a rigorous, semi-formal description of some aspects of the process of analysing & describing domains. It takes for granted the analysis & description approach – where the analysis includes, but this has not been shown in published works [17, 19, 21, 24, 29], assessments of the domain acquisition. That is: it is the rigorous description (of a model of) the method that is the focal point of our discourse. Not whether the domain analysis & description approach is appropriate.[55]

By a *method* [ι 151,π 42] we shall understand a set of *principles* [ι 177,π 49] and *procedures* [ι 179,π 49] for *selecting* and *applying* a set of *techniques* [ι 234,π 62] *using* a set of *tools* [ι 238,π 67] in order to *construct* an *artefact* [ι 12,π 13].

Endurants are those quantities of domains that we can observe (see and touch), in *space*, as "complete" entities at no matter which point in *time* – "material" entities that persists, endures – capable of enduring adversity, severity, or hardship [Merriam Webster].

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* [Merriam Webster].

The analysis & description process [to be rigorously described in the present sectom] is based on the domain analysis & description omtology [ι 167,π 46] shown i Fig. 2 on the next page.

Terms related to Fig. 2 on the facing page are:

- domain [ι 63,π 24]
- entity [ι 79,π 26]
- endurant [ι 74,π 25]
- external quality [ι 86,π 26]
- solid (entity) [ι 222,π 58]
- fluid (entity) [ι 96,π 29]
- part [ι 169,π 48]
- living species [ι 134,π 39]
- atomic part [ι 13,π 13]
- compound part [ι 38,π 20]
- composite (Cartesian part) [ι 33,π 20]
- part set [ι 170,π 48]
- internal quality [ι 119,π 34]
- unique identifier [ι 249,π 70]
- mereology [ι 149,π 42]
- attributes [ι 14,π 14]
- perdurant [ι 171,π 48]
- channel [ι 34,π 20]
- action [ι 4,π 12]
- event [ι 81,π 26]
- behaviour [ι 20,π 17]

3.2 The Stages and Steps of The DOMAIN Method

- **Stage 0. Initialisation:** Cf. Entry [ι 115,π 34] and *Method* Sect. 3.3 on page 78

[55] Issues such as *assessment of domain analysis & description*, akin to those of *requirements assessment* – so well summarised in `https://en.wikipedia.org/wiki/Requirements_analysis` – are, of course, an element of the domain analysis. But they are not covered in the above cited papers as they are of informal, human interaction flavour. But they can be reformulated for domain engineering.

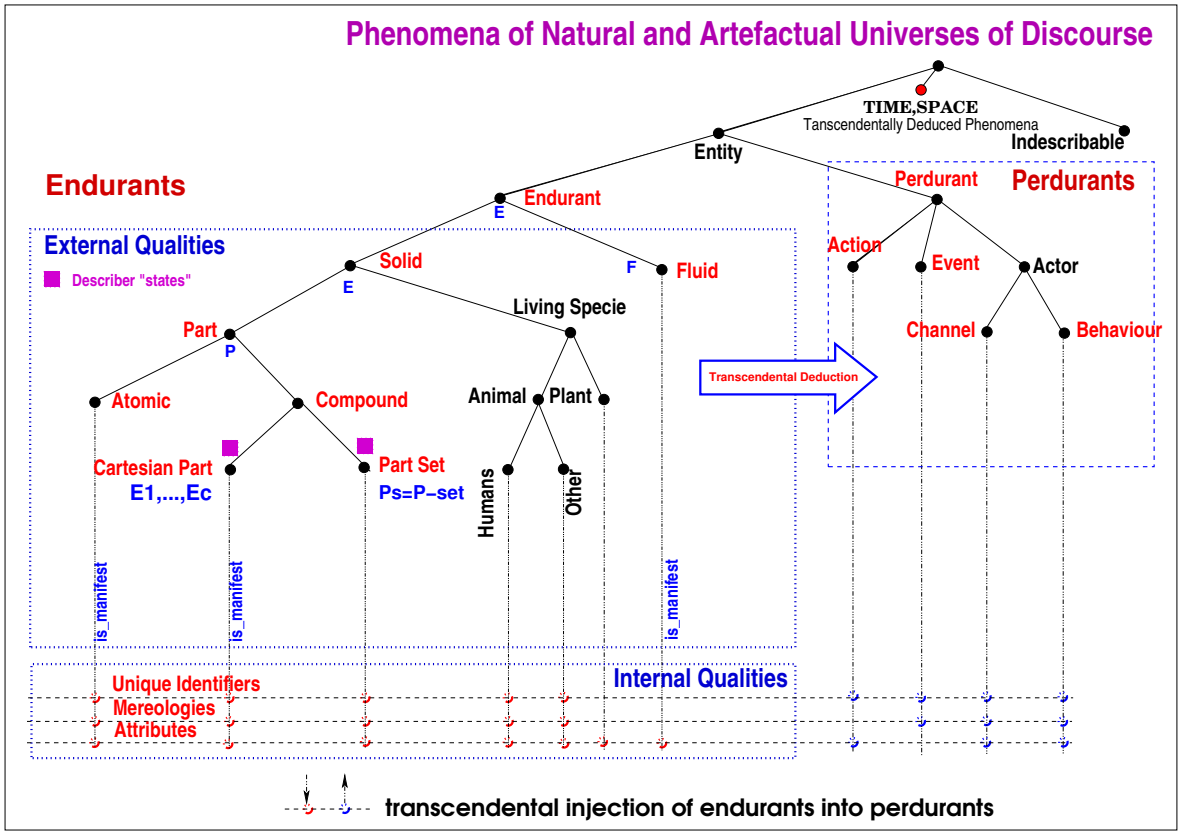


Figure 2: The Domain Analysis & Description Ontology

- **Stage 1. Universe of Discourse:** Cf. *Entry* [ι 247,π 70], *Method* Sect. 3.4 on page 79 and *Example* Sect. 4.1 on page 95.
- **Stage 2. Discover Domain:** Cf. *Method* Sect. 262 on page 79.
- **Stage 3. External Qualities:** Cf. *Entry* [ι 86,π 26], *Method* Sect. 3.6.1.1 on page 80 and *Example* Sect. 4.2.1.1 on page 95.
  - **Stage 4. Endurant Descriptions:** Cf. *Entry* [ι 75,π 25], *Method* Sect. 3.6.1.1 on page 80 and *Example* Sect. 4.2.1.1 on page 95.
  - **Stage 5. Endurant Taxonomy:** Cf. *Entry* [ι 78,π 26], *Method* Sect. 3.6.1.2 on page 83 and *Example* Sect. 4.2.1.3 on page 96.
  - **Stage 6. Endurant States:** Cf. *Entry* [ι 77,π 25], *Method* Sect. 3.6.1.3 on page 84 and *Example* Sect. 4.2 on page 95.
- **Stage 7. Internal Qualities:** Cf. *Entry* [ι 119,π 34], *Method* Sect. 3.7 on page 85 and *Example* Sect. 4.2.2 on page 96.
  - **Stage 8. Unique Identification:** Cf. Entry [ι 248,π 70] and *Method* Sect. 3.7.1 on page 85
    - **Stage 9. Unique Identifiers:** Cf. *Entry* [ι 249,π 70], *Method* Sect. 3.7.1.1 on page 86 and *Example* Sect. 4.2.2.1 on page 96.
    - **Stage 10. Unique Identifier State:** Cf. *Entry* [ι 250,π 70], *Method* Sect. 3.7.1.2 on page 87 and *Example* Sect. 4.2.2.1.1 on page 97.

    - * **Stage 11. Uniqueness:** Cf. *Entry* [ι 251,π 70], *Method* Sect. 3.7.1.3 on page 87 and *Example* Sect. 4.2.2.1.2 on page 98.
  - **Stage 12. Mereology:** Cf. *Entry* [ι 149,π 42], *Method* Sect. 3.7.2 on page 87 and *Example* Sect. 4.2.2.2 on page 98.
  - **Stage 13. Attributes:** Cf. *Entry* [ι 14,π 14], *Method* Sect. 3.7.3 on page 88 and *Example* Sect. 4.2.2.3 on page 98.
  - **Stage 14. Intentional Pulls:** Cf. *Entry* [ι 117,π 34], *Method* Sect. 3.7.4 on page 89 and *Example* Sect. 4.2.2.4 on page 100.
- **Stage 15. Perdurants:** Cf. *Entry* [ι 171,π 48], *Method* Sect. 3.8 on page 90 and *Example* Sect. 4.5 on page 104.
  - **Stage 16. Channel:** Cf. *Entry* [ι 34,π 20], *Method* Sect. 3.8.1.1 on page 91 and *Example* Sect. 4.5.1 on page 104.
  - **Stage 17. Actions:** Cf. Entry [ι 4,π 12] and *Method* Sect. 3.8.1.2 on page 91
  - **Stage 18. Behavioral Taxonomy:** Cf. *Entry* [ι 24,π 17], *Method* Sect. 3.8.1.3 on page 91 and *Example* Sect. 4.5.3 on page 104.
  - **Stage 19. Behavioural Signatures:** Cf. *Entry* [ι 22,π 17], *Method* Sect. 3.8.1.4 on page 92 and *Example* Sect. 4.5.4 on page 105.
  - **Stage 20. Behaviour Definitions:** Cf. *Entry* [ι 21,π 17], *Method* Sect. 3.8.1.5 on page 92 and *Example* Sect. 4.5.4 on page 105.
  - **Stage 21. Domain Instantiation:** Cf. *Entry* [ι 72,π 24], *Method* Sect. 3.8.1.6 on page 93 and *Example* Sect. 4.5.5 on page 108.

3.3 The "Dashboard"

The DOMAIN Method

Stage: 0 ***The Dashboard***

260. To "perform" discover_domain properly the dashboard state θ:Θ must be initialized.

(a) $\theta_{esn} : \Theta_{ESN}$ *is to hold a list of parts and their type under examination. This list will grow during "ontology traversal" examination of the of the domain.*

(b) $\theta_{dsu} : \Theta_{DSU}$ *is to hold the list of all domain specification unitss as they are issued during the analysis & description procedure. The list grows.*

(c) $\theta_{vde} : \Theta_{VDE}$ *is to hold the list of RSL*$^+$ `texts` *of value definitions.*

(d) $\theta_{beh} : \Theta_{BEH}$ *is to hold the set of sort (names) of those [parts] which are to be "morphed" into behaviours. The set is set during procedure.*

type
260a. Θ_{ESN} = (P×S)* [Pairs of values and endurant names]
260b. Θ_{DSU} = DSU* [Domain Description Units]
260c. Θ_{VDE} = RSL$^+$text* [Value definitions]
260d. Θ_{BEH} = S-set [Behaviour Sorts]
variable
260a. $\theta_{esn} : \Theta_{ESN} := \langle\rangle$
260b. $\theta_{dsu} : \Theta_{DSU} := \langle\rangle$
260c. $\theta_{vde} : \Theta_{VDE} := \langle\rangle$

260d. $\theta_{beh} : \Theta_{BEH} := \{\}$

3.4 Universe of Discourse

The DOMAIN Method

Stage: 1 ***Universe of Discourse***

261. We define the analysis & description procedure.

(a) The analyser cum describer choose a domain, and names it, say UoD.

(b) A display line ***The Universe-of-Discourse:*** *prefixes the domain specification units, and*

(c) the dashboard state θesn is initialised.

(d) And that is it, for now!

```
value
261.  discover_initialization_of_domain: Unit → Unit
261.  discover_initialization_of_domain() ≡
261a.     let (text,uod:S) [ analyser cum describer choices ] in
261b.           { θdsu := ⟨ (" The Universe-of-Discourse:, text, value uod:𝕊 ") ⟩
261c.           ‖ θesn := ⟨ (text, value uod:𝕊) ⟩ } "
261d.     end
```

3.5 Discover Domain

The DOMAIN Method

Stage: 2 ***Discover Domain*** *There is, thus, a stage, say, discover_domain, which takes no argument, but delivers a domain description of a domain which that procedure elicits!*

262. The discover_domain procedure takes no explicit arguments and "delivers" its result deposited in $\theta_{dsu} : \Theta_{DSU}$. That is: the analyser cum describer, of course, work in the environment of the domain.

(a) First the domain must be selected, that is, the universe of discourse must be chosen.

(b) Then the external qualities must be analysed and described.

(c) Then the internal qualities must be analysed and described.

(d) Finally the perdurants must be analysed and described.

```
262.  discover_domain: Unit → Unit
262.  discover_domain() ≡
262a.     initialise_domain();
```

```
262b.  discover_external_qualities() ;
262c.  discover_internal_qualities() ;
262d.  discover_perdurants()
```

3.6 External Qualities

3.6.1 The Endurants

The DOMAIN Method

Stage: 3 ***External Qualities***

263. *The discover_external_qualities*
   (a) *starts by discovering endurants;*
   (b) *then goes on to describe a domain endurant-taxonomy;*
   (c) *and to discover endurant states.*

```
value
263.  discover_external_qualities: Unit → Unit
263.  discover_external_qualities() ≡
263a.     discover_endurant_descriptions() ;
263b.     describe_taxonomy() ;
263c.     describe_endurant_states()
```

3.6.1.1 Endurant Descriptions

The DOMAIN Method

Stage: 4 ***Endurant Descriptions***

264. *The examination (i.e., analysis & description) of the domain wrt. external qualities takes place in the/a current state* $\theta : \Theta$.
   (a) θ_{esn} *"holds" the list of names of the sorts yet to be examined. For every sort examination that list either grows or shrinks. It grows when examining compound sorts. It shrinks when examining atomic sorts. So as long as there are names in the list there is examination to do !*
   (b) *The endurant part,* p*, to be examined is selected.*
   (c) *It is then described – into RSL*$^+$ `text`.
   (d) *And* p *is removed from* θ_{esn}.

```
264.  discover_endurant_descriptions: Unit → Unit
```

```
264.  discover_enduran_descriptionts() ≡
264a.     for i=1 to len co θesn do
264b.     let (p,E) = co θesn[ i ] in
264c.     traverse(p,E) end
264a.     end
```

The **co** θ_{esn} in the **for** i=1 **to len co** θ_{esn} **do** loop is to be evaluated "every time around the loop" as θ_{esn} changes within the loop !

265. The traverse step, as the name suggests, figuratively traverses the larger, **blue** lined box of Fig. 1 on page 25 – while observing a part p of type E.

We read that traversal:

(a) If p is an entity
(b) then
(c) if p is an endurant
(d) then
(e) if p is a solid
(f) then
(g) if p is a part
(h) then
(i) if p is atomic
(j) then handle that atomic
(k) else
(l) if it is a Cartesian
(m) then handle the Cartesian
(n) else handle the Part set;
(o) else it is a living species[56];
(p) else it is a fluid[57]
(q) else it is a perdurant ![58]

```
value
265.  traverse: (P×E) → Unit
265.  traverse(p,E) ≡
265a.     if is_entity(p)
265b.        then
265c.           if is_endurant(p)
265d.              then [axiom: is_entity(p)]
265e.                 if is_solid(p)
265f.                    then [axiom: is_endurant(p)]
265g.                       if is_part(p)
265h.                          then
265i.                             if is_atomic(p)
265j.                                then handle_atomic(p,E)
265k.                                 else [axiom is_compound(p)]
265l.                                   if is_Cartesian(p)
265m.                                      then handle_Cartesian(p,E)()
265n.                                      else [axiom is_Part_set(p)]
265n.                                           handle_Part_set(p,E)()
265l.                                   end
265i.                             end
265o.                          else [axiom is_living_species(p)] skip
265g.                       end
265p.                    else [axiom is_fluid(p)] then skip
265c.                 end
265q.              else [axiom: is_perdurant(p)] skip
265c.           end
```

```
265a.      end
```

We mark in **red** the functions, i.e., the steps

266. The handle_atom(p,E) function joins its sort to those of the behaviourals.

```
value
266.   handle_atom(p,E) ≡ θ_beh := co θ_beh ∪ {E},
```

267. The handle_Cartesian function behaves as follows.

   (a) The analyser cum describer observes/decides that one can observed n distinct parts, each of it own sort;
   (b) "generates" narrative text "explaining" these parts, and
   (c) updates the dashboard state accordingly – prefixing the domain specification units with a display line: **Endurant Sorts:**.
   (d) The analyser cum describer finally decides on the possibly empty, possibly full, subset of the sorts of the n Cartesian components are behavioral.

```
value
267.     handle_Cartesian: (P × E) × Unit → Unit
267.     handle_Cartesian(p,E)() ≡
267a.       let ((p1,E1),(p2,E2),...,(pn,En)) = observe(p) in
267b.       let (txt1,txt2,...,txtn) = analyser_cum_describer_choice(p) in
267c.       θ_esn := co θ_esn ⌢ ⟨ (p1,E1),(p2,E2),...,(pn,En) ⟩ ;
267c.       θ_dsu := co θ_dsu ⌢ ⟨" Endurant Sorts:
267c.                                Narrative:
267c.                                  1. txt1,
267c.                                  2. txt2,
267c.                                  ...
267c.                                  n. txtn.
267c.                                Formalisation:
267c.                                  type
267c.                                    1. E1, 2. E2, ..., En
267c.                                  value
267c.                                    1. obs_E1: E→E1
267c.                                    2. obs_E2: E→E2,
267c.                                    ...
267c.                                    n. obs_En: E→En " ⟩ ;
267d.       θ_beh := co θ_beh ∪ select_Cartesian_endurants({E1,E2,...,En}) ,
267.          end end

267d.   select_Cartesian_endurants: E-set → E-set
267d.   select_Cartesian_endurants({E1,E2,...,En}) ≡
```

267d. {Ei,Ej,...,Ek}[59] **where** {Ei,Ej,...,Ek} ⊆ {E1,E2,...,En}

268. If the part is considered a part set

(a) then actions very similar to those for Cartesian parts are "performed" by the analyser cum describer.

```
value
268.   handle_Part-set: (P × E) × Unit → Unit
268.   handle_Part-set(p,E)() ≡
268a.    let (ps:PS) = observe(p) axiom ps ≠ {}  in
268a.    let p:P • p ∈ ps  in
268a.    let (text_ps,text_p) = analyser_cum_describer_Choice(ps,p)  in
268a.    θesn := co θesn ⌢ ⟨(p,P)⟩ ;
268a.    θdsu := co θdsu ⌢ ⟨ " Endurant Sorts:
268a.                          Narrative:
268a.                            1. text_ps
268a.                            2. text_p
268a.                          Formalisation:
268a.                            type
268a.                              1. PS = P-set[60],
268a.                              2. P
268a.                            value
268a.                              1. obs_PS: E → PS " ⟩ ;
268a.    θbeh :=
268a.      co θbeh ∪ select_Part_set_endurant(P) ;
268.     end end end

268a.  select_Part_set_endurant: P → P-set
268a.  select_Part_set_endurant(P) ≡
268a.    if is_behavioral(P) then {P} else {} end
```

And so forth. For details see [28].

3.6.1.2 Endurant Taxonomy

The DOMAIN Method

Stage: 5 ***Endurant Taxonomy***

269. *An endurant taxonomy domain specification units starts with a* ***Taxonomy*** *display line, usually prefixed by the name, E, of the "overall", i.e., a "main" part that is "at the root" of the taxonomy, i.e., the usually compound part whose siblings – and again their siblings, etc., till we "reach", as leaves of the taxonomy tree, the atomic parts.*

(a) *The TAX_DSU tree-like diagram has as its root a box labeled with E.*

(b) *In general now, if the taxonomy box stands for an atom then we leave it there: no taxonomy sub-tree emanating from that box.*

(c) *If the taxonomy box, E, stands for a Cartesian endurant with components E1, E2, ..., En then we draw, on the same horizontal line below box E, n boxes, connected to box E by straight, usually slanted lines, n boxes, left-to-right, labeled E1, E2, ..., En.*

(d) *If the taxonomy box, E, stands for a part set, then we draw, below it, a box labeled with the part set name, f.ex., PS, observed from E – connected to box E by a vertical line.*

(e) *This is followed by drawing two or three boxes, "immediately" below box PS and on the same horizontal line and all identically labeled by the same endurant sort name, say P, where P is the type occurring in PS = P-**set** observed from E. n boxes, connected to box E by straight, usually slanted lines, n boxes, left-to-right, labeled E1, E2, ..., En.*

(f) *The above diagram construction instructions are followed till all domain endurants "derivable" from E have been followed.*

```
value
269. describe_endurant_taxonomy: Unit → Unit
269. describe_endurant_taxonomy() ≡
269.     let endurant_taxonomy = design_endurant_taxonomy() in
269.     θdsu := co θdsu ⌢
269.         ⟨ " Endurant Taxonomy: endurant_taxonomy " ⟩
269.     end
```

Here design_endurant_taxonomy is a "function" which is performed by the analyser cum describer.

3.6.1.3 Endurant States

The DOMAIN Method

Stage: 6 ***Endurant States***

270. The discover_endurant_state stage

(a) *initially appends an* **Endurant State**: *prefix and*

(b) *the RSL⁺ `text` **value** to θ_{dsu};*

(c) *then proceeds, for all (part,sort)s that have been "discovered" and therefor are in θ_{ves},*

(d) *to select these elements of θ_{ves},*

(e) *and append them, as domain specification unitss to θ_{dsu}.*

271. Then a notion of "global" σtates is introduced:

(a) *σ_{parts} – the union of all parts, and*

(b) $\sigma_{behavioural_{parts}}$ *the union of behavioural all parts.*

```
270.  discover_endurant_state: Unit → Unit
270.  discover_endurant_state() ≡
270a.     θdsu := co θdsu ⌢ " ⟨ Endurant State: value ⟩ " ;
270c.     for i=1 to len θves do
270d.        let (p,E) = θves[i] in
270e.        θdsu := co θdsu ⌢ " ⟨ ev:E = p ⟩ " ;
271.         θves := co θves ⌢ " ⟨ ev:E = p ⟩ " ;
271a.        σall_parts = [ ∪ of ∀ parts ] ,
271b.        σall_behavioural_parts = [ ∪ of ∀ hehaviourable parts ] "
270c.     end end
```

Here ev is a suitable, distinct value name chosen by the analyser cum describer.

3.7 Internal Qualities

The DOMAIN Method

Stage: 7 ***Internal Qualities***

272. The discover_internal_qualities procedure generates domain specification unitss for the

(a) unique identification,

(b) mereologies,

(c) attributes and

(d) for possible intentional pull descriptions/

– and in that order!

```
272.  discover_internal_qualities: Unit → Unit
272.  discover_internal_qualities() ≡
272a.     describe_identification() ;
272b.     discover_mereologies() ;
272c.     discover_attributes() ;
272d.     discover_intentional_pulls()
```

3.7.1 Unique Identification

The DOMAIN Method

Stage: 8 ***Unique Identification***

273. The describe_identification procedure generates domain specification unitss for the

(a) unique identifiers,

(b) unique identifier states,

(c) uniqueness of parts,

– and in that order!

```
273.  describe_identification() ≡
273a.       describe_unique_identifiers() ;
273b.       describe_unique_identifier_states() ;
273c.       describe_part_uniqueness() ;
```

3.7.1.1 Unique Identifiers

The DOMAIN Method

Stage: 9 ***Unique Identifiers***

274. The describe_unique_identifiers procedure is to generate

(a) a domain specification units **Unique Identification**: *display line and*

(b) records the **type** *and* ***uid_****E observers for all "behavioral" parts, that is: those parts which the analyser cum describer considers candidate to be "morphed" into behaviours – those which are dashboard recorded in* θ_{beh}:Θ_{BEH}.

i. For all E noted in the dashboard as behavioral

ii. the **type** *of the unique identifier and*

iii. the **value** *and its observer*

iv. is added to the emerging domain description θ_{dsu}.

```
274.   describe_unique_identifiers() ≡
274a.       θdsu := co θdsu ⌢ ⟨ Unique Identification: ⟩ ;
274a.       for ∀ E in co θbeh do
274(b)iv.         θdsu :=
274(b)iv.             co θdsu ⌢
274(b)ii.               ⟨ { " type
274(b)ii.                       EI
274(b)iii.                    value
274(b)iii.                       uid_E: E → EI "
274(b)i.                | (_,E) ∈ elems θesn } ⟩
274.       end
```

3.7.1.2 Unique Identifier States

The DOMAIN Method

Stage: 10 ***Unique Identifier States***

275. The describe_unique_identifier_states stage

- *(a) augments θ_{dsu} with a* **Unique Identifier States** *display line; followed by a*
- *(b) for all (p,E) known such in θ_{esn}*
- *(c) with "their"* **value**
- *(d) contribution to the unique identifier states; followed by*
- *(e) a $\sigma_{uid_{all\_parts}}$ definition, and*
- *(f) a $\sigma_{uid_{all\_behav\_parts}}$ definition.*

describe

```
275.  describe_unique_identifier_states() ≡
275a.     θdsu := co θdsu ⌢ ⟨ Unique Identifier States: ⟩ ;
275b.     for ∀ (p,E) in co θesn do
275a.        θdsu :=
275c.           co θdsu ⌢ ⟨ value ⟩
275d.            ⌢ ⟨ { " puid:EI = uid_E(p), " | (p,E) ∈ elems θesn } ⟩
275e.            ⌢ ⟨ " σuid_all_parts:UI-set = [ ... ] " ⟩
275f.            ⌢ ⟨ " σuid_all_behav_parts:UI-set = [ ... ] " ⟩
275b.     end
```

3.7.1.3 Uniqueness

The DOMAIN Method

Stage: 11 ***Uniqueness***

276. The all parts are unique **axiom** *is simple: the number of all parts equals the number of all part [unique] identifiers.*

axiom

276. **card** σ_{all_parts} = **card** $\sigma_{uid_{all\_behav\_parts}}$

3.7.2 Mereology

The DOMAIN Method

Stage: 12 ***Mereology***

277. *The mereology procedure is to generate a domain specification units which records the type and* ***obs_****E observers for all "behavioral" parts, that is: those parts which the analyser cum describer considers candidate to be "morphed"into behaviours – those which are dashboard recorded in* θ_{beh}:Θ_{BEH}.
   (a)
   (b) *For all E noted in the dashboard as behavioral the*
   (c) **type** *and*
   (d) **value**
   (e) *of the mereology definition and its observer is added to the emerging domain description.*

```
277.  discover_mereologies: Unit → Unit
277.  discover_mereologies ≡
277a.    θdsu := co θdsu ⌢ ⟨ Mereologies: ⟩ ;
277b.    for ∀ E in co θbeh do
277e.      θdsu :=
277e.          co θdsu ⌢
277c.            ⟨ " type
277c.                MereoE = ℳE(UI1,UI2,...,UIn)
277d.              value
277d.                mereo_E: E → MereoE  " ⟩
277.   end
```

3.7.3 Attributes

The DOMAIN Method

Stage: 13 ***Attributes***

278. *The* **attr***ibutes procedure is to generate*

279. *a domain specification units which records the type and* ***attr_****A observers for all "behavioral" parts, that is: those parts which the analyser cum describer considers candidate to be "morphed" into behaviours – those which are dashboard recorded in* θ_{beh}:Θ_{BEH}.
   (a) **Attributes**: *prefixes the individual attribute type and observer definitions.*
   (b) *For all E noted in the dashboard as behavioral the analyser cum describer*
   (c) *analyses that type to have a number of attributes (A1,A2,...,An);*
   (d) *with these possessing respective properties (AttrType_1,AttrType_2,...,AttrType_n).*
   AttrType_i are of the form:
      - *AttrType_i = AttrType_Expr_i × Attr_Cat [× SI_Unit]*

- *Attr_Cat* = ***sta***|***mon*** | ***prg***
- *SI_Unit* =

The [...] in [× SI_Unit] mean that this term is optional. Not all attributes can be given an SI Unit[61]

sta *refers to static attributes,* ***mon*** *refers to static attributes, and* ***prg*** *refers to programmable attributes.*

(e) θ_{dsu} *is therefore extended with a domain specification units with the list element containing*

(f) *attribute* **type** *definitions and*

(g) **value** *definitions of the* **attr***ibute observers.*

```
278.  discover_attributes: Unit → Unit
278.  discover_attributes() ≡
279a.     θdsu := co θdsu ⌢ ⟨ Attributes: ⟩ ;
279b.     for ∀ E ∈ co θbeh do
279c.        let (A1,A2,...,An) = observe_attribute_names(p:E) in
279d.        let (AttrType_1,AttrType_2,...,AttrType_n) = observe_attribute_types(p:E) in
279e.        θdsu := co θdsu ⌢
279f.           ⟨ " type
279f.                 A1 = AttrType_1, A2 = AttrType_2, ..., An = AttrType_n
279g.             value
279g.                 attr_A1: E → AttrType_1,
279g.                 attr_A2: E → AttrType_2,
279g.                 ...
279g.                 attr_An: E → AttrType_n " ⟩
278.     end end end
```

Attribute types AttrType_i are type expressions over "standard" RSL definable types, unique identifier sorts and mereo types.

Red "formulas" indicate work to be done by the analyser cum describer.

• • •

Important attributes, ones that can be ascribed to [almost all] parts, E, is the **E_history** attributes. They may not be observable, not physically measurable, but they can be *"remembered"*, can be *"talked about"* – say by humans. They therefore objectively, indisputably records the occurrence of **events**. In our domain models we record E_history attributes as ***sequences*** *of* 𝕋𝕀𝕄𝔼*-stamped events*. Examples are the events of automobiles deciding to remain at hubs or on links, or stopping, or leaving or entering these, etc.; cf. Item 336 on page 99.

3.7.4 Intentional Pulls

The 𝔻𝕆𝕄𝔸𝕀ℕ Method

Stage: 14 ***Intentional Pulls***

280. *The discover_intential_pulls procedure [possibly] generates an "Intentional Pull" domain specification units with zero, one or more, n, intentional pulls.*
   (a) θ_{dsu} *is extended*
   (b) *by an* **axiom** *and*
   (c) n, $n \geq 0$, *intentional pull predicates.*

   The $\mathcal{B}(...)$ terms are $\mathcal{B}$oolean predicates.

```
280.  discover_intentional_pulls: Unit → Unit
280.  discover_intentional_pulls() ≡
280a.    θdsu := co θdsu ⌢ ⟨ " Intentional Pulls:
280b.          axiom
280c.            Bp1(...) ⇒ Bq1(...)
280c.            ≡
280c.            Bx1(...) ⇒ By1(...) ,
280c.
280c.            Bp2(...) ⇒ Bq2(...)
280c.            ≡
280c.            Bx2(...) ⇒ By2(...) ,
280c.
280c.            ...
280c.
280c.            Bpn(...) ⇒ Bqn(...)
280c.            ≡
280c.            Bxn(...) ⇒ Byn(...)   "
280b.       ⟩
```

The **red** formulae designates work to be done by the analyser cum describer.
[Hard work !]

3.8 Perdurants

3.8.1 Disccover Perdurants

The DOMAIN Method

Stage: 15 ***Disccover Perdurants***

281. *The discover_perdurant stage has four sub-stages:*
   (a) *the description of channels,*
   (b) *the characterisation of behaviour actions,*
   (c) *the discovery of behaviours, and*

(d) the description of domain initialization;

```
281.  discover_perdurants: Unit → Unit
281.  discover_perdurants() ≡
281a.     describe_channel() ;
281b.     characterisation_of_actions() ;
281c.     discover_behaviours() ;
281d.     describe_initialization()
```

3.8.1.1 Channels

The DOMAIN Method

Stage: 16 ***Channels***

282. The describe ***channel*** *procedure is very straightforward:*

(a) θ_{dsu} *is extended with the declaration of a* ***channel*** *array*

(b) named ch, where the distinct array indexes, ui, uj [ui $\neq$ *uj] range over the unique identifiers of all behavioral parts.*

```
282.  discover_channel: Unit → Unit
282.  describe_channel() ≡
282a.     θdsu := co θdsu ⌢ " Channels:
282b.         channel { ch[{ui,uj}] | ui,uj:UI • {ui,uj}⊆σuid_parts }: M "
```

M is a type expression referring to values other than sort names, variables, ... MORE TO COME ?

3.8.1.2 Actions

The DOMAIN Method

Stage: 17 ***Actions*** *Behaviours, when observed, generally consists of sets of sequences of actions, events and behaviours. We shall "restrict" this characterization to not allow "embedded" behaviours. That is: In our understanding of the kind of domains that we have "limited' ourselves – i.e., the DOMAIN method – to deal with:*

- *(i) any behaviour is exclusively the result of a transcendental deduction of a specific part sort,*
- *(ii) and such behaviours do not invoke other part behaviours.*

3.8.1.3 Behavioural Taxonomy

The DOMAIN Method

Stage: 18 ***Behavioural Taxonomy***

283. *A behavioural taxonomy is a pictorial rendition, a two-dimensional graph, which shows "all" the behaviours, say as circles or [rounded] boxes with arrows between these. The arrows designate communications between behaviours.*

```
value
283. describe_behavioural_taxonomy: Unit → Unit
283. describe_behavioural_taxonomy() ≡
283.    let behavioural_taxonomy = design_behavioural_taxonomy() in
283.    θ_dsu := co θ_dsu ⌢
283.       ⟨ " Behavioural Taxonomy: behavioural_taxonomy " ⟩
283.    end
```

3.8.1.4 Signatures and Specifications

The DOMAIN Method

Stage: 19 ***Behaviour Signatures and Specifications*** *Behaviours transpire, as I say, by transcendental deduction, that is, are "morphed" from behaviourable parts. There is one behaviour for each such part. Their signatures follows "straight" from the composition of the internal qualities of the endurant parts from which they are "morphed":*

- *Their name is usually chose to relate to the name of the part sort from which they emerge.*
- *A first argument, by convention, is the unique identifier of "the part".*
- *A second argument is the mereology of the part.*
- *A third argument is the zero, one or more static attribute values.*
- *A possible fourth argument the names zero, one or more monitorable attribute.*
- *A "final" argument is one or more programmable attribute values.*
- *The result "value" of a behaviour is the* **Unit** *value, ().*

3.8.1.5 Behaviour Definitions

The DOMAIN Method

Stage: 20 ***Behaviour Definitions***

284. *The discover_behaviours stage "takes" the state of the dashboard and updates that dashboard.*

285. *First a* **Behaviour Definitions** *display line; then*

286. for all behavioural parts

- *(a) the analyser cum describer analyses and describes the [thus transcendentally "morphed"] part into a behaviour definition.*
- *(b) The general "pattern" of behaviour definitions start with a signature – [where we often find that we can omit the monitorable attributes] – and and a "token invocation", followed by ≡ – i.e., the definition.*
- *(c) The "body" of behavioural definitions is either of a simple form:*
  - *i. B is a function, e.g., a behaviour which terminates delivering as its result a value, called* state*.*
  - *ii. From* state *one can "extract" updated values and the mereology of behaviour ui – for the case that* behaviour *introduces new, or removes existing behavioural parts – and one or more of the programmable attributes.*
  - *iii. whereupon behaviour resumes being the behaviour, but with an updated "state"!*
- *(d) Or B follows the patterns illustrated above are "explained otherwise"!*

```
284.  discover_behaviours: Unit → Unit
284.  discover_behaviours() ≡
285.    θ_dsu := co θ_dsu ⌢ ⟨ " Behaviour Definitions: " ⟩ ;
286.    for ∀ E ∈ co θ_beh do
286.      θ_dsu := co θ_dsu ⌢
286.        ⟨ let p:E • [ where p ≡ recorded in θ as being of type E ] in
286a.         " behaviour: UI → Mereo → StatAttrs → [ MoniAttrs → ] ProgrAttrs → Unit
286b.           behaviour(ui)(mereo)(sta_atts)(moni_attrs)(prgr_attrs) ≡[62]
286(c)i.        let state = B(ui)(mereo)(sta)[ (mon) ](prgr) in
286(c)ii.       let (mereo′,prgr′) = analyse(state) in
286(c)iii.      behaviour(ui)(mereo′)(sta_atts)(moni_attrs)(prgr′) end end "  end ⟩
286.    end
286d.  or: ["explained otherwise"]
```

3.8.1.6 Domain Instantiation

The DOMAIN Method

Stage: 21 ***Domain Instantiation***

287. The describe_initialisation procedure stage first generates

- *(a) a display line:* **Domain Initialisation:**
- *(b) then, for each behavioural part, p:E, that has been encountered, i.e., each part in* $\sigma_{uid_{parts}}$
- *(c) the parallel composition of behaviour invocations, be(uid)(mereo)(sta)(moni)(prg), where the name, be, of the behaviour is a suitable name that has some connotation to the part*

[62] The analyser cum describer must, based on the internal qualities defined for E "fill in" the details of Mereo, StaticAttrs, MoniAttrs and ProgrammableAttrs. It should be obvious (!) how to do that. It is too cumbersome to lay that out here.

type E,

(d) where the unique identification, uid, is obtained from p:E,

(e) the mereology, mereo, likewise,

and so

(f) the static attribute values, sta,

(g) the monitorable attribute names, moni, and

(h) the programmable attribute values, prgr.

```
value
287.  describe_initialisation: Unit → Unit
287.  describe_initialisation() ≡
287a.     θdsu := co θdsu ⌢ ⟨ " Domain Initialisation: " ⟩
287c.     ⌢ ⟨ "  ‖ { be
287d.                  (uid_E(p))
287e.                     (mereo_E(p))
287f.                        ( { attr_A(p) | A ∈ static_attrs(p) } )
287g.                           ( { ηA[63]| A ∈ moni_attrs(p) } )
287h.                              ( { attr_A(p) | A ∈ prgr_attrs(p) } ) "
287b.               | (p,E) ∈ elems θves ∧ E ∈ co θbeh } ⟩
```

The functions static_attrs, moni_attrs and prgr_attrs follow from the category of the attribute definitions for p:E.

4 An Example

4.1 Universe of Discouse

288. The universe of discourse is that of
289. road traffic, RT –
290. whose narrative is then presented.

288. **universe of discourse**:
289. **type** RT
290. **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.

4.2 Endurants

4.2.1 External Qualities

4.2.1.1 Observing Endurants

291. In the road traffic domain we can observe a road net aggregate and an automobile aggregate.
292. In a road net aggregate we can observer an aggregate of hubs and an aggregate of links.
293. In an aggregate of automobiles we can observe a set of [atomic] automobiles.
294. Aggregates of hubs ad links are sets of [atomic] hubs and links.

type
291. RNA, AA
292. AH, AL
293. AS = A-**set**
294. HS = H-**set**, LS = L-**set**

value
291. **obs_**RNA: RT → RNA, **obs_**AA: RT → AA
292. **obs_**AH: RNA → AH, **obs_**AL: RNA → AL
293. **obs_**AS: AA → AS
294. **obs_**HS: AH → RS, **obs_**LS: AL → LS

4.2.1.2 Endurant States

295. There is the road transport [universe of discourse].
296. There is the road net aggregate.
297. There is the automobile aggregate.
298. There is the hub aggregate.
299. There is the link aggregate.
300. There is the set of automobiles.
301. There is the set of hubs.
302. There is the set of links.
303. And there is the entire set of all of these.
304. And there is the entire set of just all automobiles, hubs and links.

value
295. rt:RT
296. rna:RNA = **obs_**RNA(rt)
297. aa:AA = **obs_**AA(rt)
298. ah:AH = **obs_**AA(rna)
299. al:AL = **obs_**AL(rna)
300. as:AS = **obs_**AS(ah)
301. hs:HS = **obs_**HS(ah)
302. ls:LS = **obs_**LS(al)
303. σ_{parts} = {rt,rna,aa,ah,al} ∪ aa ∪ as ∪ hs ∪ ls
304. σ_{atoms} = aa ∪ hs ∪ ls

4.2.1.3 Endurant Taxonomy

See Fig. 3.

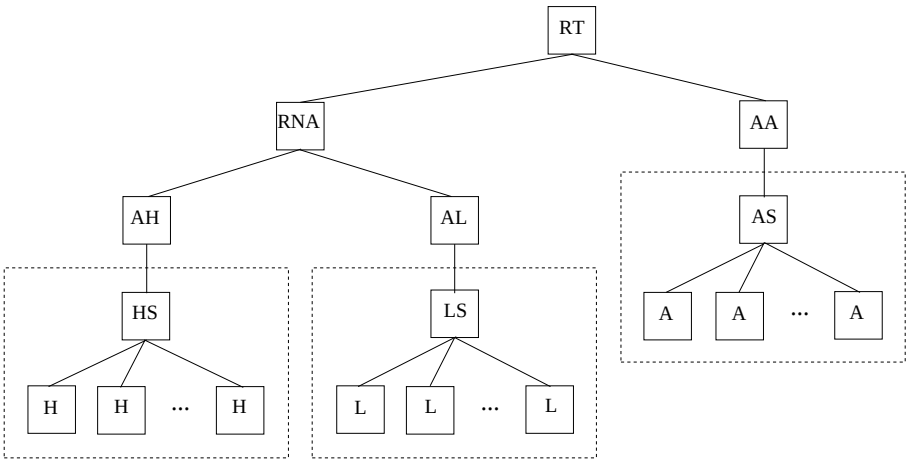


Figure 3: A Road Transport Taxonomy

4.2.2 Internal Qualities

4.2.2.1 Unique Identification

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

305. The universe of discourse has a unique identifier.

306. The road net agregate has a unique identifier.
307. The automobile aggregate has a unique identifier.
308. The hub aggregate has a unique identifier.
309. The link aggregate has a unique identifier.
310. Each hub has a unique identifier.
311. Each link has a unique identifier.
312. Each automobile has a unique identifier.

type
305. RTI
306. RNAI
307. AAI
308. HAI
309. LAI
310. HI
311. LI
312. AI

value
305. **uid_**RT: RT → RTI
306. **uid_**RNA: RNA → RNAI
307. **uid_**AA: AA → AAI
308. **uid_**HA: HA → HAI
309. **uid_**LA: LA → LAI
310. **uid_**H: H → HI
311. **uid_**L: L → LI
312. **uid_**A: A → AI .

4.2.2.1.1 Unique Identifier States

313. There is the unique identifier of the road transport.
314. There is the unique identifier of the road net aggregate.
315. There is the unique identifier of the autombile aggregate.
316. There is the unique identifier of the hub aggregate.
317. There is the unique identifier of the link aggregate.
318. There are the unique identifiers of the automobiles of the set of autombiles.
319. There are the unique identifiers of the hubs of the set of hubs.
320. There are the unique identifiers of the links of the set of links.
321. And there is the entire set of all of these.
322. And there is the entire set of all identifiers of the automobiles, hubs and links.

value
313. rt_{uid}RT: RT = **uid_**RT(rt)
314. rna_{uid}RNA: RNA = **uid_**RNA(rt)
315. aa_{uid}AA: AA = **uid_**AA(rt)
316. ah_{uid}AH: AH = **uid_**AA(rna)
317. al_{uid}AL: AL = **uid_**AL(rna)
318. as_{uid}AS: AS = {**uid_**A(a)|a:A•a∈as}

319. hs_{uid}:HS = {**uid_**H(h)|h:H•h∈hs}
320. ls_{uid}:LS = {**uid_**L(l)|l:L•a∈ls}
321. $\sigma_{parts_{uid}}$ = {rt,rna,aa,ah,al} ∪ aa ∪ hs unon ls
322. $\sigma_{atoms_{uids}}$ = aa ∪ hs ∪ ls

4.2.2.1.2 Uniqueness of Endurants

323. Parts are uniquely identfied.

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

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

324. The mereology of links is a 2 element set of hub identifiers of the road net[64].
325. The mereology of a hub is a possibly empty set of hub identifiers of the road net.
326. The mereology of an automobile is [some subset of] a set of hub and link identifiers[65]

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

value
324. **mereo_**L: L → ML
325. **mereo_**H: H → MH
326. **mereo_**A: A → MA .

4.2.2.3 Attributes Example attributes are:

- **Hubs:**
  327. 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.
  328. Hubs have state spaces, hω:HΩ: the set of hub states "signaling" which states are open/closed, i.e., **green**/**red**.
- **Links:**
  329. Links that have lengths, LEN;

[64] This is a simplified version: it allows for automoblie traffic in both directions of the link. We leave it to the reader to "cook" up othe such traffic possibilities.
[65] – a full set means that the specific automobile is allowed to travel all over the net.

330. 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.
331. Links have state spaces, lω:LΩ: the set of link states "signaling" which states are open/closed, i.e., **green**/**red**.

- **Automobiles:**

332. 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.
333. Automobiles have velocity;
334. Automobiles have acceleration;
335. Etc.[66]

- **Hubs, Links, Automobiles:**

336. 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.
337. Link positions have well-defined identifiers and fractions.

type
327. HΣ = (LI×LI)-**set** [**prg**]
328. HΩ = HΣ-**set** [**sta**]
329. LEN = **Nat** [**sta**] [m]
330. LΣ = (HI×HI)-**set** [**prg**]
331. LΩ = LΣ-**set** [**sta**]
332. APos = atH | onL [**prg**]
332a. atH :: HI
332b. onL :: LI × (fhi:HI × f:**Real** × thi:HI)
333. Veloc = **Real** [**mon**] [km/h]
334. Accel = **Real** [**mon**] [m/sec]
335. ...
336. HHis,LHis = ($\mathbb{TIME}$×AI)* [**prg**]
336. AHis = ($\mathbb{TIME}$×(HI|LI))* [**prg**]

value
327. attr_HΣ: H → HΣ
328. attr_HΩ: H → HΩ
329. attr_LEN: L → LEN
330. attr_LΣ: L → LΣ
331. attr_LΩ: L → LΩ
332. attr_APos: A → APos
333. attr_Veloc: A → Veloc
334. attr_Accel: A → Accel
336. attr_HHis: H → HHis
336. attr_LHis: L → LHis
336. attr_AHis: A → AHis

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

[66]

4.2.2.4 Intentional Pull

338. 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.
339. 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
338a. ∀ a:A • a ∈ as ⇒
338a. **let** ahist = attr_AHist(a) **in**
338a. ∀ ui:(LI|HI) • ui ∈ **dom** ahist ⇒
338b. ∀ τ:𝕋𝕀𝕄𝔼 • τ ∈ **elems** ahist(ui) ⇒
338b. **let** hist = is_LI(ui) → attr_LHist(retr_L(ui))(σ),
338b. _ → attr_HHist(retr_H(ui))(σ) **in**
338b. τ ∈ **elems** hist(uid_A(a)) **end end**
339. ∧
339c. ∀ u:(L|H) • u ∈ ls∪hs ⇒
339c. **let** uhist = attr(L|H)Hist(u) **in**
339d. ∀ ai:AI • ai ∈ **dom** uhist ⇒
339d. ∀ τ:𝕋𝕀𝕄𝔼 • τ ∈ **elems** uhist(ai) ⇒
339d. **let** ahist = attr_AHist(retr_A(ai))(σ) **in**
339d. τ ∈ **elems** uhist(ai) **end end**

4.2.2.5 Manifestation and Mobility

340. Hubs, links and automobiles are manifest.
341. Hubs and links are stationary.
342. Automobiles are mobile.

axiom
340. ∀ e:(H|L|A) • **is_**manifest(e)
341. ∀ r:(H|L) • **is_**stationary(e)
342. ∀ a:A • **is_**mobile(e)

4.3 Auxiliary Types

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

343. A path element identifier is either a link identifier or a hub identifier.

344. A path (of a road net) is a finite[67] sequence of one or more alternating hub and link identifiers
345. 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.
346. Given a hub [a link] identifier we can retrieve the identified hub [link].

type
343. PEI = LI | HI
344. Path = PEI*
345. **axiom** [Well-formed Paths]
344. ∀ path:Path •
345a. ∀ {i,i+1}⊆**inds** path ⇒
345a. ((is_HI(path[i])∧is_LI(path[i+1]) ∨ is_LI(path[i])∧is_HI(path[i+1]))
345b. ∧ (path[i]∈hs_{uis}⇒path[i+1]∈ls_{uis}
345c. ∧ **uid_**H(retr_hub(path[i]))∈**mereo_**L(retr_hub(path[i])))
345d. ∧ (path[i]∈ls_{uis}⇒path[i+1]∈hs_{uis}
345e. ∧ **uid_**L(retr_link(path[i])∈**mereo_**H(retr_link(path[i])))))
value
346. retr_hub: HI → H, retr_link: LI → L, retr_unit: UI → U
346. retr_hub(hi) **as** h • h ∈ hs ∧ **uid_**H(h)=hi
346. retr_link(li) **as** l • l ∈ ls ∧ **uid_**L(l)=li
346. retr_unit(ui) **as** u • u ∈ hs∪ls ∧ **uid_**U(u)=ui
346. **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.

4.4 Simple Function Values

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

347. The paths[68] 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. 4.3 on the facing page.

    We define the paths function in two ways.

[67] We shall only consider finite paths. The `paths` function, Item 347 below, can easily be modified to yield also infinite length paths !

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

348. Either axiomatically
349. in terms of an **as** predicate, with the result being the "largest" such set all of whose paths satisfy the wellformedness criterion;
350. or inductively[69]:
    (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
347. paths: Unit → Path-infset
348. paths() as ps
349.     such that: ∀ p:ps satisfy the wellformedness of Sect. 4.3 on page 100
```

We can also express the `paths` functions explicitly:

```
value
347. paths: Unit → Path-infset
350. paths() ≡
350a.    let ps = {⟨ni⟩ | ni:NI ∈ hs_uis}∪{⟨ei⟩ | ei:EI ∈ ls_uis}
350(b)iv.          ∪ { pi⌢⟨ui⟩⌢⟨uj⟩⌢pj | pi⌢⟨ui⟩:Path-set, ⟨uj⟩⌢pj:Path-infset
350b.                • ({pi⌢⟨ui⟩,⟨uj⟩⌢pj}⊆ps)
350(b)i.             ∧ (ui∼ ∈ elems pj ∧ uj∼ ∈ elems pi)
350(b)iii.           ∧ (ui ∈ mereo_U(retr_unit(uj)))
350(b)ii.            ∧ (uj ∈ mereo_U(retr_unit(ui)))} in
350c.    ps end
type
347. U = H|L
```

Solution to the equation, lines 350a–350(b)i, is "'obtained' by a smallest set fix-point reasoning.

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

```
value
351. paths:Path-set = paths(g)
```

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

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

- 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.

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

theorem: [All finite paths have finite reverse paths]
352. ∀ g:G,p:Path•p ∈ paths(g) ⇒ rev_path(p) ∈ paths(g)
value
352. rev_path: P → P
352. rev_path(p) ≡
352. **case** p **of**
352. ⟨⟩ → ⟨⟩,
352. ⟨ui⟩ → ⟨ui⟩,
352. ⟨ui⟩⌢p′⌢⟨uj⟩ → ⟨uj⟩⌢rev_path(p′)⌢⟨ui⟩
352. **end**

We can define further functions. For example:

353. `path_length`,
354. `shortest_path`, etc.

value
353. path_length: P → LEN
353. path_length(p) ≡
353. **case** p **of**
353. ⟨⟩ → 0
353. ⟨ui⟩ → retr_path_length(ui),
353. ⟨ui⟩⌢p′ → retr_length(ui)+path_path_length(p′)
353. **end**
353. retr_path_length: UI → LEN
353. retr_path_length(ui) ≡ (is_EI(ui)→**attr_**LEN(retr_link(ui)),is_NI(ui)→0)

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

4.5 Perdurants

4.5.1 Channel

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

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

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

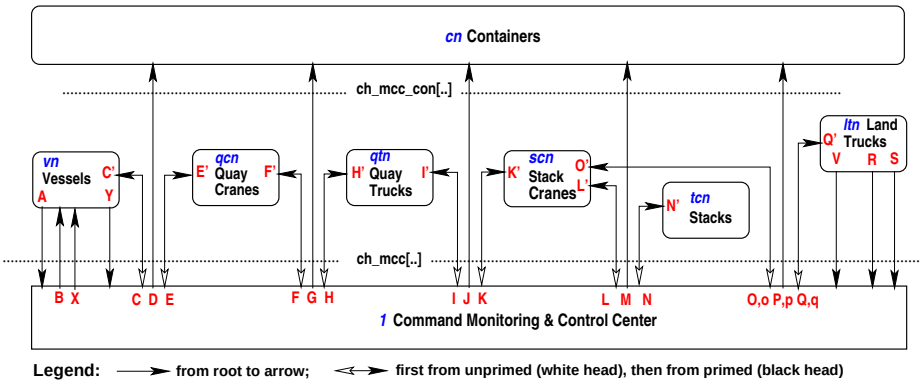
4.5.2 Variables

Not illustrated !

4.5.3 Behavioural Taxonomy

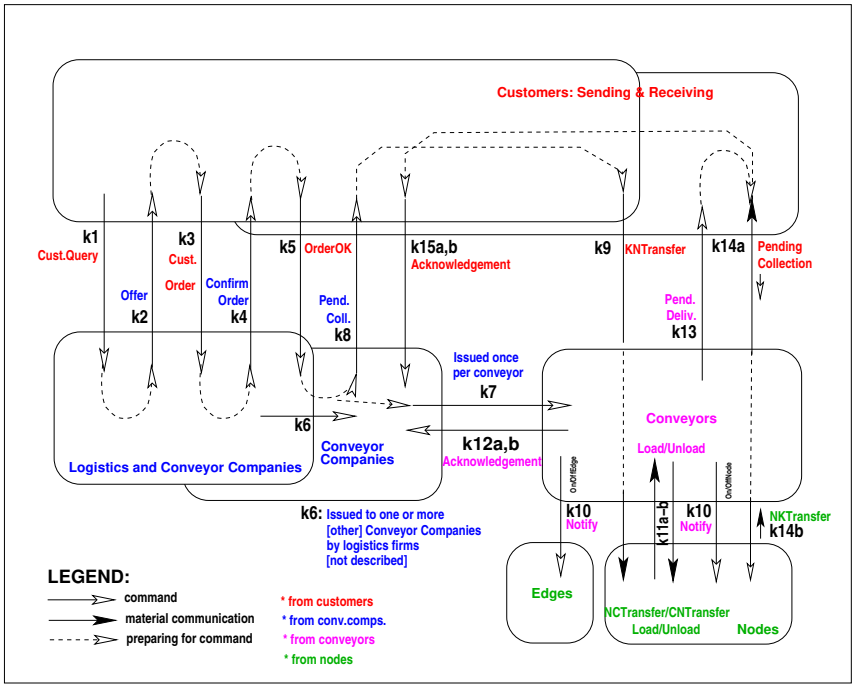
The examples are not of the road transport, RT !

4.5.3.1 A Container Terminal Port



Containers
Vessels, Quay Cranes, Quay Trucks, Stack Cranes, Stacks, Trucks
Each of the rounded boxes denote one or more parts
Command Center
[18] `www.imm.dtu.dk/~dibj/2018/yangshan/maersk-pa.pdf`

4.5.3.2 A Multi-mode Tansport System



Each of the rounded boxes designate one or more behaviours
The arrows likewise designate one or more communications
[25] `https://www.imm.dtu.dk/~dibj/2025/transport/main.pdf`

4.5.4 Behaviours: Signatures and Definitions

There are three behaviours:

357. `automobile`, corresponding to endurants `a:A`, and with appropriate argument types,
358. `hub`, corresponding to endurants `h:H`, and with appropriate argument types, and
359. `link`, corresponding to endurants `l:L`, and with appropriate argument types.

value
357. automobile: AI $\rightarrow$ AM $\rightarrow$ ... $\rightarrow$ (Apos $\times$ AHist) $\rightarrow$ **Unit**
358. hub: HI $\rightarrow$ HM $\rightarrow$ (HΩ $\times$ ...) $\rightarrow$ (HΣ $\times$ HHist) $\rightarrow$ **Unit**
359. link: LI $\rightarrow$ LM $\rightarrow$ (LEN $\times$ LΩ $\times$ ...) $\rightarrow$ (LΣ $\times$ LHist) $\rightarrow$ **Unit**

360. 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.
361. Any **hub** behaviour is "passively" awaiting **communication** from automobile behaviours as to their entering, leaving, or remaining at the hub.
362. Any **link** behaviour is "passively" awaiting **communication** from automobile behaviours as to their entering, leaving, or remaining on the link.

```
360.  automobile(ai)(ris)(...)(atH(hi),ahis)
360.  automobile(ai)(ris)(...)(onL(li,(fhi,f,thi)),ahis)
361.  hub(hi)(mh)(hω,...)(hσ,hhist)
362.  link(li)(ml)(len,lω,...)(lσ,lhist)
```

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

```
363  automobile(ai)(ris)(...)(atH(hi),ahis) ≡
363a     automobile_remain_at_hub(ai)(ris)(...)(atH(hi),ahis)
363b     ⌈⌉
363c     automobile_leaving_hub(ai)(ris)(...)(atH(hi),ahis)
363d     ⌈⌉
363e     automobile_stop(ai)(ris)(...)(atH(hi),ahis)
```

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

364. [363a] The automobile **remains** at a hub:
   (a) time is recorded,
   (b) informing the hub behaviour, whereupon
   (c) the automobile remains at that hub, "idling",

```
364  automobile_remain_at_hub(ai)(ris)(...)(atH(hi),ahis) ≡
364a    let τ = record_𝕋𝕀𝕄𝔼()() in
364b    ch[ {ai,hi} ] ! τ ;
364c    automobile(ai)(ris)(...)(atH(hi),⟨(τ,hi)⟩^ahis) end
```

365. [363c] 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.

```
365  automobile_leaving_b(ai)({li}∪ris)(...)(atH(hi),ahis) ≡
365a    let τ = record_𝕋𝕀𝕄𝔼()() in
365b    (ch[ {ai,hi} ] ! τ ‖ ch[ {ai,li} ] ! τ) ;
365c    automobile(ai)(ris)(...)(onL(li,(hi,0,_)),⟨(τ,li)⟩^ahis) end
365     pre: [hub is not isolated]
```

366. [363e] Or the automobile **stop**s, "disappears — off the radar" !

366 automobile_stop(ai)(ris),(...)(atH(hi),ahis) ≡ **stop** .

367. 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 (!)].

```
367  automobile(ai)(ris)(...)(onL(li(fhi,f,thi)),lhis) ≡
367a     automobile_remains_on_link(ai)(ris)(...)(onL(li(fhi,f,thi)),lhis)
367b   ⌈⌉
367c     automobile_leaving_linl(ai)(ris)(...)(onL(li(fhi,f,thi)),lhis)
367d   ⌈⌉
367e     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`.

368. Hubs

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

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

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

371. Links

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

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

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

4.5.5 Instantiation

374. 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
374. initialisation: Unit → Unit
374. initialisation() ≡
374a.     ‖ { hub(uid_H(h))
374a.               (mereo_H(h))
374a.                   (attr_HΩ(h),...)
374a.                       (attr_HΣ(h),attr_HΣ(h),attr_HHist(h))
374a.       | h:H • h ∈ hs }
374b.     ‖
374c.     ‖ { link(uid_L(l))
374c.               (mereo_L(l))
374c.                   (attr_LEN(l),...)
374c.                       (attr_LΣ(l),attr_LHist(l))
374c.       | l:L • l ∈ ls }
374d.     ‖
374e.     ‖ { automobile(uid_A(a))
374e.               (mereo_A(a))
374e.                   (attr_APos(a),attr_AHist(a))
374e.       | a:A • a ∈ as }
```

5 Conclusion

TO BE WRITTEN

6 Bibliography

6.1 Bibliographical Notes

TO BE WRITTEN

6.2 References

[1] J.-R. Abrial, *The B Book: Assigning Programs to Meanings*, Tracts in Theoretical Computer Science, Cambridge University Press, Cambridge, England, 1996.

[2] ——, *Modeling in Event-B: System and Software Engineering*, Cambridge University Press, Cambridge, England, 2009.

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

[4] J. W. Backus, *The syntax and semantics of the proposed international algebraic language of the zürich ACM-GAMM conference*, in ICIP Proceedings, Paris 1959, Butterworth's, London, 1960, pp. 125–132.

[5] G. Birtwistle, O.-J.Dahl, B. Myhrhaug, and K. Nygaard, *SIMULA* `begin`, Studentlitteratur, Lund, Sweden, 1974.

[6] D. Bjørner, *Domain Modeling Case Studies:*

- *2025: Transport: A Domain Description, March 2025*
  https://www.imm.dtu.dk/ dibj/2025/transport.pdf
- *2025: Banking: A Domain Description, August 2025*
  https://www.imm.dtu.dk/~dibj/2025/banking/main.pdf
- *2023: Nuclear Power Plants, A Domain Sketch, July 2023*
  https://www.imm.dtu.dk/~dibj/2023/nupopl/nupopl.pdf
- *2021: Shipping, April 2021*
  https://www.imm.dtu.dk/~dibj/2021/ral/ral.pdf
- *2021: Rivers and Canals – Endurants – A Technical Note, March 2021*
  https://www.imm.dtu.dk/~dibj/2021/Graphs/Rivers-and-Canals.pdf
- *2021: A Retailer Market, January 2021*
  https://www.imm.dtu.dk/~dibj/2021/Retailer/BjornerHeraklit27January2021.pdf
- *2019: Container Terminals, ECNU, Shanghai, China, Sept. 2018*
  https://www.imm.dtu.dk/~dibj/2018/yangshan/maersk-pa.pdf
- *2017: Documents, TongJi Univ., Shanghai, China, Sept. 2017*
  https://www.imm.dtu.dk/~dibj/2017/docs/docs.pdf

- *2017: Urban Planning, TongJi Univ., Shanghai, China, Sept. 2017*
  https://www.imm.dtu.dk/~dibj/2017/urban-planning.pdf
- *2017: Swarms of Drones, Inst. of Softw., Chinese Acad. of Sci., Peking, China, November 2017*
  https://www.imm.dtu.dk/~dibj/2017/swarms/swarm-paper.pdf
- *2013: Road Transport, Techn. Univ. of Denmark*
  https://www.imm.dtu.dk/~dibj/2013/road/road-p.pdf
- *2012: Credit Cards, Uppsala, Sweden*
  https://www.imm.dtu.dk/~dibj/2016/credit/accs.pdf
- *2012: Weather Information, Bergen, Norway*
  https://www.imm.dtu.dk/~dibj/2016/wis/wis-p.pdf
- *2010: Web-based Transaction Processing, Techn. Univ. of Vienna, Austria, April 2010*
  https://www.imm.dtu.dk/~dibj/wfdftp.pdf
- *2010: The Tokyo Stock Exchange, Tokyo Univ., Japan, November 2009*
  https://www.imm.dtu.dk/~db/todai/tse-1.pdf,
  https://www.imm.dtu.dk/~db/todai/tse-2.pdf
- *2009: Pipelines, Techn. Univ. of Graz, Austria, November 2008*
  https://www.imm.dtu.dk/~dibj/pipe-p.pdf
- *2007: A Container Line Industry Domain, Techn. Univ. of Denmark, June 2007*
  https://www.imm.dtu.dk/~dibj/container-paper.pdf
- *2002: The Market, Techn. Univ. of Denmark*
  https://www.imm.dtu.dk/~dibj/themarket.pdf
- *1995–2004: Railways, Techn. Univ. of Denmark - a compendium*
  https://www.imm.dtu.dk/~dibj/train-book.pdf

This is not a single document. It is a [printable] collection of older case studies. Some, i.e., the later ones, adhere to the emerging DDL: Domain Description Language, [27]. Earlier ones may not. Urban Planning, for example, `https://www.imm.dtu.dk/~dibj/2017/-urban-planning.pdf`, do not. It treats TIME *in an erroneous way – and should be corrected.*

[7] ——, *Programming in the Meta-Language: A Tutorial*, in The Vienna Development Method: The Meta-Language, [32], D. Bjørner and C. B. Jones, eds., LNCS, Springer–Verlag, 1978, pp. 24–217.

[8] ——, *Software Abstraction Principles: Tutorial Examples of an Operating System Command Language Specification and a PL/I-like On-Condition Language Definition*, in The Vienna Development Method: The Meta-Language, [32], D. Bjørner and C. B. Jones, eds., LNCS, Springer–Verlag, 1978, pp. 337–374.

[9] ——, *The Vienna Development Method: Software Abstraction and Program Synthesis*, in Mathematical Studies of Information Processing, vol. 75 of LNCS, Springer–Verlag, 1979. Proceedings of Conference at Research Institute for Mathematical Sciences (RIMS), University of Kyoto, August 1978.

[10] ——, *Software Engineering, Vol. 3: Domains, Requirements and Software Design*, Texts in Theoretical Computer Science, the EATCS Series, Springer, 2006. https://link.springer.com/book/10.1007/3-540-33653-2.

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

[12] ——, *Domain Engineering: Technology Management, Research and Engineering*, Research Monograph (# 4); JAIST Press, 1-1, Asahidai, Nomi, Ishikawa 923-1292 Japan; https://www.imm.dtu.dk/~dibj/jaistmono.pdf, 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.

.

[13] ——, *Domain Engineering*, in Formal Methods: State of the Art and New Directions, P. Boca and J. Bowen, eds., Eds. Paul Boca and Jonathan Bowen, London, UK, 2010, Springer, pp. 1–42. www.imm.dtu.dk/~dibj/facs-domain.pdf, https://www.booksamillion.com/p/Formal-Methods/Paul-Boca/9781848827356.

[14] ——, *Domain Science & Engineering – From Computer Science to The Sciences of Informatics, Part I: The Engineering Part*, Kibernetika i sistemny analiz, 2 (2010), pp. 100–116. www.imm.dtu.dk/~db/kiev-p1.pdf.

[15] ——, *Domain Science & Engineering – From Computer Science to The Sciences of Part II: The Science Part*, Kibernetika i sistemny analiz, 2 (2011), pp. 100–120. www.imm.dtu.dk/~db/kiev-p2.pdf.

[16] ——, *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), Springer, Heidelberg, Germany, January 2011, pp. 167–183. `www.imm.dtu.dk/~dibj/maurer-bjorner.pdf`.

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

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

[19] ——, *Domain Analysis & Description – Principles, Techniques and Modeling Languages.*, ACM Trans. on Software Engineering and Methodology, 28 (2019), p. 66 pages. www.imm.dtu.dk/~dibj/2018/tosem/Bjorner-TOSEM.pdf.

[20] ——, *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 (2019), pp. 1–67. 68 pages.

[21] ——, *Domain Science & Engineering – A Foundation for Software Development*, EATCS Monographs in Theoretical Computer Science, Springer, Heidelberg, Germany, January 2020. xii+346 pages. A revised version of this book is [29]. https:///doi.org/10.1007/978-3-030-73484-8.

[22] ——, *Domain Science & Engineering – A Foundation for Software Development*, Springer, Fall 2021.

[23] ——, *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.

[24] ——, *Domain Analysis & Description*, in Theoretical Aspects of Computing, ICTAC 2025, no. 16237 in Lecture Notes in Computer Science, Springer, November 24–28 2025, pp. 39–66. A Tutorial. DOI 10.1145/3796228.

[25] ——, *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.

[26] ——, *A Syntax for DADL – First Attempt*, tech. rep., DTU Compute, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, January–March 2026. `https://www.imm.dtu.dk/~dibj/2026/syntax/main.pdf`.

[27] ——, *DADL: An Analysis & Description Language*, tech. rep., DTU Compute, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, February 16, 2026 2026. `https://www.imm.dtu.dk/~dibj/2026/ddl/ddl.pdf`.

[28] ——, *Methodology – A Study*, tech. rep., DTU Compute, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, January–March 2026. `https://www.imm.dtu.dk/~dibj/2026/method/main.pdf`.

[29] ——, *Domain Science & Engineering, a Primer*[70] Technical University of Denmark. Significantly revised edition of [21]. xii+211 pages., February 2026 To be submitted [2026/2027].

[70] This book is currently being translated into Russian, [30], by Dr. Mikhail Chupilko and his colleagues, ISP/RAS (Institute of Systems Programming, Russian Academy of Sciences), Moscow and into Chinese, [34], by Dr. Yang ShaoFa, IoS/CAS (Institute of Software, Chinese Academy of Sciences), Beijing.

[30] D. Bjørner and M. Chupilko, *Моделирование предметной области: основы.* Translation of [29]. [Book], To be submitted [2026/2027].

[31] D. Bjørner and C. B. Jones, eds., *The Vienna Development Method: The Meta-Language*, vol. 61 of LNCS, Springer, 1978. This was the first monograph on *Meta-IV*. [7, 8, 9].

[32] ——, eds., *The Vienna Development Method: The Meta-Language*, vol. 61 of LNCS, Springer, 1978. This was the first monograph on *Meta-IV*.

[33] ——, eds., *Formal Specification and Software Development*, Prentice-Hall, London, England, 1982.

[34] D. Bjørner and Y. ShaoFa, 领域科学与工程导论. Translation of [29]. [Book], To be submitted [2026/2207].

[35] W. D. Blizard, *A Formal Theory of Objects, Space and Time*, The Journal of Symbolic Logic, 55 (1990), pp. 74–89.

[36] M. Bunge, *Treatise on Basic Philosophy: Ontology I: The Furniture of the World*, vol. 3, Reidel, Boston, Mass., USA, 1977.

[37] ——, *Treatise on Basic Philosophy: Ontology II: A World of Systems*, vol. 4, Reidel, Boston, Mass., USA, 1979.

[38] R. Carnap, *Introduction to Semantics*, Harvard Univ. Press, Cambridge, Mass., 1942.

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

[40] P. Cousot, *Principles of Abstract Interpretation*, The MIT Press, September 21, 2021.

[41] P. Cousot and R. Cousot, *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, in 4th POPL: Principles of Programming and Languages, ACM Press, 1977, pp. 238–252.

[42] ——, *Systematic Design of Program Analysis Frameworks*, in 6th POPL: Principles of Programming and Languages, ACM Press, 1979, pp. 269–282.

[43] O.-J. Dahl, E. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, 1972.

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

[45] C. A. Furia, D. Mandrioli, A. Morzenti, and M. Rossi, *Modeling Time in Computing*, Monographs in Theoretical Computer Science, Springer, 2012.

[46] C. W. George, P. Haff, K. Havelund, A. E. Haxthausen, R. Milne, C. B. Nielsen, S. Prehn, and K. R. Wagner, *The RAISE Specification Language*, The BCS Practitioner Series, Prentice-Hall, Hemel Hampstead, England, 1992. ISBN 0137528337 and 9780137528332.

[47] C. W. George, A. E. Haxthausen, S. Hughes, R. Milne, S. Prehn, and J. S. Pedersen, *The RAISE Development Method*, The BCS Practitioner Series, Prentice-Hall, Hemel Hampstead, England, 1995.

[48] C. Gunter and D. S. Scott, *Semantic domains*, in [115] — vol.B., J. Leeuwen, ed., North-Holland Publ.Co., Amsterdam, 1990, pp. 633–674.

[49] G. H. Hardy, E. M. Wright, and J. Silvermann, *An Introduction to the Theory of Numbers*, Oxford University Press, England, 6th edition ed., 2008. Editor: Roger Heath Brown.

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

[51] D. Harel and R. Marelly, *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine*, Springer-Verlag, 2003.

[52] C. A. R. Hoare, *Notes on Data Structuring*, in [43], 1972, pp. 83–174.

[53] ——, *Communicating Sequential Processes*, C.A.R. Hoare Series in Computer Science, Prentice-Hall International, 1985.

[54] ——, *Communicating Sequential Processes*. Published electronically: `usingcsp.com/-cspbook.pdf`, 2004. Second edition of [53].

[55] IEEE Computer Society, *IEEE–STD 610.12-1990: Standard Glossary of Software Engineering Terminology*, tech. rep., IEEE, IEEE Headquarters Office, 1730 Massachusetts Avenue, N.W., Washington, DC 20036-1992, USA. Phone: +1-202-371-0101, FAX: +1-202-728-9614, 1990.

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

[57] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-01715-6.

[58] M. A. Jackson, *Principles of Program Design*, Academic Press, 1969.

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

[60] ——, *Software Hakubutsushi: Sekai to Kikai no Kijutsu (Software Requirements & Specifications: a lexicon of practice, principles and prejudices)*, Toppan Company, Ltd., 2-2-7 Yaesu, Chuo-ku, Tokyo 104, Japan, 1997. In Japanese. Translated by Tetsuo Tamai (Univ. of Tokyo, tetsuo.tamai@gmail.com) and Hiroshi Sako; ISBN 4-8101-8098-0; xxv + 267 pages.

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

[62] M. A. Jackson and G. Twaddle, *Business Process Implementation — Building Workflow Systems*, Addison-Wesley, 1997.

[63] J.W. Backus and F.L. Bauer and J.Green and C. Katz and J. McCarthy and P. Naur and A.J. Perlis and H. Rutishauser and K. Samelson and B. Vauquois and J.H. Wegstein and A. van Wijngaarden and M. Woodger, *Revised Report on the Algorithmic Language Algol 60 – edited by P. Naur*, The Computer Journal, 5 (1963), p. 349–367.

[64] S. C. Kleene, *Introduction to Meta-Mathematics*, Van Nostrand, New York and Toronto, 1952.

[65] ——, *Mathematical Logic*, Dover Publications, Dover Edition, December 1, 2002. Originally published in 1967 by John Wiley & Sons, Publ., New York, NY, USA.

[66] P. J. Landin, *The Mechanical Evaluation of Expressions*, Computer Journal, 6 (1964), pp. 308–320.

[67] ——, *A Correspondence Between ALGOL 60 and Church's Lambda-Notation (in 2 parts)*, Communications of the ACM, 8 (1965), pp. 89–101 and 158–165.

[68] ——, *A Generalization of Jumps and Labels*, tech. rep., Univac Sys. Prgr. Res. Grp., N.Y., 1965.

[69] ——, *An Analysis of Assignment in Programming Languages*, tech. rep., Univac Sys. Prgr. Res. Grp., N.Y., 1965.

[70] ——, *Getting Rid of Labels*, tech. rep., Univac Sys. Prgr. Res. Grp., N.Y., 1965.

[71] ——, *A Formal Description of ALGOL 60*, in [112], 1966, pp. 266–294.

[72] ——, *A Lambda Calculus Approach*, in Advances in Programming and Non-Numeric Computations, L. Fox, ed., Pergamon Press, 1966, pp. 97–141.

[73] ——, *The Next 700 Programming Languages*, Communications of the ACM, 9 (1966), pp. 157–166.

[74] W. Little, H. Fowler, J. Coulson, and C. Onions, *The Shorter Oxford English Dictionary on Historical Principles*, Clarendon Press, Oxford, England, 1973, 1987. Two vols.

[75] M. J. Loux, *Metaphysics, a Contemporary Introduction*, Routledge Contemporary Introductions to Philosophy, Routledge, London and New York, 1998 (2nd ed., 2020).

[76] ANSI X3.9-1966, *The Fortran programming language*, tech. rep., American National Standards Institute, Standards on Computers and Information Processing, 1966.

[77] J. M. E. McTaggart, *The Unreality of Time*, Mind, 18 (October 1908), pp. 457–84. New Series. See also: [85].

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

[79] J. Mey, *Pragmatics: An Introduction*, Blackwell Publishers, 13 January, 2001. Paperback.

[80] C. S. Peirce, *Pragmatism as a Principle and Method of right thinking: The 1903 Harvard Lectures on Pragmatism*, State Univ. of N.Y. Press, and Cornell Univ. Press, 14 July 1997.

[81] G. D. Plotkin, *Structural operational semantics*, lecture notes, Aarhus University, DAIMI FN-19. Reprinted 1991, 1981. See [84, 82].

[82] ——, *The origins of structural operational semantics*, Journal of Logic and Algebraic Programming, 60–61 (2004), pp. 3–15. See [81, 84].

[83] ——, *A structural approach operational semantics*, Journal of Logic and Algebraic Programming, 60–61 (2004), pp. 17–139. Widely disseminated since 1981 as [81]. See also [82].

[84] ——, *A structural approach operational semantics*, Journal of Logic and Algebraic Programming, 60–61 (2004), pp. 17–139. Widely disseminated since 1981 as [81]. See also [82].

[85] R. L. Poidevin and M. MacBeath, eds., *The Philosophy of Time*, Oxford University Press, 1993.

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

[87] ——, *Formal Logic*, Clarendon Press, Oxford, UK, 1955.

[88] ——, *Time and Modality*, Oxford University Press, Oxford, UK, 1957.

[89] ——, *Past, Present and Future*, Clarendon Press, Oxford, UK, 1967.

[90] ——, *Papers on Time and Tense*, Clarendon Press, Oxford, UK, 1968.

[91] ——, *Changes in Events and Changes in Things*, Oxford University Press, 1993, ch. in [85].

[92] W. Reisig, *Petri Nets: An Introduction*, vol. 4 of EATCS Monographs in Theoretical Computer Science, Springer Verlag, May 1985.

[93] ——, *A Primer in Petri Net Design*, Springer Verlag, March 1992. 120 pages.

[94] ——, *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets*, Springer Verlag, December 1998. 400 pages.

[95] ——, *Understanding Petri Nets Modeling Techniques, Analysis Methods, Case Studies*, Springer, 2013. 230+XXVII pages, 145 illus.

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

[97] H. R. Rogers, *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, 1967.

[98] D. S. Scott, *Outline of a Mathematical Theory of Computation*, in Proc. 4th Ann. Princeton Conf. on Inf. Sci. and Sys., 1970, p. 169.

[99] ——, *The lattice of flow diagrams*, in Symposium on Semantics of Algorithmic Languages, E. Engeler, ed., vol. 188 of Lecture Notes in Mathematics, Springer, 1971, pp. 311–366. https://doi.org/10.1007/BFb0059703.

[100] ——, *Data types as lattices*. Unpublished Lecture Notes, Amsterdam, 1972.

[101] ——, *Lattice theory, data types and semantics*, in Symposium on Formal Semantics, R. Rustin, ed., Prentice-Hall, 1972, pp. 67–106.

[102] ——, *Data types as lattices*, SIAM Journal on Computer Science, 5 (1976), pp. 522–587.

[103] B. Smith, *Mereotopology: A Theory of Parts and Boundaries*, Data and Knowledge Engineering, 20 (1996), pp. 287—303.

[104] K. Sørlander, *Det Uomgængelige – Filosofiske Deduktioner [The Inevitable – Philosophical Deductions, with a foreword by Georg Henrik von Wright]*, Munksgaard · Rosinante, Copenhagen, Denmark, 1994. 168 pages.

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

[106] ——, *Den Endegyldige Sandhed [The Final Truth]*, Rosinante, Copenhagen, Denmark, 2002. 187 pages.

[107] ——, *Forsvaret for Rationaliteten*, Informations Forlag, Copenhagen, Denmark, 2008. 232 pages.

[108] ——, *Fornuftens Skæbne – Tanker om Menneskets Vilkår*, Informations Forlag, Copenhagen, Denmark, 2014. 238 pages.

[109] ——, *Indføring i Filosofien [Introduction to The Philosophy]*, Informations Forlag, Copenhagen, Denmark, 2016. 233 pages.

[110] ——, *Den rene fornufts struktur [The Structure of Pure Reason]*, Ellekær, Slagelse, Denmark, 2022. See [111].

[111] ——, *The Structure of Pure Reason*, Springer, February 2025. This is an English translation of [110] – done by Dines Bjørner in collaboration with the author.

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

[113] J. Sun and J. S. Dong, *Live Sequence Charts as Communicating Sequential Processes*, tech. rep., School of Computing, Dept. of Computer Science, National University of Singapore, 3 Science Drive 2, 117543 Singapore, August 2004.

[114] J. van Benthem, *The Logic of Time*, vol. 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 ed., 1983, 1991.

[115] J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Volumes A and B*, Elsevier, 1990.

[116] A. C. Varzi, *On the Boundary between Mereology and Topology*, Hölder-Pichler-Tempsky, Vienna, 1994, pp. 419–438.

[117] N. Wirth, *A Generalization of ALGOL*, Communications of the ACM, 6 (1963), pp. 547–554.

[118] N. Wirth, *The Programming Language PASCAL*, Acta Informatica, 1 (1971), pp. 35–63.

[119] ——, *Systematic Programming*, Prentice-Hall, 1973.

[120] ——, *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.

[121] ——, *Programming in Modula-2*, Springer-Verlag, Heidelberg, Germany, 1982.

[122] ——, *From Modula to Oberon*, Software — Practice and Experience, 18 (1988), pp. 661–670.

[123] ——, *The Programming Language Oberon*, Software — Practice and Experience, 18 (1988), pp. 671–690.

[124] N. Wirth and H. Weber, *EULER: A Generalization of ALGOL, and its Formal Definition*, Communications of the ACM, 9 (1966), pp. 13–23, 89–99.

[125] J. C. P. Woodcock and J. Davies, *Using Z - specification, refinement, and proof*, Prentice Hall international series in computer science, Prentice Hall, 1996. https://dblp.org/rec/books/daglib/0072139.bib.

[126] W. Yi, *A Calculus of Real Time Systems*, PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1991.