Towards a Theory of Domain Descriptions
— Bergen 8 May Mini-course Notes —

Dines Bjørner
DTU Informatics, Techn.Univ.of Denmark, DK–2800 Kgs.Lyngby
Fredsvej 11, DK-2840 Holte, Denmark
bjorner@gmail.com, www.imm.dtu.dk/~db

May 1, 2012: 16:34

Contents

Abstract

We seek foundations for a possible theory of domain descriptions. Sect. 2 informally outlines what we mean by a domain. Sect. 3 informally outlines the entities whose description form a description of a domain. Sect. 4 then suggests one way of formalising such description parts[1]. There are other ways of formally describing

[1]The exemplified description approach is model-oriented, specifically the `RAISE` [23] cum `RSL` [22] approach.

domains[2], but the one exemplified can be taken as generic for other description approaches. Sect. **??** outlines a theory of domain mereology. Sect. 5 suggests some 'domain discoverers'. 3

These research notes reflect our current thinking. Through seminar presentations, their preparation and post-seminar revisions it is expected that they will be altered and honed.

[2]Other model-oriented approaches are those of `Alloy` [28], `Event B` [1], `VDM` [7, 8, 17] and `Z` [42]. Property-oriented description approaches include `CafeOBJ` [19], `Casl` [13] and `Maude` [32, 12]

1 Introduction 4

In this section we shall cover a number of concepts ("Preliminary Notions" and "An Ontology of Descriptions", Sects. 1.2–1.4) that lie at the foundation of the theory and practice of domain science and engineering. These are general issues such as (i) software engineering as consisting of domain engineering, requirements engineering, and software design, (ii) types and values, and (iii) algebras. But first we shall put the concept of domain engineering in a proper perspective.

1.1 Rôles of Domain Engineering 5

By domain engineering we shall understand the engineering[3] of domain descriptions, their study, use and maintenance. In this section (Sect. 1.1) we shall focus on the use of domain descriptions (i) in the construction of requirements and, from these, in the design of software, and (ii) more generally, and independent of requirements engineering and software design, in the study of man-made domains in a search for possible laws.

1.1.1 Software Development 6

We see domain engineering as a first in a triptych phased software engineering: (I) domain engineering, (II) requirements engineering and (III) software design. Sections 3–4 cover some engineering aspects of domain engineering.

7 **Requirements Construction** As shown elsewhere [3, 4, 5, 6] domain descriptions, $\mathcal{D}$, can serve as a firm foundation for requirements engineering. This done is by systematically "deriving" major part of the requirements from the domain description. The 'derivation' is done in steps of refinements and extensions. Typical steps reflect such 'algebraic
8 operations' as projection, instantiation, determination, extension, fitting, etcetera In "injecting" a domain description, $\mathcal{D}$, in a requirements prescription, $\mathcal{R}$, the requirements engineer endeavors to satisfy *goals*, $\mathcal{G}$, where goals are meta-requirements, that is, are a kind of higher-order requirements which can be uttered, that is, postulated, but cannot be formalised in a way from which we can "derive" a software design. For the concept of 'goal' we refer to [30, Axel van Lamsweerde].

So, to us, domain engineering becomes an indispensable part of software engineering. In [6] we go as far as suggesting that current requirements engineering (research and practice) rests on flawed foundations !

9 **Software Design** Finally, from the requirements prescription, $\mathcal{R}$, software, $\mathcal{S}$, can be designed through a series of refinements and transformations such that one can prove

[3] Engineering is the discipline, art, skill and profession of acquiring and applying scientific, mathematical, economic, social, and practical knowledge, in order to design and build structures, machines, devices, systems, materials and processes ... [http://en.wikipedia.org/wiki/Engineering]

$\mathcal{D}, \mathcal{S} \models \mathcal{R}$, that is, the software design, $\mathcal{S}$, models, i.e., is a correct implementation of the requirements, $\mathcal{R}$, where the proof makes assumptions about the domain, $\mathcal{D}$.

1.1.2 Domain Studies "In Isolation" 10

But one can pursue developments of domain descriptions whether or not one subsequently wishes to pursue requirements and software design. Just as physicists study "mother nature" in order just to understand, so domain scientists cum engineers can study, for example, man-made domains — just to understand them. Such studies of man-made domains 11
seem worthwhile. Health care systems appear to be quite complex, embodying hundreds or even thousands of phenomena and concepts: parts, actions, events and behaviours. So do container lines, manufacturing, financial services (banking, insurance, trading in securities instruments, etc.), liquid and gaseous material distribution (pipelines), etcetera. Proper studies of each of these entails many, many years of work.

1.2 Additional Preliminary Notions 12

We first dwell on the "twinned" notions 'type' and 'value', Sect. 1.2.1. And then we summarise, Sect. 1.2.2, the notions of (universal, or abstract) algebras, heterogeneous algebras and 'behavioural' algebras. The latter notion, behavioural algebra, is a "home-cooked" term. (Hence the single quotes.) The algebra section, Sect. 1.2.2, is short on definitions and long on examples.

1.2.1 Types and Values 13

Values $(0, 1, 2, \ldots)$ have types (**integer**). We observe values (**false, true**)), but we speak of them by their types (**Bool**ean); that is: types are abstract concepts whereas (actual) values are (usually) concrete phenomena. By a type we shall here, simplifying, mean a way of characterising a set of entities (of similar "kind"). Entity values and types are related:when we observe an entity we observe its value; and when we say that an entity is of a given type, then we (usually) mean that the observed entity is but one of several entities *of that type.* 14

Example 1 (Types and Values of Parts) Three naïve examples

When we say, or write, *the* [or *that*] *net*, we mean

1. an entity, a specific value, n,
2. of type *net*, N.

When we say, or write, *the* [or *that*] *account*, we mean

3. an entity, a specific value, a,
4. of type *account*, A.

When we say, or write, *the* [or *that*] *container*, we mean

5. an entity, a specific value, c,
6. of type *container*, C.

type	**type**	**type**
2. N	4. A	6. C
value	**value**	**value**
1. n:N	3. a:A	5. c:C

•

15

Example 2 (Types and Values of Actions, Events and Behaviours) We continue the example above: A set of actions that all insert hubs in a net have the common signature:

value
insert: $H \rightarrow N \xrightarrow{\sim} N$

The type expression $H \rightarrow N \xrightarrow{\sim} N$ demotes an infinite set of functions from Hubs to partial functions from Nets

to Nets. The **value** clause insert: $H \rightarrow N \xrightarrow{\sim} N$ names a function value in that infinite set insert
16 and non-deterministically selects an arbitrary value in that infinite set. The functions are partial ($\xrightarrow{\sim}$) since an argument Hub may already "be" in the N in which case the insert function is not defined. A set of events that all result in a link of a net being broken can be characterised by the same predicate signature:

value
link_disappearance: $N \times N \rightarrow$ **Bool**

The set of behaviours that focus only on the insertion and removal of hubs and links in a net have the common signature:

type
Maintain = Insert_H | Remove_H | Insert_L | remove_L
value
maintain_N: $N \rightarrow \text{Maintain}^* \rightarrow N$
maintain_N: $N \rightarrow \text{Maintain}^\omega \rightarrow$ **Unit**

If insertions and removals continue ad infinitum, i.e., $^\omega$, then the maintenance behaviour do likewise: **Unit**.

•

Inquiry: Type and Value

The concept of type and its study in the last 50 years, is, perhaps, the finest contribution that computer science have made to mathematics. It all seems to have started with Bertrand Russel who needed to impose a type hierarchy on sets in order to understand the problem posed by the question: "is the set of all sets a member of itself". Explicit types were (one may claim) first introduced into programming languages in `Algol 60` *[2].*

The two concepts: 'type' and 'value' go hand-in-hand. MORE TO COME •

1.2.2 Algebras 17

Abstract Algebras By an *abstract algebra* we shall understand a (finite or infinite) set of parts (e_1, e_2, ...) called the *carrier*, A (a type), of the algebra, and a (usually finite) set of functions, f_1, f_2, ..., f_n, [each] in Ω, over these. Writing $f_i(e_{j_1}, e_{j_2}, \ldots, e_{j_m})$, where f_i is in Ω of signature:

$$\textbf{signature}\, \omega : A^n \;\rightarrow\; A$$

and each e_{j_ℓ} (ℓ : $\{1..m\}$) is in A. The operation $f_i(e_{j_1}, e_{j_2}, \ldots, e_{j_m})$ is then meant to designate either **chaos** (a totally undefined quantity) or some e_k in A.

18

Heterogeneous Algebras A *heterogeneous algebra* has its carrier set, A, consist of a number of usually disjoint sets, also referred to as sub-types of A: A_1, A_2, ..., A_n, and a set of operations, ω:Ω, such that each operation, ω, has a *signature*:

$$\textbf{signature}\, \omega : A_i \times A_j \times \cdots \times A_k \;\rightarrow\; A_r$$

where $A_i, A_j, \ldots, A_k$ and A_r are in $\{A_1, A_2, \ldots, A_n\}$. 19

Example 3 (Heterogeneous Algebras: Platoons) We leave it to the reader to fill in missing narrative and to decipher the following formalisation.

7. There are vehicles.
8. A platoon is a set of one or more vehicles.

type
7. V
8. P = {| p • p:V-**set** ∧ p≠{} |}

9. A vehicle can join a platoon.
10. A vehicle can leave a platoon.
11. Two platoons can be merged into one platoon.
12. A platoon can be split into two platoons.

20

9. join_0: V × P → P
9. join_0(v,p) ≡ p ∪ {v} **pre**: v ∉ p

10. leave_0: V × P → P
10. leave_0(v,p) ≡ p\{v} **pre**: v ∈ p

11. merge_0: P × P → P

11. merge_0(p,p′) ≡ p ∪ p′ **pre**: p ≠ {} ≠ p′ ∧ p ∩ p′ = {}

12. split_0: P → P-**set**
12. split_0(p) ≡ **let** p′,p″:P • p′ ∪ p″ = p **in** {p′,p″} **end pre**: **card** p ≥ 2

The above formulas define a heterogeneous algebra with types V and P and operations (or actions) *join_0*, *leave_0*, *merge_0*, and *split_0*.

•

21

Behavioral Algebras An abstract algebra is characterised by the one type, A, of its parts and by its operations all of whose signatures are of the form $A{\times}A{\times}\cdots{\times}A{\to}A$. A heterogeneous algebra is an abstract algebra and is further characterised by two or more types, A_1, A_2, $\ldots$, A_m, and by a set of operations of usually distinctly typed signatures. A behavioral algebra is a heterogeneous algebra and is further characterised by a set of events and by a set of behaviours where events are like actions and behaviours are sets of
22 sequences of actions, events and behaviours.

Example 4 (**A Behavioural Algebra: A System of Platoons and Vehicles**) Our example may be a bit contrived. We have yet to unfold, as we do in this paper, enough material to give more realistic examples.

13. A well-formed platoon/vehicle system consists of a pair:
    a *convoys* which is a varying set of [non-empty] platoons and
    b *reservoir* which is a varying set of vehicles —
    c such that the *convoys* platoons are disjoint, no vehicles in common, and
    d such that *reservoir* have no vehicle in common with any platoon in *convoys*.

23

14. Platoons are characterised by unique platoon identifiers.
15. These identifiers can be observed from platoons.
16. Vehicles from the *reservoir* behaviour may join [leave] a platoon whereby they leave [respectively join] the pool.
17. Two platoons may merge into one, and a platoon may split into two.
18. Finally, vehicles may enter [exit] the system by entering [exiting] *reservoir*.

24

type
13. S = {| (c,r):C×R•r ∩ ∪ c = {} |}
13a. C = {| c:P-**set** • wf_C(c) |}
value

13c. wf_C: $C \rightarrow$ **Bool**
13c. wf_C(c) $\equiv \forall\ p,p':P \bullet \{p,p'\} \subseteq c \Rightarrow p \neq \{\} \neq p' \wedge p \cap p' = \{\}$
type
13b. R = V-**set**
value
16. join_1: $S \xrightarrow{\sim} S$
16. leave_1: $S \xrightarrow{\sim} S$
17. merge_1: $S \xrightarrow{\sim} S$
17. split_1: $S \xrightarrow{\sim} S$
18. enter_1: $S \xrightarrow{\sim} S$
18. exit_1: $S \xrightarrow{\sim} S$

25

19. join_1 selects an arbitrary vehicle in *r:R* and an arbitrary platoon *p* in *c:C*, joins *v* to *p* in *c* and removes *v* from *r*.
20. leave_1 selects a platoon *p* in *c* and a vehicle *v* in *p*, removes *v* from *p* in *c* and joins *v* to *r*.
21. merge_1 selects two distinct platoons *p*,*p′* in *c*, removes them from *c*, takes their union and adds to *c*.
22. split_1 selects a platoon *p* in *c*, one which has at least to vehicles,
23. and partitions *p* into *p′* and *p″*, removes *p* from *c* and joins *p′* and *p″* to *c*.
24. enter_1 joins a fresh vehicle *v* to *r*.
25. exit_1 removes a vehicle *v* from a non-empty *r*.

26

19. join_1(c,r) $\equiv$
19. **let** $v:V \bullet v \in r, p:P \bullet p \in c$ **in**
19. $(c \setminus \{p\} \cup \{$**join_0(v,p)**$\}, r \setminus \{v\})$ **end**

20. leave_1(c,r) $\equiv$
20. **let** $v:V, p:P \bullet p \in c \wedge v \in p$ **in**
20. $(c \setminus \{p\} \cup \{$**leave_0(v,p)**$\}, r \cup \{v\})$ **end**

21. merge_1(c,r) $\equiv$
21. **let** $p,p':P \bullet p \neq p' \wedge \{p,p'\} \subseteq c$ **in**
21. $(c \setminus \{p,p'\} \cup \{$**merge_0(p,p')**$\}, r)$ **end**

22. split_1(c,r) $\equiv$
23. **let** $p:P \bullet p \in c \wedge$ **card** $p \geq 2$ **in**

23. **let** p′,p″:P•p ∪ p′ = p **in**
23. (c\{p} ∪ **split_0(p)**,r) **end end**

24. enter_1(c,r) ≡ (c,**let** v:V•v∉ r ∪ ∪ c **in** r ∪ {v} **end**)
25. exit_1()(c,r) ≡ (c,**let** v:V•v ∈ r **in** r\{v} **end**) **pre**: r≠{}

The r ∪ ∪c in enter_1(c,r) expresses the union (with the vehicles of r) of all the vehicles in
27 all the platoons of c, i.e., the distributed union of c (∪c).

The above model abstracts an essence of the non-deterministic behaviour of a platooning system. We make no assumptions about which vehicles are joined to or leave which platoons, which platoons are merged, which platoon is split nor into which sub-platoons,
28 and which vehicle enters and exits the reservoir state.

26. We model the above *system* as a behaviour which is composed from a pair of concurrent behaviours:
    a a *convoys* behaviour and
    b a *reservoir* behaviour
    c where these behaviours interact via a channel *cr_ch* and
    d where the entering of "new" and exiting of "old" vehicles occur on a channel *io_ch*
27. Hence the communications between the *reservoir* behaviour and the *convoys* behaviour are of three kinds: *J*oining (moving) a vehicle to a ("magically"[4]) named platoon from the *reservoir* behaviour, *R*emoving [moving] a vehicle from a named platoon to (*mkV(v)*) the *reservoir* behaviour

29

type
27. M == mkJ(v:V) | mkR | mkV(v:V)
channel
26c. cr_ch:M
26d. io_ch:V
value
26. system: S → **Unit**
26. system(c,r) ≡ convoys(c) ∥ reservoir(r)

30

28. The *convoys* behaviour non-deterministically (⌈⌉) chooses either to
    a merge platoons, or to

[4]In this example we skip the somewhat 'technical' details as to how the *reservoir* behaviour obtains knowledge of platoon names.

b split platoons, or to

c interact with the *reservoir* behaviour via channel *ct_ch*

d and based on that interactions

i. to either join a[n arbitrary] vehicle *v* to a platoon, or

ii. to remove a named vehicle, *v*, from a platoon

iii. while "moving' that vehicle to *reservoir*.

31

```
28.       convoys: C → in,out cr_ch  Unit
28.       convoys(c) ≡ convoys(merge(c)) ⌈⌉ convoys(split(c)) ⌈⌉ convoys(interact(c))

28c.      interact: C → in,out cr_ch  C
28c.      interact(c) ≡
28c.         let m = cr_ch ? in
28d.         case m of
28(d)i.            mkJ(v) → join_vehicle(v,c),
28(d)ii.           mkR  → let (c′,v)=remove_vehicle(c) in
28(d)iii.                     ct_ch!mkV(v) ; c′
28c.         end end end
```

32

29. The *merge_platoons* behaviour

a non-deterministically chooses two platoons of *convoys* (*p*,*p*′),

b removes the two platoons from *convoys* and adds the *merge* of these two platoons to *convoys*.

c If *convoys* contain less than two platoons then *merge_platoons* is undefined.

```
29.       merge_platoons: C → C
29.       merge_platoons(c) ≡
29a.         let p,p′,p″:P • p≠p′∧{p,p′}⊆ c in
29b.         c\{p,p′} ∪ {merge_0(p,p′)} end
29b.         pre: card c ≥ 2
```

33

30. The *split_platoons* function

a non-deterministically chooses a platoon, *p*, of two or more vehicles in *convoys*,

b removes the chosen platoon from *convoys* and inserts the split platoons into *convoys*.

c If there are no platoons in *c* with two or more vehicles then *split_platoons* is undefined.

```
30.  split_platoons: C ~→ C
30.  split_platoons(c) ≡
30a.     let p:P • p ∈ c ∧ card p ≥ 2 in
30b.     c\{p} ∪ {split_0(p)} end
30c.     pre: ∃ p:P • p ∈ c ∧ card p ≥ 2
```

34

31. The *reservoir* behaviour interacts with the *convoys* behaviour and with "an external", that is, undefined behaviour through channels *ct_ch* and *io_ch*.
   The *reservoir* behaviour [external] non-deterministically chooses between

   a importing a vehicle from "the outside",

   b exporting a vehicle to "the outside",

   c moving a vehicle to the *convoys* behaviour, and

   d moving a vehicle from the *convoys* behaviour.

35

```
31.  reservoir: R → in,out cr_ch, io_ch  Unit
31.  reservoir(r) ≡
31a.       (r ∪ {io_ch?}),
31b.     ⌷ let v:V • v ∈ t in io_ch!mkV(v) ; reservoir(r\{v}) end
31c.     ⌷ let v:V • v ∈ t in ct_ch!mkJ(v) ; reservoir(r\{v}) end
31d.     ⌷ let mkV(v) = ct_ch? in reservoir(r ∪ {v}) end
```

We may consider Items 31a–31b as designating events.

This example designates a behavioural algebra.

•

Inquiry: Algebra

Algebra is a mathematical notion. We shall use this notion in seeking to describe domains as algebras.

MORE TO COME

•

1.3 On 'Method' and 'Methodology' 36

Inquiry: Method and Methodology

We present our characterisation of the concepts of 'method' and 'methodology'. When we use these terms then our characterisation is what we mean by their use. There are other characterisations. Be that as it may. •

By a *method* we shall understanda set of *principles*, *techniques* and *tools* where the principles help *select* and *apply* these techniques and tools such that an *artifact*, here a domain description, can be constructed.

By *methodology* we shall understand the *knowledge* and *study* of one or more methods.

Languages, whether informal, as English, or formal, as RSL, are *tools*.

1.4 An Ontology of Descriptions 37

"By ontology we mean the philosophical study of the nature of being, existence, or reality as such, as well as the basic categories of being and their relations. Traditionally listed as a part of the major branch of philosophy known as metaphysics, ontology deals with questions concerning what entities exist or can be said to exist, and how such entities can be grouped, related within a hierarchy, and subdivided according to similarities and differences."[5]

1.4.1 Entities and Properties 38

A main stream of philosophers [34, 20, 18] appear to agree that there are two categories of discourse: entities[6] and properties. Once we say that, a number of questions arise: (Q_1) What counts as an entity ? (Q_2) What counts as a property ? (Q_3) Are properties entities ? (Q_4) Can properties predicate properties ? We shall take no and yes to be answers to Q_3 and Q_4. These lecture notes shall answer Q_1 and Q_2

1.4.2 Categories of Entities 39

We shall promulgate the following classes of entities: parts, and operations. where we further "sub-divide" operations into actions, events and behaviours That is, we can predicate entities, e, as follows: $\mathbb{IS\_PART}(e)$, $\mathbb{IS\_OPERATION}(e)$, that is, $\mathbb{IS\_ACTION}(e)$, $\mathbb{IS\_EVENT}(e)$ and $\mathbb{IS\_BEHVAIOUR}(e)$. We shall justify the above categorisation through these lecture notes. So parts, actions, events and behaviours form an ontology of descriptions.

1.5 Structure of Paper 40

[5] http://en.wikipedia.org/wiki/Ontology

[6] The literature [31, 10, 11, 34, 20, 18, 41] alternatively refer to entities by the term individuals.