Wolt♠

A DOMAIN Description

Dines Bjørner*
Fredsvej 11, DK-2840 Holte, Danmark
E–Mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

Work in Progress – Incomplete Draft

July 29, 2026, 21:16

Abstract

We present an experimental research & engineering description of Wolt[1]-like grocery delivery systems.[2,3]

Contents

*© Dines Bjørner – July 29, 2026

[2] We refer to Appendix Sect. E on page 91 – as the reader may wish to follow how domain modeling may take place in "real life"!

[3] **Warning:** Some narratives and many formula need be double checked! I have started, Sunday July 26, 2026, at 08:48 am, review-editing the entire document before contemplating the writing of Sects. 4.3 and 5.7–5.8. I have marked, in margins, **[ho:mi;da:mo]**, progress wrt. this editing.

[2] ♠ `https://explore.wolt.com/en/dnk/about` Use of the Wolt logo may need permission. Will inquire once this report is "completed" – with `DoorDash` – – or Wolt `https://about.doordash.com/en-us` – but near-impossible to find their corporate e-mail addresses!

1 Introduction

This experimental research document is based on our work on DOMAIN science & engineering [14, 17, 18, 26, 29, 30].

- It was thought out and written first for myself, as an exploratory research, domain engineering and documentation effort. I am, in 2026, continually "testing" the DOMAIN method. One – sort of – new ``erkenntniss'' obtained in thinking out and writing this report is the importance of carefully, yet informally narrating domain actors: actions and event. Thus Sect. 5.3.
- As I wrote it the idea occurred to me to "turn" the document into a *school example:* one that, more-or-less authoritatively, shows readers
  - the more-or-less sequential DOMAIN analysis & description development process, and
  - also how an "end product", DOMAIN descriptions, might/should look like.

1.1 Survey

Thus this document can serve as a "school" example of how a domain description is constructed and presented. In careful phases (•), stages (–) and steps (*). One after the other. Sequentially – such as shown in this document.

In reading this document the reader should be reasonably well-informed of these phases (3, •), stages (6, –) and steps (10, *) of DOMAIN modeling. These are the determination, i.e., analysis & description of:

- the universe of discourse Sect. 3
- endurants Sect. 4

That is: the reader knows that when studying this document's coverage of earlier sections, e.g., `external qualities`, that `parts` have `internal qualities`, like `unique identification`, etc., and that there is the notion that some `endurants`, i.e., `parts`, are `manifest`, and, if so, are either `static` or `mobile`. Etcetera, etc.!

1.2 A Colourful Report

We have pursued two **colouring** styles:

- **Section**, **subsection**, **sub-subsection** and **paragraph** display lines are shown in "living **colours**: "![4]
- A number of formula keywords and cross-references are also **coloured** – some in the margins[5]

[4] I like that! I think it should help the reader stop up, pause, reflect on what has just been covered in the previous text, "saddle around", ready for and properly informed, though the next, **colour**-signaled display line, of what is "coming up". Others "feel" intimated, sorry!

[5] These markings should help readers "find their way around": more quickly identifying the essential description points!

1.3 Page Shifts

This document is meant, if read, to be primarily read *"from a computer display screen"*. Two remarks on this:

- I have taken the, if for paper-printing, luxurious decision to split the text onto many pages.[6] That is: there a new pages for various sections, subsections and paragraphs. And – in so far as reasonable – I have tried to keep **narration**s **and** **formalisation**s of many individual domain concepts *on the same page*[7] !
- So, for *"computer display screen reading"* of, for example behaviors, I [additionally] suggest the following:
  - while reading any specific page
  - keep separate display windows open for each of the domain concepts otherwise, more-or-less implicitly, referred to on that page:
    - endurant external qualities,
    - mereologies,
    - attributes,
    - actors,
    - messages,
    - signatures

2 Epistemological Remarks: A Skeleton Mini Essay[8]

You may wish to view this document, i.e., 𝔻𝕆𝕄𝔸𝕀ℕ engineering, being based, relying, reflecting on four dimensions:

- **Domains versus Computable Functions**
- **Thought, Language and Writing**
- **Endurants "versus" Perdurants**
- **Values versus Object**

2.1 Domains Versus Computable Functions

Here we go !

- Some 300.000 years ago Homo Sapiens "appeared"[9].
- And some 5.000 years ago, between the rivers of [today's] Euphrates Tigris the first manifestations of human-created, orderly societies appeared [36].
- The Greeks gave us the first examples of rational thought for half a millennium before Christ. Philosophers have thought about "the world around us": *what is it made of ?.*[10]

[6] Without these page shifts this, presently 66 page, document would be 46 pages !
[7] – trying to avoid paper-turning to & from formulas and narratives
[8] You can skip this section in any reading of this document
[9] `https://en.wikipedia.org/wiki/Human`
[10]

- Christianity gave us "food for thought" of that which cannot be rationally understood.
- Mankind developed – let us say since the Middle Ages [such as the 'West', i.e., Europe, has labeled them] – various man-made institutions: orderly governments, managed road nets, "institutionalized" commerce, regulated banks, schools, hospitals, etc.
- Since basically the late Middle Ages scientists have studied nature: Gallilei, Kopernicus, Brahe, Newton, etc., up to and beyond today – using and developing mathematics.
- But nobody has really studied the mathematics of man-made institutions – perhaps with the exception of economics.

With **domain modeling**, I unashamed/unabashed claim, that that is changing!

- Now we can study man-made institutions such as [5]:
  - *Air Traffic [6, 22]*
  - *Assembly Lines [20]*
  - *Banking [25]*
  - *Book-keeping [23]*
  - *Container Terminals [16]*
  - *Credit Cards [11]*
  - *Pipelines [24]*
  - *Railways [33]*
  - *Retailer Markets [7, 19]*
  - *Shipping [21]*
  - *Stock Exchanges [9]*
  - *Swarms of Drones [13]*
  - *Transportation [8, 27]*
  - *Urban Planning [15]*
  - *Weather Prognosis [12]*

Up till now **Informatics** – the confluence of computer & computing science and mathematics – saw computing from the basis of the hardware computers and other IT equipment: from the computer "out" towards its application "domains".

Now we can also see **Informatics** from the basis of the domains.

Physicists never "see" physics phenomena from the side of computing! Do they [now]?

2.2 Thought, Language and Writing

More than five thousand years ago, again in the land between the rivers [36], the *cunei form script*[11] emerged. Over the next two and a half millennia various *writing systems* [55, 41, 50, 39, 42, 48, 58] emerged – with the invention of writing systems based on pronounceable characters which, when composed into words, "mimicked" how these words were indeed spoken.

- Thales of Milet (624–545): *everything originates from* `water` [56]
- Anaximander (610–546): *`apeiron' (the 'un-differrentiated', 'the unlimited') is the origin* [38]
- Anaximenes (586–526): *air is the basis for everything* [53]
- Heraklit of Efesos (540–480): *fire is the basis and "battle"* [4]
- Parminedes (515–470): *everything that exists is* `eternal and immutable` [46]
- Empedokles (490–430): *there are four base elements:* `fire, water, air and soil` [68]
- Demokrit (460–370): *all is built from* `atoms` [1]

[11] `https://en.wikipedia.org/wiki/Cuneiform`. Cuneiform is one of the world's earliest systems of writing, developed by the ancient Sumerians in Mesopotamia (modern-day Iraq and Syria) around 3200 BCE. It is characterized by distinctive wedge-shaped marks pressed into soft clay tablets using a cut reed stylus

Euklid of Alexandria[12] formulated his geometry in greek words. It was not till well after the early Renaissance that special, what became mathematical, notations[13] for the rigorous expression of mathematics, including rational arguments, emerged.

Since then we have witnessed a plethora of formal notations: as mathematics expanded and programming languages saw the scene!

Linguists and philosophers have speculated about relationships between language and thought. Notably *Benjamin Lee Whorf* [14]. *Noam Chomsky*[15] likewise contributed, in his, different way, to the understanding of language and thought. *Ludwig Wittgenstein*'s [16] book `Tractatus Logico-Philosophicus`, had the broad goal: of trying to identify the relationship between language and reality, and to define the limits of science. His basic approach was to study the problem on the basis of language analysis. We may say that he failed his goal: properties of the universe exists before languages emerged. [Such is the conclusion arrived at in [60, 61, 62, 63, 64].]

2.3 Domains: Endurants & Perdurants

Domains are the man-made institutions *"within which we act"*! Endurants are their *"static"* quantities. Perdurants are their *"dynamic"* qualities. We encircle the concept of

Characterisation: 1 **Domain:** 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* `[DB]` ■

We encircle the concept of

Characterisation: 2 **Endurants:** In the context of domains, **endurants** are those entities 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] ■

We encircle the concept of

Characterisation: 3 **Perdurants:** **Perdurants**, in the context of domains, **perdurants** are those entities 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] ■

At every one moment we observe endurants and perdurants. We see the physical (static) form (of endurants) and [somehow also] 'states' of the (dynamically) unfolding (perdurant) behaviours and the instantaneous events of their interactions.

We describe, i.e., narrate in our national language(s), and formalize in some novel form of mathematics, endurants and perdurants.

[12] `[https://da.wikipedia.org/wiki/Euklid]`

[13] Florian Cajori (1928) *A History of Mathematical Notations.* Chicago: Open Court Publishing. ISBN 9780486161167

[14] Benjamin Atwood Lee Whorf (April 24, 1897 – July 26, 1941) was an American linguist ... best known for proposing the `Sapir-Whorf Hypothesis.` He believed that the structures of different languages shape how their speakers perceive and conceptualize the world. Whorf saw this idea, named after him and his mentor Edward Sapir, as having implications similar to those of Einstein's principle of physical relativity.

[15] https://en.wikipedia.org/wiki/Noam_Chomsky

[16] https://en.wikipedia.org/wiki/Ludwig_Wittgenstein

2.4 Values Versus Objects – and Functions

Major aspects, i.e., properties of what we describe, is expressed as values, functions over these, and as objects. We shall make a distinction between

- **value**s and
- **object**s

Specifically – in the context of domains:

- **Value**s are typically internal qualities such as **unique identifiers, mereologies, attributes** and values "composed" or "extracted" from these.
- **Object**s are typically the "aggregate" of the external quality endurants and their transcendentally deduced perdurants. So endurant part behaviours embody an essence of being an object.
- **Function**s `apply` to `values` and `yield values`. Behaviours are defined as functions that apply to the `internal qualities` of `objects`.
- **However:** In this domain description we shall model some, not *behaviourable* endurants, viz. *groceries*, as values that can be communicated between behaviours.[17]

• • •

We have summarised some aspects of four concepts. We do not claim that they form some kind of orthogonal concept "sphere". You, the reader, may pause to reflect on these four, whether they together form a "fifth" concept, etc.,! Have fun!

3 Universe of Discourse [30, *Sect. 4.1*]

There are basically two "parties" to grocery delivery systems (**GDS**): the delivery service (**DS**) and its customers (**KC**).

Specification Unit: 1 **Universe of Discourse: A Grocery Delivery System:**

Narrative:

1. A general grocery delivery system, **GDS**, has:
   - a set, **KS**, of **customers**, **K**, who order **G**roceries to be delivered;
   - a **grocery store**, **GST**[18] with groceries for sale [and delivery];
   - a **courier management, CM** which organizes deliveries; and
   - a set, **CS**, of **couriers, C**.

Formalisation:

type 1. GDS. **value** 1. gds:GDS. **axiom** 1. **is_**{structure(gds:GDS)}.

[17] This kind of modeling is also present in [16, *Container Terminal Ports*] where containers, as endurant parts, are conveyed by container vessels, quay cranes, quay trucks, etc.

[18] Modeled, conceptually, as one, but that one can be interpreted [say in *requirements development* phase] as a set of branch grocery stores, one per land district

Expanded Narrative:

⋆ *The four kinds of "players" of the General Delivery System are as follows:*

- ***Customers*** *contact the grocery store; is either rejected because the customers address is outside the operating area of the store, or accepted – then given a catalog. The customer then selects one or more kinds of groceries and informs the store of the desired quantities. The customer then either abandons the ordering or completes it in an orderly fashion, paying cash for delivery. After that the customer awaits acceptance, notification of time of delivery and, eventually the delivery – during and after which the customer may revert to begin further orders and await deliveries.*
- *The* ***Grocery Store*** *has a store of several kinds of groceries which – at cost – can be ordered by customers and delivered; customer orders are either rejected (because the customers address is outside the operating area of the store) or accepted. The grocery store then in one or more contacts from the customer accepts its sub-orders. Finally the grocery store accepts (or rejects) the completed order, notifies the customer so, collects, as a package of accepted orders, and hands them over to courier management.*
- ***Courier Management*** *accepts order packages, schedules deliveries, notify customers, and provides consolidated delivery routes for one or more identified customers to couriers.*
- ***Couriers*** *are given delivery routes from courier management, travels these routes, intermittently deliver packages to awaiting customers and eventually returns to courier management.*
- *There is a* ***road net*** *along which couriers can travel. It is understood by delivery system (customers, the store, courier management and couriers) in terms of a* ***road map*** *which faithfully abstracts the road net.*
- *And there are* ***monies***[19] ***M****. Customers pay for orders by their cash – which is then collected by the grocery store.*
- *Any universe of discourse is considered a [manifest, structure] object.*

4 Endurants

Endurants are elements of objects.

4.1 External Qualities [30, *Sects. 4.2–4.5*]

External qualities are not objects, but elements of objects.

4.1.1 The Endurants

Specification Unit: 2 **Main Endurants:**

Narrative:

2. From a `gds:GDS` one can observe two main endurants (`a,b`)
   (a) a compound delivery service, **ds:DS**, which is here will be considered a structure; and

[19] "Monies" (or "moneys") is a formal variant primarily used in legal and financial contexts to denote several distinct or discrete sums of money being collected or paid out [https://writingexplained.org/monies-vs-moneys-difference]

(b) a compound [of] customers, **ks:KS**[20], which is a structure.

There is a road net along which couriers of the delivery service deliver groceries. In this domain description we model these road nets as a road map attribute of the delivery service. We refer to Appendix A on page 47.

Formalisation:

type
2. DS, KS
value
2a. **obs_**DS: GDS → DS
2b. **obs_**KC: GDS → KS
axiom
2a. **is_**structure(ds:DS)
2b. **is_**structure(ks:KS)

Model Choice: 1 **Endurant Parts:** We could have analysed the domain into a different "constellation" of endurant parts and/or analysed properties of the chosen ones differently. Delivery service, for example, could have been predicated as `manifest,` instead of, as here, as a `structure.` In choosing it as manifest we might have in mind that the two [Cartesian] components, grocery store and courier management, might be "overall managed" by the delivery service. Now we leave it open, in a *requirements prescription* phase, for the *requirements engineers,* as directed by their marketing concerns, to institute such a relationship, or, to make the two into separately owned & managed companies. Etc., etc. ▪

4.1.1.1 Customer Endurants

3. Customer compounds are [structure] part sets of customers.
4. Customers are atomic, manifest, static and behaviourable[21].

type
3. KS = K-**set**
4. K
value
3. **obs_**KS: KC → KS
axiom
3. **is_**structure(ks:KS)
4. **is_**manifest(k:K) ∧ **is_**static(k:K) ∧ **is_**behaviourable(k:K)

[20] We spell customer endurant sort names with 'k' in order to distinguish them from couriers!
[21] Customers pay for delivered groceries in cash, i.e., with monies. From where these monies come from we do not model!

Model Choice: 2 **The KS Structure:** **KS** is modeled as a `structure`. Some customers may, of course, choose to be members of one or more `consumer group`s whose purposes may be to protect *purchase and consumption rights,* on behalf of `consumers`, vis-a-vis for example `delivery services`. In such a case, and were we to include that aspect in our description of ***Wolt***, then **KS** may be further decomposed into also having [Cartesian] `consumer group` components ▪

4.1.1.2 Grocery Endurants

5. Groceries are atomic.
6. Groceries are manifest, mobile and **not** behaviourable.

type
5. G

axiomr
6. **is_**manifest(g:G), **is_**mobile(g:G) ∼**is_**behaviourable(g:G)

Model Choice: 3 **Groceries:** Groceries are modeled as elements of attributes of customers, the grocery store, courier management and couriers. This is a choice. That choice makes the model simpler ▪

4.1.1.3 Grocery Store Endurant

Specification Unit: 3 **Delivery Service Endurants:**

Narrative:

7. From the [compound] delivery service, **DS**, one can observe the following two components:
   (a) an atomic grocery store, **GST**, which is manifest, static and behaviourable[22];
   (b) a courier management, **CM**,which is manifest, static and behaviourable; and

Formalisation:

type
7. GST, CM

value
7a. **obs_**GST: DS → GST
7b. **obs_**CM: DS → CM

axiom
7a. **is_**manifest(gst:GST) ∧ **is_**static(gst:GST)∧ **is_**behaviourable(gst:GST)
7b. **is_**manifest(cc:CM) ∧ **is_**static(cm:CM) ∧ **is_**behaviourable(c:CM)

[22] As we shall later see: it contains, as a programmable attribute a set of groceries: many different kinds, and, for each kind, 0, 1 or more instances. From where these groceries "arrive" we do not model!

Model Choice: 4 **Separation of Parts:** We have chosen to separate the two, the grocery store and the management of couriers, into those two. The one, i.e., the latter: courier management could be an integral part. Etc. ▪

4.1.1.4 Courier Management Endurant

Narrative:

8. From courier management, **cm:CM**, we can observe a part sets, **cs:CS** [of couriers], a structure.
9. Couriers, **c:C**, are atomic, manifest, mobile and behaviourable.

Formalisation:

type
8. CS = C-**set**
9. C

value
8. **obs_**SC: CM → CS

axiom
8. **is_**structure(cd:CS)
9. **is_**manifest(c:C) ∧ **is_**mobile(c:C) ∧ **is_**behaviourable(c:C)

Model Choice: 5 **Courier Status:** Whether these couriers are self-employed – hired on a per delivery or other base, or [possibly "unionized"] salaried staff, we leave open ▪

4.1.1.5 Courier Endurant

10. Couriers are atomic. They "emerge" from courier management's courier pat sets.
11. Couriers are manifest, mobile and behaviourable.

type
10. C

axiom
11. **is_**manifest(c:C), **is_**mobile(c:C) **is_**behaviourable(c:C)

4.1.1.6 Money Endurants

12. Money endurants are atomic, manifest, static and **not** behaviourable.

type
12. M

axiom
12. **is_**manifest(m:M) ∧ **is_**mobile(m:M) ∧ ∼**is_**behaviourable(m:M)

Model Choice: 6 **Monies / Cash:** As for groceries, we have modeled cash/payment/monies as attributes of customers, grocery stores and courier management. That is sufficient – and it makes the model simpler, without loss of generality. We shall not model where monies come from! ∎

4.1.2 A Behaviour State [30, *Sect. 4.8*]

We select the behaviourable parts to form a state σ_{parts}

Narrative:

13. We recall the arbitrarily chosen value of the universe of discourse, cf. Item 1 on page 9.
    (a) The grocery store, *gst*, is a state component.
    (b) The courier management, *cmgt*, is a state component *cmgt* with which the grocery behaviour communicates.
    (c) The set of atomic couriers, *cs*, likewise contributes to the state.
    (d) The set of atomic customers, *ks*, of likewise contributes to the state.
    (e) The state is then the union of all the above-listed state components.

Formalisation:

value
13. gds:GDS
??. ds:DS = **obs_**DS(gds)
13a. gst:GST = **obs_**GST(ds)
13b. $cmgt$:CM = **obs_**CM(ds)
13c. cs: = { c | c:C • c ∈ **obs_**CS($gmgt$) }
13d. ks: = { k | k:K • k ∈ **obs_**KS(**obs_**KC(gds)) }
13e. σ_{parts} = $\{gst\}\cup\{cmgt\}\cup cs\cup ks$

Model Choice: 7 **Groceries:** The set of groceries of grocery atoms, `g:G`,

- is a programmable attribute of the grocery store, `gst:GST`, atom;
- the courier management atom, `cm:GM`, is a programmable attribute of that atom;
- courier atoms, `c:C`, is a programmable attribute of those atoms; and
- customer atoms, `k:K`, is a programmable attribute of those atoms;

Thus groceries, taken individually, and as a set, **is not part of the state.** That is a "unifying" modeling choice ∎

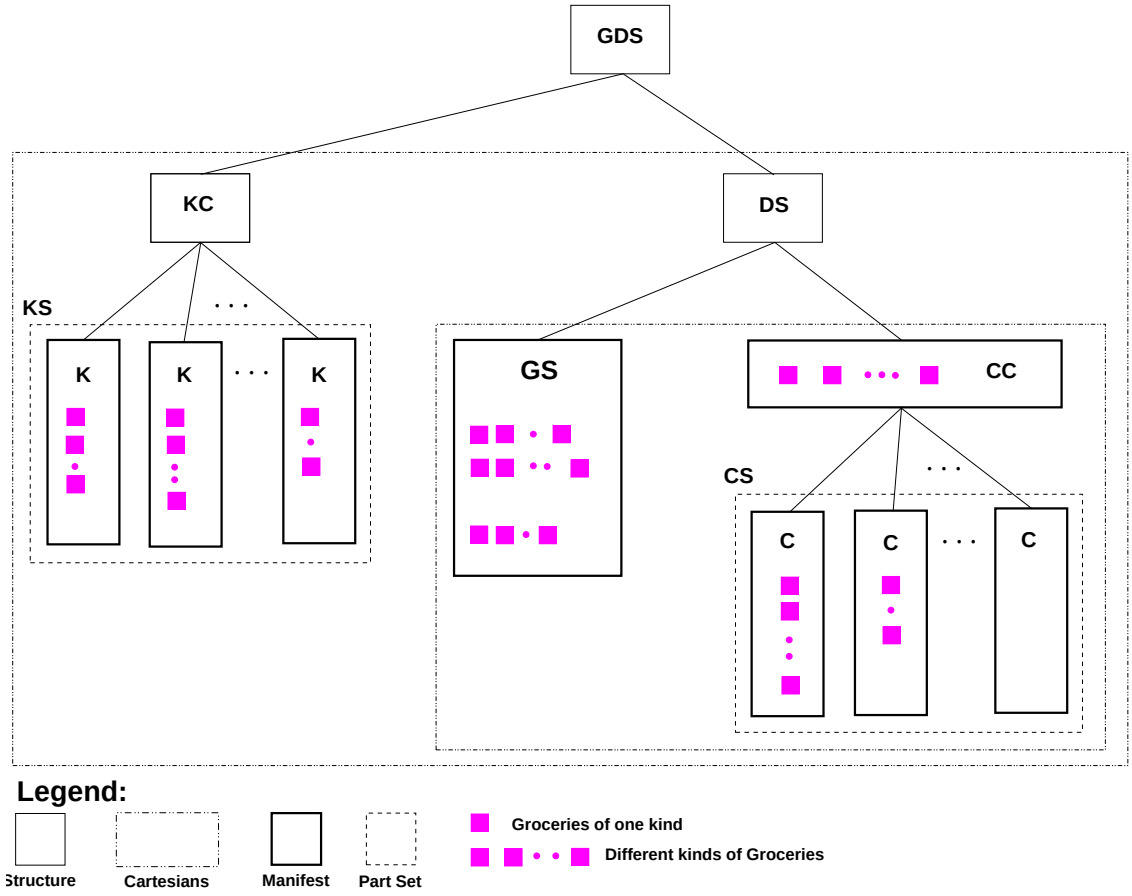

Figure 1: A **Wolt** Endurant Taxonomy

4.1.3 Endurant Taxonomy

Model Choice: 8 **Grocery Attributes:** Customers, **k:K**, the grocery store, **gst:GST**, the courier management, **cmgt:CM**, and couriers, **c:C** are all atoms, as we shall later see, in Sect. 4.2.3 on page 18, they all have, as attributes, possibly empty sets, **gs:G-set**, of all groceries, **g:G**, of all kinds! Groceries are shown as ■s in below taxonomy ■

• • •

The parts we have so far described really have no properties – such as we have described them up till now. In the next sections' four subsections we shall "endow" the state parts with properties.

4.2 Internal Qualities

The internal qualities of unique identifiers, meeologies and attibutes are values. Elements and composition of [elements of] these internal qualities are values. There are four kinds of internal qualities:

- **Unique Identification** Sect. 4.2.1 on the following page

Intentional pulls are properties of domains – and can not be treated as values.

4.2.1 Unique Identification [30, *Sect. 5.2*]

Manifest entities have unique identity. No two, seemingly similar, but otherwise distinct entities can have the same identity: they necessarily must "occupy" different, non-overlapping 𝕊ℙ𝔸ℂ𝔼s.

14. Unique identifiers are not further defined tokens.
15. Let `E` be the type of all manifest endurants.
16. With each endurant, `e:E`, we can associate, **uid_E(e)**, a unique identifier.

```
type
14. UI
15. E = GST | CM | G | C | K
value
16. uid_E: E → UI
```

17. Let σ_{uis} be the union of the unique identifiers of all state elements.

```
value
17. σuis = { uid_E(p) | p:E • p ∈ σparts }
```

4.2.1.1 Uniqueness

18. All state parts are unique alone through the uniqueness of their identifications:

```
axiom
18. card σparts = card σuis
```

4.2.1.2 Unique Identifier State For subsequent use – in expressing the well-formedness of mereologies – we calculate:

```
value
  gsti = uid_(gst)
  cmi = uid_(cm)
  cis = { uid_(c)|c:C•c ∈ cs }
  kis = { uid_(k)|k:K•k ∈ ks }
```

4.2.1.3 Disjointness of Part Identifiers We could introduce separate sets of unique identifiers:

- **KI:** customers identifiers,
- **GSTO:** grocery store identifier,
- **CMI:** courier management identifier,
- **CI:** courier identifiers.

But need not ! So here we go:

19. Let `KI, GSTI, CMI, CI` stand for the sort types of customers, grocery stores (there is but one), courier managements (there is but one), an couriers.
20. These sorts are disjoint.
21. We can define separate observers.

type
19. UI = KI | GSTI | CMI | CI
19. KI, GSTI, CMI, CI

axiom
20. (KI∩GSTI ∪ KI∩CMI ∪ KI∩CI ∪ KI∩CI ∪ GSTI∩CMII ∪ GSTI∩CI ∪ CMII∩CI) = {}

value
21. **uid_**K: K→KI, **uid_** GST: GST→GSTI, **uid_**CM: CM→CMI, **uid_**C: C→CI

4.2.2 Mereology [30, *Sect. 5.3*]

We shall examine the mereologies of all σ_{parts} elements. The mereology of:

22. **customers**, `k:K`, is a triplet of (i) the identifier, `gsti:GSTI [∈UI]`, of the store (ii) the courier management identifier, `cmgti:CMI [∈UI]`, and (iii) the set, `cis:CI-set [⊂UI]`, of all courier identifiers;[23]
23. the **grocery store,** `gst:GST`, is a pair: (i) the set, `kis:KI-set [⊂UI]`, of customer identifiers, and the (ii) single identifier, `cmi:CMI [∈UI]`, of the courier manager, ;[24]
24. the **courier management,** `cmgt:CM`, is a triplet of (i) the [single] identifier, `gsti:GSTI [∈UI]`, of the grocery store, (ii) the set, `kis:UI-set [⊂UI]`, of all customer identifiers, and (iii) the set, `cis:CI-set [⊂UI]`, of all courier identifiers[25] and
25. the mereology of **couriers,** `c:C`, is a pair (i) of the identifier, `cmgti:CMI [∈UI]`, of courier management and (ii) of all customer identifiers, `kis:UI-set [⊂UI]`.[26]

type
22. MK = GSTI × CMI × CI-**set**
23. MGST = KI-**set** × CMI
24. MCM = GSTI × KI-**set** × CI-**set**
25. MC = CMI × KI-**set**

[23] **Analysis:** Customer behaviours – in this domain description – only communicate with the grocery store, courier management and couriers.
[24] **Analysis:** These are the behaviours with which the grocery store interacts.
[25] **Analysis:** Courier management interacts with all its colleagues in the delivery service.
[26] **Analysis:** Couriers interact with its courier department and [potential] customers.

value
22. **mereo_**K: K → MK
23. **mereo_**GS: GC → MGC
24. **mereo_**CM: CC → MCM
25. **mereo_**C: C → MC

We refer to Sect. 4.2.1.2 on page 16 for the definition of the unique identifier state.

axiom
for ∀ k:K • k ∈ ks, c:C • c ∈ cs ⇒
22. **let** (gsti′,cmi′,cis′) = **mereo_**K(k) **in** gsti′=$gsti$ ∧ cmi′=cmi ∧ cis′=cis **end** ∧
23. **let** (kis′,cci′) = **mereo_**GST(gst) **in** kis′=kis ∧ cmi′=cmi **end** ∧
24. **let** (gsti′,kis′,cis′) = **mereo_**CC(cc) **in** gsi′=$gsti$ ∧ ksi′=ksi ∧ cis′=cis **end** ∧
25. **let** (cmi′,kis′)=**mereo_**C(c) **in** cmi′=cmi ∧ kis′=kis **end**

4.2.3 Attributes [30, *Sect. 5.4*]

4.2.3.1 Customers, K

Narrative:

26. For the purposed of this model Customers have names and addresses and this customer information is static.
    (a) Addresses contain the road net link or hub identifier (of the street section or intersection)[27] and numbered location at that road element.
27. Customers have – optionally – a "most recent" grocery catalog.
28. Customers have a current order status – here modeled as a map from grocery names to ordered quantity. The order status is empty if the mode is **idl**e or **rej**ected
29. Customers have programmable cash-on-hand.
30. Customers have a collection of delivered, i.e., accepted, groceries.
31. And finally customers are either in a **idl**e, or an **order**ing, or a **rej**ected mode.

Formalization:

type
26. CustInfo = Name × CustAddr ×.. `static`
26a. CustAddr = .(LI|HI) × LocNo
27. Ctlg = [GrocCtlg] `programmable`
28. Ordr = OrdrNo × (GrocName $_{m}\!\!\rightarrow$Quantity) `programmable`
35a. GrocName, Quantity = **Nat**
29. CustCash = M `programmable`
30. AGS = G-**set** `programmable`

[27] We refer to Appendix refwolt:road-net for the story of *Road Nets*.

31. Mode = **idl** | **ord** | **rej** `programmable`

value

26. **attr_**CustInfo: K → CustInfo
27. **attr_**Ctlg: K → GrocCtlg
28. **attr_**Ordr: K → GrocCtlg
29. **attr_**CustCash: K → CustCash
30. **attr_**AGS: K → AGS
31. **attr_**Mode: K → Mode

4.2.3.2 Groceries, G. Although we do not model groceries as behaiourable endurants, hence no as behaviours, we can associate a number of attributes with groceries.

32. Groceries have Grocery Names.
33. Groceries "come at" / have a Price.
34. Groceries have a Manufacturing Date anda Due Date.
35. Groceries have a Weight and a Volume

    Etc.

value

32. GrocNam
33. Price
34. MfgDate , DueDate
35. Weight , Volume , ...

type

32. **attr_**GrocNam: G→GrocNam,
33. **attr_**Price : G→Price,
34. **attr_**MfgDate: G→MfgDate, **attr_**DueDate: G→DueDate
35. **attr_**Weight: G→Weight, **attr_**Volume G→Volume

4.2.3.3 Grocery Store, GST

Narrative:

36. Grocery Stores have [programmable] Groceries.
    (a) Groceries are here modeled as a map from Grocery Names to a set of groceries of that name.
    (b) Any specific Grocery have a unique identity and a sell-by-date, etc.
37. Grocery stores have a [programmable] Catalog modeled as maps from Grocery Names to Grocery Information:
    (a) Price, Weight, Durability, Spatial Description, Packaging Information, etc.

(b) Grocery Names are further unspecified.

36. Grocery Stores have a Customer Catalog here modeled as a [possibly empty (no customers yet !)] map from Customer Identifiers to Customer Information. The map usually contains a subset of the identifiers of all customers and is updated every time a customer issues and closes an order.
37. Customer Information maps ordr numbers to orders.
38. Orders maps dates to maps from order numbers to orders. grocery names to quantity and cost.
39. Grocery Stores have Cash.

Formalization:

You will find the narratives for the formluas below on Page 19

type
36a. Groceries = GrocNames $\xrightarrow{m}$ G-**set**
36b. G = GI × ... × SellBy
36b. SellBy, GI=UI
37. GrocCtlg = GrocName $\xrightarrow{m}$ (Quantity × GrocInfo)
35a. Quantity = **Nat**
35a. GrocInfo = Price × Weight × Durability × SpaceInfo × PackInfo × ... ..
35a. Price, Weight, Durability, SpaceInfo, PackInfo, ...
35b. GrocName
36. CustCtlg = KI $\xrightarrow{m}$ CustInfo
37. CustInfo = OrdNo $\xrightarrow{m}$ Orders
38. Orders = Date $\xrightarrow{m}$ (OrdrNo $\xrightarrow{m}$ Order)
??. Order = (GrocName $\xrightarrow{m}$ Quantity) × Cost
39. Cash
value
36a. **attr_**Groceries: GST → Groceries
36b. **uid_**GI: G → UI
37. **attr_**GrocCtlg: GST → Catalog
36. **attr_**CustCtlg: GST → CustCtlg
39. **attr_**Cash: GS → M

4.2.3.4 Courier Management, CM

Narrative:

40. 'Courier Management' has a [static] road map [28] which it uses in order to determine a suitable *'delivery route'*, a *'path'*, [29] of the road net.
41. 'Courier Management' has a [static] Roster of 'Courier's. The roster is modeled as a map from Courier Identifiers to Courier Information: Courier Name, Address, Current Delvery Routes, etc.

[28] We refer to Appendix A on page 47
[29] See Sect. A.2 on page 48

42. 'Courier Management' has a [programmable] Groceries-to-be-Delivered "storage" which is modeled as a map from Customer Identifiers to triplets
    (a) of Link or Hub identifiers, Number (of location on or at identified hib or link, and the actual grocery to be delivered..
    (b) Locations are further undefined.
43. 'Courier Management' has a [programmable] plan for grocery deliveries modeled as maps from courier identifiers to delivery routes.
    (a) Deliver Routes are modeled as lists of pairs: A hub or link identifer and the possible deliveries along the identified hub or link.
    (b) These possibl delveries are here modeled as a map from cutomer identifiers to pairs of the Number of the location at the hun or along the link and a customer order

You will find further narratives for the formluas below on Page 20

type
40. RM [static]
41. CR = CI $\xrightarrow{m}$ CInfo × CName × CAddr × CurrDelRoute × ... [progammable]
41. CInfo, CName, CAddr, ...
41. CurrDelRoute = ((LI|HI) × No)*
42. GD = KI $\xrightarrow{m}$ SDL [programmable]
42a. SDL = (LI|HI) × Loc × G-**set**
42b. Loc
43. PL = CI $\xrightarrow{m}$ DR
43a. DR = (HI|LI) × Delvs
43b. Delvs = KI $\xrightarrow{m}$ (Loc × GI-**set**)

value
40. **attr_**RM: CM → RM
41. **attr_**CR: CM → GD
42. **attr_**GD: CM → CR
43. **attr_**PL: CM → PL

As we shall see next, 'couriers' are given a modified form of 'delivery routes:' Instead of its 'grocery identifiers' there are the actual groceries of those 'identifiers' – "moved over" from the 'groceries to be delivered'.

Note: *The reader may have noticed that we "juggle" around: sometimes we state* `G-set`*, sometimes* `GI-set`*. Instead of this we could have introduce, and maybe we shall do so, a notion of customer order – as a map from* `CustOrdNo` *to* `GI-set` *– and have a courier management attribute that maps* `CustOrdNo` *to* `G-set` ■

4.2.3.5 Couriers, C

Narrative:

44. Couriers have a Concrete Delivery Route. The delivery route is a list of pairs
    - a road element identifier, and
    - a map from road element locations to pairs of
      - a customer identifier,
      - and a set of groceries.

Formalisation:

You will find the narratives for the formluas below on Page 22

type
44. CDL = ((LI|HI) × (Loc $\xrightarrow{m}$ (KI × G-**set**)))* `progammable`
value
44. **attr_**CDL: C → Idx

4.2.4 Intentional Pull [30, *Sect. 5.4*]

45. An example intentional pull is – perhaps [30]
    (a) With respect to groceries:
        i. the sum total of all groceries of the general delivery system is at
            A. the grocery store plus
            B. the courier management plus
            C. the various couriers plus
            D. the customers
        ii. and that sum, such as we shall model it, is invariant: does not change:
            A. to start with all groceries are with the grocery store, i.e., none are with the other "players": courier management, couriers, and customers.
            B. The customers order and some groceries are moved to courier management.
            C. Step-by-step these are moved on to couriers,
            D. and finally "end" up with customers.
    (b) With respect to monies:
        i. Some are with customers,
        ii. other are with the grocery store.
        iii. As orders are accepted
            A. some move from customers

[30] Sun., July 19, 2026, *:39: I wrote the below yesterday. I am really not quite sure that it is about *intentional pull;* the intent of monies is here to buy groceries – so the intentional pull must be about the relations between the two! Well: all that this situation about me not being sure, illustrates that I have yet to understand, more fully, what *intentional pull* is all about!

B. to grocery stores.

(c) Over time these two sets: groceries and monies remain invariant: "stays the same"

46. There could be other intentional pulls!

We presently refrain from formalising the above!

4.3 Endurant Axioms & Proofs

TO BE WRITTEN

4.4 Endurant Theory

TO BE WRITTEN

5 Perdurants

These are the perdurants:

- **Channels** Sect. 5.1, pg. 23
- **Perdurant Taxonomy** Sect. 5.2, pg. 23
- **Actors** Sect. 5.3, pg. 24
- **Messages** Sect. 5.4, pg. 28
- **Behaviours** Sect. 5.5, pg. 31
- **Initialisation** Sect. 5.6, pg. 40

And these are the behaviours:

- 'CUSTOMER's,
- 'GROCERY STORE',
- 'COURIER MANAGEMENT' and
- 'COURIER's.

5.1 Channels

47. Behaviours interact/communicate with one-another "via" a set of channels –

48. "transferring" messages of various kinds.

channel
47. { chan[{ui,uj}] | ui,uj:UI • {ui,uj}⊆σ_{uis} } : M
type
48. M = ...

5.2 Perdurant Taxonomy

Figure 2 on the next page suggests a perdurant taxonomy for a *Wolt* domain. The "flow" of interactions between 'CUSTOMER's and the 'GROCERY STORE' is basically "left-to-right". The flow is indicated by straight- and dashed-line arrows.

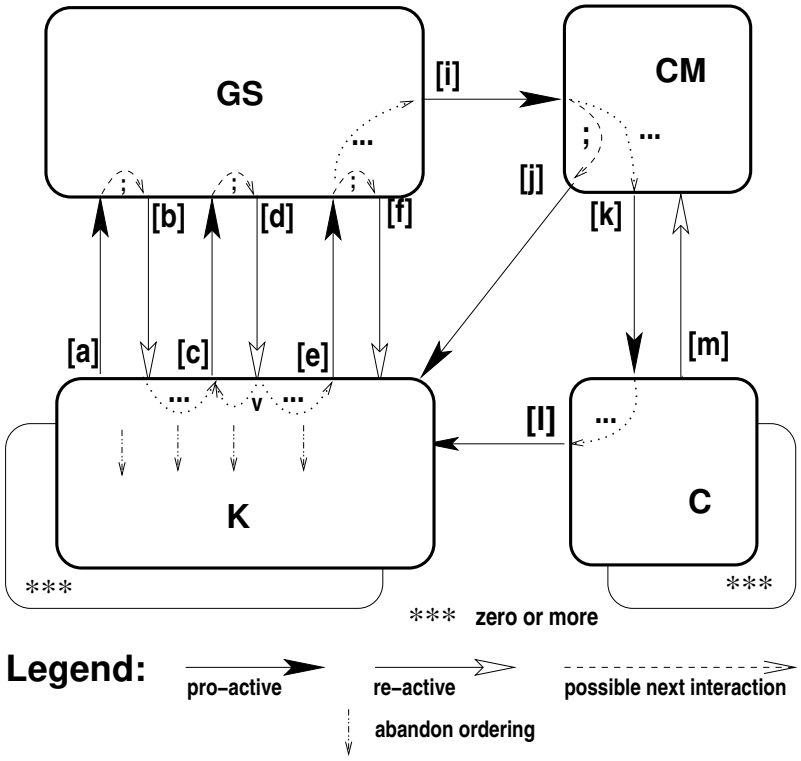

Figure 2: A Perdurant Taxonomy

5.3 Actors

We[31] sketch an **analysis** of actions and events of behaviours. Their narrative and formal **description** will appear in Sect. 5.5.2 on page 31.

We font spell:

- 'BEHAVIOUR' names in 'UNDERLINED SMALL CAPITALS';
- `behaviour interactions' in `teletype';
- 'actions' and *'events'* in '', resp. *'italic'*;
- *'attribute values'* in *'slanted'* text.

We 'emphasize' these domain concepts by enclosing them in citation marks: '...'.

• • •

Reading Note: In the text below You will find two kinds of "cross-referencing" texts:

[x↓] [x↑]

- One are margin notes like [x↓] with "matching [x↑],
- the other are Item/Page references in red, respectively blue.
- You may choose to ignore these cross-references, or, "with two hands/fingers" to keep track of their "cross-meaning"!

[31] **Note:** This text was begun on Sat.11 July, 2026, 14:43, after having structured, but not provided text for Sect. 4.2.3.

5.3.1 Customer Actions

Customer ↔ Grocery Store Interactions

49. At some, internal non-deterministically 'chosen' time a 'CUSTOMER' decides to possibly order some *'grocery'*.

    (a) The 'CUSTOMER' therefore `contacts' [ι 52a, π 26][32,33] the 'GROCERY STORE' informing it of this intent, its [unique] *'customer identification'* and its *'address.'* [a↓]

    (b) From the 'GROCERY STORE' the 'CUSTOMER' then `receives' [ι 52b, π 26] [↑b]

    - α. the accepting case:
      - (i) Either one or two *'order numbers'*: two the identified customer has a pending order, and in any case a new order number;
      - (ii) a *'catalog'* of *'groceries'* currently offered; thus if the 'CUSTOMER' [as an **event**[34]] has previously abandoned an *'order,'* the message from the 'GROCERY STORE' offers that abandoned *'order'*;
      - (iii) the 'GROCERY STORE', in fact, offers the 'CUSTOMER' to "repeat" any one of previously completed order – either for just ordering the same, or for selecting orders from one such [and possibly augment that order].
    - β. the rejecting case:
      - the 'GROCERY STORE' does not accept order to e delivered outside it *'operating area.'*

    (c) The 'CUSTOMER' 'decides' to

    - either abandon [an **event**] the inquiry [if so informed, cf. Item β just above],
    - or continue the order process:
      - either continue that, the previously abandoned *'order'*,
      - or to ignore that order and start a fresh *'order'*.

    i. The 'CUSTOMER' 'selects' a possible previously completed or pending order and 'browses' the *'catalog'* and [internal non-deterministically] 'selects' a *'grocery'* and a *'quantity'* of that *'grocery'*.

    ii. By 'select'ing and *'quantify'*ing a *'grocery'*the 'CUSTOMER' "simultaneously" `inform's [ι 52c, π 26] the 'GROCERY STORE' of that – together with the chosen order number. [c↓]

    (d) The 'CUSTOMER' 're-iterates' steps 49(c)ii and 49e, zero, one or more times.

    (e) In response to each *'grocery & quantity'* 'select'ion, the 'CUSTOMER' `receive's [ι 52(f)iA, π 26] from the 'GROCERY STORE' an *'order numbered tally'* of which *'groceries'* have so far been selected, in what *'quantity'*, and of individual and total *'costs'*. [↑d]

    (f) At some point the 'CUSTOMER' 'decide's [ι 52f, π 26]. to 'end' the 'order'ing either in an "orderly" manner or to "abandon" [an **event**] the order: [e↓]

    i. In 'completing an order' [in an orderly fashion] the 'CUSTOMER'

[32] – the **red ιtem πage** references are matched by corresponding **blue ιtem πage** "cross" references
[33] The character-labeled margin entry should be understood in the context of Fig. 2 on the preceding page,
[34] Events are so marked. All other are trans**action**s

ii. `transfers` the appropriate (i) *'order number'*, (ii) *'cash'* payment and information about that customer: (iii) *'name, address, delivery time'*.

[↑f] iii. The 'CUSTOMER' then awaits an `acknowledgment` [ι 52(f)iA, π 26] response from the 'GROCERY STORE:'
- either *'acceptance'* of the order and confirmation of delivery time,
- or *'rejection'* of the order – in which case *'cash'* is declined.

iv. Whereupon the 'CUSTOMER' "closes" down contact with the 'GROCERY STORE'.

50. Customers receive deliveries from couriers with no interaction.

51. There may be other ... 'CUSTOMER'/'GROCERY STORE' interactions.

5.3.2 Grocery Store Actions

Grocery Store ↔ Customer Interactions

52. The 'GROCERY STORE' [primarily and] external non-deterministically

[↑a] (a) awaits [ι 49a, π 25][35] `inquiries` from 'CUSTOMER's wishing to shop for groceries.

[b↓] (b) The 'GROCERY STORE' `respond`s [ι 49b, π 25] to such *'inquiries'* by finding out: can it service this customer (is the customer outside the grocery store's operating area), and if it can service, has it serviced this customer before. Thus:
- either sending the customer:
  - (i) a pair or a single, the new, of *'order numbers:'* two if pending order;
  - (ii) a *'catalog'* of `groceries` currently offered and in *'stock'*;
  - (iii) if the customer has some pending order, then the number of that order, and, in any case, a [new] order number;
  - (iv) a display of all previous, if any, completed orders.
- or informing the 'CUSTOMER' that the 'GROCERY STORE' cannot handle deliveries "outside" its *'operating area'*.

(c) If within its *'operating area'* the 'GROCERY STORE' awaits for that 'CUSTOMER'
[↑c] to `inform` [ι 49(c)ii, π 25] it of which *'grocery'* and in which *'quantity'* that 'CUSTOMER' would/might wish to 'acquire'.– by *'order number'*.

[d↓] (d) The 'GROCERY STORE' makes a 'note' of that *'order'* and [ι 49(c)ii, π 25] `acknowledges` the sub-order – by number.

(e) The 'GROCERY STORE' re-iterates steps 52c–52d zero, one or more times.

[↑e] (f) At some time the 'CUSTOMER' [somehow] `indicates` [ι 49f, π 25] that it

i. either 'completes' the *'order'* in an orderly fashion – by transferring to the 'GROCERY STORE' the *'full order: cash payment, and delivery address information'* – by *'order number'*.

[f↓] A. in which case the 'GROCERY STORE' `responds` [ι 49e, π 25] by
- (α) either accepting the order – by *'order number'*, i.e., the cash payment,

[35] – the **blue ιtem πage** references are matched by corresponding **red ιtem πage** "cross" references

- (β) or rejecting the order – by *'order number'*, i.e., returning the cash payment.

ii. or 'abandons' the *'order'*, an **event**,
   A. in which case 'GROCERY STORE' 'registers' the order – by *'order number'* – as *'pending'*
   B. and [thus] "terminates" this order-interaction with that customer;

iii. Or, the 'GROCERY STORE' – say through a "time-out" – [somehow] decides that the 'CUSTOMER' may have abandoned [an **event**] the order in which case it 'registers' the order as *'pending'* and [thus] "terminates" this order-interaction with that customer.

Grocery Store → Courier Management Interactions

(g) The 'GROCERY STORE' collects *'customer order'* packages `convey' [ι 53a, π 27] this package – by *'order number'* – to 'COURIER MANAGEMENT'. [i↓]

5.3.3 Courier Management Actions

Courier Management ↔ Courier Interactions:

53. The 'COURIER MANAGEMENT' iterates, non-deterministically, between 53a-54c and 52g.

(a) The 'COURIER MANAGEMENT' `receives' [ι 52g, π 27] from the 'GROCERY STORE' a *'delivery order'* – an *'order numbered set'* of [physical] *'groceries'* – to be `delivered' to a 'CUSTOMER'. [↑i]

(b) The 'COURIER MANAGEMENT' informs [ι 55a, π 28] The 'COURIER' the 'CUSTOMER' of the *'pending delivery'* – by *'order number'*, etc. [j↓]

(c) 'COURIER MANAGEMENT' then [ι 53c, π 27]↔[ι 53c, π 27] arranges for an available 'COURIER' to be selected[36], – such that that courier may have several deliveries as part of one circular route – and given the *'delivery'* (*'order number'* identified *'set of groceries'* and *'customer designation'*. [k↓] [↑k]

(d) At more-or-less irregular times the 'COURIER MANAGEMENT' *'welcomes'*[ι 54c, π 27] a *'returning'* 'COURIER' – possibly also returning rejects.[37] [n↓]

5.3.4 Courier Actions

Courier ↔ Customer Interactions:.

54. The selected 'COURIER' commences the *'delivery route'*, visiting potentially several 'CUSTOMER's.

(a) At each 'CUSTOMER' the 'COURIER' `delivers' [ι 52a, π 26] – [l↓]

(b) receiving in return an acknowledgment [ι 52b, π 26] and continues; [↑m]

(c) or ends his delivery route, *'returning'* [ι 53d, π 27] to the 'COURIER MANAGEMENT' – possibly also returning rejects.[38] [m↓]

[36] This "arrangement" involves some work: consulting a *'road map'* [See Appendix A on page 47] to decide upon a suitable courier and route. Details are given in Sect. 5.5.2.4.

[37] We shall not model this facet.

[38] We shall not model this facet.

55. At some, internal non-deterministically 'chosen' time
    [↑j] (a) 'CUSTOMER's `receives'[ι 54a, π 27] from 'COURIER MANAGEMENT' a *'order number identified notification of pending delivery'*.
    [↑l] (b) the 'CUSTOMER' `receives'[ι 54a, π 27] from a 'COURIER' a *'delivery'*.
    [m↓] (c) The 'CUSTOMER' `acknowledges or rejects' [ι 54a, π 27] the *'delivery'*.
    (d) and nothing more happens! That is: we do not model how the delivered groceries :are handled – i.e., consumed and thus "disappears"!

5.4 Messages

This section relates strongly to Sect. 5.1 on page 23: It details the message type M.

By messages we mean the "contents" of the interactions, i.e., communications, between behaviours. These have been hinted at in the previous section on Actors. They have been highlighted in that section in two ways: (i) by margin notes of the kind: [x↓] and [x↑] and by ιtem/πage cross-references – in corresponding colours![39]

56. There are the interactions marked "on" Fig. 2 on page 24's arrows `[a, b, c, d, e, f, i, j, k, l, m, n]`. Hence we type these messages `Ma, Mb, Mc, Md, Me, Mf, Mi, Mj, Mk, Ml` and `Mm`.

type
56. M = Ma|Mb|Mc|Md|Me|Mf|Mi|Mj|Mk|Ml|Mm

- In Fig. 2 on page 24 we have indicated [between 'BEHAVIOURS'] message transactions by arrows:
  - **black** arrowheads mean that the 'BEHAVIOUR'.from which the arrow emanates is `pro-active`;
  - white arrows means tht the 'BEHAVIOUR'.from which the arrow emanates is `re-active`;
  - dashed arrows between receipt of pro-active and re-active messages means that the re-actor responds in one-and-the-same composite, i.e., synchronised action – these arrows are, additionally, "sugared" with semi-cola: ";"; first receipt, then send;
  - dotted arrows between receipt of messages means that the re-actor may respond, sooner or later – these arrows are, additionally, "sugared" with semi-cola: "..."; send after some time!

[39] **Note:** *Great care must be taken in designing message types: they form crucial interfaces between behaviours. "Behind" these interfaces the domain analyser cum describer, the requirements and the software engineer can do their work while trusting in the stability of the interface: that it does not change!* Maybe I have not taken "enough great care". Please understand that as I write this [i.e., defining these message types] I have yet to define the 'CUSTOMER, GROCERY STORE, COURIER MANAGEMENT' and 'COURIER' behaviours. As I define these I may have to "correct" the message types.

- We have highlighted TIME and **OrdrNo** for the convenence of the reader: to better "see" that almos all messages are TIME-stamped and that **OrdrNo**s are an essential element of most messages.

In Items 57– **??** on page ?? we narrate and formalise these as (::) disjoint types. We define the type of these messages, corresponding to [**a**↓]–[**n**↓][40]

57. **a:** The customer conveys its intent to start ordering by sending it is address: [ι **49a**, π **25**] → [ι **52a**, π **26**]

    57. **type** Ma = StartOrdr :: TIME × Addr

58. **b:** The grocery store [ι **52b**, π **26**] → [ι **49b**, π **25**]

    (a) may reject the inquiry

    (b) or accept and then sends the inquiring customer its current catalog.

    type
    58. Mb = TIME × (RejMsg | AccptMsg)
    58a. RejMsg :: **reject**
    58b. AccptMsg :: Catalog × ([**OrdrNo**]×**OrdrNo**) × (**OrdrNo** $_{m}\!\!\rightarrow$(Date×Ordr))

59. **c:** The customer selects an order number (a pending or "the" newly offered) and a single or several different kinds of groceries and for each of these a quantity and informs the grocery store of this: [ι **49(c)ii**, π **25**] → [ι **52c**, π **26**]

    type
    59. Mc = SubOrdr :: TIME × **OrdrNo** × (GrocNam $_{m}\!\!\rightarrow$Quantity)
    59. **OrdrNo** , GrocNam
    59. Quantity = **Nat**

60. **d:** The grocery store responds to each sub-order by sending the customer a tally of what has been ordered so far and its cost: [ι **52(f)iA**, π **26**] → [ι **49e**, π **25**]

    type
    60. Md = Tally :: TIME × **OrdrNo** × (GrocNam $_{m}\!\!\rightarrow$Quantity) × Cost
    60. Cost = **Nat**

61. **e:** The customer [ι **49f**, π **25**] → [ι **52f**, π **26**]

    (a) either end the ordering by stating so in an orderly fashion,

    (b) or just ceases interacting with the grocery store.

[40] [**g**↓]–[**h**↓] and [**g**↑]–[**h**↑] currently absent !

type
61. Me = EndOrdr = TIME × (Orderly | Abandon)
61a. Orderly :: **OrdrNo** × Payment
61a. Payment = Cash
61b. Abandon :: [time−**out**]

[ι 52(f)iA, π 26] → [ι 49(f)ii, π 26]

62. **f:** The grocery store either
    (a) accepts the order,
    (b) or rejects and returns cash paid.

type
62. Mf :: TIME × (Accept | Return)
62a. Accept :: **OrdrNo**
62b. Return :: **OrdrNo** × Cash

[ι 52g, π 27] → [ι 53a, π 27]

63. **i:** The grocery store [packs the order and] transfers
    (a) the order numbered package to courier management
    (b) with necessary customer information.

type
63. Mi = XferPack :: TIME × Package × CustInfo
63a. Package = **OrdrNo** × G-**set**
63b. CustInfo = Ki × (LI|HI) × No

[ι 55a, π 28] → [ι 53b, π 27]

64. **j:** Courier management informs customers of pending deliveries: tally, day and hour.

type
64. Mj = Pending :: TIME × **OrdrNo** × Tally × Date
64. Date = DayMonthHour

[ι 53c, π 27] → [ι 53c, π 27]

65. **k:** Once courier management has assigned a courier it `hands over' to that courier a pair: the [appropritaly calculated and to be current] delivery route and a schematic delivery to that person together with, i.e., as an element of a delivery route.

type
65. Mk = HandOvr :: TIME × CurrDelRoute × GD

[ι 52a, π 26] → [ι 54a, π 27]

66. **l:** The courier has reached a customer for delivery and delivers the order numbered packages of groceries.

66. **type** Ml = Dlvr :: TIME × **OrdrNo** × G-**set**

[ι 54c, π 27] → [ι 53d, π 27]

67. **m:** The courier has ended the current delivery route and signs back at the office.

??. **type** Mm = Return :: TIME × CI

[If the "return" map is nil, i.e., empty ([]), the delvery has been accepted.]

5.5 Behaviours

There are four 'BEHAVIOUR's to be defined:

- 'CUSTOMER's,
- 'GROCERY STORE',
- 'COURIER MANAGEMENT' and
- 'COURIER's.

5.5.1 Signatures

The arguments to all behaviours [informally] are the part :

- identifier,
- mereology,
- attributes:
  - static,
  - monitorable[41] and
  - programmable.

The formal signatures of:

68. 'CUSTOMER'
69. 'GROCERY STORE'
70. 'COURIER MANAGEMENT'
71. 'COURIER'

are:

68. cust: KI → MK → (CustInfo×CustAddr) → ... → (Ctlg×Ordr×CustCash×AGS×Mode) → **Unit**
69. groc_store: GSI → MGS → GrocCtlg → ... → (Groceries×CustCtlg×Cash) → **Unit**
70. courier_mgt: CMI → MCM → (RM×CR) → ... → (GD×PL) → **Unit**
71. courier: CI → MC → ... → (CDL×Idx) → **Unit**

5.5.2 Definitions

5.5.2.1 An Idle Behaviour. Evaluation of expressions and elaboration of statements mathematical semantics "takes no time"! Occurs instantaneously. To model that time may pass between interpretation of formal description [RSL] clauses[42], we introduce an `idle` behaviour.

72. We postulate, i.e., describe, a behaviour, `idle`.
    (a) As a behaviour `idle` ...
    (b) We just hint at the/a time-out!

value
72. idle: UI → Mereology → Static → Monitorable → Programmable → **Unit**
72. idle(ui)(mer)(sta)(...)(prg) ≡ time_out()
72a. time_out: **Unit** → **Unit**
72b. time_out() ≡ ...

[41] – of which there are none in this model
[42] (expressions, statements and output/input [CSP] interactions[43])

5.5.2.2 Customers.

We refer to Sect. 5.3.1 on page 25 for an informal arrative of customer actions and to Sect. 5.4 on page 28 for both narrative and formal descriptions of customer messages.

Customer Invocation:

73. 'Customer' invocation
    (a) alternates between being
    (b) **idle**: ready to order,
    (c) **ordr**: "within" an ordering phase, or
    (d) **rej**: being rejected ordering.

```
value
73.  cust(ki)(mk)(kinfo,kaddr)(...)(cash,ags,mode) ≡
73a.     case mode of
73b.         idle  → cust_idle(ki)(mk)(kinfo,kaddr)(...)(cash,ags,idl)
73c.         ordr  → cust_ordr(ki)(mk)(kinfo,kaddr)(...)(cash,ags,ord)
73d.         rej   → cust_rej(ki)(mk)(kinfo,kaddr)(...)(cash,ags,rej)
73a.     end
```

Idle Customer:

74. An **idl**e 'customer' external non-deterministically alternates between
    (a) "idling", here modeled by a **skip** [sequentially] followed by an idle customer; **or**
    (b) **inquiring** the grocery store as to commencing an order
    (c) [sequentially] followed by an idle customer; **or**
    (d) **awaiting** a response from the grocery store:
        i. the response can be any one of two:
        ii. a **rejection**
        iii. in which case the customer transits to **rej**ect mode, **or**
        iv. an **acceptance** – in which case the customer
            A. arbitrarily selects either a past order number, if not **nil**, or a new, offered, order number;
            B. if a [non-**nil**] past order number is chosen then that pending order is chosen, otherwise a "fresh, empty" order is to e used;
            C. whereupon the customer becomes an ordering customer with an updated order.

ι49a,π25
ι52a,π26
ι52b,π26
ι49b,π25

```
value
74.  cust_idl(ki)(gsi,cmi,cis)(info,addr)(...)(ctlg,ordr,cash,ags,idl) ≡
74a.       ( idle() ; cust_idl(ki)(gsi,cmi,cis)(info,addr)(...)(ctlg,ordr,cash,ags,idl) )
74b.[a] [] (chan[ {ki,gsi} ] ! mk_StartOrdr(obs_TIME(),addr) ;
74c.       cust_ord(ki)(mk)(info,addr)(...)(ctlg,ordr,cash,ags,idl) )
```

```
74d.[b] [] (let mb = chan[ {ki,gsi} ] ?  in
74(d)i.        case mb of
74(d)ii.          mk_RejMsg(τ,reject) →
74(d)iii.             cust_idl(ki)(gsi,cmi,cis)(info,addr)(...)(ctlg,ordr,cash,ags,rej),
74(d)iv.          mk_AcceptMsg(τ,((pon,non),ords)) →
74(d)ivA.             (let no:OrdrNo • no ∈ {pon,non} \ {nil} in
74(d)ivB.              let ordr′ = if no=pno then ords(pno) else [ ] end in
74(d)ivC.              cust_ord(ki)(gsi,cmi,cis)(info,addr)(...)(ctlg,ordr′,cash,ags,no,ord)
74(d)i.        end end )
74.         end end )
```

Ordering Customer:

75. An **ord**ering 'CUSTOMER' internal non-deterministically alternates between
    (a) **either** idling ;
    (b) **ot** selecting and [sub-]ordering
    (c) **or** deciding to complete an ordering, in an orderly fashion:
    (d) **or** simply **abandoning** the ordering by reverting to be an **idl**e consumer !
    (e) **or** receiving notification of delivery of past order ι52g,π27
    (f) **or** awaiting delivery of past order ι53a,π27
    – in all cases **ord**ering customer resumes being an ordering customer. ι52g,π27 ι53a,π27

```
value
75.  cust_ord(ki)(mereo)(static)(...)(prgr) ≡
75a.     idle() ; cust_ord(ki)(mereo)(static)(...)(prgr) ≡
75b.  [] cust_ord_select(ki)(mereo)(static)(...)(prgr)
75c.  [] cust_ord_end_ok(ki)(mereo)(static)(...)(prgr)
75d.  [] cust_ord_end_abandon(ki)(mereo)(static)(...)(prgr)
75e.  [] cust_ord_await_note(ki)(mereo)(static)(...)(prgr)
75f.  [] cust_ord_await_delv(ki)(mereo)(static)(...)(prgr)
```

Selecting Customer:

76. A **select**ing 'CUSTOMER'
    (a) selects quantities of one or more kinds of groceries, ι49(c)ii,π25
    (b) sub-ordering these from the grocery store, ι52c,π26
    (c) resuming being an **ord**ering customer ;

```
value
76.  cust_select(ki)(gsi,cmi,cis)(info,addr)(...)(ctlg,ordr,cash,ags,on,ord) ≡
76a.       (let ordr′ = sel_grocs(ctlg,ordr) in
76b.[c]     chan[ {ki,gsi} ] ! mk_SubOrdr(obs_TIME(),on,ordr′) ;
76c.        cust_ord(ki)(gsi,cmi,cis)(info,addr)(...)(ctlg,ordr′,cash,ags,on,ord) end)
```

Completing Customer:

77. The 'CUSTOMER' decides to complete the ordering, in an orderly fashion, hence:
    (a) calculates the payment,
    (b) notifies the grocery store,
    (c) awaits its response,
    (d) examines that response:
    (e) if acceptance
    (f) it reverts to become an idle customer with payment deducted from its cash,
    (g) if reject,
    (h) it also reverts to become an idle customer but now with no monies deducted from its cash.

```
value
77.  cust_end_ok(ki)(gsi,cmi,cis)(info,addr)(...)(ctlg,ordr,cash,ags,on,ord) ≡
77a.     let payment = topay(...) in
77b.[e]  chan[ {ki,gsi} ] ! mk_EndOrdr(obs_TIME(),mk_Orderly(on,payment)) ;
77c..[f] let mk_Mf(τ,aor) = chan[ {gsi,ki} ] ? in
77d.     case aor of
77e.          mk_Accept(ono) →
77f.             cust_idl(ki)(gsi,cmi,cis)(info,addr)(...)(ctlg,ordr,cash−payment,ags,idl),
77g.          mk_Return(ono,payment) →
77h.             cust_idl(ki)(gsi,cmi,cis)(info,addr)(...)(ctlg,ordr,cash,ags,idl)
77.      end end end
```

Abandoning Customer:

78. The 'CUSTOMER' simply just **abandon**s the ordering –
    (a) reverting to be an **idl**e consumer!

```
77.  cust_end_abandon(ki)(gsi,cmi,cis)(info,addr)(...)(ctlg,ordr,cash,ags,on,ord) ≡
78a.     cust_ord(ki)(gsi,cmi,cis)(info,addr)(...)(ctlg,ordr,cash,ags,idl).
```

Customer Receiving Notification of Delivery:

79. 'CUSTOMER'
    (a)
    (b)

```
value
79. cust_ord_await_notification(ki)(mereo)(static)(...)(prgr) ≡
79a.
79b.
```

Customer Awaiting Delivery:

80. The 'CUSTOMER' awaits
    (a) delivery of past order[s]
    (b) and resumes being an ordering customer with updated groceries.

ι52g,π27
ι53a,π27

```
80.  cust_ordr_await_delivery(ki)(gsi,cmi,cis)(info,addr)(...)(ctlg,ordr,cash,ags,on,ord) ≡
80a.[l]   let mk_Dlvr(tai,on′,gs) in ▯{ chan[ {ki,ci} ] ? | ci ∈ cis } in
80b.      cust_ord(ki)(gsi,cmi,cis)(info,addr)(...)(ctlg,ordr,cash,ags ∪ gs,on,ord) end
```

Customer Rejecting Delivery:

81. A **rej**ected 'CUSTOMER' simply ceases to be a customer!

```
value
81.  cust_reject(ki)(mk)(kinfo,kaddr)(...)(cash,ags,rej) ≡ stop
```

5.5.2.3 Grocery Store. We refer to Sect. 5.3.2 on page 26 for an informal arrative of grocery store actions and to Sect. 5.4 on page 28 for both narrative and formal descriptions of grocery store messages.

General Grocery Store:

82. A 'GROCERY STORE'[44] external non-deterministically alternates between
    (a) **either idling**;
    (b) **or:** awaiting from customers either of three kinds of inquiries:
        i. **either:** customers inquire wrt. commencing an order informing the grocery store of its [delivery] address;
        ii. **or:** customers order groceries:
        iii. **or:** customers end their ordering:
    (c) **or:** customers receive delivery notifications;
    (d) **or:** customers receive deliveries.

```
value
82.  groc_store(gsi)(mereo)(static)(...)(prgr) ≡
82a.     idle() ; groc_store(gsi)(mereo)(static)(...)(prgr)
82b.  ▯ (let msg = ▯ { chan[{gsi,ki}] ? | ki ∈ kis } in
82b.     case msg of
82(b)i. [a]      mk_StartOrdr(τ,c_addr)
82(b)i.          → groc_store_start_ordr(gsi)(mereo)(static)(...)(prgr)(msg)
82(b)ii. [c]     mk_SubOrdr(τ,on,order)
```

[44] **Dear reader:** in the formalisation below – in order to keep the formula within lines – I have had to use very short, rather inscrutable names for attributes – sorry!

```
82(b)ii.          → groc_store_ordr(gsi)(mereo)(static)(...)(prgr)(msg)
82(b)iii.[e]    mk_EndOrdr(τ,ooa)
82(b)iii.         → groc_store_end_ordr(gsi)(mereo)(static)(...)(prgr)(msg)
82b.    end end )
82c.[j] [] groc_store_rcv_notes(gsi)(mereo)(static)(...)(prgr)
82d.[l] [] groc_store_rcv_delvs(gsi)(mereo)(static)(...)(prgr)
```

New Order:

83. A ‘GROCERY STORE’ handling a customer’s new order

    (a) checks whether the ‘CUSTOMER’ resides within the grocery store’s operating area,

    (b) if the ‘CUSTOMER’ [delivery] address is outside the ‘GROCERY STORE’’s operating area the customer is so informed – and the inquiry ends:

    (c) the ‘GROCERY STORE’ resumes being a grocery store;

    (d) if the ‘CUSTOMER’ [delivery] address is within the ‘GROCERY STORE’’s operating area the ‘GROCERY STORE’ examines whether the ‘CUSTOMER’ has shopped before and informs the ‘CUSTOMER’ of previous and pending orders and offers a new order number

```
value
83.  groc_store_start_ordr(gsi)(gsi,kis,cmi)(g_area,g_ctlg)(...)(gs,c_ctlg,cash)
83.                                                    (mk_StartOrdr(τ,c_addr)) ≡
83a.      if ~ witin_area(c_addr,g_area)(rm)[45]
83a.        then
83b.[b]       chan[{si,ki}] ! (obs_TIME(),mk_RejMsg(reject))
83c.          groc_store(gsi)(gsi,kis,cmi)(g_area,g_ctlg)(...)(gs,c_ctlg,cash)
83d.        else
83d.          let ((o,n,os).ncc) = examine_customer(cust_ctlg) in
83d.[b]       chan[{si,ki}] ! (obs_TIME(),mk_AccptMsg(g_ctlg,(o,n),os));
83d.          groc_store(gsi)(gsi,kis,cmi)(g_area,g_ctlg)(...)(gs,ncc,cash) end
83.       end
```

We defer description of the `witin_area` and, `examine` functions – till later!

Ongoing Order:

84. A ‘GROCERY STORE’ handling a customer’s ongoing ordering:

    (a) the grocery store notes the order, checks that there is the desired quantity and marks the [possibly lower] quantity for reserved,

    (b) informs the customer with an updated order, and

    (c) resumes being a grocery store;

[45] The `rm=(ls,hs)` road map is a global value. See Appendix A.

```
value
84.   groc_store_ordr(gsi)(gsi,kis,cmi)(g_area,g_ctlg)(...)(gs,c_ctlg,cash)
84.                                                    (mk_SubOrdr(τ,on,order)) ≡
84a.      let (new_ordr,ncc,ngs,cost) = marks_the_order(gs,order) in
84b.[d]   chan[{gsi,ki}] ! (obs_TIME(),mk_Tally(on,new_ordr,cost)) ;
84c.      groc_store(gsi)(gsi,kis,cmi)(g_area,g_ctlg)(...)(ngs,c_ctlg,cash) end
```

We defer description of the `marks_the_order` function – till later!

Orderly End:

85. A ‘GROCERY STORE’ handles an orderly end to a[ny] ‘CUSTOMER’s ordering
    (a) by *‘accepting’* the order,
    (b)
    (c)
    (d) and resume being a ‘GROCERY STORE’ .

```
85.   groc_store_end_ordr(gsi)(gsi,kis,cmi)(g_area,g_ctlg)(...)(gs,c_ctlg,cash)
85.                                                    (mk_Orderly(ono,payment)) ≡
85a.     let (package,gs′) = collect_order(gs,...) in
85b.[f]   ( chan[{gsi,ki}] ! (obs_TIME(),mk_Mf(mk_Accept(on))) ‖
85b.[i]     chan[{gsi,cmi}] ! (obs_TIME(),mk_Package(ki[46],ono,package)) ) ;
85d.      groc_store(gsi)(gsi,kis,cmi)(g_area,g_ctlg)(...)(gs′,ncc,cash+payment) end
```

Abandon:

86. A ‘GROCERY STORE’ *‘rejects’* a ‘COSTUMER’s payment
    (a) by first `` `returning' `` the *‘payment’* and
    (b) then resume being a ‘GROCERY STORE’ .

```
86.   groc_store_end_ordr(gsi)(gsi,kis,cmi)(g_area,g_ctlg)(...)(gs,c_ctlg,cash)
86.                                                    (mk_Abandon(...)) ≡
86a.[f]   chan[{gsi,ki}] ! (obs_TIME(),mk_Mf(mk_Reject(on,payment))) ;
86b.      groc_store(gsi)(gsi,kis,cmi)(g_area,g_ctlg)(...)(gs′,ncc,cash)
```

5.5.2.4 Courier Management.

87. ‘COURIER MANAGEMENT’ engages in four interactions:
    - **Reactive:**
      - receive `packages` to customers `ki` from grocer management, and
      - receive `returning` couriers;

[46] **Editorial Notes:** **Check** that `ki` is “accessible”.

and

- **Proactive:**
  - `send` notifications of delivery to customers, and
  - `hand` over packages and delivery routes to couriers:

Hence:

(a) `` `receive' `` appropriately "addresses" *'packages'* from the 'GROCERY STORE':
   i. 'schedule' package dispatches: delivery routes and couriers;
   ii. `` `notify' `` 'CUSTOMER's of pending deliveries;
   iii. `` `hands [there]' `` over to and [thereby] dispatches 'COURIER's; and
   iv. 'revert' to being 'COURIER MANAGEMENT'.

(b) `` `receive' `` returning 'COURIERS'
   i. update courier roster and
   ii. 'revert' to being 'COURIER MANAGEMENT'.

```
value
87.  courier_mgt(gmi)(mcm)(rm)(...)(cr,gd,pl) ≡
87a.     let (msg,ui) = [] { chan[{ui,cmi}] ? | ui ∈ {gs}∪kis } in
87a.     case msg of
87a.         (τ,mk_Package(ki,ono,package)) →
87(a)i.             let ((dlr,ci),(gd′,pl′)) = schedule_dispatch(rm,cr,...,gd,pl) in
87(a)ii.[j]         ( chan[{cmi,ki}] ! mk_    ||
87(a)iii.[k]          chan[{cmi,ci}] ! mk_    ) ;
87(a)iv.              courier_mgt(gmi)(mcm)(rm)(...)(cr,gd′,pl′) end
87b.         (τ,mk_Mn(mk_Return(ui))) →
87(b)i.             let (cr′,pl′) = update_courier_roster_and_plan(cr,ui) in
87(b)ii.            courier_mgt(gmi)(mcm)(rm)(...)(crr′,gd′,pl′) end
87.      end end
```

We presently leave the **schedule_dispatch** – as well as the other **magenta coloured** [auxiliary] functions unspecified.

- They are – **of course** – major functions.
- About these presently unspecified functions we can say this:
  - their formal domain description, in terms of properties, may no be so difficult,
  - but their software implementations, following an appropriate requirements prescription, imply oftentimes "tricky" *data structures & algorithms*.
- So there is one aspect that – sort of – characterises *domain engineering* in contrast to *domain engineering*.

5.5.2.5 Couriers.

Couriers, `ci:CI`, external non-deterministically alternate between being 'IDLE' and being 'EN ROUTE' and, at any time, at a position of that route, *'cdl:CDL'*. A courier "en route" travels the `road net`.[47]

$\iota 44, \pi, 22$ The courier's delivery route, *'cdl:CDL'* is an attribute of couriers. It alternates between road links (street segments) and hubs (street intersections), `ri`; *'cdl'* states the *'number, no'* of the customer on that road element, and the groceries, *'gs'*, to be delivered.

type
$\iota 44, \pi, 22.$ CDL = ((LI|HI) × (KI $\xrightarrow[m]{}$ (on:OrdrNo × gs:G-**set**)))*

88. The courier external non-deterministically awaits
    (a) being requested by courier management to deliver groceries to customers along a circular[48] delivery route and then proceeds to travel that route:
        i. Examines the delivery route.
        ii. If it is empty it means that the courier has no job to do at the moment and we can assume (as an **axiom**) that it is "at home", i.e., at the courier management.
        iii. If it is not empty, then it is on road element, `ri`, it has a possibly empty, customer indexed set, `delvs` of deliveries.
        iv. The couriers then delivers order-numbered groceries to all recorded customers;
        v. If that was the last road elements on the delivery route
        vi. then that road element "leads" to the courier management, the courier is back at its base and notifies the courier management.
        vii. For each road element visited, i.e., for each element of the delivery list, the courier "advances" to the next element or is back home!

```
value
88.  courier: CI → MC → Stat → Mereo → CDL → Unit
88.  courier(ci)(cmi,kis)(...)(cdl′) ≡
88a.     let (τ,mk_Del(cdl)) = chan[{ci,cmi}] ? in          [ axiom sel_ri(hd cdl) = sel_ri(tl cdl) ]
88(a)i.  case cdl of
88(a)ii.    ⟨⟩ → (idle() ; courier(ci)(cmi,kis)(...)(tl cdl)),
88(a)iii.   ⟨(ri,delvs)⟩^cdl′ →
88(a)iv.       → [] ( { chan[{ci,ki}] ! mk_Ml(mk_Dlvr(obs_TIME(),(on:delvs(ki),gs:delvs(ki))))
88(a)iv.                                                  | ki:KI • ki ∈ dom delvs } ;
88(a)v.                  if cdl′ = ⟨⟩
88(a)vi.                    then chan[{ci,cmi}] ! mk_(Return(obs_TIME(),ci))
88(a)vi.                    else skip end )
88(a)i.  courier(ci)(cmi,kis)(...)(cdl′) end
88(a)vii. end
```

[47] Cf. Appendix A on page 47
[48] See Sect. A.2.1 on page 49 for definition os circular routes.

5.6 Initialisation

We refer to Sect. 5.5.1 on page 31 for recalling behaviour signatures and to Sect. 35 on page 60 for recalling the global state values of customers ks, the grocery store gst, the courier management $cmgt$ and the couriers cs.

89. The 'INITIALISATION' behaviour invokes
    (a) all customers ks,
    (b) the grocery store gst,
    (c) the courier management$cmgt$, and
    (d) all couriers cs.

```
value
89.  initialisation() ≡
89a.      { cust(uid_K(k))(mereo_K(k))
89a.          (attr_CustInfo(k),attr_CustAddr(k))(...)
89a.              (attr_Ctlg(k),attr_Ordr(k),attr_CustCash(k),attr_AGS(k),attr_Mode(k))
89a.        | k:K • k ∈ ks }
89b.  ‖ groc_store(uid_GSgst)(mereo_GS(gst))
89b.          (attr_GrocCtlggst)(...)
89b.              (attr_Groceries(gst),attr_CustCtlg(gst),attr_Cash(gst))
89c.  ‖ courier_mgt(uid_CM(cmgt))(mereo_CM(cmgt))
89c.          (attr_RM(cmgt),attr_CR(cmgt))(...)
89c.              (attr_GD(cmgt),attr_PL(cmgt))
89d.  ‖ { courier(uid_C(c))(mereo_C(c))
89d.          (...)(...)
89d.              (attr_CDL(c),attr_PL(c))
89d.        | c:C • c ∈ cs }
```

5.7 Perdurant Axioms & Proofs

5.8 Perdurant Theory

6 Conclusion

6.1 What has been Achieved

TO BE WRITTEN

6.2 What has Not been Achieved

We have not – it seems – done a "good-enough job" of:

90. expressing all relevant endurant and perdurant axioms;
91. expressing any relevant endurant and perdurant theories, let alone their proofs;
92. ...

6.3 A Critique of The Model

The model does not describe a number of aspects.

93. Timing: Yes, we have made sure to possibly prepare for that, for example by supplying all communications between behaviours with time-stamps. But what about modeling that (i) customers can require specific delivery times; (ii) that it takes time, for the grocery store, to assemble packages; (iii) that it takes time, for courier management, to schedule deliveries; (iv) that the delivery along a route takes time; (v) etc. ?
94. Where does the monies come from: customers must somehow have an income / a fortune ?
95. Where does the groceries come from originally ? One could model a [set of] grocery suppliers from where the grocery store orders these;
96. What happens to groceries, in the grocery store, whose "due-date" "expires" ?
97. Groceries lost in transport ?
98. Customers ordering groceries in quantities not available at the store ? Replacement or delivery of fewer than ordered, etc. ?
99. Etc. ?

6.4 Acknowledgment

To **Kari**, my dear wife of 60+ years, for my getting up – often several times – during late evenings and nights, and away from being "downstairs" with here during 2/3 the day ! THANKS.

To **Klaus Havelund** for patiently following my incessant questions about a role for Generative AI: the possibility of letting a **LLM** try express, as it has been done in this document, a narrative and an RSL/CSP formalised description of the *Wolt* DOMAIN – say based on an English text input as presented in Sect. 3: the *Expanded Narrative* Page 10.

7 Bibliography

References

[1] Scott Aaronson. *Quantum Computing since Democritus*. Cambridge University Press, 2013.

[2] Jean-Raymond Abrial. *Modeling in Event-B – System and Software Engineering*. Cambridge University Press, 2010. https://doi.org/10.1017/CBO9781139195881.

[3] Egidio Astesiano, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. CASL: The common algebraic specification language. *Theoretical Computer Science*, 286:153–196, 2 2002.

[4] Jonathan Barnes. *The Presocratic Philosophers: The Natural Philosophy of Heraclitus*. Routledge, Taylor & Francis Group, 1982. 43–62.

[5] Dines 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, [28]. Earlier ones may not. *Urban Planning*, for example, https://www.imm.dtu.dk/~dibj/2017/urban-planning.pdf, do not. It treats $\mathbb{TIME}$ in an erroneous way – and should be corrected.

[6] Dines Bjørner. Software Systems Engineering — From Domain Analysis to Requirements Capture: An Air Traffic Control Example. In *2nd Asia-Pacific Software Engineering Conference (APSEC '95)*. IEEE Computer Society, 6–9 December 1995. Brisbane, Queensland, Australia.

[7] Dines Bjørner. Domain Models of "The Market" — in Preparation for E–Transaction Systems. In *Practical Foundations of Business and System Specifications (Eds.: Haim Kilov and Ken Baclawski)*, The Netherlands, December 2002. Kluwer Academic Press. www2.imm.dtu.dk/~dibj/themarket.pdf.

[8] Dines Bjørner. Transportation Systems Development. In *2007 ISoLA Workshop On Leveraging Applications of Formal Methods, Verification and Validation; Special Workshop Theme: Formal Methods in Avionics, Space and Transport*, ENSMA, Futuroscope, France, December 12–14 2007. www.imm.dtu.dk/ db/isola-pa.pdf.

[9] Dines Bjørner. The Tokyo Stock Exchange Trading Rules www.imm.dtu.dk/~db/todai/tse-1.pdf, www.imm.dtu.dk/ db/todai/tse-2.pdf. R&D Experiment, Techn. Univ. of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, January, February 2010.

[10] Dines Bjørner. A Rôle for Mereology in Domain Science and Engineering. In *Mereology and the Sciences*, Synthese Library (eds. Claudio Calosi and Pierluigi Graziani), pages 323–357, Amsterdam, The Netherlands, October 2014. Springer. https://www.imm.dtu.dk/~dibj/2011/urbino/urbino-colour.pdf.

[11] Dines Bjørner. A Credit Card System: Uppsala Draft www.imm.dtu.dk/~dibj/2016/credit/accs.pdf. Technical Report: Experimental Research, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, November 2016.

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

[13] Dines Bjørner. A Space of Swarms of Drones. www.imm.dtu.dk/~dibj/2017/swarms/swarm-paper.pdf. Research Note, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, December 2017.

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

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

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

[17] Dines Bjørner. Domain Analysis & Description – Principles, Techniques and Modeling Languages. *ACM Trans. on Software Engineering and Methodology*, 28(2):66 pages, March 2019. www.imm.dtu.dk/~dibj/2018/tosem/Bjorner-TOSEM.pdf.

[18] Dines Bjørner. *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 [30]. https:///doi.org/10.1007/978-3-030-73484-8.

[19] Dines Bjørner. A Retailer Market: Domain Analysis & Description. A Comparison Heraklit/DS&E Case Study. www.imm.dtu.dk/ dibj/2021/Retailer/BjornerHeraklit27January2021.pdf. Technical Report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, January 2021.

[20] Dines Bjørner. Automobile Assembly Plants. www.imm.dtu.dk/~dibj/2021/assembly/-assembly-line.pdf. Technical Report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, September 2021.

[21] Dines Bjørner. Shipping. www.imm.dtu.dk/~ dibj/2021/ral/ral.pdf. Technical Report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, April 2021.

[22] Dines Bjørner. Air Traffic Control. www.imm.dtu.dk/~ dibj/2022/nasa-fm/atc-nasa-fm.pdf. Technical Report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, Spring 2022.

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

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

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

[26] Dines Bjørner. Domain Analysis & Description. In *Theoretical Aspects of Computing, ICTAC 2025*, number 16237 in Lecture Notes in Computer Science, pages 39–66. Springer, November 24–28 2025. A Tutorial. DOI 10.1145/3796228.

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

[28] Dines Bjørner. DADL: An Analysis & Description Language. Technical report, 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`.

[29] Dines Bjørner. The 𝔻𝕆𝕄𝔸𝕀ℕ Method. Technical report, DTU Compute, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, March 2026.

[30] Dines Bjørner. Domain Science & Engineering, a Primer[49] Technical University of Denmark. Significantly revised edition of [18]. xii+211 pages., February 2026 To be submitted [2026/2027].

[31] Dines Bjørner and Mikhail Chupilko. : . Translation of [30]. [Book], To be submitted [2026/2027].

[32] Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer, Heidelberg, Germany, 1978. https://doi.org/10.1007/3-540-08766-4.

[33] Dines Bjørner and Martin Pěnička. Towards a **TRAIN** Book for **T**he **RAI**lway Domai**N**. Techn. reports, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, August 2004. http://www.imm.dtu.dk/ dibj/train-book.pdf.

[34] Dines Bjørner and Yang ShaoFa. . Translation of [30]. [Book], To be submitted [2026/2207].

[35] E. Börger and R. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springe, 2003. ISBN 3-540-00702-4.

[36] Bartle Bull. *Land between the rivers – A 5,000 year history of Iraq*. Atlantic Books, London, England, 2025. 546 pages.

[37] Roberto Casati and Achille C. Varzi. *Parts and Places: the structures of spatial representation*. MIT Press, 1999.

[38] Dirk L. Couprie and Radim Kocandrle. *Anaximander: Anaximander on Generation and Destruction*. x, Springer (Briefs in Philosophy Series).

[49] This book is currently being translated into Russian, [31], 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.

[39] W. W. Davies. *Egyptian Glyphs*. British Museum Press, London, England, 1987.

[40] Razvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. AMAST Series in Computing - Vol. 6. World Scientific Publishing Co., Pte. Ltd., 5 Toh Tuck Link, Singapore 596224, July 1998. 196pp, ISBN 981-02-3513-5, US$30.

[41] Musée du Louvre. *naissande de l'écriture: cunéiforms et hiéroglyphes*. Musée du Louvre, Paris, France, 1982.

[42] Carl Faulmann. *Buch der Schrift*. Greno Press, Nördligen, Germany, 1880/1985.

[43] Chris W. George, Peter Haff, Klaus Havelund, Anne Elisabeth Haxthausen, Robert Milne, Claus Bendix Nielsen, Søren Prehn, and Kim Ritter Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992. ISBN 0137528337 and 9780137528332.

[44] James Gosling and Frank Yellin. *The Java Language Specification*. Addison-Wesley & Sun Microsystems. ACM Press Books, 1996. 864 pp, ISBN 0-10-63451-1.

[45] Michael Reichhardt Hansen and Hans Rischel. *Functional Programming Using F#*. Cambridge University Press, 2013.

[46] Martin Heidegger. *Parminedes*. Indiana University Press, 1998.

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

[48] J. T. Hooker. *Reading the Past: Ancient Writing from Cuneiform to the Alphabet*. British Museum Press, London, England, 1990. 384 pages.

[49] Michael Jackson. *Software Requirements & Specifications — a lexicon of practice, principles and prejudices*. Addison–Wesley, 1995.

[50] K. Khazai. *Naissance et Evolution de l'Ecriture*. Societé des Banques, Luxembourg, 1985. 232 pages.

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

[52] W. Little, H.W. Fowler, J. Coulson, and C.T. Onions. *The Shorter Oxford English Dictionary on Historical Principles*. Clarendon Press, Oxford, England, 1973, 1987. Two vols.

[53] J. Edward Mercer. *The Mysticism Of Anaximenes And The Air*. Kessinger Publishing, LLC, 2010.

[54] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Mass., USA and London, England, 1990.

[55] Akira Nakanishi. *Writing Systems of the World – Alphabets, Syllabaries, Pictograms*. Charles E. Tuttle, Rutland, Vermont, USA & Tokyo, Japan, 1080. 182 pages.

[56] Patricia O'Grady. *Thales of Miletus*. Routledge (Western Philosophy Series), 2002.

[57] Wolfgang Reisig. *Understanding Petri Nets Modeling Techniques, Analysis Methods, Case Studies*. Springer, 2013. 230+XXVII pages, 145 illus.

[58] Andrew Robinson. *The Story of Writing*. Thames & Hudson, London, England, 1995. 224 pages.

[59] Peter Sestoft. *Java Precisely*. The MIT Press, 25 July 2002.

[60] Kai Sørlander. *Det Uomgængelige – Filosofiske Deduktioner [The Inevitable – Philosophical Deductions, with a foreword by Georg Henrik von Wright]*. Munksgaard · Rosinante, Copenhagen, Denmark, 1994. 168 pages.

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

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

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

[64] Kai Sørlander. *Den rene fornufts struktur [The Structure of Pure Reason]*. Ellekær, Slagelse, Denmark, 2022. See [65].

[65] Kai Sørlander. *The Structure of Pure Reason*. Springer, February 2025. This is an English translation of [64] – done by Dines Bjørner in collaboration with the author.

[66] N. Wirth. The Programming Language Oberon. *Software — Practice and Experience*, 18:671–690, 1988.

[67] J. C. P. Woodcock and M. Loomes. *Software Engineering Mathematics*. Pitman, London, 1988.

[68] M.R. Wright. *Empedokles: The Extant Fragments*. Hackett Publishing Company, Inc., 1995.

A Road Maps

This is **not** a description of the domain of road nets. For that we refer to [27] or `https://www.-imm.dtu.dk/~dibj/2025/transport.pdf`/ Instead it is a [classical] abstract specification of road maps [of road nets] as attributes of, for example, grocery stores **GC** – cf. Sect. **??**.

A.1 The Road Map Structure

100. A road map consists of a set of links and a set of hubs.

101. Links have unique identifiers, mereology, and length attributes.

102. Hub have unique identifiers and mereology.

103. Link and hub identifiers are further unspecified. .
104. Link lengths are natural numbers designating meters.
105. The mereology of links is a set of hub identifiers:
106. two elements
107. and of the road map.
108. The mereology of hubs is a set of zero, one or more link identifiers
109. of the road map.

```
type
100.  RM = L-set × H-set
101.  L = sli:LI × LM × s_length:LEN
102.  H = shi:HI × HM
103.  LI, HI
104.  LEN = Nat
105.  LM = HI-set
108.  HM = LI-set
axiom
106.  ∀ lm:LM • card lm=2
107.    ∧ lm ∈ his
109.  ∀ hm:HM • hm ∈ lis

value
    (ls,hs):RM
    lis = { sli(l) | l:L • l ∈ ls }
    his = { shi(h) | h:K • h ∈ hs }
```

A.2 Paths

110. A path is a list of alternating link and hub identifiers.
111. A path of a road map
112. is a path and
113. has adjacent identifiers be those of adjacent links and hubs of the road map.

```
type
110.  P = (LI|HI)*
axiom [Alternating Path Identifiers]
110.  ∀ p:P •
110.    ∀ {i,i+1}:Nat • {i,i_1} ⊆ inds p •
110.      is_LI(p(i)) ∧ is_HI(p(i+1)) ∨ is_HI(p(i)) ∧ is_LI(p(i+1))
value
```

```
111.  paths: RN → P-set
111.  paths(rn) as ps
112.    post: ∀ p ∈ ps satisfies axiom [Alternating Identifiers]
113.      ∧ ∀ (li,hj):(LI×HI) • ⟨...,li,hj,...⟩ ∈ ps ⇒ hj ∈ adjacent_hubs(rn)(li)
113.      ∧ ∀ (hi,lj):(HI×LI) • ⟨...,hi,lj,...⟩ ∈ ps ⇒ lj ∈ adjacent_links(rn)(hi)

113.  adjacent_hubs: RN × LI → HI-set
113.  adjacent_hubs(ls,hs)(li) ≡
113.    let {(hai,ham),(hbi,hbm)}⊆hs • li ∈ ham∧li ∈ hbm in {hai,hbi} end

113.  adjacent_links: RN × HI → LI-set adjacent_links
113.  adjacent_links(rn)(hi) ≡
113.    let {(lai,lam,_),(lbi,lbm,_)}⊆ls • hi ∈ lam∧hi ∈ lbm in {lai,lbi} end
```

Above was a property-oriented specification of `paths`. Next is an "algorithmic" specification:

114. **Basis clause:** Any link and any hub identifier of a road map is a path of that road map.
115. **Induction clause:** If `li:LI` [`hj:HI`] in path of a road map `p^⟨li⟩` [`p^⟨hj⟩`] `is adjacent to hj:HI [li:LI] in a path of a road map ⟨hi⟩^p′ [⟨li⟩^p′]` then `p^⟨li,hj⟩^p′` [`p′^⟨hj,li⟩^p`] is a paths of the road map.
116. **Extremal clause:** Only such paths which can be constructed by a finite set of applications of the above clauses are paths of the road map.

```
value
111.  paths: RM → P-set
111.  paths(ls,hs) ≡
111.  let ps = { ⟨li⟩ | li ∈ lis } ∪ { ⟨hi⟩ | hi ∈ his }
115.         ∪ { p^⟨li,hj⟩^p′ | p^⟨li⟩ ∈ ps ∧ ⟨hj⟩^p′ ∈ ps ∧ li ∈ adjacent_links(rn)(hj) }
115.         ∪ { p^⟨hj,li⟩^p′ | p^⟨hj⟩ ∈ ps ∧ ⟨li⟩^p′ ∈ ps ∧ li ∈ adjacent_links(rn)(hj) } in
116.   ps end
```

A.2.1 Circular Paths / Routes

117. A path is circular if it contains the same identifier more than once.

```
117.  is_circular: Path → Bool
117.  is_circular(p) ≡ card elems p < len p
```

A.2.2 Circle Paths

118. A path, `⟨i⟩^p^⟨j⟩` (of the road map) is a circle path if `i=j`.

```
value
118.  is_circle_path: Path → RN → Bool
118.  is_circle_path(p)(rn) ≡
```

```
118.    ∃ (i,j):(LI|HI),p′:P•p′∈paths(rn)
118.      ⇒ i=j ∧ p=⟨i⟩⌢p′⌢⟨j⟩
118.  pre: p ∈ paths(rn)
```

A.2.3 Path Lengths

119. The length of a path is the sum of the lengths of all its [possibly repeated] links.

```
value
119.  length: P → Nat
119.  length(p) ≡
119.    case p of
119.      ⟨⟩ → 0,
119.      ⟨li:LI⟩⌢p′ → s_length(retr_L(rn)(li)) + length(p′),
119.      ⟨hi:HI⟩⌢p′ →  length(p′)
119.    end
```

120. From a road map and a link identifier of that road map we can retrieve the link of that identification of the road map.

```
value
120.  retr_L: RN → LI → LEN
120.  retr_L(ls,hs)(li) ≡ let l:L • l ∈ ls ∧ uid_E(l)=li in l end
120.  pre: li ∈ lis
```

A.2.4 Shortest Paths

121. Given a road map we can calculate the set of shortest paths,

122. i.e., the paths

123. all of whose lengths are the same, and are shorter than all other paths of the road map.

```
value
121.  shortest_paths: RM → P-set
121.  shortest_paths(rn) ≡
121.    let ps = paths(rn) in
122.    let sps = { p | p:P • p ∈ ps ∧ ∀ p′:P • p′ ∈ ps ∧ length(p)≤length(p′) } in
122.    sps end end
```

A.2.5 Connected Road Net

124. A road map is connected if any node [or link] can be reached from any other node [or link], i.e., that there are paths between these!

```
value
124. is_connected: RM → Bool
124. is_connected(ls,hs) ≡
124.   let ps = paths(ls,hs) in
124.   ∀ hi,hi′:HI • {hi,hi′} ⊆ his • ∃ p:P • p ∈ ps ∧ {hi,hi′} ⊆ elems p end
```

B A Wolt & Domain Terminology

Peter Naur[50] taught me, way back in the 1960s:

- *The perhaps most crucial item to be dispensed with in any programming project*
- *is the establishment, from the very start of such a project, of a* `terminology document`.
- *A terminology of the terms of the application domain and the software being developed.*
- *To adhere to, daily, and update, daily, this terminology,*
- *To ensure that all development documents do so!*
- *To possible assign a more-or-less full time staff to take care of this!*
- *Failure to do so, he claimed, would surely lead to problems:*
  - *for the individual project programmer*
  - *as well in communication between these.*

Over the years I have seen the truth in Peter's diction!

• • •

This terminology is, perhaps, a bit unusual in that it, in addition to proper, natural language characterisations of terms, also "recall" the formal definition of some of the **Wolt** entities.

[50] `https://en.wikipedia.org/wiki/Peter_Naur`

DOMAIN: 1.

1. **Abstraction:** A perhaps the, most important method principle is:

Characterisation: 4 **Abstraction:** Quoting **C.A.R. Hoare:**

- 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 ■

DOMAIN: 2.

2. **Action:** The dynamic details of domains are revealed in their actions:

Characterisation: 5 **Action:** By an **action** we shall here mean anything, in a domain, that potentially `changes the values of a state` ■

DOMAIN: 3.

3. **Actor:** Actors are "the players" that get "things done"!

Characterisation: 6 **Actor:** By an **actor** we shall here mean anything, in a domain, that can `change the values of a state` ■

DOMAIN: 4.

4. **Atom:** The domain analyser cum describer decides which endurants are to be modeled as atoms, which as compounds.

Characterisation: 7 **Atomic Part:** By an **atomic part** we shall understand a part which the domain analyser considers to be indivisible in the sense of not meaningfully divisible, for the purposes of the domain under consideration, that is, to not meaningfully consist of sub-parts ■

DOMAIN: 5.

5. **Attribute:** The real properties of endurants are revealed in their attribute types and values.

Characterisation: 8 **Attributes:** Solids [and fluids] have properties beyond those of unique identification and mereology. [We do not associate unique identity nor mereology with fluids.] These properties are more free-wheeling. Some are measurable others can be spoken about. **Attributes** are either `static, monitorable` or `programmable` ■

DOMAIN: 6.

6. **Behaviour:** The overall domain dynamics is revealed in the number, composition and interaction between behaivours.

Characterisation: 9 **Behaviour:** By a **behaviour** we shall here mean a set of sequences of actions, events and behaviours ■

DOMAIN: 7.

7. **Cartesian:** Named so after the French Mathematician (etc.) René Descartes[51] some compound parts are composed from a definite number of parts, some of which we analyse and describe.

Characterisation: 10 **Cartesian: Cartesian parts** consist of two or more distinctly sort-named endurants (solids or fluids) ■

DOMAIN: 8.

8. **Channel:** For lack of a better term we use CSP's concept name.

Characterisation: 11 **Channel:** By a **channel** we shall mean any abstraction notion, i.e., a concept, not – in domain descriptions – anything concrete, but a notion that allows communication, i.e., synchronisation and exchange of messages between bahevours ■

DOMAIN: 9.

9. **Compound:** Parts that are not atomic are compound!

Characterisation: 12 **Compound: Compound**s are those parts which can be considered composed from either a definite collection (a Cartesian) of two or more parts (!) or an indefinite collection (a part set) of zero, one or more parts ■

Wolt: 1.

10. **Courier**, **C:** Couriers are the messengers that deliver groceries to Your door, be they humans, walking, bicycling or by car, or be they WayMo[52]-like robots,!

- **Index:**

  Part: Cf. Sect. **??** on page ??, Item 9.
  State: *cs* Cf. Sect. 35 on page 60, Item 13c.
  Meeology: Cf. Sect. 22, Item 25 on page 17

  type
  25. MC = CMI × KI-**set**
  value
  25. **mereo_**C: C → MC

  Meeology: Cf. Item 25 on page 17
  Attributes: Cf. Sect. 4.2.3.5 on page 22
  You will find the narratives for the formluas below on Page 20

[51] [https://en.wikipedia.org/wiki/René_Descartes]
[52] WayMo: https://waymo.com/

type
40. RM [static]
41. CR = CI $\xrightarrow{m}$ CInfo × CName × CAddr × CurrDelRoute × ... [progammable]
41. CInfo, CName, CAddr, ...
41. CurrDelRoute = ((LI|HI) × No)*
42. GD = KI $\xrightarrow{m}$ SDL [programmable]
42a. SDL = (LI|HI) × Loc × G-**set**
42b. Loc
43. PL = CI $\xrightarrow{m}$ DR
43a. DR = (HI|LI) × Delvs
43b. Delvs = KI $\xrightarrow{m}$ (Loc × GI-**set**)

value
40. **attr_**RM: CM → RM
41. **attr_**CR: CM → GD
42. **attr_**GD: CM → CR
43. **attr_**PL: CM → PL

Actions: Cf. Sect. 5.3.4 on page 27

Signature: Sect. 5.5.1 on page 31

71. courier: CI→MC→..→(CDL×Idx)→**Unit**

Behaviour:

Wolt: 2. 11. **Courier Management**, **CM:** Courier nanagement receieves packages from the grocery store, notify customers of pending delveriesm schedules deliveries these and hand over packages to couriers for delivery.

- **Index:**

Part: Cf. Sect. **??** on page ??, Items 7+7b.

State: *cmgt* Cf. Sect. 35 on page 60, Item 13b.

Mereology: Cf. Sect. 22, Item 24 on page 17

type
24. MCM = GSTI × KI-**set** × CI-**set**

value
24. **mereo_**CM: CC → MCM

Attributes: Cf. Sect. 4.2.3.4 on page 20

You will find the narratives for the formluas below on Page 20

type
40. RM [static]
41. CR = CI $\xrightarrow{m}$ CInfo × CName × CAddr × CurrDelRoute × ... [progammable]
41. CInfo, CName, CAddr, ...
41. CurrDelRoute = ((LI|HI) × No)*
42. GD = KI $\xrightarrow{m}$ SDL [programmable]
42a. SDL = (LI|HI) × Loc × G-**set**
42b. Loc
43. PL = CI $\xrightarrow{m}$ DR

43a. DR = (HI|LI) × Delvs
43b. Delvs = KI $\xrightarrow{m}$(Loc × GI-**set**)
value
40. **attr_**RM: CM → RM
41. **attr_**CR: CM → GD
42. **attr_**GD: CM → CR
43. **attr_**PL: CM → PL

Actions: Cf. Sect. 5.3.3 on page 27
Signature: Sect. 5.5.1 on page 31
70. courier_mgt: CMI→MCM→(RM×CR)→...→(GD×PL)→**Unit**

Behaviour:

12. **Customer**, **K:** Customers are the You and me: wishing to and actually also able to acquire groceries from the grocery delivery service: orders them, get notified about their pending arrival from courier management and, finally, receives these. *Wolt*:3.

- **Index:**

Part: Cf. Sect. 4.1.1.1 on page 11, Item 4.
State: *kst* Cf. Sect. 35 on page 60, Item 13d.
Mereology: Sect. 22, Item 22 on page 17

type
22. MK = GSTI × CMI × CI-**set**
value
22. **mereo_**K: K → MK

Attributes: Cf. Sect. 4.2.3.2 on page 19
You will find the narratives for the formluas below on Page 18

type
26. CustInfo = Name × CustAddr ×.. `static`
26a. CustAddr = .(LI|HI) × LocNo
27. Ctlg = [GrocCtlg] `programmable`
28. Ordr = OrdrNo × (GrocName $\xrightarrow{m}$Quantity) `programmable`
35a. GrocName, Quantity = **Nat**
29. CustCash = M `programmable`
30. AGS = G-**set** `programmable`
31. Mode = **idl** | **ord** | **rej** `programmable`
value
26. **attr_**CustInfo: K → CustInfo
27. **attr_**Ctlg: K → GrocCtlg
28. **attr_**Ordr: K → GrocCtlg
29. **attr_**CustCash: K → CustCash
30. **attr_**AGS: K → AGS
31. **attr_**Mode: K → Mode

Actions: Cf. Sect. 5.3.1 on page 25

Signature: Sect. 5.5.1 on page 31

68. cust: KI→MK >(CustInfo×CustAddr)→.. →(Ctlg×Ordr×CustCash×AGS×Mode)→**Unit**

Behaviour:

DOMAIN: 10.

13. **Domain:** This is what this document is all about: a narrative & formal description of a domain: ***Wolt***.

Characterisation: 13 **Domain:** 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* [DB] ■

DOMAIN: 11.

14. **Endurant:** Endurants are the staiic entities of a domain.

Characterisation: 14 **Endurants:** In the context of domains, **endurants** are those entities 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] ■

DOMAIN: 12.

15. **Entity:** Entities are what we can talk about rationally.

Characterisation: 15 **Entity:** By an **entity**, in the context of domains, we shall understand a phenomenon whose properties can be rationally described ■

DOMAIN: 13.

16. **Event:** Events occur, not always as we wish them!

Characterisation: 16 **Event:** By an **event** we shall here mean anything, in a domain, that potentially and surreptitiously `changes the values of a state` ■

DOMAIN: 14.

17. **Formalisation:** Well, well: We can describe matters in our mother-tongue, and we can also describe them – more-or-less – mathematically, That is: we can formalise our [otherwise narrative] descriptions.

There are many formal specification languages – alphabetically listed:

- **Alloy:** [49]
- **ASM:** [35]
- **Event B:** [2]
- **Cafe OBJ:** [40]
- **CASL:** [3]
- **Petri nets:** [57]
- **RSL:** [43]
- **TLA$^+$:** [51]
- **VDM:** [32]
- **Z:** [67]

18. **Fluid:** Water, oil, air, gases, plasma – are here considered fluids: The *Wolt* domain does not exhibit "fluids-of-interest"! DOMAIN: 15.

Characterisation: 17 **Fluid:** By a **fluid** 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 [52, Vol. I, pg. 774] ■

19. **Grocery**, **G:** See Sect. 4.1.1.2 on page 12. *Wolt*: 4.

20. **Grocery Store**, **GST:** *Wolt*: 5.

- **Index:**

Part: Cf. Sect. **??** on page ??, Items 7+7a.

State: *gst* Cf. Sect. 35 on page 60, Item 13a.

Mereology: Cf. Sect. 22, Item 23 on page 17

type
23. MGST = KI-**set** × CMI

value
23. **mereo_**GST: GC → MGC

Attributes: Cf. Sect. 4.2.3.3 on page 19

You will find the narratives for the formluas below on Page 19

type
36a. Groceries = GrocNames $\xrightarrow{m}$ G-**set**
36b. G = GI × ... × SellBy
36b. SellBy, GI=UI
37. GrocCtlg = GrocName $\xrightarrow{m}$ (Quantity × GrocInfo)
35a. Quantity = **Nat**
35a. GrocInfo = Price × Weight × Durability × SpaceInfo × PackInfo × ... ..
35a. Price, Weight, Durability, SpaceInfo, PackInfo, ...
35b. GrocName
36. CustCtlg = KI $\xrightarrow{m}$ CustInfo
37. CustInfo = OrdNo $\xrightarrow{m}$ Orders
38. Orders = Date $\xrightarrow{m}$ (OrdrNo $\xrightarrow{m}$ Order)
??. Order = (GrocName $\xrightarrow{m}$ Quantity) × Cost
39. Cash

value
36a. **attr_**Groceries: GST → Groceries
36b. **uid_**GI: G → UI
37. **attr_**GrocCtlg: GST → Catalog
36. **attr_**CustCtlg: GST → CustCtlg
39. **attr_**Cash: GS → M

Actions: Cf. Sect. 5.3.2 on page 26

Signature: Sect. 5.5.1 on page 31

69. groc_store: GSI→MGST→GrocCtlg→...→(Groceries×CustCtlg×Cash)→**Unit**

Behaviour:

DOMAIN: 16.

21. **Manifest:**

Characterisation: 18 **Manifest:** By **manifest** we shall in general mean an endurant which is : readily perceived by the senses and especially by the sense of sight [Merriam-Webster], more specifically mean an endurant which one can also touch and which can potentially be transcendentally deduced into a behaviour [DB] ▪

↓ 14:02;29:07

DOMAIN: 17.

22. **Mereology:**

Characterisation: 19 **Mereology:** **Mereology** is the study and knowledge of part relations[53] ▪

DOMAIN: 18.

23. **Method:**

Characterisation: 20 **Method:** By a **method** we shall understand a *set of* **principles** and a *set of* **procedures** for *selecting* and *applying* a *set of* **techniques** *using* a *set of* **tools** in order to *construct* an *artefact* [DB]

DOMAIN: 19.

24. **Monitorable Attribute:**

Characterisation: 21 **Monitorable Attribute:** By a **monitorable attribute** we mean an attribute whose value changes *"at its own volition"*, that is: must be *"somehow obtained!"* ▪

25. **Narration:** See ι 26, π 58

DOMAIN: 20.

26. **Narrative:**

DOMAIN: 21.

27. **Ontology:**

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

We refer to Fig. 3 on the facing page for the *Wolt* analysis & description ontology.

DOMAIN: 22.

28. **Part:**

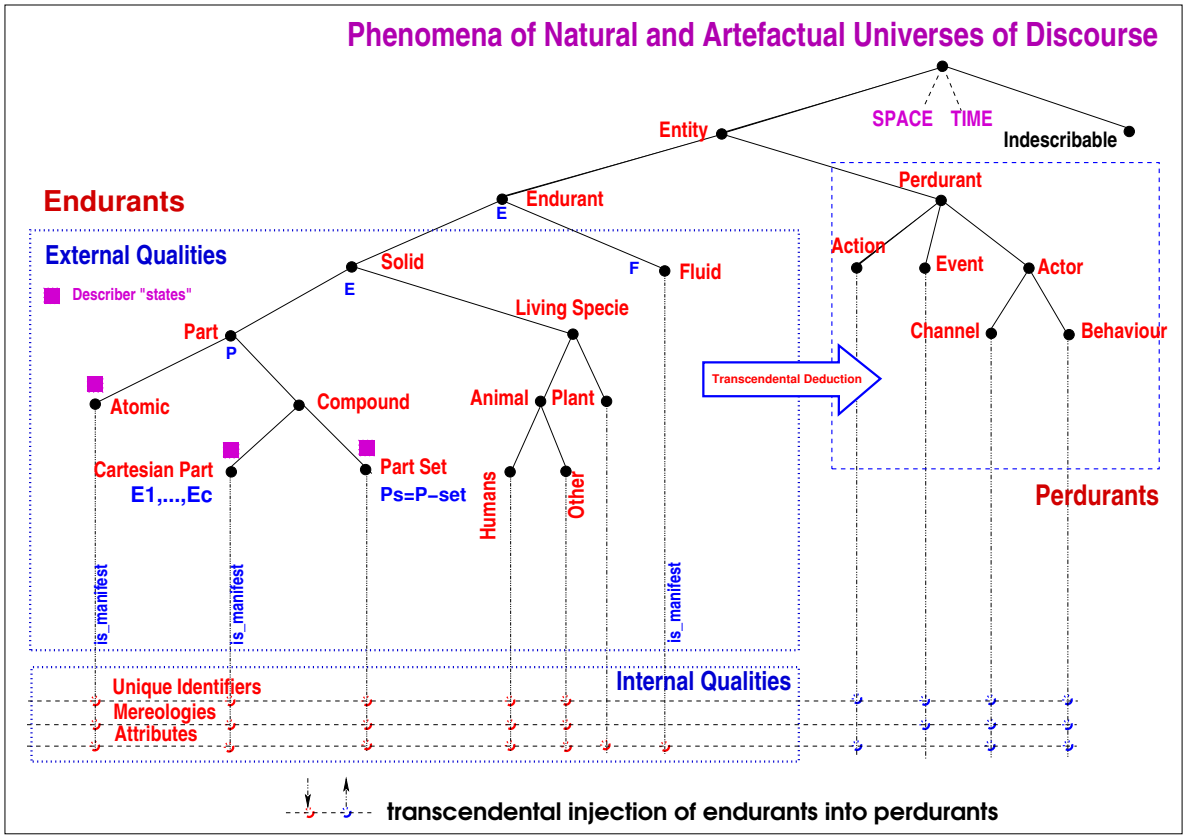

Figure 3: The **Wolt** Domain Analysis & Description Ontology

Characterisation: 22 **Part:** TO COME ■

29. **Part Set**: DOMAIN: 23.

Characterisation: 23 **Part Set:** **Part sets** consist of an indefinite number of zero, one or more parts ■

30. **Perdurant**: DOMAIN: 24.

Characterisation: 24 **Perdurants:** **Perdurants**, in the context of domains, **perdurants** are those entities 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] ■

31. **Programmable Attribute**: DOMAIN: 25.

[53] Mereology, as a logical/philosophical discipline, can perhaps best be attributed to the Polish mathematician/logician Stanisław Lesniewski (1886–1939) [37, 10].

Characterisation: 25 **Progammable Attribute:** By a **progammable attribute** we mean an attribute value whose value can be set ■

DOMAIN: 26. 32. **Phenomenon:**

Characterisation: 26 **Phenomenon:** By a **phenomenon**, in the context of domains, we shall understand a fact that is observed to exist or happen ■

DOMAIN: 27. 33. **Solid:**

Characterisation: 27 **Solid:** By a **solid** [or **discrete**] endurant we shall understand an entity which is separate, individual or distinct in form or concept, or, rephrasing: have body [or magnitude] of three-dimensions: length, breadth and depth `[52, Vol. II, pg. 2046]` ■

DOMAIN: 28. 34. **Sort:**

Characterisation: 28 **Sort:** By a **sort** we shall mean a set of domain entities `[DB]` ■

DOMAIN: 29. 35. **State:**

Characterisation: 29 **State:** By a **state** – in the contex of *domain*s – we mean a set of *manifest parts* ■

DOMAIN: 30. 36. **Static Attribute:**

Characterisation: 30 **Static Attribute:** By a **static attribute** we mean an attrbute whose value cannot/does not change ■

DOMAIN: 31. 37. **Sort:**

Characterisation: 31 **Sort:** By a **sort** we shall mean a set of domain entities `[DB]` ■

38. **Store:** See ι 20, π 57

DOMAIN: 32. 39. **Type:**

Characterisation: 32 **Type:** By a **type** we shall mean a set of values `[DB]` ■

DOMAIN: 33. 40. **Value:**

Characterisation: 33 **Value:** By a **value** we shall mean a mathematical value (`Booleans, numbers, sets, Cartesians, lists, maps, functions,` or other), or a domain entity (`endurants, unique identifier, mereology, attribute,` or `behaviour`) `[DB]` ■

C A RAISE Specification Language Primer

We present an RSL Primer. Indented text, in slanted font, such as this, presents informal material and examples. Non-indented text, in roman font, presents narrative and formal explanation of RSL constructs.

This RSL Primer omits treatment of a number of language constructs, notably the RSL module concepts of *schemes, classes* and *objects*. Although we do cover the imperative language construct of [declaration of] variables and, hence, assignment, we shall omit treatment of structured imperative constructs like **for** ..., **do** s **while** b, **while** b **do** s loops.

Section C.13 on page 86 introduces additional language constructs, thereby motivation the $^+$ in the RSL$^+$ name.

C.1 Specification Units

An RSL specification consists of a set (textually ordered in any linear sequence) of *specification units*. We shall only treat five kinds of such units:

- *type:* Prefixed by the keyword (literal) **type**, type specification units introduce distinct type names. The general form of a type specification unit is:

  type T = Type_Expression

  where T is a distinct (type) identifier. Type specification units may introduce several (new, distinct) types:

  type T1 = Type_Expression_1, T2 = Type_Exression_2, ..., Tn = Type_Expression_n

  For more on types, see Sect. C.2.

- *value:* Prefixed by the keyword (literal) **value** value specification units introduce distinct value names. The general form of a value specification unit is:

  value a:A = $\mathcal{E}$(...)

  where $\mathcal{E}$(...) is a value expression. Value specification units may introduce several (new, distinctly named) values:

  value a_1:A_1 = $\mathcal{E}_1$(...), a_2:A_2 = $\mathcal{E}_2$(...), ..., a_n:A_n = $\mathcal{E}_n$(...)

  Quite often the value specification is of the form:

  value f: A → B, f(arg) ≡ $\mathcal{E}$(...[f]...)

  that is: the value is a function, f, whose *signature* gives the name, f, and the type of the functionality A → B (or A $\xrightarrow{\sim}$ B). Most of this chapter will cover aspects of value expressions.

- *axiom:* Prefixed by the keyword (literal) **axiom**, axiom specification units serve to limit values. The general form of an axiom specification unit is:

  axiom $\mathcal{A}$(...)

  where $\mathcal{A}$(...) is some predicate expression over (specification unit) defined quantities. For more on axiomatic expressions, see Sect. C.3.

- *variable:* Prefixed by the keyword (literal) **variable**, variable specification units introduce distinct variable names. The general form of a variable specification unit is:

  variable v:T := expression

  where v is ..., T is ..., and expression is a value expression. For variables, see Sect. C.9.2.

- *channel:* Prefixed by the keyword (literal) **channel**, channel specification units introduce distinct channel names.

  The general form of a channel specification unit is:

  channel { ch[{i,j}] | i,j:UI • ... }: M

  where ch is a distinct, here channel, name,we are declaring an array of channels. ch[$\{i,j\}$] expresses that , $\{i,j\}$ ranges of so-called unique identifier indices of type UI, and M is a type expression. For more on channels, see Sect. C.10.1.

This chapter will otherwise be conventionally structured.

C.2 Types and Values

Types are, in general, set-like structures[54] of things, i.e., values, having common characteristics.

A bunch of zero, one or more apples (type *apples*) may thus form a [sub]set of type *Belle de Boskoop* apples. A bunch of zero, one or more pears (type *pears*) may thus form a [sub]set of type *Concorde* pears. A union of zero, one or more of these apples and pears then form a [sub]set of entities of type *fruits*.

C.2.1 Sort and Type Expressions

Sort and type expressions are expressions whose values are types, that is, possibly infinite set-like structures of values (of "that" type).

[54] We shall not, in this primer, go into details as to the mathematics of types.

C.2.1.1 Atomic Types: Identifier Expressions and Type Values Atomic types have (atomic) values. That is, values which we consider to have no proper constituent (sub-)values, i.e., cannot, to us, be meaningfully "taken apart".

RSL has a number of [so-called] *built-in* atomic types. They are expressed in terms of literal identifiers. These are the **Boolean**s, **int**egers, **Nat**ural numbers, **Real**s, **Char**acters, and **Text**s. **Text**s are free-form texts and are more general than just texts of RSL-like formulas. RSL$^+$Text's will be introduced in Sect. C.13 on page 86.

We shall not need the base types **Char**acters, nor the general type **Text**s for domain modelling in this primer. They will be listed below, but not mentioned further.

The base types are:

Basic Types

type
[1] **Bool**
[2] **Int**
[3] **Nat**
[4] **Real**
[5] **Char**
[6] **Text**

1. The Boolean type of truth values **false** and **true**. indexsymbindaaab@**true**
2. The integer type on integer values ..., –2, –1, 0, 1, 2, ...
3. The natural number type of positive integer values 0, 1, 2, ...
4. The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period ("."), followed by a natural number (the fraction).
5. The character type of character values ″a″, ″bb″, ...
6. The text type of character string values ″aa″, ″aaa″, ..., ″abc″, ...

C.2.1.2 Composite Types: Expressions and Type Values Composite types have composite values. That is, values which we consider to have proper constituent (sub-)values, i.e., can, to us, be meaningfully "taken apart".

From these one can form type expressions: finite sets, infinite sets, Cartesian products, lists, maps, etc.

Let A, B and C be any type names or type expressions, then these are the composite types, hence, type expressions:

Composite Type Expressions

[7] A-**set**
[8] A-**infset**
[9] A × B × ... × C
[10] A*

[11] A^{ω}
[12] $A_{m}\!\!\rightarrow\!\!B$
[13] $A \rightarrow B$
[14] $A \xrightarrow{\sim} B$
[15] A | B | ... | C
[16] mk_id(sel_a:A,...,sel_b:B)
[17] sel_a:A ... sel_b:B

The following are generic type expressions:

7. The set type of finite cardinality set values.
8. The set type of infinite and finite cardinality set values.
9. The Cartesian type of Cartesian values.
10. The list type of finite length list values.
11. The list type of infinite and finite length list values.
12. The map type of finite definition set map values.
13. The function type of total function values.
14. The function type of partial function values.
15. The postulated disjoint union of types A, B, ..., and C.
16. The record type of mk_id-named record values mk_id(av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.
17. The record type of unnamed record values (av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

Section C.13 on page 86 introduces the extended RSL concepts of type name values and the type, $\mathbb{T}$, of type names.

C.2.2 Type Definitions

C.2.2.1 Sorts — Abstract Types Types can be (abstract) sorts in which case their structure is not specified:

Sorts

type
A, B, ..., C

C.2.2.2 Concrete Types Types can be concrete in which case the structure of the type is specified by type expressions:

Type Definition

```
type
  A = Type_expr
```

RSL Example: Sets. Narrative: H stand for the domain type of street intersections – we shall call then hubs, and let L stand for the domain type of segments of streets between immediately neighboring hubs – we shall call then links. Then Hs and Ls are to designate the types of finite sets of zero, one or more hubs, respectively links. **Formalisation:**

```
type H, L, Hs=H-set, Ls=L-set •
```

RSL Example: Cartesians. Narrative: Let RN stand for the domain type of road nets consisting of hub aggregates, HA, and link aggregates, LA. Hub and link aggregates can be observed from road nets, and hub sets and link sets can be observed from hub, respectively link aggregates. **Formalisation:**

```
type RN = HA×LA, Hs, Ls
value obs_HA: RN→HA, obs_LA: RN− LA, obs_Hs: HA→Hs, obs_Ls: LA→Ls
```

Observer functions, obs_... are not further defined – beyond their signatures. They will (subsequently) be defined through axioms over their results •
Some schematic type definitions are:

Variety of Type Definitions

```
[18] Type_name = Type_expr /∗ without |s or subtypes ∗/
[19] Type_name = Type_expr_1 | Type_expr_2 | ... | Type_expr_n
[20] Type_name ==
         mk_id_1(s_a1:Type_name_a1,...,s_ai:Type_name_ai) |
         ... |
         mk_id_n(s_z1:Type_name_z1,...,s_zk:Type_name_zk)
[21] Type_name :: sel_a:Type_name_a  ...  sel_z:Type_name_z
[22] Type_name = {| v:Type_name′ • P(v) |}
```

where a form of [19–20] is provided by combining the types:

Record Types

```
[23] Type_name = A | B | ... | Z
[24] A == mk_id_1(s_a1:A_1,...,s_ai:A_i)
[25] B == mk_id_2(s_b1:B_1,...,s_bj:B_j)
[26] ...
[27] Z == mk_id_n(s_z1:Z_1,...,s_zk:Z_k)
```

Of these we shall almost exclusively make use of [23–27].

Disjoint Types. Narrative: A pipeline consists of a finite set of zero, one or more [interconnected][55] pipe units. Pipe units are either wells, or are pumps, or are valves, or are joins, or are forks, or are sinks. **Formalisation:**

```
type PL = P-set, P == WU|PU|VA|JO|FO|SI, Wu,Pu,Vu,Ju,Fu,Su
     WU::mkWU(swu:Wu), PU::mkPU(spu:Pu), VA::mkVU(svu:Vu),
     JO::mkJu(sju:Ju), FO::mkFu(sfu:Fu), SI::mkSi(ssu:Su)
```

where we leave types Wu, Pu, Vu, Ju, Fu and Su further undefined •
Types A, B, ..., Z are disjoint, i.e., shares no values, provided all mk_id_k are distinct and due to the use of the disjoint record type constructor ==.

```
axiom
  ∀ a1:A_1, a2:A_2, ..., ai:Ai •
    s_a1(mk_id_1(a1,a2,...,ai))=a1 ∧ s_a2(mk_id_1(a1,a2,...,ai))=a2 ∧
    ... ∧ s_ai(mk_id_1(a1,a2,...,ai))=ai ∧
  ∀ a:A • let mk_id_1(a1′,a2′,...,ai′) = a in
    a1′ = s_a1(a) ∧ a2′ = s_a2(a) ∧ ... ∧ ai′ = s_ai(a) end
```

Note: Values of type A, where that type is defined by A::B×C×D, can be expressed A(b,c,d) for b:B, c:D, d:D.

C.2.2.3 Subtypes

In RSL, each type represents a set of values. Such a set can be delimited by means of predicates. The set of values b which have type B and which satisfy the predicate $\mathcal{P}$, constitute the subtype A:

Subtypes

```
type
  A = {| b:B • 𝒫(b) |}
```

Subtype. Narrative: The subtype of even natural numbers.
Formalisation: `type ENat = {| en | en:Nat • is_even_natural_number(en) |}` •

C.3 The Propositional and Predicate Calculi

C.3.1 Propositions

In logic, a proposition is the meaning of a declarative sentence. [A declarative sentence is a type of sentence that makes a statement]

C.3.1.1 Propositional Expressions

Propositional expressions, informally speaking, are quantifier-free expressions having truth (or **chaos**) values. ∀, ∃ and ∃! are quantifiers, see below.

[55] Although interconnected we shall not model pipelines as lists.

Below, we will first treat propositional expressions all of whose identifiers denote truth values. As we progress, in sections on arithmetic, sets, list, maps, etc., we shall extend the range of propositional expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values (**true** or **false** [or **chaos**]). Then:

Propositional Expressions

false, **true**
a, b, ..., c ∼a, a∧b, a∨b, a⇒b, a=b, a≠b

are propositional expressions having Boolean values. ∼, ∧, ∨, ⇒, =, ≠ and □ are Boolean connectives (i.e., operators). They can be read as: **not**, **and**, **or, if then** (or **implies**), **equal, not equal** and **always**.

C.3.1.2 Propositional Calculus Propositional calculus is a branch of logic. It is also called propositional logic, statement logic, sentential calculus, sentential logic, or sometimes zeroth-order logic. It deals with propositions (which can be true or false) and relations between propositions, including the construction of arguments based on them. Compound propositions are formed by connecting propositions by logical connectives. Propositions that contain no logical connectives are called atomic propositions `[Wikipedia]`
A simple two-value Boolean logic can be defined as follows:

type
  Bool
value
  true, **false**
  ∼: **Bool** → **Bool**
  ∧, ∨, ⇒, =, ≠, ≡: **Bool** × **Bool** → **Bool**
axiom
  ∀ b,b′:**Bool** •
    ∼b ≡ **if** b **then false else true end**
    b ∧ b′ ≡ **if** b **then** b′ **else false end**
    b ∨ b′ ≡ **if** b **then true else** b′ **end**
    b ⇒ b′ ≡ **if** b **then** b′ **else true end**
    b = b′ ≡ **if** (b∧b′)∨(∼b∧∼b′) **then true else false end**
    (b ≠ b′) ≡ ∼(b = b′)
    (b ≡ b′) ≡ (b = b′)

We shall, however, make use of a three-value Boolean logic. The model-theory explanation of the meaning of propositional expressions is now given in terms of the *truth tables* for the logic connectives:

∨, ∧, **and** ⇒ **Syntactic Truth Tables**

∨	true	false	chaos
true	true	true	true
false	true	false	chaos
chaos	chaos	chaos	chaos

∧	true	false	chaos
true	true	false	chaos
false	false	false	false
chaos	chaos	chaos	chaos

⇒	true	false	chaos
true	true	false	chaos
false	true	true	true
chaos	chaos	chaos	chaos

The two-value logic defined earlier 'transpires' from the **true,false** columns and rows of the above truth tables.

C.3.2 Predicates

Predicates are mathematical assertions that contains variables, sometimes referred to as predicate variables, and may be true or false depending on those variables' value or values[56]

C.3.2.1 Predicate Expressions Let x, y, ..., z (or term expressions) designate non-Boolean values, and let $\mathcal{P}(x)$, $\mathcal{Q}(y)$ and $\mathcal{R}(z)$ be propositional or predicate expressions, then:

Simple Predicate Expressions

[28] ∀x:X • $\mathcal{P}(x)$
[29] ∃y:Y • $\mathcal{Q}(y)$
[30] ∃!z:Z • $\mathcal{R}(z)$

are quantified, i.e., predicate expressions. ∀, ∃ and ∃! are the quantifiers.

C.3.2.2 Predicate Calculus They are "read" as:

[28] For all x (values in type X) the predicate $\mathcal{P}(x)$ holds – if that is not the case the expression yields truth value **false**.

[29] There exists (at least) one y (value in type Y) such that the predicate $\mathcal{Q}(y)$ holds – if that is not the case the expression yields truth value **false**.

[30] There exists a unique z (value in type Z) such that the predicate $\mathcal{R}(z)$ holds – if that is not the case the expression yields truth value **false**.

[28–30] The predicates $\mathcal{P}(x)$, $\mathcal{Q}(y)$ or $\mathcal{R}(z)$ may yield **chaos** in which case the whole expression yields **chaos**.

C.4 Arithmetics

RSL offers the usual set of arithmetic operators. From these the usual kind of arithmetic expressions can be formed.

Arithmetic

[56] https://calcworkshop.com/logic/predicate-logic/, and: predicate logic, first-order logic or quantified logic is a formal language in which propositions are expressed in terms of predicates, variables and quantifiers. It is different from propositional logic which lacks quantifiers https://brilliant.org/wiki/predicate-logic/.

```
type
  Nat, Int, Real
value
  +,−,∗: Nat×Nat→Nat | Int×Int→Int | Real×Real→Real
  /: Nat×Nat~→Nat | Int×Int~→Int | Real×Real~→Real
  <,≤,=,≠,≥,> (Nat|Int|Real) → (Nat|Int|Real)
```

C.5 Comprehensive Expressions

Comprehensive expressions are common in mathematics texts. They capture properties conveniently abstractly

C.5.1 Set Enumeration and Comprehension

C.5.1.1 Set Enumeration Let the below a's denote values of type A:

Set Enumerations

$\{\{\}, \{a\}, \{e_1,e_2,...,e_n\}, ...\} \in$ A-**set**
$\{\{\}, \{a\}, \{e_1,e_2,...,e_n\}, ..., \{e_1,e_2,...\}\} \in$ A-**infset**

C.5.1.2 Set Comprehension The expression, last line below, to the right of the ≡, expresses set comprehension. The expression "builds" the set of values satisfying the given predicate. It is abstract in the sense that it does not do so by following a concrete algorithm.

Set Comprehension

```
type
  A, B
  P = A → Bool
  Q = A ~→ B
value
  comprehend: A-infset × P × Q → B-infset
  comprehend(s,P,Q) ≡ { Q(a) | a:A • a ∈ s ∧ P(a)}
```

C.5.1.3 Cartesian Enumeration Let e range over values of Cartesian types involving A, B, ..., C, then the below expressions are simple Cartesian enumerations:

Cartesian Enumerations

```
type
  A, B, ..., C
  A × B × ... × C
```

```
value
  (e1,e2,...,en)
```

C.5.2 List Enumeration and Comprehension

C.5.2.1 List Enumeration Let a range over values of type A, then the below expressions are simple list enumerations:

List Enumerations

```
{⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ...} ∈ A*
{⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ..., ⟨e1,e2,...,en,... ⟩, ...} ∈ Aω

⟨ a_i .. a_j ⟩
```

The last line above assumes a_i and a_j to be integer-valued expressions. It then expresses the set of integers from the value of e_i to and including the value of e_j. If the latter is smaller than the former, then the list is empty.

C.5.2.2 List Comprehension The last line below expresses list comprehension.

List Comprehension

```
type
  A, B, P = A → Bool, Q = A ~→ B
value
  comprehend: Aω × P × Q ~→ Bω
  comprehend(l,P,Q) ≡ ⟨ Q(l(i)) | i in ⟨1..len l⟩ • P(l(i))⟩
```

C.5.3 Map Enumeration and Comprehension

C.5.3.1 Map Enumeration Let (possibly indexed) u and v range over values of type $T1$ and $T2$, respectively, then the below expressions are simple map enumerations:

Map Enumerations

```
type
  T1, T2
  M = T1 m→T2
value
  u,u1,u2,...,un:T1, v,v1,v2,...,vn:T2
  [ ], [ u↦v ], ..., [ u1↦v1,u2↦v2,...,un↦vn ] ∀ ∈ M
```

C.5.3.2 Map Comprehension

The last line below expresses map comprehension:

Map Comprehension

```
type
  U, V, X, Y
  M = U m→ V
  F = U ~→ X
  G = V ~→ Y
  P = U → Bool
value
  comprehend: M×F×G×P → (X m→ Y)
  comprehend(m,F,G,P) ≡ [ F(u) ↦ G(m(u)) | u:U • u ∈ dom m ∧ P(u) ]
```

C.6 Operations

C.6.1 Set Operations

C.6.1.1 Set Operator Signatures

Set Operator Signatures

```
value
  18 ∈: A × A-infset → Bool
  19 ∉: A × A-infset → Bool
  20 ∪: A-infset × A-infset → A-infset
  21 ∪: (A-infset)-infset → A-infset
  22 ∩: A-infset × A-infset → A-infset
  23 ∩: (A-infset)-infset → A-infset
  24 \: A-infset × A-infset → A-infset
  25 ⊂: A-infset × A-infset → Bool
  26 ⊆: A-infset × A-infset → Bool
  27 =: A-infset × A-infset → Bool
  28 ≠: A-infset × A-infset → Bool
  29 card: A-infset ~→ Nat
```

C.6.1.2 Set Operation Examples

Set Operation Examples

```
examples
  a ∈ {a,b,c}
  a ∉ {}, a ∉ {b,c}
  {a,b,c} ∪ {a,b,d,e} = {a,b,c,d,e}
  ∪{{a},{a,b},{a,d}} = {a,b,d}
  {a,b,c} ∩ {c,d,e} = {c}
```

```
∩{{a},{a,b},{a,d}} = {a}
{a,b,c} \ {c,d} = {a,b}
{a,b} ⊂ {a,b,c}
{a,b,c} ⊆ {a,b,c}
{a,b,c} = {a,b,c}
{a,b,c} ≠ {a,b}
card {} = 0, card {a,b,c} = 3
```

C.6.1.3 Informal Set Operator Explication The following is **not** a definition of RSL semantics. In RSL formulas we present an explication of RSL operators. Read, what appears as definitions, ≡, as [a kind of] identities.

18. ∈: The membership operator expresses that an element is a member of a set.
19. ∉: The nonmembership operator expresses that an element is not a member of a set.
20. ∪: The infix union operator. When applied to two sets, the operator gives the set whose members are in either or both of the two operand sets.
21. ∪: The distributed prefix union operator. When applied to a set of sets, the operator gives the set whose members are in all of the operand sets.
22. ∩: The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.
23. ∩: The prefix distributed intersection operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.
24. \: The set complement (or set subtraction) operator. When applied to two sets, the operator gives the set whose members are those of the left operand set which are not in the right operand set.
25. ⊆: The proper subset operator expresses that all members of the left operand set are also in the right operand set.
26. ⊂: The proper subset operator expresses that all members of the left operand set are also in the right operand set, and that the two sets are not identical.
27. =: The equal operator expresses that the two operand sets are identical.
28. ≠: The nonequal operator expresses that the two operand sets are *not* identical.
29. **card**: The cardinality operator gives the number of elements in a finite set.

C.6.1.4 Set Operator Explications The set operations can be "equated" as follows:

Set Operator Explications

```
value
```

```
  s′ ∪ s″ ≡ { a | a:A • a ∈ s′ ∨ a ∈ s″ }
  s′ ∩ s″ ≡ { a | a:A • a ∈ s′ ∧ a ∈ s″ }
  s′ \ s″ ≡ { a | a:A • a ∈ s′ ∧ a ∉ s″ }
  s′ ⊆ s″ ≡ ∀ a:A • a ∈ s′ ⇒ a ∈ s″
  s′ ⊂ s″ ≡ s′ ⊆ s″ ∧ ∃ a:A • a ∈ s″ ∧ a ∉ s′
  s′ = s″ ≡ ∀ a:A • a ∈ s′ ≡ a ∈ s″ ≡ s⊆s′ ∧ s′⊆s
  s′ ≠ s″ ≡ s′ ∩ s″ ≠ {}
  card s ≡
    if s = {} then 0 else
    let a:A • a ∈ s in 1 + card (s \ {a}) end end
    pre s /∗ is a finite set ∗/
  card s ≡ chaos /∗ tests for infinity of s ∗/
```

C.6.2 Cartesian Operations

Cartesian Operations

```
type
  A, B, C
  g0: G0 = A × B × C
  g1: G1 = ( A × B × C )
  g2: G2 = ( A × B ) × C
  g3: G3 = A × ( B × C )

value
  va:A, vb:B, vc:C, vd:D
  (va,vb,vc):G0,
  (va,vb,vc):G1
  ((va,vb),vc):G2
  (va3,(vb3,vc3)):G3

decomposition expressions
  let (a1,b1,c1) = g0,
      (a1′,b1′,c1′) = g1 in .. end
  let ((a2,b2),c2) = g2 in .. end
  let (a3,(b3,c3)) = g3 in .. end
```

C.6.3 List Operations

C.6.3.1 List Operator Signatures

List Operator Signatures

```
value
  hd: A^ω ~→ A
  tl: A^ω ~→ A^ω
  len: A^ω ~→ Nat
  inds: A^ω → Nat-infset
  elems: A^ω → A-infset
  .(.): A^ω × Nat ~→ A
  ⌢: A* × A^ω → A^ω
  =: A^ω × A^ω → Bool
```

$\neq$: $A^\omega \times A^\omega \to$ **Bool**

C.6.3.2 List Operation Examples

List Operation Examples

examples
hd⟨a1,a2,...,am⟩=a1
tl⟨a1,a2,...,am⟩=⟨a2,...,am⟩
len⟨a1,a2,...,am⟩=m
inds⟨a1,a2,...,am⟩={1,2,...,m}
elems⟨a1,a2,...,am⟩={a1,a2,...,am}
⟨a1,a2,...,am⟩(i)=ai
⟨a,b,c⟩⌢⟨a,b,d⟩ = ⟨a,b,c,a,b,d⟩
⟨a,b,c⟩=⟨a,b,c⟩
⟨a,b,c⟩ ≠ ⟨a,b,d⟩

C.6.3.3 Informal List Operator Explication The following is **not** a definition of RSL semantics. In RSL formulas we present an explication of RSL operators. Read, what appears as definitions, ≡, as [a kind of] identities.

- **hd**: Head gives the first element in a nonempty list.
- **tl**: Tail gives the remaining list of a nonempty list when Head is removed.
- **len**: Length gives the number of elements in a finite list.
- **inds**: Indices give the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems**: Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$: Indexing with a natural number, i larger than 0, into a list ℓ having a number of elements larger than or equal to i, gives the ith element of the list.
- $\frown$: Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.
- =: The equal operator expresses that the two operand lists are identical.
- $\neq$: The nonequal operator expresses that the two operand lists are *not* identical.

The operations can also be defined as follows:

C.6.3.4 List Operator Explications

The following is **not** a definition of RSL semantics. In RSL formulas we present an explication of RSL operators. Read, what appears as definitions, ≡, as [a kind of] identities.

List Operator Explications

```
value
  is_finite_list: A^ω → Bool

  len q ≡
    case is_finite_list(q) of
      true → if q = ⟨⟩ then 0 else 1 + len tl q end,
      false → chaos end

  inds q ≡
    case is_finite_list(q) of
      true → { i | i:Nat • 1 ≤ i ≤ len q },
      false → { i | i:Nat • i≠0 } end

  elems q ≡ { q(i) | i:Nat • i ∈ inds q }

  q(i) ≡
    if i=1
      then
        if q≠⟨⟩
          then let a:A,q′:Q • q=⟨a⟩^q′ in a end
          else chaos end
      else q(i−1) end

  fq ^ iq ≡
      ⟨ if 1 ≤ i ≤ len fq then fq(i) else iq(i − len fq) end
      | i:Nat • if len iq≠chaos then i ≤ len fq+len end ⟩
    pre is_finite_list(fq)

  iq′ = iq″ ≡
    inds iq′ = inds iq″ ∧ ∀ i:Nat • i ∈ inds iq′ ⇒ iq′(i) = iq″(i)

  iq′ ≠ iq″ ≡ ∼(iq′ = iq″)
```

C.6.4 Map Operations

C.6.4.1 Map Operator Signatures

Map Operator Signatures

```
value
```

[30] $\cdot(\cdot)$: M → A $\xrightarrow{\sim}$ B
[31] **dom**: M → A-**infset** [domain of map]
[32] **rng**: M → B-**infset** [range of map]
[33] †: M × M → M [override extension]
[34] ∪: M × M → M [merge ∪]
[35] \: M × A-**infset** → M [restriction by]
[36] /: M × A-**infset** → M [restriction to]
[37] =,≠: M × M → **Bool**
[38] °: (A $\xrightarrow[m]{}$ B) × (B $\xrightarrow[m]{}$ C) → (A $\xrightarrow[m]{}$ C) [composition]

C.6.4.2 Map Operation Examples

Map Operation Examples

value
[30] m(a) = b
[31] **dom** [a1↦b1,a2↦b2,...,an↦bn] = {a1,a2,...,an}
[32] **rng** [a1↦b1,a2↦b2,...,an↦bn] = {b1,b2,...,bn}
[33] [a↦b,a′↦b′,a″↦b″] † [a′↦b″,a″↦b′] = [a↦b,a′↦b″,a″↦b′]
[34] [a↦b,a′↦b′,a″↦b″] ∪ [a‴↦b‴] = [a↦b,a′↦b′,a″↦b″,a‴↦b‴]
[35] [a↦b,a′↦b′,a″↦b″]\{a} = [a′↦b′,a″↦b″]
[37] [a↦b,a′↦b′,a″↦b″]/{a′,a″} = [a′↦b′,a″↦b″]
[38] [a↦b,a′↦b′] ° [b↦c,b′↦c′,b″↦c″] = [a↦c,a′↦c′]

C.6.4.3 Informal Map Operation Explication

- $m(a)$: Application gives the element that a maps to in the map m.
- **dom**: Domain/Definition Set gives the set of values which *maps to* in a map.
- **rng**: Range/Image Set gives the set of values which *are mapped to* in a map.
- †: Override/Extend. When applied to two operand maps, it gives the map which is like an override of the left operand map by all or some "pairings" of the right operand map.
- ∪: Merge. When applied to two operand maps, it gives a merge of these maps.
- \: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements that are not in the right operand set.
- /: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements of the right operand set.
- =: The equal operator expresses that the two operand maps are identical.
- ≠: The nonequal operator expresses that the two operand maps are *not* identical.

- °: Composition. When applied to two operand maps, it gives the map from definition set elements of the left operand map, m_1, to the range elements of the right operand map, m_2, such that if a is in the definition set of m_1 and maps into b, and if b is in the definition set of m_2 and maps into c, then a, in the composition, maps into c.

C.6.4.4 Map Operator Explication

The following is **not** a definition of RSL semantics. In RSL formulas we present an explication of RSL operators. Read, what appears as definitions, ≡, as [a kind of] identities.
The map operations can also be defined as follows:

Map Operator Explications

```
value
  rng m ≡ { m(a) | a:A • a ∈ dom m }

  m1 † m2 ≡
    [ a↦b | a:A,b:B •
      a ∈ dom m1 \ dom m2 ∧ b=m1(a) ∨ a ∈ dom m2 ∧ b=m2(a) ]

  m1 ∪ m2 ≡ [ a↦b | a:A,b:B •
              a ∈ dom m1 ∧ b=m1(a) ∨ a ∈ dom m2 ∧ b=m2(a) ]

  m \ s ≡ [ a↦m(a) | a:A • a ∈ dom m \ s ]
  m / s ≡ [ a↦m(a) | a:A • a ∈ dom m ∩ s ]

  m1 = m2 ≡
    dom m1 = dom m2 ∧ ∀ a:A • a ∈ dom m1 ⇒ m1(a) = m2(a)
  m1 ≠ m2 ≡ ∼(m1 = m2)

  m°n ≡
    [ a↦c | a:A,c:C • a ∈ dom m ∧ c = n(m(a)) ]
    pre rng m ⊆ dom n
```

C.7 λ-Calculus + Functions

The λ-Calculus is a foundation for the abstract specification language that RSL is

C.7.1 The λ-Calculus Syntax

λ-Calculus Syntax

```
type /∗ A BNF Syntax: ∗/
    ⟨L⟩ ::= ⟨V⟩ | ⟨F⟩ | ⟨A⟩ | ( ⟨A⟩ )
    ⟨V⟩ ::= /∗ variables, i.e. identifiers ∗/
    ⟨F⟩ ::= λ⟨V⟩ • ⟨L⟩
    ⟨A⟩ ::= ( ⟨L⟩⟨L⟩ )
```

```
value /∗ Examples ∗/
    ⟨L⟩: e, f, a, ...
    ⟨V⟩: x, ...
    ⟨F⟩: λ x • e, ...
    ⟨A⟩: f a, (f a), f(a), (f)(a), ...
```

C.7.2 Free and Bound Variables

Free and Bound Variables

Let x, y be variable names and e, f be λ-expressions.

- ⟨V⟩: Variable x is free in e.
- ⟨F⟩: x is free in $\lambda y \bullet e$ if $x \neq y$ and x is free in e.
- ⟨A⟩: x is free in $f(e)$ if it is free in either f or e (i.e., also in both).

C.7.3 Substitution

In RSL, the following rules for substitution apply:

Substitution

- **subst**([N/x]x) ≡ N;
- **subst**([N/x]a) ≡ a,

  for all variables a≠ x;
- **subst**([N/x](P Q)) ≡ (**subst**([N/x]P) **subst**([N/x]Q));
- **subst**([N/x](λx•P)) ≡ λ y•P;
- **subst**([N/x](λ y•P)) ≡ λy• **subst**([N/x]P),

  if x≠y and y is not free in N or x is not free in P;
- **subst**([N/x](λy•P)) ≡ λz•**subst**([N/z]**subst**([z/y]P)),

  if y≠x and y is free in N and x is free in P
  (where z is not free in (N P)).

C.7.4 α-Renaming and β-Reduction

α and β Conversions

- α-renaming: λx•M

If x, y are distinct variables then replacing x by y in λx•M results in λy•**subst**([y/x]M). We can rename the formal parameter of a λ-function expression provided that no free variables of its body M thereby become bound.

- β-reduction: (λx•M)(N)

  All free occurrences of x in M are replaced by the expression N provided that no free variables of N thereby become bound in the result. (λx•M)(N) ≡ **subst**([N/x]M)

C.7.5 Function Signatures

For sorts we may want to postulate some functions:

Sorts and Function Signatures

```
type
  A, B, C
value
  obs_B: A → B,
  obs_C: A → C,
  gen_A: B×C → A
```

C.7.6 Function Definitions

Functions can be defined explicitly:

Explicit Function Definitions

```
value
  f: Arguments → Result
  f(args) ≡ DValueExpr

  g: Arguments ~→ Result
  g(args) ≡ ValueAndStateChangeClause
  pre P(args)
```

Or functions can be defined implicitly:

Implicit Function Definitions

```
value
  f: Arguments → Result
  f(args) as result
  post P1(args,result)

  g: Arguments ~→ Result
```

g(args) **as** result
pre P2(args)
post P3(args,result)

The symbol $\tilde{\rightarrow}$ indicates that the function is partial and thus not defined for all arguments. Partial functions should be assisted by preconditions stating the criteria for arguments to be meaningful to the function.

C.8 Other Applicative Expressions

RSL offers the usual collection of applicative constructs that functional programming languages (Standard ML [54, 54] or F# [45]) offer

C.8.1 Simple let Expressions

Simple (i.e., nonrecursive) **let** expressions:

Let Expressions

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

is an "expanded" form of:

$(\lambda a.\mathcal{E}_b(a))(\mathcal{E}_d)$

C.8.2 Recursive let Expressions

Recursive **let** expressions are written as:

Recursive **let** Expressions

let f = λa:A • E(f) **in** B(f,a) **end**

is "the same" as:

let f = **Y**F **in** B(f,a) **end**

where:

F ≡ λg•λa•(E(g)) and YF = F(YF)

C.8.3 Predicative let Expressions

Predicative **let** expressions:

Predicative **let** Expressions

let a:A • $\mathcal{P}$(a) **in** $\mathcal{B}$(a) **end**

express the selection of a value a of type A which satisfies a predicate $\mathcal{P}$(a) for evaluation in the body $\mathcal{B}$(a).

C.8.4 Pattern and "Wild Card" let Expressions

Patterns and *wild cards* can be used:

Patterns

let {a} ∪ s = set **in** ... **end**
let {a,_} ∪ s = set **in** ... **end**

let (a,b,...,c) = cart **in** ... **end**
let (a,_,...,c) = cart **in** ... **end**

let ⟨a⟩^ℓ = list **in** ... **end**
let ⟨a,_,b⟩^ℓ = list **in** ... **end**

let [a↦b] ∪ m = map **in** ... **end**
let [a↦b,_] ∪ m = map **in** ... **end**

C.8.4.1 Conditionals

Various kinds of conditional expressions are offered by RSL:

Conditionals

```
if b_expr then c_expr else a_expr
end

if b_expr then c_expr end ≡ /∗ same as: ∗/
   if b_expr then c_expr else skip end

if b_expr_1 then c_expr_1
elsif b_expr_2 then c_expr_2
elsif b_expr_3 then c_expr_3
...
elsif b_expr_n then c_expr_n end

case expr of
   choice_pattern_1 → expr_1,
```

```
    choice_pattern_2 → expr_2,
    ...
    choice_pattern_n_or_wild_card → expr_n
end
```

C.8.5 Operator/Operand Expressions

Operator/Operand Expressions

⟨Expr⟩ ::=
⟨Prefix_Op⟩ ⟨Expr⟩
| ⟨Expr⟩ ⟨Infix_Op⟩ ⟨Expr⟩
| ⟨Expr⟩ ⟨Suffix_Op⟩
| ...
⟨Prefix_Op⟩ ::=
− | ∼ | ∪ | ∩ | **card** | **len** | **inds** | **elems** | **hd** | **tl** | **dom** | **rng**
⟨Infix_Op⟩ ::=
= | ≠ | ≡ | + | − | ∗ | ↑ | / | < | ≤ | ≥ | > | ∧ | ∨ | ⇒
| ∈ | ∉ | ∪ | ∩ | \ | ⊂ | ⊆ | ⊇ | ⊃ | ⌢ | † | °
⟨Suffix_Op⟩ ::= !

C.9 Imperative Constructs

RSL offers the usual collection of imperative constructs that imperative programming languages (Java [44, 59] or Oberon (!) [66]) offer

C.9.1 Statements and State Changes

Often, following the RAISE method, software development starts with highly abstract-applicative constructs which, through stages of refinements, are turned into concrete and imperative constructs. Imperative constructs are thus inevitable in RSL.

Statements and State Change

```
  Unit
value
  stmt: Unit → Unit
  stmt()
```

- Statements accept no arguments.
- Statement execution changes the state (of declared variables).

- **Unit** → **Unit** designates a function from states to states.
- Statements, stmt, denote state-to-state changing functions.
- Writing () as "only" arguments to a function "means" that () is an argument of type **Unit**.

C.9.2 Variables and Assignment

Variables and Assignment

0. **variable** v:Type := expression
1. v := expr

C.9.3 Statement Sequences and skip

Sequencing is expressed using the ';' operator. **skip** is the empty statement having no value or side-effect.

Statement Sequences and **skip**

2. **skip**
3. stm_1;stm_2;...;stm_n

C.9.4 Imperative Conditionals

Imperative Conditionals

4. **if** expr **then** stm_c **else** stm_a **end**
5. **case** e **of**: p_1→S_1(p_1),...,p_n→S_n(p_n) **end**

C.9.5 Iterative Conditionals

Iterative Conditionals

6. **while** expr **do** stm **end**
7. **do** stmt **until** expr **end**

C.9.6 Iterative Sequencing

Iterative Sequencing

8. **for** e **in** list_expr • P(b) **do** S(b) **end**

C.10 Process Constructs

RSL offers several of the constructs that CSP [47] offers

C.10.1 Process Channels

As for channels we deviate from common RSL [43] in that we directly *declare* channels – and not via common RSL *objects* etc.

Let A and B stand for two types of (channel) messages and i:KIdx for channel array indexes, then:

Process Channels

channel c:A
channel { k[i]:B • i:Idx }
channel { k[i,j,...,k]:B • i:Idx,j:Jdx,...,k:Kdx }

declare a channel, c, and a set (an array) of channels, k[i], capable of communicating values of the designated types (A and B).

C.10.2 Process Composition

Let P and Q stand for names of process functions, i.e., of functions which express willingness to engage in input and/or output events, thereby communicating over declared channels. Let P() and Q stand for process expressions, then:

Process Composition

P ‖ Q Parallel composition
P □ Q Nondeterministic external choice (either/or)
P ⌈⌉ Q Nondeterministic internal choice (either/or)
P ‡ Q Interlock parallel composition

express the parallel (‖) of two processes, or the nondeterministic choice between two processes: either external (□) or internal (⌈⌉). The interlock (‡) composition expresses that the two processes are forced to communicate only with one another, until one of them terminates.

C.10.3 Input/Output Events

Let c, k[i] and e designate channels of type A and B, then:

Input/Output Events

```
c ?, k[i] ?      Input
c ! e, k[i] ! e  Output
```

expresses the willingness of a process to engage in an event that "reads" an input, respectively "writes" an output.

C.10.4 Process Definitions

The below signatures are just examples. They emphasise that process functions must somehow express, in their signature, via which channels they wish to engage in input and output events.

Process Definitions

```
value
  P: Unit → in c out k[i]
  Unit
  Q: i:KIdx →  out c in k[i] Unit

  P() ≡ ... c ? ... k[i] ! e ...
  Q(i) ≡ ... k[i] ? ... c ! e ...
```

The process function definitions (i.e., their bodies) express possible events.

C.11 RSL Module Specifications

We shall not include coverage nor use of the RSL module concepts of *schemes, classes* and *objects*.

C.12 Simple RSL Specifications

Often, we do not want to encapsulate small specifications in schemas, classes, and objects, as is often done in RSL. An RSL specification is simply a sequence of one or more types, values (including functions), variables, channels and axioms:

Simple RSL Specifications

```
type
  ...
variable
  ...
channel
  ...
value
```

```
  ...
axiom
  ...
```

C.13 RSL⁺: Extended RSL

Section C.2 on page 62 covered standard RSL types. To them we now add two new types: Type names and RSL⁺Text.

C.13.1 Type Names and Type Name Values

C.13.1.1 Type Names

- Let T be a type name.
- Then ηT is a type name value.
- And $\eta\mathbb{T}$ is the type of type names.

C.13.1.2 Type Name Operations

- η can be considered an operator.
  - It (prefix) applies, then, to type (T) identifiers and yields the name of that type.
  - Two type names, nT_i, nT_j, can be compared for equality: $\mathsf{nT}_i = \mathsf{nT}_j$ iff $i = j$.
- It, vice-versa, suffix applies to type name (nT) identifiers and yields the name, T, of that type: $\mathsf{nT}\eta = \mathsf{T}$.

C.13.2 RSL⁺Text

C.13.2.1 The RSL⁺Text Type and Values

- RSL⁺Text is the type name for ordinary, non-extended RSL texts.

We shall not here give a syntax for ordinary, non-extended RSL texts – but refer to [43].

C.13.2.2 RSL⁺Text Operations

- RSL⁺Texts can be compared and concatenated:
  - $\mathsf{rsl\text{-}text}_a$=$\mathsf{rsl\text{-}text}_b$
  - $\mathsf{rsl\text{-}text}_a\,\widehat{}\,\mathsf{rsl\text{-}text}_b$

The $\widehat{}$ operator thus also applies, besides, lists (tuples), to RSL texts – treating RSL texts as (if they were) lists of characters.

D A Wolt & An ℝSL Index

D.1 The Wolt ℝOMAIN Index

There are 257 formal *Wolt* DOMAIN entities.

D.2 The RSL Index

E Notes

E.1 General Notes

I offer this working document on the `Internet`[57] for Your perusal: to see the day-to-day "progress" of the work:

- **Wed Jul 8 08:53:55:** Started work on this document.
- **Sat Jul 11 09:15:02:** I have yesterday completed up to and including `Mereology`.
  I still do not know how the various behaviours actually interact!
  Today I will work on "fixing" some *undefined references* [done by 10:15] and start on `Attributes`.
- **Wed Jul 15 16:25:12:**I finished the 'CUSTOMER'/'GROCERY STORE' actkions – almost!

E.2 Modeling Notes

30. **Modeling Part Sets:**
    (a) as endurants:
        i. as we have done for the part sets of couriers:of the courier management (**CC**)
        ii. although their number may increase/decrease (hiring/laying off)
        iii. we decide to not model that.
    (b) as attributes:
        i. as we have done for groceries of the grocery store (**GS**);
        ii. here we model that their numebr do indeed decrease/increase.
31. **Road Nets & Maps**
    (a) In many previous publications and reports we model domains involving road, and, in general, transport nets.
    (b) In this document we shall not model domains of roads, but specify road maps.
30. Do I need KO, customer acconts ? Should it not be an atribute in GS ?
31.
32.

E.3 Project Notes

33. We distinguish between
    - one- or two-person exploratory
    - and properly staffed, say commecial

    domain modeling efforts.

    The current report is a one-person exploratory effort.

[57] `/home/dibj/public_html/2026/wolt/main.pdf`

34. A properly staffed domain modeling effort would typically have one or two domain analyser cum describers per major endurant.

    In the present case that would, for example, mean:

    - One chief domain analyser cum describer plus an assistant for the modeling of the delivery service, **DS**;
    - two persons to handle the grocery store, **GS**;
    - One person each to handle
      - customers, **KS, K**;
      - income and supply, **IN, SU**;
      - courier management **CC**;
      - couriers, **C**; and
      - the order and bookkeepin departments, **KO, BK**.

    – for a staff total of typically eight !

35.

36.

37.