4 Describing Domain Entities 111

4.1 On Describing

The purpose of description is to use for example informal text to present an entity (simple, action, event or behaviour) so that the reader may "picture" ("envisage"), that which is being described. The text describing the entity is said to be a syntactic quantity. and the entity is then said to be a semantic quantity: the syntactic text *denotes* the semantic quantity. We also say that the syntactic quantity designates, denotes, indicates, specifies, 112
points out, gives a name or title to, or characterises[18] the semantic quantity.

4.1.1 Informal Descriptions

In the many examples[19] of Sects. 2–3 we have made several references to quite a few domain entities. We do not claim that we have described these entities.

113

Domain Instances Versus Domains What we can observe are instances of a specific domain or fragments (perhaps parts) of a specific domain. What we describe are either abstractions of these instances or abstractions of a set (i.e., a type) of these instance. If 114
someone describes me as an atomic part with the action(s) and event(s) of my behaviour, then that someone describes an instance of a person, not the domain of all persons, but in that description it is expected that *many fragments* of the description is also valid for either a lot of persons or all persons. We say that these *many fragments* describe not an instances but fragments of abstractions of a domain of persons.

115

Non-uniqueness of Domain Descriptions We say 'a domain', not 'the domain'. Two or more domain describers may not exactly focus on the same entities and their properties. A domain description is always an abstraction. Something is left out. Not all entities and not all properties of those entities included may be deemed worthwhile to be included.

A good domain description, to us, is a domain description that covers what most stake holders can agree on to be relevamt aspects of the domain, that reveals generally unknown facets of the domain, and that is terse and precise.

116

A Criterion for Description For us, to informally describe an entity ideally means the following: *Let there be given what we can agree on to be an entity, call it e. Let there be given what is claimed to be a description of that entity. Let a person read and claim to have understood that description. Now that person is confronted with some phenomenon e'. Either that phenomenon is the same or it is of the same kind (type) as e or it is not. If e' is of the same kind as e then the person must identify it as such, unequivocally. If e' is not of the same kind as e then the person must identify it as not being so, likewise unequivocally.* 117

[18]— eight alternative terms for the same idea!
[19]Examples 5–23

If a description does not satisfy the above then it is not a *proper description*.

The above "criterion" suffers, seriously, from our not having made precise what we mean by *"same"* and *"same kind"*.

These notes are not the place for a much needed investigation of the "sameness" problem. It is basically a philosophical question. But we should not overlook the fact that it is the domain describer and the domain stake holders who, finally, decide on "sameness".

118

Reason for 'Description' Failure There can be three reasons for a description to not be proper:

1. *either all phenomena are entities as described — the description is vacuous;*
2. *or there are entities which were meant to be of the type or not meant to be of the type described but which "fall outside", respectively "fall inside" the description;*
3. *or the description does not make sense, is "gibberish", ambiguous, or otherwise.*

That is: a proper description, when applied to entities, "divides" their world into two non-empty and disjoint sets: the set of all entities being described by the description, and the rest !

119

Failure of Description Language But we have a problem ! One cannot give a precise definition of exactly the *denoting language*, that is, of exactly, all and only those informal texts which designate entities. Firstly, we have not given a sufficiently precise informal text characterisation of entities, Secondly, natural (cum national) languages, like English, defy such characterisations. We must do our best with informal language descriptions.

120

Guidance But there is help to be gotten! The whole purpose of Sect.3 was to establish the pointers, i.e., guidelines, as to what must be described, generally: parts, actions, events and behaviours, and specifically: whether atomic or composite parts, their attributes, and, optionally, their mereology, and, for composite parts, their subparts; and, as a starter, the signatures of actions, events and behaviours. This section will continue the line reviewed just above and provide further hints, pointers, guidelines.

4.1.2 Formal Descriptions 121

We shall, in addition to the *description components*[20], outlined in Sect. 3 now join the possibility of improved description precision through the use of formal description. We argue that formal description, while being used in-separately with precise informal narrative. improves precision while enabling formal proofs of properties of that which is denoted by the description. 122

[20]parts, actions, events and behaviours; attributes and possibly unique identifiers of parts, and mereology of composite (atomic) parts; subparts of composite parts; etc.

We shall here use the term 'formal' in the sense of mathematics. A formal description language is here defined to have a formal syntax, that is, a set of syntax rules which define precisely and unambiguously, which texts over the alphabet of the language are indeed sentences of that language[21], a formal semantics, , that is, something which to every syntactically valid sentence of the language, ascribes a meaning in terms of a mathematical quantity[22], and a proof system, that is, a consistent and relative complete set of axioms and proof rules using which one can prove properties of descriptions.

We shall "unravel" an example formal description language, `FDL`, in this section. `FDL` has similarities to the `RAISE` [22] Specification Language, `RSL` [21], but, as our informal explanation of the meaning of `FDL` will show, it is not `RSL`. The similarities are "purely" syntactical.

4.2 A Formal Description Language 123

4.2.1 Observing and Describing Entities

We make the obvious distinction between observing semantic values but expressing syntactic structures. We observe parts, but first express types and then their properties; actions, but first express their signatures and then their definitions; events, but first express their signatures and then their definitions; and behaviours, but first express their signatures and then their definitions.

4.2.2 Observing and Describing Parts 124

In order to describe a part we use such phrases as: *a patient* (whom we here consider to be an atomic part)) is characterised by as set of properties a `name.` a `central personal registration identifier,` a `gender,` a `birth date,` a `birth place,` a `nationality,` a `weight,` a `height,` a `insurance policy,` a `medical record,` etcetera; and *a transport* 125 *net* (whom we here consider to be a composite part)) is characterised by a set of properties a structure of, in this case two subparts, ie., a set of hubs and a set of links, and their mereology. Thus we take the nouns *name, central personal registration identifier, gender, birth date, birth place, nationality, weight, height, insurance policy, medical record, . . . , set of hubs,* and *set of links* as type names. The names 'patient' and 'transport net' are also domain names. That is, we go from instance of part to the type of all parts "of the 126 same kind". One must take great care in not confusing the two: type and value). Later we shall clarify the distinction between type and domain names.

127

Abstract Types By an *abstract type* we generally mean some further unexplained set of mathematical quantities. Abstract types are in contrast to concrete types by which

[21]that is, the alphabet and sentences can be considered mathematical quantities

[22]a set, a Cartesian, a list, a function, or some such mathematical item which can be characterised by a number of properties

we mean such mathematical quantities as sets, Cartesians, lists, maps and functions (in general). Abstract types are also referred to as *sorts*.

The `FDL` clause:

type A

defines what we shall here, simplifying, take as a set of values said to be of type A. A is said to be a *type name* (here, more specifically, a *sort name*).

128

Concrete Types Borrowing from `RSL`, and, in general, discrete mathematics, we introduce `FDL` clauses for expressing set, Cartesian, list, map and function types. Let A, B, ..., C be (type or sort) names which denote some (not necessarily distinct) types, then

A-**set**, $(A \times B \times ... \times C)$, A^*, A^ω, $A \xrightarrow[m]{} B$, $A \rightarrow B$, $A \overset{\sim}{\rightarrow} B$

are *type expressions* which denote the following (left-to-right) concrete types: set of sets of type A values; sets of Cartesian values, (a,b,...,c), over types A, B, ..., C; set of finite lists of elements of type A values; set of possibly infinite lists of elements of type A values; set of maps, that is enumerable functions from type A into type B values; set of total functions from type A into type B values; respectively set of partial functions from
129 type A into type B values. Choosing to describe a part as a sort rather than a concrete type reflects a *principle* of abstraction. Modelling a concrete type in terms of, for example, a map type ($A \xrightarrow[m]{} B$) rather than as type of indexed sets ((A×B)-**set**) reflects a modelling *technique*.

130

Type Definitions Besides the sort type definitions, e.g., **type** A there are the concrete type definitions.

Let D be some (unused)type name, then

type D = Type_Expression

is a concrete type definition where Type_Expression is of either of the forms A-**set**, A-**infset**, (A×B×...×C), A^*, A^ω, $A \xrightarrow[m]{} B$, $A \rightarrow B$, $A \overset{\sim}{\rightarrow} B$ and A|B|...|C, where A, B, ..., C are either type names or, more generally, other such type expressions and where A|B|...|C expresses
131 the "union" type of the A. B. ..., and C types.

Example 24 (Transport Net Types) Let us exemplify the above by starting a series of examples all focused on a domain of transport nets.
132 Figure 9 on the facing page shows a net with eight hubs and seven links.

To be able — here, in this tect — to refer to fragments (here sub-parts), of what is shown in Fig. 9, we label the parts with names (Fig. 10); these names stand for the designated parts. They are not properties of the parts, they are the parts. Also: they are
133 not the unique indentifiers of the parts.

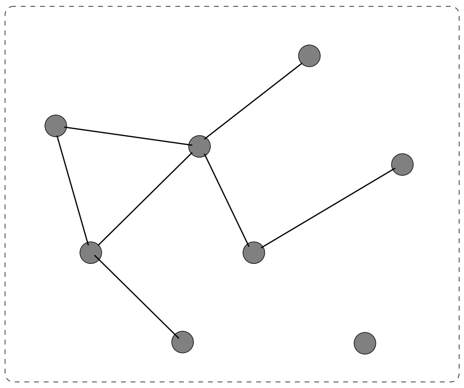
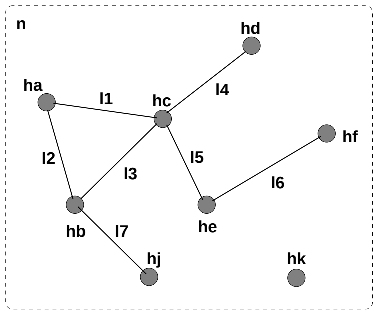


Figure 9: A transport net



obs_Hs(n) = {ha,hb,hc,hd,he,hf,hg,hj,hk}

obs_Ls(n) = {l1,l2,l3,l4,l5,l6,l7,l8,l9}

Figure 10: A transport net

32. We focus on the transport net domain. That domain is "dominated" by the composite parts of nets, *n:N*.

32. **type** N

33. There are two subparts of nets:

    a sets, hs:HS, of hubs (seen as one part) and

    b sets, ls:LS, of links (also seen as one, but another part).

33a. HS   33b. LS

As part of identifying the composite net type, N, we also identify two observers: obs_HS (observe [set of] hubs) and obs_LS (observe [set of] links) That is: 134

33a. obs_HS: N → HS   33b. obs_LS: N → LS

34. Hubs subparts of HS ('sets of hubs'), and
35. links are subparts of respectively LS ('sets of links'),

    and are of types

    a H, respectively
    b L.

type
35a. H
35b. L

135 We may, for convenience, bypassing a step (i.e., Items 33a–33b) instead express:

type	**value**
35a. Hs = H-**set**	35a. obs_Hs: N → H-**set**
35b. Ls = L-**set**	35b. obs_Ls: N → L-**set**

•

136 **Type Properties** In the following we shall be introducing a number of functions which analyse parts with respect to respective *properties* [17, 33, 20]. There are three kinds of properties of interest to us: the subparts of composite parts, the mereology of composite parts, and general attributes of parts (apart from their possible subparts and mereology). 137 Every entity, whether simple, or an action, or an event, or a behaviour, has a unique identification. The mere existence, in time and space, endows a part with a unique identification as follows. No two spatial parts can occupy overlapping space, so some abstract spatial location is a form of unique identification. We consider the unique identity of a part of type A, say AI, as a general attribute. We use the attribute values of AI to formulate 138 mereologies.

Thus there are three kinds of property analysis functions.

- *Subpart observer functions*, obs_B, obs_C, ..., obs_D, which apply to composite parts (say of type A), and yield their constituent subparts, say of type B, C, ..., D:

  obs_B: A → B, obs_C: A → C, ..., obs_D: A → D;

- *Mereology functions* which apply to composite parts (say of type A), and yields elements of their mereologies.

  Let parts of type B, C, ..., D be in some mereology relation to parts of type A. That is, there are mereology functions mereo_Ax: B → AIx, mereo_Ay: C → AIy, ..., mereo_Az: D → AIz

where Ax, Ay and Az are some distinct identifiers and AIx, AIy and Ayz are some type expressions over type A, typically

AI, AI-**set**, (AI×...×AI), etc.

139

- *Attribute Functions* which apply to parts (say of type A) and yield their attributes (short of the mereologies) (say of types E, F, ..., G:

  attr_E: A → E, attr_F: A → F, ..., attr_G: A → G;

  Among the general attribute functions are the unique identification functions, say attr_A.

140

Subpart Type Observers Given a composite sort, named, say, A, and postulating that its values contains subparts of type B, one can observe these type B subparts of A using the likewise postulated *observer function*:

obs_B: A → B

If A values also contain subparts of types C, ..., E, then there also exists the additional observer functions: obs_C, ..., obs_E. We say that the observer functions are postulated. We postulate them. And we endow them with properties so that they "stand out" from one another. First examples of properties are given by the observer function signatures: from type A values observer function obs_B yields B values. Further properties may be expressed through axioms. 141

Example 25 (Subpart Type Observers) From nets we observe

36. sets of hubs and

37. sets of links.

in either of two ways:

```
value                          value
36.  obs_HS: N → HS            35a.  obs_Hs: N → H-set
37.  obs_LS: N → LS            35b.  obs_Ls: N → L-set
```

•

142

Unique Identifier Functions All parts have unique identifiers. This is a dogma. We may never need some (or any) of these unique part identifiers. But they are there nevertheless.

Example 26 (Unique Hub and Link Identifiers) From hubs and links we observe their unique

38. hub and

39. link

identifier attributes and their 'observers':

type
38. HI
39. LI

value
38. uid_HI: H → HI
39. uid_LI: L → LI

143 Figure 11 shows the labelling of links with unique link identifiers, and of hubs with unique hub identifiers. It also shows sample unique identifier observer functions.

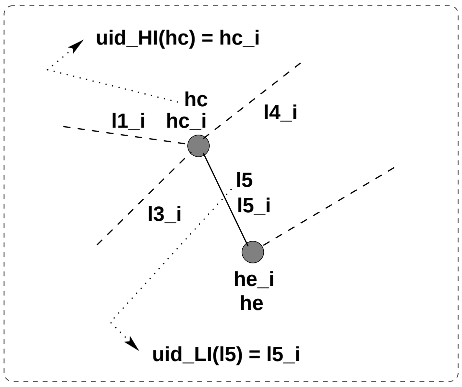


Figure 11: Fragment of a transport net emphasizing unique part identfiers and their observers

•

144

Mereologies and Their Functions

Example 27 (**Transport Net Mereology**) To express the mereology of transport nets we build on the unique identifications of hubs and links.

40. Links connect exactly two distinct hubs, mereo_HIs.

41. Hubs are connected to zero, one or more distinct links, mereo_LIs.

type
40. HIs = HI-**set**
axiom
40. ∀ his:HIs•**card** his=2
type
41. LIs = LI-**set**
value
40. mereo_L: L → HIs
41. mereo_H: H → LIs

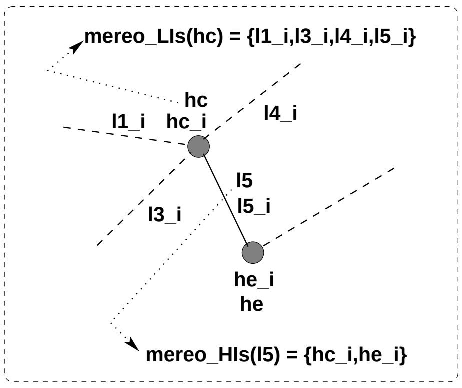

mereo_LIs(hc) = {l1_i,l3_i,l4_i,l5_i}
hc
l1_i
hc_i
l4_i
l5
l5_i
l3_i
he_i
he
mereo_HIs(l5) = {hc_i,he_i}

Figure 12: Fragment of a transport net emphasizing mereology observers

145
146 Figure 12 illustrates the idea of mereology observer. The above (Items 40–41) form the basis for expressing the constraints on how hubs and links are connected.

42. Given a net, its link and hub observers and the derived link and hub identifier extraction functions, the mereology of all nets must satisfy the following:

    a All link identifiers observed from hubs must be of links of that net and

    b All hub identifiers observed from links must be of hubs of that net.

43. We introduce two auxiliary functions for extracting all hub and link identifiers of a net.

147

```
axiom
42.  ∀ n:N,
42.     let ls=obs_LS(n),hs=obs_HS(n),
42.         lis=xtr_LIs(n),his=xtr_HIs(n) in
42a.    ∀ h:H•h ∈ hs ⇒ mereo_H(h)⊆lis ∧
42b.    ∀ l:L•l ∈ ls ⇒ mereo_L(l)⊆his end
value
43.  xtr_LIs: N → LI-set, xtr_HIs: N → HI-set
43.  xtr_LIs(n)≡{uid_LI(l)|l:L•l ∈ obs_LS(n)}
43.  xtr_HIs(n)≡{uid_HI(h)|h:H•h ∈ obs_HS(n)}
```

•

148

General Attributes and Their Functions

Example 28 (Hub States and Hub State Spaces) In addition to the unique identifiers and the mereology of parts there are the general attributes. An example are the states of hubs and links where these states indicate the direction of traffic for which the hubs and links are open. 149

44. With any hub, h, we thus associate
    - a a hub state, $h\sigma$, consisting of a set of pairs of link identifiers (with (li_j, li_k) in $h\sigma$ expressing that traffic is open from link l_j to link l_k via hub h), and
    - b a hub state space, $h\omega$, consisting of a set of hub states.
45. The relations between
    - a the link identifiers of the hub,
    - b the hub states, and
    - c the hub state spaces
46. must satisfy the following
    - a wrt. the potential set, hs, which is the "largest possible" hub state for h, one that allows traffic from any link l_i incident upon h to any link l_j emanent from h:
    - b the hub state is any subset of hs, and
    - c and such hub state is in that hub's state space.

150

```
type
44a.  HΣ = (LI×LI)-set
44b.  HΩ = HΣ-set
value
44a.  attr_HΣ: H → HΣ
44b.  attr_HΩ: H → HΩ
axiom
45.   ∀ n:N,h:H • h ∈ obs_Hs(n) ⇒
41.       let hlis = mereo_H(h),
44a.          hσ = attr_HΣ(h),
44b.          hω = attr_HΩ(h),
43.           lis = xtr_LIs(n) in
46a.      let hs = {(li,lj),(lk,li)|li:LI•li ∈ hlis∧{lj,lk}⊆lis} in
46b.      hσ⊆hs  ∧
46c.      hσ ∈ hω end end
```

•

151

Example 29 (Further Atomic Attributes) In addition to the unique link identifier links also have, for example, lengths, widths, possibly heights, geographic (spatial) locations, etc.

```
type
    LEN, WID, HEI, LOC, ...
value
    attr_LEN: L → LEN
    +: LEN × LEN → LEN
    >: LEN × LEN → Bool
    attr_WID: L → WID
    attr_HEI: L → HEI
    attr_LOC: L → LOC
    ...
```

●

4.2.3 Describing Actions 152

Function Names Actions potentially change states. Actions are here considered deliberate phenomena in that they are caused by willful applications (by agents within the domain being described) of functions having a specific, deliberate purpose, i.e., state change in mind. 153

Example 30 (Transport Net Action Names) Some examples are: *create* an *empty net* (no hubs, no links); *insert* a (new) hub; *insert* a (new) *link* (between a pair of distinct hubs of the net); *delete* (existing) *hub* (having no links into or out from it); and *delete* (existing) *link*.

●

154

Informal Function Descriptions The above examples just listed some actions by their function names. We did not describe these functions. We now do so, for two of these functions.

Example 31 (Informal Transport Net Action Descriptions) We detail two informal function descriptions. The *create empty net* function

47. applies to nothing and yields a net, n, of no hubs and no links.

The *insert link* function

48. applies to a net, n, of at least two distinct hubs, as identified, hi_j, hi_k, by the function application arguments, and a new link ℓ not in n, and yields a net, n′. 155

    a The inserted link l is to be connected to the two distinct hubs identified by hj_i and hk_i, and these designate hubs of the net.

b l is not in the original net.

c The mereology of l designate hj_i and hk_i.

d The state of these identified hubs do not allow any traffic.

e No new hubs are added, hence the set of hub identifiers of the net is unchanged.

f Only one link is added to the net, hence the set of net link identifiers is changed by only the addition of the l link identifier.

g Let the hubs identified by hj_i and hk_i be hj and hk, respectively, before the insertion,

h and by hj' and hk' after the insertion.

i Now the mereology contributions by the two changed hubs reflect only the addition of link l.

156 We leave the informal descriptions of

49. *delete_L*
50. *insert_H*
51. *delete_H*

to the reader.

•

157 **Formal Function Descriptions** We observe actions, but describe the functions which when applied amount to actions. There are two parts to describe a function: (i) the function signature, a:A→B: a distinct function name, say f, and a function type, A→B, that is, type of arguments A and type of results B, and (ii) the function definition, f(a,b)≡$\mathcal{C}$(a): a symbolic function invocation, f(a,b), and a definition body, $\mathcal{C}$(a). $\mathcal{C}$(a) is a clause, i.e., an expression in FDL, whose evaluation yields the function value. 158

There are other ways than:

```
value
    f: A → B
    f(a,b) ≡ $\mathcal{C}$(a)
```

in which to define a function. For example:

```
value
    f: A → B
    f(a) as b
        pre $\mathcal{P}$(a)
        post $\mathcal{Q}$(a,b)
```

159

Example 32 (Formal Transport Net Action Descriptions) The *create net* function:

```
value
47.  create_N: Unit → N
47.  create_N() as n
47.     post obs_HS(n)={} ∧ obs_LS(n)={}
```

160

The *insert link* function:

```
value
48.  insert_L: (HI × HI) × L → N → N
48.  insert_L((hij,hik),l)(n) as n′
48a.    pre  hji≠hki ∧ {hji,hki} ⊆ xtr_HIs(n) ∧
48b.          ∧ l ∉ obs_LS(n)
48c.          ∧ mereo_L(l) = {hji,hki}
48d.          ∧ attr_HΣ(get_H(hji))(n)={}=attr_HΣ(get_H(hki))(n)
48e.    post xtr_HIs(n) = xtr_HIs(n′)
48f.          ∧ xtr_LIs(n′) = xtr_LIs(n) ∪ {uid_LI(l)}
48g.          ∧ let hj=get_H(hij)(n), hk=get_H(hik)(n),
48h.                hj′=get_H(hij)(n′),vhk′=get_H(hik)(n′) in
48i.          ∧ mereo_H(hj′) = mereo_H(hj) ∪ {uid_LI(l)}
48i.          ∧ mereo_H(hk′) = mereo_H(hk) ∪ {uid_LI(l)} end
```

161

52. From the postulated observer and attribute functions one can define the auxiliary *get* function:

```
value
52.  get_H: HI → N ~→ H
52.  get_H(hi)(n) ≡
52.     let h:H•h ∈ obs_HS(n)∧uid_HI(h)=hi
52.     in h end
52.     pre hi ∈ xtr_HIs(n)
```

162

We do not narrate the informal description of "remaining" net actions (cf. Items 49– 51 on the preceding page), just their function signatures and pre-conditions.

```
49.  delete_L: LI → N ~→ N
49.     pre delete_L(li): li ∈ xtr_LIs(n)
50.  insert_H: H → N ~→ N
50.     pre insert_H(h): h ∉ obs_HS(n)∧mereo_H(h)={}∧mereo_Ω(h)={{}}
51.  delete_H: HI → N ~→ N
51.     pre delete_H(hi): hi ∈ xtr_HIs(n)∧mereo_H(get_H(hi))(n)={}
```

Given appropriate post-conditions the following theorems must be provable:

theorems:
  ∀ h:H • **pre**−conditions are satisfied ⇒ delete_H(uid_HI(h))(insert_H(h)(n))=n
  ∀ l:L • **pre**−conditions are satisfied ⇒ delete_L(uid_LI(l))(insert_L(l)(n))=n

•

163 **Agents** An *agent* is a behaviour which invokes functions, hence cause actions. So we simply "equate" agents with behaviours.

4.2.4 Describing Events 164

We observe events. But we describe logical properties characterising classes of "the same kind" of events.

Deliberate and Inadvertent (Internal and External) Events Events are like actions: somehow a function was applied either *deliberately* by an agent outside (that is, *external* to) the domain being described, or *inadvertently* by a behaviour of (that is, *internal* to) the domain, but for another purpose than captured by the event. 165

Example 33 (A Deliberate [External] Event) We narrate a simple external cause / "internal" effect example: When one or more bank customers default on their loans and declare themselves unable to honour these loans then the bank may go bankrupt.

•

166

Example 34 (An Inadvertent [Internal] Action Event) We narrate a simple internal cause / "internal" effect example: When a bank customer, the agent, withdraws monies from an account the balance of that account, if the withdrawal is completed, may go negative, or may go below the credit limit. In either case we say that, the withdrawal action, as was intended, succeeded, but that an *"exceeded credit limit"* event occurred.

•

167

Event Predicates Instead of describing events by directly characterising the deliberate external, respectively inadvertent internal actions we suggest to describe these events indirectly, by characterising the logical effects, say, in terms of predicates over *before/after* states. 168

Example 35 (Formalisation of An External Event) The event is that of a *"link segment disappearance"*.

53. Generally we can explain *"link segment disappearances"*, for example, as follows:

54. A l_i-identified link, l, between hubs hf and ht (identified in l) is *removed*.

55. Two hubs, *hf*″ and *ht*″, and two links, *lf* and *lt*, are *inserted* — where

   a hub values *hf* and *ht* (the hubs in the original net) become hub values *hf'* and *ht'* in the resulting net, that is, hub values *hf* and *ht* have same respective hub identifiers as *hf*′ and *ht*′,

   b hubs *hf*″ and *ht*″ are new,

   c links *l*′ and *l*″ are new,

   d link *lf* is *inserted* between *hf*′ and *hf*″, that is, link *lf* identifies hubs *hf*′ and *hf*″, and link *lt* is inserted between *ht*′ and *ht*″, that is, link *lt* identifies hubs *ht*′ and *ht*″, 169

   e hub *hf*′ is, in the resulting net, connected to the links hub *hf* was connected to in the original net “minus” link *l* but “plus” link *lf*,

   f hub *ht*′ is, in the resulting net, connected to the links it was connected to in the original net “minus” link *l* but “plus” link *lt*,

   g hub *hf*″ is connected only to link *lf*
   and hub *ht*″ is connected only to link *lt*,

   h the state space of *hf*′ suitably includes all the possibilities of entering link *lf*[23],

   i the state space of *ht*′ suitably includes all the possibilities of entering link *lt*[24],

   j the state spaces of *hf*″ and *ht*″ both contains just the empty set,

   k the states of *ht*″ and *ht*″ are both the empty set: “dead ends !”,

   l the sum of the lengths of links *lf* and *lt* is less than the length of link *l*, and

   m all other non-mereology attributes of *lf* and *lt* are the same as those of link *l*.

56. All other links and hubs are unchanged.

170

```
value
53.  link_segment_disappearance: N × N → Bool
53.  link_segment_disappearance(n,n′) ≡
53.     ∃ l:L, hf″,ht″:H, lf,lt:L •
54.         {l} = obs_LS(n) \ \obs_LS(n′)
55a.          ∧ let hfi=uid_HI(hf), hti=uid_HI(ht) in
55a.             let hf′=get_H(hfi)(n′), ht′=get_H(hti)(n′) in
55b.             {hf″,ht″}∩ obs_HS(n)={} ∧ {hf″,ht″}⊆obs_HS(n′)
55c.          ∧ {lf,lt}∩ obs_LS(n)={} ∧ {lf,lt}⊆obs_LS(n′)
55d.          ∧ mereo_L(l′)={hfi,uid_HI(hf″)} ∧ mereo_L(l″)={hti,uid_HI(ht″)}
55e.          ∧ mereo_H(hf)=mereo_H(hf′)\{uid_LI(l)}∪{uid_LI(lf)}
55f.         ∧ mereo_H(ht)=mereo_H(ht′)\{uid_LI(l)}∪{uid_LI(lt)}
```

[23] A substitution function replaces all link *l* identifiers with link *lf* identifiers.
[24] A substitution function replaces all link *l* identifiers with link *lt* identifiers.

55g. $\wedge$ mereo_H(hf$''$)={uid_LI(lf)} $\wedge$ mereo_H(ht$''$)={uid_LI(lt)}
55h. $\wedge$ attr_HΩ(hf$'$)=subst(uid_LI(l),uid_LI(lf),attr_HΩ(hf))
55i. $\wedge$ attr_HΩ(ht$'$)=subst(uid_LI(l),uid_LI(lt),attr_HΩ(ht))
55j. $\wedge$ attr_HΩ(hf$''$)={{}} $\wedge$ attr_HΩ(ht$''$)={{}}
55k. $\wedge$ attr_HΣ(hf$''$)={} $\wedge$ attr_HΣ(ht$''$)={}
55l. $\wedge$ attr_LEN(lf)+attr_LEN(t)<attr_LEN(l)
55m. $\wedge$ $\forall$ X:non_mereo_attributes(l)•attr_X(lf)=attr_X(lt)=attr_X(l)[25]
56. $\wedge$ obs_HS(n$'$)\{hf$'$,hf$''$,ht$'$,ht$''$} = obs_HS(n)\{hf,ht}
56. $\wedge$ obs_LS(n$'$)\{lf,lt} = obs_LS(n)\{l} **end end**

171 We can express a theorem relating the above to the *rem*ove and *ins*ert functions.

theorem: link_segment_disappearance(n,n$'$) $\Rightarrow$
  let l:L, hf$''$,ht$''$:H, lf,lt:L • [Lines 54–55, 55a–55m, 56] **in**
  n$'$ = ins_L(lt)(ins_L(lf)(ins_H(ht$''$)(ins_H(hf$''$)(rem_L(uid_LI(l))(n)))))
  end

•

4.2.5 Describing Behaviours 172

Behaviour Description Languages As for the description of parts, actions and events[26] there exists formal ways of describing behaviours as of sequences of actions, events and behaviours: some are "textual"[27]: `CSP` [25], some are "graphical", for example: `MSC` [Message Sequence Charts] [26], `Petri Nets` [36] and `State Charts` [24].

173 **Simple Sequential Behaviours:** A simple sequential behaviour is a sequence of actions and events.

— ***Snapshot Description of a Simple Sequential Behaviour:*** Snapshot of a behaviour, as it unfolds, could be described:

let σ' = action_1(arg_1)(σ) $\lceil\rceil$ event_1(σ)(σ') **in**
let σ'' = action_1(arg_1)(σ') $\lceil\rceil$ event_1(σ')(σ'') **in**
...
let $\sigma''...'$ = action_1(arg_1)($\sigma'...'$) $\lceil\rceil$ event_1(`sigm'...'a)($\sigma''...'$) **in**
$\sigma''...'$ **end** ... **end end**

[25]The predicate in Line 55m on the preceding page is to be explained.
[26] Part, action and event description languages were first mentioned in the 'Abstract' footnotes 1– 2 on page 4: `Alloy` [27], `CafeOBJ` [18], `Casl` [13] `Event B` [1], `Maude` [30, 12] `RAISE/RSL` [22, 21], `VDM` [7, 8, 16] and `Z` [38].
[27]The languages mentioned in Footnote 26 are textual.

where the internal non-deterministic operator, $\lceil\rceil$, expresses that either its left side or right operand is chosen. The seemingly recursive equation:

$$\textbf{let}\ \sigma' = \text{act}(\text{arg})(\sigma)\ \lceil\rceil\ \text{event}(\text{arg})(\sigma)(\sigma')$$

expresses a further non-determinism: any σ' satisfying the equation is a valid next state. We can name such simple sequential behaviours, for example: P.

Simple Concurrent Behaviours: A simple concurrent behaviour is a set of two or more simple sequential behaviours. 174

We describe a simple concurrent behaviour by a list of named behaviour descriptions separated by the parallel behaviour composition operator ‖, for example, P‖Q‖...‖R. A variant form is ‖{P(i)|i:Index•predicate(i)} which expresses 'distribution' of the behaviour composition operator (‖) over 'expanded' terms, that is, $\mathsf{P}(\mathsf{i}_j)\|\mathsf{P}(\mathsf{i}_k)\|\ldots\|\mathsf{P}(\mathsf{i}_\ell)$.

Communicating Behaviours: A communicating behaviour is a behaviour which expresses willingness to engage in a (synchronisation and) communication with another communicating behaviour or with the environment. 175

In order to express 'communication' (between behaviours) a notion of an output/input *channel* is introduced with behaviours allowed 'access' to channel (which are therefore shared). In CSP channels are typed with the type of the values that can be output on a channel "between" behaviours. In CSP output of a value (say of expression e) onto channel 176
ch is expressed by the statement ch!e whereas input of a value from channel ch is expressed by the expression ch?.

```
type
    M
channel
    ch:M
value
    S: Unit → Unit,               S() = P() ‖ Q()
    P: Unit → out ch  Unit,       P() ≡ ... ch!e ...
    Q: Unit → in ch Unit,         Q() ≡ ... ch? ...
```

We thus describe a communicating behaviour by allowing one or more clauses: statements of the kind ch!e and expressions of the kind ch?.

External Non-deterministic Behaviours: A behaviour, $\mathcal{P}$, composed from behaviours 177
$\mathcal{P}_i, \mathcal{P}_j, \ldots, \mathcal{P}_k$ is said to exhibit internal non-determinism if the behaviour is either as is behaviour $\mathcal{P}_i$, or as is behaviour $\mathcal{P}_j$, ..., or as is behaviour $\mathcal{P}_k$, and is influenced in being so by the environment of behaviour $\mathcal{P}$.

We describe such behaviours as follows: P: P_i ⌷ P_j ⌷ ... ⌷ P_k where P_i (etcetera) describes behaviour $\mathcal{P}_i$ (etcetera). A variant description of internal non-determinism is
⌷{P(i)|i:Index•predicate(i)}. 178

External influence is, for example, expressed if behaviour descriptions P_i (etcetera) contain either an output (ch!e) or an input (ch?) clause and the environment offers to accept input, respectively offers output "along" the name channel.

179 **Internal Non-deterministic Behaviours:** A behaviour, $\mathcal{P}$, composed (somehow) from behaviours $\mathcal{P}_i, \mathcal{P}_j, \ldots, \mathcal{P}_k$ is said to exhibit internal non-determinism if the behaviour is either as is behaviour $\mathcal{P}_i$, or as is behaviour $\mathcal{P}_j$, ..., or as is behaviour $\mathcal{P}_k$, and is not influenced in being so by the environment of behaviour $\mathcal{P}$.

We describe such behaviours as follows: P: P_i ⌈⌉ P_j ⌈⌉ ... ⌈⌉ P_k where P_i (etcetera) describes behaviour $\mathcal{P}_i$ (etcetera). A variant description of internal non-determinism is
180 ⌈⌉{P(i)|i:Index•predicate(i)}.

For internal non-determinism to work for expressions like the above we must assume that they do not contain such output (ch!e) or an input (ch?) clauses for which the environment may accept input, respectively offer output.

181 **General Communicating Behaviours:** A general communicating behaviour is a set of sequences of actions, events and (simple sequential, simple concurrent, communicating and non-deterministic) behaviours such that at least two separately identifiable behaviours
182 of a set share at least one channel and contain respective ch!e and ch? clauses.

Example 36 (A Road Pricing (Transport) System Behaviour) This example is quite extensive.

57. A road pricing (transport) system, Δ_{RPS} contains
    a a net n — as outlined in earlier examples — of hubs and links,
    b a fleet f of vehicles and
    c a central road pricing monitor m.
58. From Δ_{RPS} we can observe the
    a a net, n:N,
    b a fleet of vehicles, f:F, and
    c a road pricing monitor, m:M.
59. From the net, n:N, we observe
    a a set of hubs and
    b a set of links
60. From the fleet, f:F, we observe
    a a set of vehicles.

183

```
type
57.  Δ_RPS, N, F, M
value
57a.   obs_N: Δ_RPS → N
57b.   obs_F: Δ_RPS → F
57c.   obs_M: Δ_RPS → M
59a.   obs_Hs:  N → H-set
59b.   obs_Ls:  N → L-set
60a.   obs_Vs:  F → V-set
```

We need "prepare" some part names:

```
type
57a.     n:N                          59a.     hs:Hs=obs_Hs(n)
57b.     f:F                          59b.     ls:Ls=obs_Ls(n)
57c.     m:M                          60a.     vs:Vs=obs_VSs(f)
```

184

61. With the road pricing behaviour we associate separate behaviours,
    a one for the net which is seen as the parallel composition of
        i. a set of hub behaviours
        ii. a set of link behaviours;
    b one for the fleet of vehicles which is seen as the parallel composition of
        i. a set of vehicle behaviours;
    c and a central road pricing monitor behaviour.

185

```
61.      road_pricing_system: Unit → Unit
61.      road_pricing_system() ≡ net()‖fleet()‖monitor(...)

61a.     net() ≡
61(a)i.        ‖ {hub(uid_HI(h))(h)(vis)|h:H•h ∈ hs} ‖
61(a)ii.       ‖ {link(uid_LI(l))(l)(vis)|l:L•l ∈ ls}

61b.     fleet() ≡
61(b)i.        ‖ {vehicle(obs_VI(v))(v)(vp)|v:V•v ∈ vs}

61c.     monitor(...) ≡ ...
```

186

The *vis* arguments of the *hub* and *link* behaviours "carry" the identifiers of current *v*ehicles currently at the hub or on the link. The *vp* argument of the *vehicle* behaviour "carries" the current *v*ehicle *p*osition. The (...) argument of the *monitor* behaviour records the history status of all vehicles on the net. We omit details of how these arguments are initialised. 187

62. We associate channels as follows:
    a one for each pair of vehicles and hubs,
    b one for each pair of vehicles and links and
    c one for monitor (connected to vehicles).

```
62.  channel
62a.    {vh_ch[ uid_VI(v),uid_HI(h) ]|v:V,h:H•v ∈ vd∧h ∈ hs}:VH_Msg
62b.    {vl_ch[ uid_VI(v),uid_LI(h) ]|v:V,l:L•v ∈ vs∧h ∈ ls}:VL_Msg
62c.    mon:VM_Msg
```

188 We omit detailing the channel message types.

63. Vehicles are positioned
    a either on a link, in direction from one hub to a next, some fraction down that link,
    b or at a hub, in direction from one link to a next where
    c the fraction is a real between 0 and 1.

```
type
63.    VP = onL | atH
63a.   onL == mk_onL(li:LI,fhi:HI,f:FRA,thi:HI)
63b.   atH == mk_atH(hi:HI,fli:LI,tli:LI)
63c.   FRA = Real axiom ∀ fra:FRA•0≤fra≤1
```

189

64. The *vehicle* behaviour is modelled as a CSP process which communicates with hubs, links and the monitor.

65. The *vehicle* behaviour is a relation over its position.
    If on a link, at some position,
    a then the vehicle may "remain" at that position,
    b chosen so internally non-deterministically,
    c or, if the vehicle position is not "infinitesimally" close to the "next" hub,
    d then the vehicle will move further on along the link,
    e some small fraction δ,
    f else the vehicle moves into the next hub in direction of the link named *li′*
    g where *li′* is in the set of links connected to that hub —
    h while notifying the link, the hub and the monitor of its entering the link and entering the hub.

190

```
value
65e. $\delta$:**Real axiom** $0<\delta \ll 1$vehc6
64.  vehicle: VI $\rightarrow$ V $\rightarrow$ VP $\rightarrow$
64.        **out**,**in** {vl_ch[ vi,li ]|li:LI•li $\in$ xtr_LIs(ls)} **out** m_ch **Unit**
65.  vehicle(vi)(v)(vp:mk_onL(li,fhi,f,thi)) $\equiv$
65a.     vehicle(vi)(v)(vp)
65b.     $\sqcap$
65c.     **if** f + $\delta$<1
65d.        **then** vehicle(vi)(v)(mk_onL(li,fhi,f+$\delta$,thi))
65e.        **else let** li′:LI•li′ $\in$ mereo_H(get_H(thi)(n)) **in**
65g.              vh_ch[ vi,thi ]!enterH ‖ vl_ch[ vi,li ]!leaveL ‖ m_ch!leaveL_enterH(vi,li,thi);
65h.              vehicle(vi)(v)(mk_atH(thi,li,li′)) **end end**
```

191

66. If the vehicle is at a hub,
    a then the vehicle may “remain” at that same position,
    b chosen so internally non-deterministically,
    c or move on to the next link,
    d in direction of a next hub,
    e while notifying the hub and monitor of leaving the hub and the link and the monitor of entering the link.

```
66.  vehicle(vi)(v)(vp:mk_atH(hi,fli,tli)) $\equiv$
66a.    vehicle(vi)(v)(vp)
66b.    $\sqcap$
66d.    **let** {hi′,thi}=mereo_L(getL(tli)(n)) **in assert:** hi′=hi
66e.    vh_ch[ vi,hi ]!leaveH ‖ vl_ch[ vi,tli ]!enterL ‖ m_ch!leaveH_enterL(vi,hi,tli);
66c.    vehicle(vi)(v)(ml_onL(tli,hi,0,thi)) **end**
```

192

67. The monitor behaviour records the (dynamic) history of all vehicles on the net: alternating sequences of hub and link identifiers.

68. The monitor contains a price table which to every link and hub records the fee for moving along that link or hub.

```
type
67.  VW′ = VI $\overrightarrow{m}$ (HI|LI)$^*$
67.  VW = {|vw:VW′•wf_VW(vh)|}
68.  Fee, PT = (LI|HI) $\overrightarrow{m}$ Fee
```

value
67. wf_VW(vh) $\equiv$
67. $\forall$ hll:(HI|LI)* • hll $\in$ **rng** vh
67. $\forall$ i:**Nat**•{i,i+1}$\subseteq$**inds** hll $\Rightarrow$
67. is_HI(hll(i))$\wedge$is_LI(hll(i+1))$\vee$is_LI(hll(i))$\wedge$is_HI(hll(i+1))

193

69. The monitor behaviour non-deterministically externally alternates between
    a input of messages from vehicles
        i. either when entering a link in which case the vehicle history is updated with that link's identifier (for that vehicle),
        ii. or when entering a hub in which case the vehicle history is updated with that hub's identifier (for that vehicle).
    b and accepting inquiries and requests relating vehicle histories and fees (designated by (...) below).

194

value
69. monitor: PT $\rightarrow$ VH $\rightarrow$ **in** m_ch **Unit**
69. monitor(pt)(vh) $\equiv$
69a. (**case** m_ch? **of**
69(a)i. leaveH_enterL(vi,hi,li) $\rightarrow$ monitor(pt)(vh $\dagger$ [vi $\mapsto$ vh(vi)$^\frown\langle$li$\rangle$])
69(a)ii. leaveL_enterH(vi,li,hi) $\rightarrow$ monitor(pt)(vh $\dagger$ [vi $\mapsto$ vh(vi)$^\frown\langle$hi$\rangle$])
69a. **end**)
69b. $\lceil\rceil$ (...)

195 We omit description of other monitor actions (Line 69b).

70. Link behaviours maintain a state which records the set of vehicles "currently" on the link.
71. The link behaviour expresses willingness to
    a accept messages from vehicles
    b entering links in which case the *"vehicle vi on link"* state has *vi* added, or
    c leaving links in which case the *"vehicle vi on link"* state has *vi* removed,
    d where these vehicles range over all fleet vehicles.

196

type
70. VIS = VI-**set**
value
71. link: li:LI $\rightarrow$ L $\rightarrow$ VIS $\rightarrow$ **in** cl_Unitlink1

```
71.   link(li)(l)(vis) $\equiv$
71a.      $\lceil\rceil$ {let m = cl_vl[ vi,li$'$ ]? in assert: li$'$=li
71a.          case m of
71b.             enterL $\rightarrow$ link(li)(l)(vis $\cup$ {vi})
71c.             leaveL $\rightarrow$ link(li)(l)(vis $\setminus$ {vi})
71d.          end | vi:V•v $\in$ xtr_VIs(vs) end}
```

We leave it to the reader to suggest a *hub* behaviour description.

•