5 Discovering Domain Entities 219

Section 2 briefly characterised, informally and also a bit more formally, what we mean by a domain. Section 3 informally and systematically characterised the four categories of entities: parts, actions, events and behaviours. Section 4 more-or-less "repeated" Sect. 3's material but by now giving more terse narratives (that is, informal descriptions) and, for the fist time, also formalisations. Section 4 did not hint at how one discovers domain parts (i.e., their types), actions, events and behaviours. In this section we try unravel a set of techniques and tools — so-called 'discoverers' and 'analysers' — using which the domain describer (scientist and/or engineer) can more-or-less systematically discover, analyse and describe a domain, informally and formally.

5.1 Preliminaries 220

Before we present the discoverers and analysers we need establish some concepts.

5.1.1 Part Signatures 221

Let us consider a part $p : P$. Let $p : P$, by definition, be the *principal part* of a domain. Now we need to identify (i) the type, P, of that part; (ii) the types, $\mathsf{S}_1, \ldots, \mathsf{S}_m$, of its proper sub-parts (if p is composite); (iii) the type, PI, of its unique identifier; (iv) the possible types, $\mathsf{MI}_1, \ldots, \mathsf{MI}_n$, of its mereology; and (v) the types, $\mathsf{A}_1, \ldots, \mathsf{A}_o$, of its attributes. We shall name that cluster of type identifications the *part signature*. We refer to P as identifying the part signature. Each of the S_i (for $i : \{1..m\}$) identifies sub-parts and hence
222 sub-part, i.e., part signatures.

Example 41 (Net Domain and Sub-domain Part Signatures) The part signature of the hubs and the links are here chosen to be those of(i) the (root) net type, N, (ii) the (sub-domain) set of hubs type Hs, (ii) the (sub-domain) set of links type Ls and (v) the type of net attributes Net_name, Net_owner, etc. The part signatures Hs and Ls are (ii) Hs = H-set, H (iii,iv) HI, LI-set (v) Hub_Nm, Location, HΣ, HΩ, ... (ii) Ls = L-set, L (iii,iv) LI, HI-set (v) Link_Nm, LΣ, LΩ, LEN, etc. •

5.1.2 Domain Indices 223

By a domain index we mean a list of part type names that identify a sequence of part signatures. More specifically The domain Δ has index $\langle\Delta\rangle$. The sub-domains of Δ, with part types A, B, ..., C, has indices $\langle\Delta,A\rangle$, $\langle\Delta,B\rangle$, $\ldots$, $\langle\Delta,C\rangle$. The sub-domains of sub-domain with index ℓ and with part types A, B, ..., C has indices $\ell^\frown\langle A\rangle$, $\ell^\frown\langle B\rangle$, $\ldots$, $\ell^\frown\langle C\rangle$.
224

Example 42 (Indices of a Road Pricing Domain) We refer to the the *Road-pricing Transport Domain*, cf. Example 36 on page 56.

The sub-domain indices of the road-pricing transport domain, Δ, are: ⟨Δ⟩, ⟨Δ,N⟩, ⟨Δ,F⟩, ⟨Δ,M⟩, ⟨Δ,N,Vs⟩, ⟨Δ,N,Ls⟩, ⟨Δ,N,H⟩, ⟨Δ,N,L⟩ and ⟨Δ,F,V⟩. •

5.1.3 Inherited Domain Signatures 225

Let ⟨Δ,A,B,C,D⟩ be some domain index. Then ⟨Δ,A,B,C⟩ ⟨Δ,A,B⟩ ⟨Δ,A⟩ ⟨Δ⟩ are the inherited domain indices of ⟨Δ,A,B,C,D⟩.

5.1.4 Domain and Sub-domain Categories 226

By the domain category of the domain indexed by $\ell^\frown\langle$D$\rangle$ we shall mean the domain signature of D, and the action, event and behaviour definitions whose signatures involves just the types given in the domain signature of D or in inherited domain signatures. 227

Example 43 (The Road-pricing Domain Category) The road-pricing domain category consist of the types N, F and M, the create_Net create_Fleet and create_M actions, and corresponding Net, Fleet and M behaviours •

228

By a sub-domain category, of index ℓ, we shall mean the sub-domain types of the sub-domain designated by index ℓ, and the actions, events and behaviours whose signatures involves just the types of the ℓ indexed sub-domain or of any prefix of ℓ indexed sub-domain or of the root domain. 229

Example 44 (A Hub Category of a Road-pricing Transport Domain) The ancestor sub-domain types of the hub sub-domain are: *HS*, *N* and Δ. The hub category thus includes the part (etc.) types H, HI, ..., the insert_Hub and the delete_Hub actions, perhaps some saturated_hub (and/or other) event(s), but probably no hub behaviour as it would involve at least the type LI which is not in an ancestor sub-domain of the Hub sub-domain. •

5.1.5 Simple and Compound Indexes 230

By a simple index we mean a domain or a sub-domain index. By a compound index we mean a set of two or more distinct indices of a domain Δ. Compound indices, c_{idx} : $\{\ell_i, \ell_j, \ldots, \ell_k\}$, designate parts, actions, events and behaviours each of whose types and signatures involve types defined by all of the simple indexes of c_{idx}.

Example 45 (Compound Indices of the Road-pricing System) We show just one compound index: {⟨Δ,N,HS,H⟩,⟨Δ,N,LS,L⟩}. •

5.1.6 Simple and Compound Domain Categories 231

By a simple domain category we shall mean any ℓ-indexed [sub-]domain category. By the compound domain category of compound index $c_{idx} : \{\ell_i, \ell_j, \ldots, \ell_k\}$, we shall mean the set of types, actions, events and behaviours as induced by compound index c_{idx}, that is, parts, actions, events and behaviours each of whose types and signatures involve types defined by all of the simple indexes of c_{idx}. 232

Example 46 (The Compound Domain Category of Hubs and Links) The compound domain category designated by $\{\langle\Delta,N,HS,H\rangle,\langle\Delta,N,LS,L\rangle\}$ includes:

```
type
    HIs = HI-set
        axiom ∀ his:HIs•card his=2
    LIs = LI-set
    HΣ = (LI×LI)-set
    LΣ = (HI×HI)-set
    HΩ = HΣ-set
    LΩ = LΣ-set
value
    mereo_L: L → HIs,
    mereo_H: H → LIs
    attr_HΣ: H → HΣ
    attr_LΣ: L → LΣ
    attr_HΩ: H → HΩ
    attr_LΩ: L → LΩ
axiom
    ∀ h:H•attr_HΣ(h)⊆attr_HΩ(h)
    ∀ l:L•attr_LΣ(l)⊆attr_LΩ(l)
```

•

5.1.7 Examples 233

We repeat some examples, but now "formalised".

Example 47 (The Root Domain Category) We start at the root, Δ, of the Road Pricing Domain. See Fig. 13.

Figure 13: The $\langle\Delta\rangle$ Root

At the root we 'discover' the net, fleet and road pricing monitor. See Fig. 14 on the facing page. 234

When observing the very essence of the road pricing domain "at the $\langle\Delta\rangle$ level" one observes: 235

Figure 14: Exploring the root index ⟨Δ⟩ Index

type
N, F, M
value
obs_N: Δ → N
obs_F: Δ → F
obs_M: Δ → M
attr_...: Δ → ...

where ... stands for types of road pricing domain attributes.

•

236

Example 48 (The Net Domain Category) We then proceed to explore the domain at index ⟨Δ,N⟩. See Fig. 15.

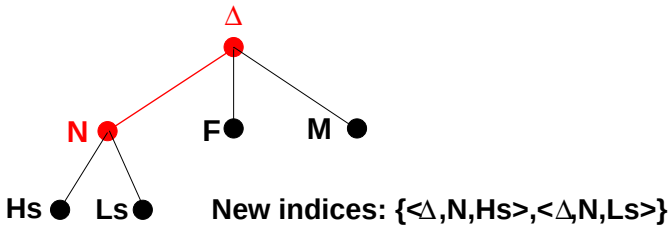

Figure 15: Exploring the ⟨Δ,N⟩ Index

237

When observing the very essence of the Net domain, "at the ⟨Δ, N⟩ level" one observes:

type
Hs = H-**set**
Ls = L-**set**
H
L
...
value
obs_Hs: N → Hs
obs_Ls: N → Ls
attr_Hs: Hs → ...
attr_Ls: Ls → ...

where ... stand for attributes of the Hs and the Ls parts of N.

•

238

Example 49 (The Fleet Domain Category) We then proceed to explore the domain at index ⟨Δ,F⟩. See Fig. 16.

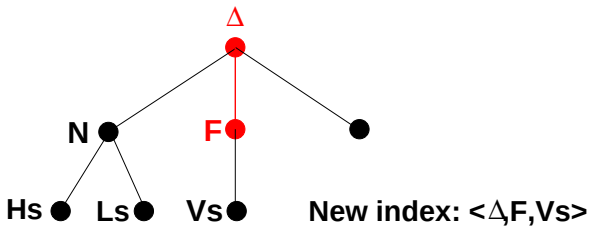

Figure 16: Exploring the ⟨Δ,F⟩ Index

239

When observing the very essence of the Fleet domain, "at the ⟨Δ, F⟩ level" one observes:

type
  Vs = V-**set**
  V
  ...
value
  obs_Vs: F → Vs
  attr_... Vs → ...

where ... stand for attributes that we may wish to associate with Fleets of vehicles.

•

240

Example 50 (The Hub Domain Category) We now switch "back" to explore the domain
241 at index ⟨Δ,N,Hs⟩. See Fig. 17 on the next page.

When observing the very essence of the Fleet domain, "at the ⟨Δ, N,Hs,H⟩ level" one observes:

type
  HI
  ...
value
  uid_HI: H → HI
  attr_...: H → ...

where ... stand for LOCation, etc.

•

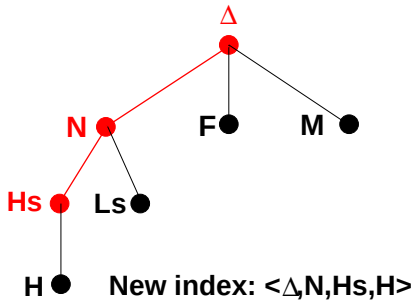

Figure 17: Exploring the ⟨Δ, N,Hs,H⟩ Index

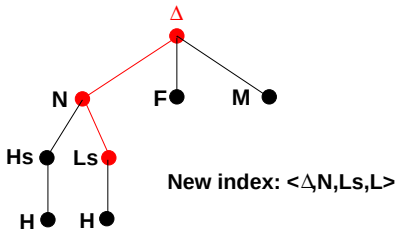

Figure 18: Exloring the ⟨Δ, N,Ls,L⟩ Index

242

243 **Example 51 (The Link Domain Category)** Next we explore the link domain. See Fig. 18.

When observing the very essence of the Fleet domain, "at the ⟨Δ, N,Ls,L⟩ level" one observes:

type
  LI
  ...
value
  uid_LI: L → LI
  attr_...: L → ...

where ... stand for LOCation, LENgth, etc.

•

244

Example 52 (The Compound Hub and Link Domain Category) We next explore a compound domain. See Fig. 19 on the next page. 245

When observing the very essence of the Fleet domain, at the {⟨Δ, N,Hs,H⟩, ⟨Δ, N,Ls,L⟩} level one observes:

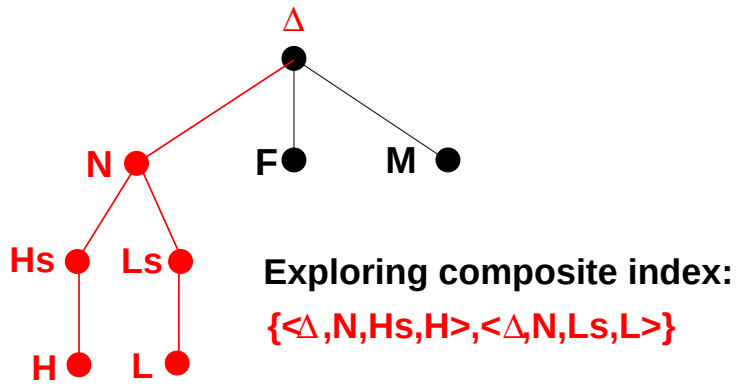

Figure 19: Exploring composite index $\{\langle \Delta, \mathsf{N}, \mathsf{Hs}, \mathsf{H}\rangle, \langle \mathsf{N}, \mathsf{Ls}, \mathsf{L}\rangle\}$

type
  HΣ = (LI×LI)-**set**, HΩ = HΣ-**set**,
  LΣ = (HI×HI)-**set**, LΩ = LΣ-**set**
value
  attr_HΣ: H → HΣ, attr_HΩ: H → HΩ
  attr_LΣ: L → LΣ, attr_LΩ: L → LΩ
  mereo_L: L → HI-**set axiom** ∀ l:L:**card** mereo_L(l)=2
  mereo_H: H → LI-**set** (= LI-**set**)
  remove_H: HI → N ~→ N
  insert_L: L → N ~→ N
  remove_L: LI → N ~→ N
  ...
axiom
  ∀ hσ:HΣ ..., ∀ hω:HΩ ...
  ∀ lσ:HΣ ..., ∀ lω:HΩ ...

•

5.1.8 Discussion 246

The previous examples (47–52), especially the last one (52), illustrates the complexity of a domain category; from just observing sub-part types and attributes (as in Examples 47–49), Example 52 observations grow to intricate mereologies etcetera. The 'discoverers' that we shall propose aim at structuring the discovery process by focusing, in turn, on part sorts, concrete part types, unique identifier types of parts, part mereology, part attributes, action signatures, event signatures, etc.

5.2 Proposed Type and Signature 'Discoverers' 247

By a 'domain discoverer' we shall understand a tool and a set of principles and techniques for using this tool in the discovery of the entities of a domain.

In this section we shall put forward a set of type and signature discoverers. Each discoverer is indexed by a simple or a compound domain index. And each discoverer is dedicated to some aspect of some entities. Together the proposed discoverers should cover the most salient aspects of domains. Our presentation of type and signature discoverer does not claim to help analyse "all" of a domain. 248

We need formally define what an index is.

```
type
    Index = Smpl_Idx | Cmpd_Idx
    Smpl_Idx = {| ⟨Δ⟩^idx | idx:Type_Name* |}
    Cmpd_Idx′ = Smpl_Idx-set
    Cmpd_Idx = {| sis:Cmpd_Idx′ • wf_Cmpd_Idx(sis) |}
value
    wf_Cmpd_Idx: Cmpd_Idx′ → Bool
    wf_Cmpd_Idx(sis) ≡ ∀ si,si′:Smpl_Idx • {si,si′}⊆sis ∧ si≠si′

    DISCOVERER_KIND: Index → Text
    DISCOVER_KIND(ℓ^⟨t⟩) as text
        pre: ℓ^⟨t⟩ is a valid index beginning with Δ
        post: text is some, in our case, RSL text
```

The idea of the $\ell^\frown\langle t\rangle$ index is that it identifies a sub-domain, t, of Δ where $\mathcal{DISCOVERER\_KIND}$ is any of the several different "kinds" of domain forms: 249

[90 (Page 76)] PART_SORTS,

[91 (Page 77)] HAS_A_CONCRETE_TYPE,

[92 (Page 78)] PART_TYPES,

[93 (Page 79)] UNIQUE_ID,

[96 (Page 79)] MEREOLOGY,

[98 (Page 80)] ATTRIBUTES,

[100 (Page 82)] ACTION_SIGNATURES,

[101 (Page 83)] EVENT_SIGNATURES and

[103 (Page 84)] BEHAVIOUR_SIGNATURES.

In a domain analysis (i.e., discovery) the domain description emerges "bit-by-bit". Initially types are discovered and hence texts which define unique identifier types and functions, mereology types and functions, and attribute types and functions. Then the signatures of actions, events and behaviours.

You may consider these "piece-wise" texts as being "added" to a (hence) growing reservoir of (RSL) texts with this reservoir being continually inspected by the domain analyser.

5.2.1 Analysing Domain Parts 250

The two most important aspects of an algebra are those of its parts and its operations. Rather than identifying, that is, discovering or analysing individual parts we focus on discovering their types — initially by defining these as sorts. And rather than focusing on defining what the operations achieve we concentrate on the signature, i.e., the types of the 251 operations.

It (therefore) seems wise to start with the discovery of parts, and hence of their types. Part types are present in the signatures of all actions, events and behaviours. When observing part types we also observe a variety of part type analysers: possible unique identities of parts, the possible mereologies of composite parts, and the types of the attributes of these parts.

252

Domain Part Sorts and Their Observers Initially we "discover" parts — by deciding upon their types, in the form, first of sorts, subsequently and possibly in the form of concrete types.

253 **A Domain Sort Discoverer:**

90. A part type discoverer applies to a simply indexed domain, *index*, and yields
    a a set of type names
    b each paired with a part (sort) observer.

value
90. $\mathbb{PART\_SORTS}$: Index $\xrightarrow{\sim}$ **Text**
90. $\mathbb{PART\_SORTS}(\ell^\frown\langle \mathrm{T}\rangle)$:
90a. tns:{T_1,T_2,...,T_m}:TN-**set** ×
90b. { obs_T_j: T → T_j | T_j:tns}

254

Example 53 (Some Part Sort Discoveries) We apply a concrete version of the above sort discoverer to the road-pricing transport domain Δ:

```
PART_SORTS(⟨Δ⟩):
    type
        N, F, M
    value
        obs_N: Δ → N
        obs_F: Δ → F
```

```
        obs_M: Δ → M

PART_SORTS(⟨Δ,N⟩):
    type
        Hs, Ls
    value
        obs_Hs: N → Hs
        obs_Ls: N → Ls


PART_SORTS(⟨Δ,F⟩):
    type
        Vs
    value
        obs_Cs: F → Vs
```

•

255

Domain Part Types and Their Observers

Do a Sort Have a Concrete Type ? Sometimes we find it expedient to endow a "discovered" sort with a concrete type expression, that is, "turn" a sort definition into a concrete type definition.

91. Thus we introduce the "discoverer":

```
91  HAS_A_CONCRETE_TYPE: Index → Bool
91  HAS_A_CONCRETE_TYPE(ℓ^⟨t⟩):true|false
```

256

Example 54 (Some Type Definition Discoveries) We exemplify two true expressions:

```
HAS_A_CONCRETE_TYPE(⟨Δ,N,Hs⟩)
HAS_A_CONCRETE_TYPE(⟨Δ,N,Ls⟩)
~ HAS_A_CONCRETE_TYPE(⟨Δ,N,Hs,H⟩)
~ HAS_A_CONCRETE_TYPE(⟨Δ,N,Ls,L⟩)
```

•

A Domain Part Type Observer: The PART_TYPES(ℓ⌢⟨t⟩) invocation yields one or more sort definitions of part types together with their observer functions. The domain analyser can decide that some parts can be immediately analysed into concrete types. Thus, together with yielding a type name, the PART_TYPES can be expected to yield also a type definition, that is, a type expression (paired with the type name). Not all type expressions make sense. We suggest that only some make sense. 257 258

92. The PART_TYPES discoverer applies to a composite type, t, and yields
    a. a type definition, T = TE,
    b. together with the sort and/or type definitions of so far undefined type names of TE.
    c. The PART_TYPES discoverer is not defined if the designated sort is judged to not warrant a concrete type definition.

92. PART_TYPES: Index $\xrightarrow{\sim}$ **Text**
92. PART_TYPES(ℓ⌢⟨t⟩):
92a. **type** t = te,
92b. T_1 or T_1 = TE_1
92b. T_2 or T_2 = TE_2
92b. ...
92b. T_n or T_n = TE_n
92c. **pre**: HAS_A_CONCRETE_TYPE(ℓ⌢⟨t⟩)

259

Example 55 (Some Part Type Discoveries) We exemplify two discoveries:

```
PART_TYPES(⟨Δ,N,Hs⟩):
    type
        H
        Hs = H-set

PART_TYPES(⟨Δ,N,Ls⟩):
    type
        L
        Ls = L-set

PART_TYPES(⟨Δ,F⟩):
    type
        V
        Vs = V-set
```

•

260 **Concrete Part Types:** In Example 55 on the facing page we illustrated one kind of concrete part type: sets. Practice shows that sorts often can be analysed into sets. Other analyses of part sorts are Cartesians, list, and simple maps:

90b. te: tn1 × tn2 × ... × tnm
90b. te: tn*
90b. te: Token $\xrightarrow[m]{}$ tn

where tn's are part type – usually sort – names some of which may have already been defined, and where Token is some simple atomic (non-part) type.

261 **Part Type Analysers** There are three kinds of analysers: *unique identity* analysers, *mereology* analysers and *general attribute* analysers. and

262 **Unique Identity Analysers:** We associate with every part type T, a unique identity type TI.

93. So, for every part type T we postulate a unique identity analyser function uid_TI.

```
value
93. UNIQUE_ID: Index → Text
93. UNIQUE_ID(ℓ^⟨T⟩):
93.     type
93.         TI
93.     value
93.         uid_TI: T → TI
```

263 **Mereology Analysers:** We remind the reader of Sects. 3.1.6 on page 32. Given a part, p, of type T, the mereology, MEREOLOGY, of that part is the set of all the unique identifiers of the other parts to which part p is partship-related as "revealed" by the mereo_TI$_i$ functions applied to p.

94. Let types T_1, T_2, ..., T_n be the types of all parts of a domain.
95. Let types TI_1, TI_2, ..., TI_n[29], be the types of the unique identifiers of all parts of that domain.
96. The mereology analyser MEREOLOGY is a generic function which applies to an index and yields the set of

    a zero,

[29] We here assume that all parts have unique identifications.

b one or

c more

mereology observers.

264

type
94. T = T_1 | T_2 | ... | T_n
95. T_{idx} = TI_1 | TI_2 | ... | TI_n
96. MEREOLOGY: Index → **Text**
96. MEREOLOGY($\{\ell_i^\frown\langle T_j\rangle,...,\ell_k^\frown\langle T_l\rangle\}$):
96a. **either:** {}
96b. **or:** mereo_TI_x: T → (TI_x|TI_x-**set**)
96c. **or:** { mereo_TI_x: T → (TI_x|TI_x-**set**),
96c. mereo_TI_y: T → (TI_y|TI_y-**set**),
96c. ...,
96c. mereo_TI_z: T → (TI_z|TI_z-**set**) }

where none of TI_x, TI_y, ..., TI_z are equal to TI and each is some T_{idx}.

265 **General Attribute Analysers:** A general attribute analyser analyses parts beyond their unique identities and possible mereologies.

97. Part attributes have names. We consider these names to also abstractly name the corresponding attribute types, that is, the names function both as attribute names and sort names. Finally we allow attributes of two or more otherwise distinct part types to be the same.

98. ATTRIBUTES applies to parts of any part type t and yields

99. the set of attribute observer functions attr_at, one for each attribute sort at of t.

266

type
97. AT = AT_1 | AT_2 | ... | AT_n
value
98. ATTRIBUTES: Index → **Text**
98. ATTRIBUTES($\ell^\frown\langle T\rangle$):
99. **type**
99. AT_1, AT_2, ..., AT_m
99. **value**
99. attr_AT_1: T → AT_1
99. attr_AT_2: T → AT_2
99. ...,
99. attr_AT_m: T → AT_m, m≤n

267

Example 56 (Example Part Attributes) We exemplify attributes of composite and of atomic parts:

```
ATTRIBUTES(⟨Δ⟩):
    type
        Domain_Name, ...
    value
        attr_Name: Δ → Domain_Name
        ...
```

where Domain_Name could include *State Roads* or *Rail Net*. etcetera. 268

```
ATTRIBUTES(⟨Δ,N⟩):
    type
        Sub_Domain_Location, Sub_Domain_Owner, Kms, ...
    value
        attr_Location: N → Sub_Domain_Location
        attr_Owner: N → Sub−Domain_Owner
        attr_Length: N → Kms
        ...
```

where Sub_Domain_Location could include *Denmark*, Sub_Domain_Owner could include *The Danish Road Directorate*[30], respectively *BaneDanmark*[31], etcetera. 269

```
ATTRIBUTES(⟨Δ,N,Hs,L⟩):
    type
        LOC, LEN, ...
    value
        attr_LOC: L → LOC
        attr_LEN: L → LEN
        ...


ATTRIBUTES({⟨Δ,N,Hs,L⟩.⟨Δ,N,Hs,H⟩}):
    type
        LΣ = HI-set, LΩ − LΣ-set
        HΣ = LI-set, HΩ − HΣ-set
    value
        attr_LΣ: L → LΣ
```

[30] http://www.vejdirektoratet.dk/roaddirectorate.asp?page=dept&objno=1024
[31] http://uk.bane.dk/default_eng.asp?artikelID=931

attr_LΩ: L → LΩ
attr_HΣ: H → HΣ
attr_HΩ: H → HΩ

where LOC might reveal some Bezier curve[32] representation of the possibly curved three dimensional location of the link in question, LEN might designate length in meters, LΣ [HΣ] designates the state of the link [hub], LΩ [HΩ] designates the space of all allowed states of the link [hub], etcetera.

•

270 — ***Attribute Sort Exploration:*** Once the attribute sorts of a part type have been determined there remains to be "discovered" the concrete types of these sorts. We omit treatment of this point in the present version of these these research notes.

5.2.2 Discovering Action Signatures 271

General We really should discover actions, but actually analyse function definitions. And we focus, in these research notes, on just "discovering" the function signatures of these actions. By a function signature, to repeat, we understand a functions name, say fct, and a function type expression (te), say dte$\xrightarrow{\sim}$rte where dte defines the type of the function's definition set and rte defines the type of the function's image, or range set.

272 **Function Signatures Usually Depend on Compound Domains** We use the term 'functions' to cover actions, events and behaviours.

We shall in general find that the signatures of actions, events and behaviours depend on types of more than one domain. Hence the schematic index set $\{\ell_1\hat{}\langle \mathsf{t}_1\rangle,\ell_2\hat{}\langle \mathsf{t}_2\rangle,...,\ell_n\hat{}\langle \mathsf{t}_n\rangle\}$ is used in all actions, events and behaviours discoverers.

273 **The ACTION_SIGNATURES Discoverer**

100. The ACTION_SIGNATURES meta-function applies to an index set and yields
   a a set of action signatures each consisting of an action name and a pair of definition set and range type expressions where
   b the type names that occur in these type expressions are defined by in the domains indexed by the index set.

100 ACTION_SIGNATURES: Index $\xrightarrow{\sim}$ **Text**
100 ACTION_SIGNATURES($\{\ell_1\hat{}\langle \mathrm{T}_1\rangle,\ell_2\hat{}\langle \mathrm{T}_2\rangle,...,\ell_n\hat{}\langle \mathrm{T}_n\rangle\}$):
100a act_fct$_i$: te$_{i_d}$ $\xrightarrow{\sim}$ te$_{i_r}$,
100a act_fct$_j$: te$_{j_d}$ $\xrightarrow{\sim}$ te$_{j_r}$,

[32] http://en.wikipedia.org/wiki/Bézier_curve

100a ... ,
100a act_fct$_k$: te$_{k_d} \xrightarrow{\sim}$ te$_{k_r}$
100b **where:**
100b type names in (te$_{(i|j|...|k)_d}$) and in (te$_{(i|j|...|k)_r}$) are
100b type names defined by the indices which are prefixes of
100b $\ell_m\frown\langle T_m\rangle$ and where T_m is in some signature $\mathsf{act\_fct}_{i|j|...|k}$.

5.2.3 Discovering Event Signature

274

Events are from the point of view of signatures very much like actions.

101. The EVENT_SIGNATURES meta-function applies to an index set and yields
   a a set of action signatures each consisting of an action name and a pair of definition set and range type expressions where
   b the type names that occur in these type expressions are defined either in the domains indexed by the index set or by the environment (i.e., "outside" the domain Δ).

275

101 EVENT_SIGNATURES: Index $\xrightarrow{\sim}$ **Text**
101 EVENT_SIGNATURES({$\ell_1\frown\langle$T$_1\rangle$,$\ell_2\frown\langle$T$_2\rangle$,...,$\ell_n\frown\langle$T$_n\rangle$}):
101a evt_fct$_i$: te$_{i_d} \xrightarrow{\sim}$ te$_{i_r}$,
101a evt_fct$_j$: te$_{j_d} \xrightarrow{\sim}$ te$_{j_r}$,
101a ... ,
101a evt_fct$_k$: te$_{k_d} \xrightarrow{\sim}$ te$_{k_r}$
101b **where:**
101b type names of $\mathsf{te}_{(i|j|...|k)_d}$ and $\mathsf{te}_{(i|j|...|k)_r}$ are type names
101b defined by the indices which are prefixes of $\ell_m\frown\langle t_m\rangle$
101b and where t_m is in some signature $\mathsf{act\_fct}_{i|j|...|k}$ or may
101b refer to types definable only "outside" Δ

5.2.4 Discovering Behaviour Signatures

276

We choose, in these research notes, to model behaviours in CSP[33]. This means that we model (synchronisation and) communication between behaviours by means of messages m of type M, CSP channels (**channel** ch:M) and CSP

output: ch!e [offer to deliver value of expression e on channel ch], and
input: ch? [offer to accept a value on channel ch].

[33] Other behaviour modelling languages are Petri Nets, MSCs: Message Sequence Charts, Statechart etc. We invite the reader to suggest corresponding 'discovery' techniques and tools.

We allow for the declaration of single channels as well as of one, two, ..., n dimensional arrays of channels with indexes ranging over channel index types

type Idx, CIdx, RIdx ...:
channel ch:M, { ch_v[vi]:M′|vi:Idx }, { ch_m[ci,ri]:M″|ci:CIdx,ri:RIdx }, ...

277 etcetera. We assume some familiarity with RSL/CSP.

A behaviour usually involves two or more distinct sub-domains.

Example 57 (The Involved Subdomains of a Vehicle Behaviour) Let us illustrate that behaviours usually involve two or more distinct sub-domains. A vehicle behaviour, for example, involves the vehicle subdomain, the hub subdomain (as vehicles pass through hubs), the link subdomain (as vehicles pass along links) and, for the road pricing system, also the monitor subdomain. •

278

102. The BEHAVIOUR_SIGNATURES is a meta function.

103. It applies to a set of indices and results in a text,

104. The text contains

    a a set of zero, one or more message types,

    b a set of zero, one or more channel index types,

    c a set of zero, one or more channel declarations,

    d a set of one or more process signatures with each signature containing a behaviour name, an argument type expression, a result type expression, usually just **Unit**, and

    e an input/output clause which refers to channels over which the signatured behaviour may interact with its environment.

279

103. BEHAVIOUR_SIGNATURES: Index $\xrightarrow{\sim}$ **Text**
103. BEHAVIOUR_SIGNATURES($\{\ell_1\hat{}\langle T_1\rangle,\ell_2\hat{}\langle T_2\rangle,...,\ell_n\hat{}\langle T_n\rangle\}$):
104a. **type** $M = M_1 \mid M_2 \mid ... \mid M_m$, $m \geq 0$
104b. $I = I_1 \mid I_2 \mid ... \mid I_n$, $n \geq 0$
104c. **channel** ch,vch[i],{vch[i]:M|i:I_a},{mch[j,k]:M|j:I_b,k:I_c},...
104d. **value**
104d. bhv_1: $ate_1 \rightarrow inout_1\ rte_1$,
104d. bhv_2: $ate_2 \rightarrow inout_2\ rte_2$,
104d. ... ,
104d. bhv_m: $ate_m \rightarrow inout_m\ rte_m$,
104d. **where** type expressions $atei_i$ and rte_i for all i involve at least two
104d. types t'_i and t''_j of respective indexes $\ell_i\hat{}\langle t_i\rangle$ and $\ell_j\hat{}\langle t_j\rangle$
104e. **where** $inout_i$: **in** k | **out** k | **in**,**out** k
104e. **where** k: ch | ch[i] | {ch[i]|i ∈ I_a} | {mch[j,k]:M|i:I_b,j:I_c} | ...

280

Example 58 (A Vehicle Behaviour Signature Discovery) We refer, for example, to Examples 36 (Pages 56–61) and 40 (Pages 66–67).

```
let ih=⟨Δ.N.LS,H⟩,il=⟨Δ.N.HS,L⟩,iv=⟨Δ,F,V⟩,im=⟨Δ,Monitor⟩ in
BEHAVIOUR_SIGNATURES({iv,ih,iv,im}) as text
    let n:N, hs=obs_HS(n), ls=obs_LS(n), vs=obs_F(PART_SORTS)(⟨Δ⟩) in
    where text:
        type
            VL_Msg, VH_Msg, VM_Msg
        channel
62a.    {vh_ch[ attr_VI(v),attr_HI(h) ]|v:V,h:H•v ∈ vd∧h ∈ hs}:VH_Msg
62b.    {vl_ch[ attr_VI(v),attr_LI(h) ]|v:V,l:L•v ∈ vs∧h ∈ ls}:VL_Msg
62c.    m_ch:VM_Msg
        value
64.     vehicle: VI → V → VP →
83.         out,in {vl_ch[ vi,li ]|li:LI•li ∈ xtr_LIs(ls)}
83.                {vh_ch[ vi,hi ]|hi:HI•hi ∈ xtr_HIs(hs)} out m_ch,... Unit
end end
```

•

5.3 What Does Application Mean ? 281

Now what does it actually mean "to apply" a discover function ? We repeat our list of discoverers.

[90 (Page 76)] PART_SORTS,

[91 (Page 77)] HAS_A_CONCRETE_TYPE,

[92 (Page 78)] PART_TYPES,

[93 (Page 79)] UNIQUE_ID,

[96 (Page 79)] MEREOLOGY,

[98 (Page 80)] ATTRIBUTES,

[100 (Page 82)] ACTION_SIGNATURES,

[101 (Page 83)] EVENT_SIGNATURES and

[103 (Page 84)] BEHAVIOUR_SIGNATURES.

It is the domain engineer cum scientist[34] who "issues" the "commands". The first "formal" domain inquiry is that of PART_SORTS(⟨Δ⟩). We refer to Item 90 on page 76, for example as captured by the formulas, Items 90–90b (Page 86). 282

For the domain engineer to 'issue' one of the 'discovery commands' means that that person has (i) prepared his mind to study the domain and is open to impressions, (ii) decided which *DISCOVERER_KIND* to focus on, and (iii) studied the "rules of engagement" of that command, that is which pre-requisite discoverers must first have been applied, with which index, that is, in which context the command invocation should be placed, and which results the invocation is generally expected to yield.

5.3.1 PART_SORTS 283

Let us review the PART_SORTS discoverer:

```
value
90.   PART_SORTS: Index ~→ Text
90.   PART_SORTS(ℓ^⟨T⟩):
90a.      tns:{T1,T2,...,Tm}:TN-set ×
90b.      { obs_Tj: T → Tj | Tj:tns}
```

The domain analyser[35] has decided to "position" the search at domain index $\ell^\frown\langle \mathsf{T}\rangle$ where $\mathsf{T} = \Delta$ if $\ell = \langle\rangle$ and where T is some "previously discovered part type. 284

From Item 90a the domain analyser is guided (i.e., advised) to analyse the domain "at position $\ell^\frown\langle \mathsf{T}\rangle$: is the domain type T a composite type of one or more subpart types ? If so then decide which they are, that is: $\mathsf{T}_1,\mathsf{T}_2,...,\mathsf{T}_m$, that is, the "generation" of the text **type** $\mathsf{T}_1,\mathsf{T}_2,...,\mathsf{T}_m$, if not then tns={} and no text is "generated".

Item 90b, and given the domain analyser's resolution of Item 90a, then directs the "generation" of m observers obs_T_j: $\mathsf{T} \to \mathsf{T}_j$ (for $j : \{1..m\}$).

5.3.2 HAS_A_CONCRETE_TYPE 285

Let us review the HAS_A_CONCRETE_TYPE analyser:

```
91   HAS_A_CONCRETE_TYPE: Index → Bool
91   HAS_A_CONCRETE_TYPE(ℓ^⟨T⟩):true|false
```

Item 91 directs the domain analyser to decide whether the domain type T at "position" $\ell^\frown\langle \mathrm{t}\rangle$ should be given a concrete type definition. It is a decision sôlely at the discretion of the domain analyser whether domain type T should be given a concrete type definition, and, as we shall see next, which concrete type it should then be "given", that is, how it should be "concretely abstractly" modelled.

5.3.3 PART_TYPES 286

Let us review the PART_TYPES analyser:

```
92.   PART_TYPES: Index ~→ Text
92.   PART_TYPES(ℓ^⟨t⟩):
92a.      type T = TE,
```

[34] When we write: *domain engineer cum scientist* we mean to say that the domain engineer really is performing a scientific inquiry.

[35] We use the alternative, synonymous terms: 'domain engineer', 'domain describer'. 'domain scientist'.

92b. T_1 or T_1 = TE_1
92b. T_2 or T_2 = TE_2
92b. ...
92b. T_n or T_n = TE_n
92c. **pre**: HAS_A_CONCRETE_TYPE($\ell^\frown\langle t\rangle$)

The domain analyser has decided to "position" the search at domain index $\ell^\frown\langle T\rangle$ where $T = \Delta$ if $\ell = \langle\rangle$ and where T is some "previously discovered part type. 287

From Item 92 the domain analyser is guided (i.e., advised) to analyse the domain "at position $\ell^\frown\langle T\rangle$: can a reasonably abstract, yet concrete type definion be given for T ? If so then decide which it should be, that is, should it be an atomic type a number type, **Intg, Rat, Real**, a Boolean type, **Bool**, or a token type, f.ex. TOKENA token type is a further undefined atomic type — typically used to model identifiers.; or should it be a composite type either a set type: TE: T_s**-set** of te: T_s**-infset**, or a Cartesian type: TE: $T_1 \times T_2 \times ... T_m$, or a list type: TE: $T_\ell^\star$ or te: T_ℓ^ω or a map type: TE: $T_d \xrightarrow[m]{} T_t$? In either case the text type T = TE, T_1 or T_1=TE_1, T_2 or T_1=TE_2, ..., T_n or T_n=TE_n is generated where TE (TE_x) is a type expression whose so far undefined type names T_1, T_2, ..., T_n must be defined, either as sorts, or a concrete types. 288

5.3.4 UNIQUE_ID 289

Let us review the UNIQUE_ID analyser:

```
value
93.  UNIQUE_ID: Index → Text
93.a UNIQUE_ID(ℓ^⟨T⟩):
93.b     type
93.c         TI
93.d     value
93.e         uid_TI: T → TI
```

Item 93.a inquires as to the Line 93.b **type** name Line 93.c of the inquired part type's unique identifiers Line 93.d and the function signature **value** Line 93.e of the observer, uid_TI, name, the definition set type (T, of course) and the range set type (TI — obviously). Thus, the only real "new" "discovery" here is the name, TI, of the unique identifier type. 290

Etcetera, etcetera.

5.4 Discussion

We have presented a set of discoverers:

[90 (Page 76)] PART_SORTS,

[91 (Page 77)] HAS_A_CONCRETE_TYPE,

[92 (Page 78)] PART_TYPES,

[93 (Page 79)] UNIQUE_ID,

[96 (Page 79)] MEREOLOGY,

[98 (Page 80)] ATTRIBUTES,

[100 (Page 82)] ACTION_SIGNATURES,

[101 (Page 83)] EVENT_SIGNATURES and

[103 (Page 84)] BEHAVIOUR_SIGNATURES.

291

There is much more to be said: About a meta-state component in which is kept the "text" so far generated. A component from which one can see which indices and hence which type names have so far been "generated", and on the basis of which one can perform tests of well-formedness of generated text, etcetera, etcetera,

6 Conclusion 292