# Domain Science & Engineering
# From Computer Science to The Sciences of Informatics

Dines Bjørner
Fredsvej 11, DK-2840 Holte, Denmark
bjorner@gmail.com -- www.imm.dtu.dk/~db

14 February 2010: Compiled: March 13, 2010: 00:00 ECT

**Abstract**

In this paper we wish to **advocate** (i) that schools, institutes and departments of computer science, software engineering, informatics, cybernetics, and the like, **re-orient** themselves along two lines: (i.1) more **emphasis on teaching** programming and software engineering based on **formal methods**; and (i.2) more **emphasis on research** into **formal methods** for the trustworthy development of software that meets customers' expectations and is correct, that is, the right software and that the software is right. We also wish to **advocate** (ii) that the concepts of **domain science** and **domain engineering** become an indispensable part of the **science of informatics** and of **software engineering**. And we finally wish to **advocate** (iii) that informatics research centers embark on **path-finder projects** which **research** and **experimentally develop** domain models for infra-structure components, for example, (iii.1) **financial service industries** (banks, stock exchanges, etc.), (iii.2) **health-care** (hospitals, clinics, private physicians, etc.) (iii.3) **pipeline systems** (oil, gas), (iii.4) **transportation** (such as railways, shipping, air traffic, etc.).

## 1 Introduction

The background postulates of this paper are the following: (i) half a century of computer science research may very well have improved our understanding of computing devices (automata etc.), but it has yet to contribute significantly to the quality of software products; (ii) our students, the future leading software engineers, those of them who go into industry rather than "remaining" in academia, are being mislead by too many foundational courses to believe that these are relevant for the practice of software engineering; (iii) a significant re-orientation of university teaching and research into both 'computer science' and software engineering must occur if we are to improve the relevance of 'computer science' to software engineering. In this paper we shall, unabashedly, suggest the kind of re-orientation that we think will rectify the situation alluded to in Items (i–iii).

## 1.1 Some Definitions of Informatics Topics 7

Let us first delineate our field of study. It first focuses on *computer science*, *computing science*, *software* and *software engineering*.

8

**Definition 1** – ***Computer Science:*** By *computer science* we shall understand the study and knowledge of the properties of the *'things'* that can *'exist'* inside computers: data and processes.

Examples of *computer science* disciplines are: automata theory (studying automata [finite or otherwise] and state machines [without or with stacks]), formal languages (studying, mostly the syntactic the "foundations" and "recognisability" of abstractions of of computer programming and other "such" languages), complexity theory, type theory, etc.

9

Some may take exception to the term *'things'*[1] used in the above and below definition. They will say that it is imprecise. That using the germ conjures some form of reliance on Plato's Idealism, on his Theory of Forms. That is, *"that it is of Platonic style, and thus, is disputable. One could avoid this by saying that these definitions are just informal rough explanations of the field of study and further considerations will lead to more exact definitions."*[2] Well, it may be so. It is at least a conscious attempt, from this very beginning, to call into dispute and discuss "those things". Section 4 of this paper ("A Specification Ontology and Epistemology") has as one of its purposes to encircle the problem.

10

**Definition 2** – ***Computing Science:*** By *computing science* we shall understand the study and knowledge of the how to construct the *'things'* that can *'exist'* inside computers: the software and its data.

Conventional examples of *computing science* disciplines are: algorithm design, imperative programming, functional programming, logic programming, parallel programming, etc. To these we shall add a few in this paper.

11

**Definition 3** – ***Software:*** By *software* we shall understand not only the code intended for computer execution, but also its use, i.e., programmer manuals: installation, education, user and other guidance documents, as well all as its development documents: domain models, requirements models, software designs, tests suites, etc. "zillions upon zillions" of documents.

12

The fragment description of the example Pipeline System of this paper exhibits, but a tiny part of a domain model.

**Definition 4** – ***Software Engineering:*** By *software engineering* we shall understand the methods (analysis and construction principles, techniques and tools) needed to carry out, manage and evaluate software development projects as well as software product marketing, sales

[1] and also to the term *'exist'*.

[2] Cf. personal communication, 12 Feb., 2010, with Prof. Mikula Nikitchenko, Head of the Chair of Programming Theory of Shevchenko Kyiv National University, Ukraine

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

and service — whether these includes only domain engineering, or requirements engineering, or software design, or the first two, the last two or all three of these phases. *Software engineering*, besides documents for all of the above, also includes all auxiliary project information, stakeholder notes, acquisition units, analysis, terminology, verification, model-checking, testing, etc. documents

### 1.2 The Triptych Dogma 13

**Dogma 1** – ***Triptych:*** By *the triptych dogma* we shall understand a dogma which insists on the following: Before software can be designed one must have a robust understanding of its requirements; and before requirements can be prescribed one must have a robust understanding of their domain.

14

**Dogma 2** – ***Triptych Development:*** By *triptych development* we shall understand a software development process which starts with one or more stages of *domain engineering* whose objective it is to construct a *domain description*, which proceeds to one or more stages of *requirements engineering* whose objective it is to construct a *requirements prescription*, and which ends with one or more stages of *software design* whose aim it is to construct the *software*.

### 1.3 Structure of This Paper 15

In Sect. 2 we present a non-trivial example. It shall serve to illustrate the new concepts of *domain engineering*, *domain description* and *domain model*. In Sect. 3 we shall then discuss ramifications of the triptych dogma. Then we shall follow-up, Sect. 4, on what we have advocated above, namely a beginning discussion of our logical and linguistic means for description, of "the kind of '*things*' that can '*exists*' or the things (say in the domain, i.e., "real world") that they reflect".

## 2 Example: A Pipeline System 16

The example is to be read "hastily". That is, emphasis, by the reader, should be on the narrative, that is, on conveying what a domain model describes, rather than on the formulas.

The example is that of domain modelling an pipeline system Figure 1 on the following page show the planned Nabucco pipeline system. 17

### 2.1 Pipeline Basics 18

Figure 2 on page 5 conceptualises an example pipeline. Emphasis is on showing a pipeline net consisting of units and connectors (•). 19


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

Figure 1: The Planned Nabucco Pipeline: http://en.wikipedia.org/wiki/Nabucco_Pipeline

These are some non-temporal aspects of pipelines. nets and units: wells, pumps, pipes, valves, joins, forks and sinks; net and unit attributes; and units states, but not state changes. We omit consideration of "pigs" and "pig"-insertion and "pig"-extraction units. 20

## Pipeline Nets and Units:

1. We focus on nets, $n : N$, of pipes, $\pi : \Pi$, valves, $v : V$, pumps, $p : P$, forks, $f : F$, joins, $j : J$, wells, $w : W$ and sinks, $s : S$.
2. Units, $u : U$, are either pipes, valves, pumps, forks, joins, wells or sinks.
3. Units are explained in terms of disjoint types of PIpes, VAlves, PUmps, FOrks, JOins, WElls and SKs.

```
type
  1 N, PI, VA, PU, FO, JO, WE, SK
  2 U = Π | V | P | F | J | S| W
  2 Π == mkΠ(pi:PI)
  2 V == mkV(va:VA)
  2 P == mkP(pu:PU)
  2 F == mkF(fo:FO)
  2 J == mkJ(jo:JO)
  2 W == mkW(we:WE)
  2 S == mkS(sk:SK)
```

21

## Unique Identifiers:

4. We associate with each unit a unique identifier, $ui : UI$.
5. From a unit we can observe its unique identifier.
6. From a unit we can observe whether it is a pipe, a valve, a pump, a fork, a join, a well or a sink unit.

```
type
  4 UI
value
  5 obs_UI: U → UI
  6 is_Π: U → Bool
    is_Π(u) ≡
       case u of mkPI(_)→true,_→false end
  6 is_V: U → Bool
    is_V(u) ≡
       case u of mkV(_)→true,_→false end
  6 ...
  6 is_S: U → Bool
    is_S(u) ≡
       case u of mkS(_)→true,_→false end
```

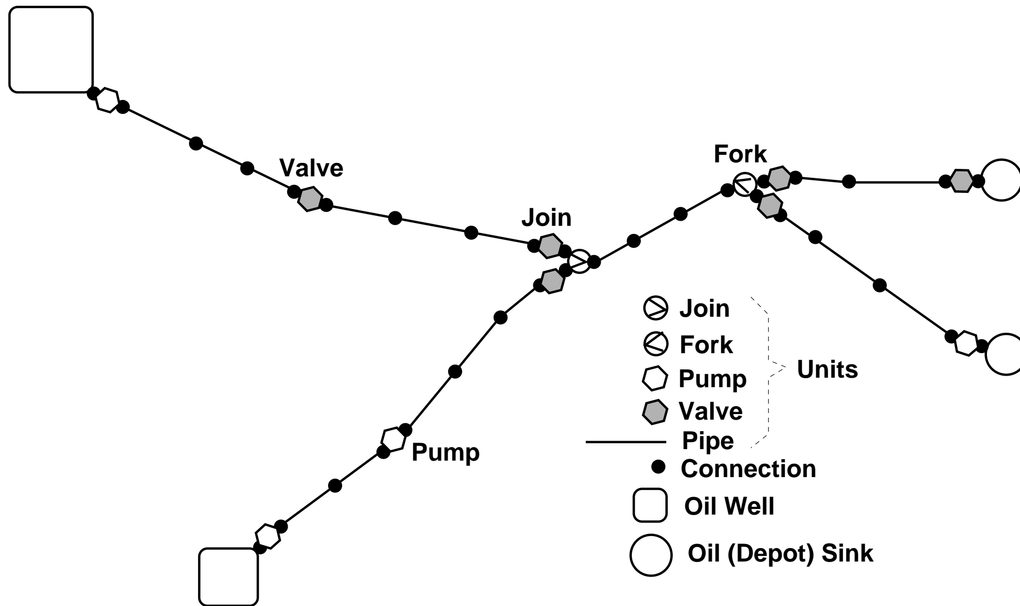


Figure 2: An oil pipeline system

A connection is a means of juxtaposing units. A connection may connect two units in which case one can observe the identity of connected units from "the other side".

### Pipe Unit Connectors:

7. With a pipe, a valve and a pump we associate exactly one input and one output connection.
8. With a fork we associate a maximum number of output connections, $m$, larger than one.
9. With a join we associate a maximum number of input connections, $m$, larger than one.
10. With a well we associate zero input connections and exactly one output connection.
11. With a sink we associate exactly one input connection and zero output connections.

```
value
  7 obs_InCs,obs_OutCs: Π|V|P → {|1:Nat|}
  8 obs_inCs: F → {|1:Nat|}
  8 obs_outCs: F → Nat
  9 obs_inCs: J → Nat
  9 obs_outCs: J → {|1:Nat|}
  10 obs_inCs: W → {|0:Nat|}
  10 obs_outCs: W → {|1:Nat|}
  11 obs_inCs: S → {|1:Nat|}
  11 obs_outCs: S → {|0:Nat|}
axiom
  8 ∀ f:F • obs_outCs(f) ≥ 2
  9 ∀ j:J • obs_inCs(j) ≥ 2
```

If a pipe, valve or pump unit is input-connected [output-connected] to zero (other) units, then it means that the unit input [output] connector has been sealed. If a fork is input-connected to zero (other) units, then it means that the fork input connector has been sealed. If a fork is output-connected to $n$ units less than the maximum fork-connectability, then it means that the unconnected fork outputs have been sealed. Similarly for joins: "the other way around".

### Observers and Connections:


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

12. From a net one can observe all its units.
13. From a unit one can observe the the pairs of disjoint input and output units to which it is connected:
    a) Wells can be connected to zero or one output unit — a pump.
    b) Sinks can be connected to zero or one input unit — a pump or a valve.
    c) Pipes, valves and pumps can be connected to zero or one input units and to zero or one output units.
    d) Forks, $f$, can be connected to zero or one input unit and to zero or $n$, $2 \leq n \leq$obs_Cs($f$) output units.
    e) Joins, $j$, can be connected to zero or $n$, $2 \leq n \leq$obs_Cs($j$) input units and zero or one output units.

```
value
  12 obs_Us: N → U-set

13 obs_cUIs: U → UI-set × UI-set
   wf_Conns: U → Bool
   wf_Conns(u) ≡
      let (iuis,ouis)=obs_cUIs(u) in
      iuis ∩ ouis={}∧
      case u of
13a  mkW(_) →
         card iuis ∈{0}∧card ouis ∈{0,1},
13b  mkS(_) →
         card iuis ∈{0,1}∧card ouis ∈{0},
13c  mkΠ(_) →
         card iuis ∈{0,1}∧card ouis ∈{0,1},
13c  mkV(_) →
         card iuis ∈{0,1}∧card ouis ∈{0,1},
13c  mkP(_) →
         card iuis ∈{0,1}∧card ouis ∈{0,1},
13d  mkF(_) →
         card iuis ∈{0,1}∧
         card ouis ∈{0}∪{2..obs_inCs(j)},
13e  mkJ(_) →
         card iuis ∈{0}∪{2..obs_inCs(j)}∧
         card ouis ∈{0,1}
      end end
```

26

### Wellformedness:

14. The unit identifiers observed by the obs_cUIs observer must be identifiers of units of the net.

```
axiom
  14 ∀ n:N,u:U • u ∈ obs_Us(n) ⇒
  14   let (iuis,ouis) = obs_cUIs(u) in
  14   ∀ ui:UI • ui ∈ iuis ∪ ouis ⇒
  14     ∃ u′:U •
  14       u′ ∈ obs_Us(n)∧u′≠u∧obs_UI(u′)=ui
  14   end
```

## 2.2 Routes

27

### Routes:

15. By a route we shall understand a sequence of units.
16. Units form routes of the net.

```
type
  15 R = UI^ω

value
  16 routes: N → R-infset
  16 routes(n) ≡
  16   let us = obs_Us(n) in
  16   let rs = {⟨u⟩|u:U•u ∈ us}
  16     ∪ {r⌢r′|r,r′:R• {r,r′}⊆rs∧adj(r,r′)} in
  16   rs end end
```

28

### Adjacent Routes:

17. A route of length two or more can be decomposed into two routes
18. such that the last unit of the first route "connects" to the first unit of the second route.

**value**

```
17  adj: R × R → Bool
17  adj(fr,lr) ≡
17    let (lu,fu)=(fr(len fr),hd lr) in
18    let (lui,fui)=(obs_UI(lu),obs_UI(fu)) in
18    let ((_,luis),(fuis,_)) =
18        (obs_cUIs(lu),obs_cUIs(fu)) in
18    lui ∈ fuis ∧ fui ∈ luis end end end
```

29

## No Circular Routes:

19. No route must be circular, that is, the net must be acyclic.

**value**

```
19  acyclic: N → Bool
19    let rs = routes(n) in
19    ∼∃ r:R•r ∈ rs ⇒
19      ∃ i,j:Nat•{i,j}⊆inds r∧
19      i≠j∧r(i)=r(j) end
```

30

## Wellformed Nets, Special Pairs, wfN_SP:

20. We define a "special-pairs" well-formedness function.
    a) Fork outputs are output-connected to valves.
    b) Join inputs are input-connected to valves.
    c) Wells are output-connected to pumps.
    d) Sinks are input-connected to either pumps or valves.

**value**

```
20  wfN_SP: N → Bool
20  wfN_SP(n) ≡
20    ∀ r:R • r ∈ routes(n) in
20      ∀ i:Nat • {i,i+1}⊆inds r ⇒
20        case r(i) of
20a         mkF(_) → ∀ u:U•adj(⟨r(i)⟩,⟨u⟩)
20a                        ⇒ is_V(u),
20a         _→true end ∧
20        case r(i+1) of
20b         mkJ(_) → ∀ u:U•adj(⟨u⟩,⟨r(i)⟩)
20b                        ⇒ is_V(u),
20b         _→true end ∧
20        case r(1) of
20c         mkW(_) → is_P(r(2)),
20c         _→true end ∧
20        case r(len r) of
20d         mkS(_) → is_P(r(len r−1))
20d                    ∨ is_V(r(len r−1)),
20d         _→true end
```

The **true** clauses may be negated by other **case** distinctions' is_V or is_V clauses.

### 2.2.1 Special Routes, I

31

21. A pump-pump route is a route of length two or more whose first and last units are pumps and whose intermediate units are pipes or forks or joins.
22. A simple pump-pump route is a pump-pump route with no forks and joins.
23. A pump-valve route is a route of length two or more whose first unit is a pump, whose last unit is a valve and whose intermediate units are pipes or forks or joins.
24. A simple pump-valve route is a pump-valve route with no forks and joins.


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

25. A valve-pump route is a route of length two or more whose first unit is a valve, whose last unit is a pump and whose intermediate units are pipes or forks or joins.

26. A simple valve-pump route is a valve-pump route with no forks and joins.

27. A valve-valve route is a route of length two or more whose first and last units are valves and whose intermediate units are pipes or forks or joins.

28. A simple valve-valve route is a valve-valve route with no forks and joins.

32

**value**

21-28 ppr,sppr,pvr,spvr,vpr,svpr,vvr,svvr: R → **Bool**
  **pre** {ppr,sppr,pvr,spvr,vpr,svpr,vvr,svvr}(n): **len** n≥2

21 ppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_P(fu) ∧ is_P(lu) ∧ is_πfjr(ℓ)
22 sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
23 pvr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_P(fu) ∧ is_V(r(**len** r)) ∧ is_πfjr(ℓ)
24 sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
25 vpr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_V(fu) ∧ is_P(lu) ∧ is_πfjr(ℓ)
26 sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
27 vvr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_V(fu) ∧ is_V(lu) ∧ is_πfjr(ℓ)
28 sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)

is_πfjr,is_πr: R → **Bool**
is_πfjr(r) ≡ ∀ u:U•u ∈ **elems** r⇒is_Π(u)∨is_F(u)∨is_J(u)
is_πr(r) ≡ ∀ u:U•u ∈ **elems** r⇒is_Π(u)

### 2.2.2 Special Routes, II 33

Given a unit of a route,

29. if they exist (∃),

30. find the nearest pump or valve unit,

31. "upstream" and

32. "downstream" from the given unit.

34

**value**

29 ∃UpPoV: U × R → **Bool**
29 ∃DoPoV: U × R → **Bool**
31 find_UpPoV: U × R $\stackrel{\sim}{\to}$ (P|V), **pre** find_UpPoV(u,r): ∃UpPoV(u,r)

```
32 find_DoPoV: U × R ~→ (P|V), pre find_DoPoV(u,r): ∃DoPoV(u,r)
29 ∃UpPoV(u,r) ≡
29   ∃ i,j Nat•{i,j}⊆inds r∧i≤j∧{is_V|is_P}(r(i))∧u=r(j)
29 ∃DoPoV(u,r) ≡
29   ∃ i,j Nat•{i,j}⊆inds r∧i≤j∧u=r(i)∧{is_V|is_P}(r(j))
31 find_UpPoV(u,r) ≡
31   let i,j:Nat•{i,j}⊆indsr∧i≤j∧{is_V|is_P}(r(i))∧u=r(j) in r(i) end
32 find_DoPoV(u,r) ≡
32   let i,j:Nat•{i,j}⊆indsr∧i≤j∧u=r(i)∧
32     {is_V|is_P}(r(j))
32   in r(j) end
```

## 2.3 State Attributes of Pipeline Units 35

By a state attribute of a unit we mean either of the following three kinds: (i) the open/close states of valves and the pumping/not_pumping states of pumps; (ii) the maximum (laminar) oil flow characteristics of all units; and (iii) the current oil flow and current oil leak states of all units. 36

**Unit Attributes:**

33. Oil flow, $\phi : \Phi$, is measured in volume per time unit.
34. Pumps are either pumping or not pumping, and if not pumping they are closed.
35. Valves are either open or closed.
36. Any unit permits a maximum input flow of oil while maintaining laminar flow. We shall assume that we need not be concerned with turbulent flows.
37. At any time any unit is sustaining a current input flow of oil (at its input(s)).
38. While sustaining (even a zero) current input flow of oil a unit leaks a current amount of oil (within the unit).

```
type
  33 Φ
  34 PΣ == pumping | not_pumping
  34 VΣ == open | closed
value
     −,+: Φ × Φ → Φ,
     <,=,>: Φ × Φ → Bool
  34 obs_PΣ: P → PΣ
  35 obs_VΣ: V → VΣ
  36–38 obs_LamiΦ.obs_CurrΦ,obs_LeakΦ: U → Φ
  is_Open: U → Bool
    case u of
      mkΠ(_)→true,
      mkF(_)→true,
      mkJ(_)→true,
      mkW(_)→true,
      mkS(_)→true,
      mkP(_)→obs_PΣ(u)=pumping,
      mkV(_)→obs_VΣ(u)=open
    end
  accept_LeakΦ,excess_LeakΦ: U → Φ
axiom
  ∀ u:U • excess_LeakΦ(u) > accept_LeakΦ(u)
```

37

The sum of the current flows into a unit equals the the sum of the current flows out of a unit minus the (current) leak of that unit. This is the same as the current flows out of a unit equals the current flows into a unit minus the (current) leak of that unit. The above represents an interpretation which justifies the below laws. 38


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

### Flow Laws (I):

39. When, in Item 37, for a unit u, we say that at any time any unit is sustaining a current input flow of oil, and when we model that by obs_CurrΦ(u) then we mean that obs_CurrΦ(u) - obs_LeakΦ(u) represents the flow of oil from its outputs.

```
value
 39  obs_inΦ: U → Φ
 39  obs_inΦ(u) ≡ obs_CurrΦ(u)
 39  obs_outΦ: U → Φ
law:
 39  ∀ u:U • obs_outΦ(u) =
 39    obs_CurrΦ(u)−obs_LeakΦ(u)
```

39

### Flow Laws (II):

40. Two connected units enjoy the following flow relation, if
    a) two pipes, or
    b) a pipe and a valve, or
    c) a valve and a pipe, or
    d) a valve and a valve, or
    e) a pipe and a pump, or
    f) a pump and a pipe, or
    g) a pump and a pump, or
    h) a pump and a valve, or
    i) a valve and a pump

    are immediately connected

41. then
    a) the current flow out of the first unit's connection to the second unit
    b) equals the current flow into the second unit's connection to the first unit

```
law:
 40  ∀ u,u′:U •
 40    {is_Π,is_V,is_P,is_W}(u′|u′′)
 40    ∧ adj(⟨u⟩,⟨u′⟩)
 40    ∧ is_Π(u)∨is_V(u)∨is_P(u)∨is_W(u)
 40    ∧ is_Π(u′)∨is_V(u′)∨is_P(u′)∨is_S(u′)
 41    ⇒ obs_outΦ(u)=obs_inΦ(u′)
```

40

A similar law can be established for forks and joins. For a fork output-connected to, for example, pipes, valves and pumps, it is the case that for each fork output the out-flow equals the in-flow for that output-connected unit. For a join input-connected to, for example, pipes, valves and pumps, it is the case that for each join input the in-flow equals the out-flow for that input-connected unit. We leave the formalisation as an exercise.

## 2.4 Pipeline Actions

41

### Simple Pump and Valve Actions:

42. Pumps may be set to pumping or reset to not pumping irrespective of the pump state.
43. Valves may be set to be open or to be closed irrespective of the valve state.
44. In setting or resetting a pump or a valve a desirable property may be lost.

```
value
 42  to_pump, to_not_pump: P→N→N
 43  vlv_to_op, vlv_to_clo: V→N→N
 42  to_pump(p)(n) as n′
 42    pre p ∈ obs_Us(n)
 42    post let p′:P•obs_UI(p)=obs_UI(p′) in
 42      obs_PΣ(p′)=pumping
```

```
42       ∧ else_equal(n,n′)(p,p′) end
42  to_not_pump(p)(n) as n′
42    pre p ∈ obs_Us(n)
42    post let p′:P•obs_UI(p)=obs_UI(p′) in
42      obs_PΣ(p′)=not_pumping
42      ∧ else_equal(n,n′)(p,p′) end
43  vlv_to_op(v)(n) as n′
42    pre v ∈ obs_Us(n)
43    post let v′:V•obs_UI(v)=obs_UI(v′) in
42      obs_VΣ(v′)=open
42      ∧ else_equal(n,n′)(v,v′) end
43  vlv_to_clo(v)(n) as n′
42    pre v ∈ obs_Us(n)
43    post let v′:V•obs_UI(v)=obs_UI(v′) in
42      obs_VΣ(v′)=close
42       ∧ else_equal(n,n′)(v,v′) end
  else_equal: (N×N) → (U×U) → Bool
  else_equal(n,n′)(u,u′) ≡
      obs_UI(u)=obs_UI(u′)
    ∧ u ∈ obs_Us(n) ∧ u′ ∈ obs_Us(n′)
    ∧ omit_Σ(u) = omit_Σ(u′)
    ∧ obs_Us(n)\{u} = obs_Us(n) \ {u′}
    ∧ ∀ u″:U•u″ ∈ obs_Us(n)\{u}
                ≡ u″ ∈ obs_Us(n′) \ {u′}
  omit_Σ: U → U_no_state — "magic" function
  =: U_no_state × U_no_state → Bool
axiom
  ∀ u,u′:U•omit_Σ(u)=omit_Σ(u′)
              ≡ obs_UI(u)=obs_UI(u′)
```

## Unit Handling Events:

45. Let $n$ be any acyclic net.
45. If there exists $p, p', v, v'$, pairs of distinct pumps and distinct valves of the net,
45. and if there exists a route, $r$, of length two or more of the net such that
46. all units, $u$, of the route, except its first and last unit, are pipes, then
47. if the route "spans" between $p$ and $p'$ and the *simple desirable property*, sppr(r), does not hold for the route, then we have a possibly undesirable event — that occurred as soon as sppr(r) did not hold;
48. if the route "spans" between $p$ and $v$ and the *simple desirable property*, spvr(r), does not hold for the route, then we have a possibly undesirable event;
49. if the route "spans" between $v$ and $p$ and the *simple desirable property*, svpr(r), does not hold for the route, then we have a possibly undesirable event; and
50. if the route "spans" between $v$ and $v'$ and the *simple desirable property*, svvr(r), does not hold for the route, then we have a possibly undesirable event.

```
events:
  45  ∀ n:N • acyclic(n) ∧
  45    ∃ p,p′:P,v,v′:V • {p,p′,v,v′}⊆obs_Us(n)⇒
  45      ∧ ∃ r:R • r ∈ routes(n) ∧
  46        ∀ u:U•u ∈ elems(r)\{hd r,r(len r)}⇒
  47          is_Π(i) ⇒
  47            p=hd r∧p′=r(len r) ⇒ ∼sppr_prop(r) ∧
  48            p=hd r∧v=r(len r) ⇒ ∼spvr_prop(r) ∧
  49            v=hd r∧p=r(len r) ⇒ ∼svpr_prop(r) ∧
  50            v=hd r∧v′=r(len r) ⇒ ∼svvr_prop(r)
```

42

43

## Wellformed Operational Nets:

51. A well-formed operational net
52. is a well-formed net
    a) with at least one well, $w$, and at least one sink, $s$,
    b) and such that there is a route in the net between $w$ and $s$.

```
value
  51  wf_OpN: N → Bool
  51  wf_OpN(n) ≡
  52    satisfies axiom 14 on page 6
  52    ∧ acyclic(n): Item 19 on page 7
  52    ∧ wfN_SP(n): Item 20 on page 7
  52    ∧ satisfies 39 on the preceding page and 40 on the facing p
  52a     ∧ ∃ w:W,s:S • {w,s}⊆obs_Us(n)
  52b       ⇒ ∃ r:R• ⟨w⟩⌢r⌢⟨s⟩ ∈ routes(n)
```


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

44

### Initial Operational Net:

53. Let us assume a notion of an initial operational net.
54. Its pump and valve units are in the following states
    a) all pumps are not_pumping, and
    b) all valves are closed.

```
value
  53  initial_OpN: N → Bool
  54  initial_OpN(n) ≡ wf_OpN(n) ∧
  54a     ∀ p:P • p ∈ obs_Us(n) ⇒ obs_PΣ(p)=not_pumping ∧
  54b     ∀ v:V • v ∈ obs_Us(n) ⇒ obs_VΣ(p)=closed
```

45

### Oil Pipeline Preparation and Engagement:

55. We now wish to prepare a pipeline from some well, $w : W$, to some sink, $s : S$, for flow.
    a) We assume that the underlying net is operational wrt. $w$ and $s$, that is, that there is a route, $r$, from $w$ to $s$.
    b) Now, an orderly action sequence for engaging route $r$ is to "work backwards", from $s$ to $w$
    c) setting encountered pumps to pumping and valves to open.

In this way the system is well-formed wrt. the desirable sppr, spvr, svpr and svvr properties. Finally, setting the pump adjacent to the (preceding) well starts the system.

```
value
  55   prepare_and_engage: W × S → N ~→ N
  55   prepare_and_engage(w,s)(n) ≡
  55a     let r:R • ⟨w⟩^r^⟨s⟩ ∈ routes(n) in
  55b     act_seq(⟨w⟩^r^⟨s⟩)(len⟨w⟩^r^⟨s⟩)(n) end
  55    pre ∃ r:R • ⟨w⟩^r^⟨s⟩ ∈ routes(n)
  55c  act_seq: R → Nat → N → N
  55c  act_seq(r)(i)(n) ≡
  55c     if i=1 then n else
  55c     case r(i) of
  55c       mkV(_)→
  55c         act_seq(r)(i−1)(vlv_to_op(r(i))(n)),
  55c       mkP(_)→
  55c         act_seq(r)(i−1)(to_pump(r(i))(n)),
  55c       _→act_seq(r)(i−1)(n)
  55c     end end
```

## 2.5 Connectors

46

The interface , that is, the possible "openings", between adjacent units have not been explored. Likewise the for the possible "openings" of "begin" or "end" units, that is, units not having their input(s), respectively their "output(s)" connected to anything, but left "exposed" to the environment. We now introduce a notion of connectors: abstractly you may think of connectors as concepts, and concretely as "fittings" with bolts and nuts, or "weldings", or "plates" inserted onto "begin" or "end" units.

47

### Connectors:

56. There are connectors and connectors have unique connector identifiers.
57. From a connector one can observe its unique connector identifier.
58. From a net one can observe all its connectors
59. and hence one can extract all its connector identifiers.
60. From a connector one can observe a pair of "optional" (distinct) unit identifiers:
    a) An optional unit identifier is

   b) either a unit identifier of some unit of the net
   c) or a ‘‘nil’’ “identifier”.

61. In an observed pair of “optional” (distinct) unit identifiers
   - there can not be two ‘‘nil’’ “identifiers”.
   - or the possibly two unit identifiers must be distinct

```
type
  56 K, KI
value
  57 obs_KI: K → KI
  58 obs_Ks: N → K-set
  59 xtr_KIS: N → KI-set
  59 xtr_KIs(n) ≡ {obs_KI(k)|k:K•k ∈ obs_Ks(n)}
type
  60 oUIp′ = (UI|{|nil|})×(UI|{|nil|})
  60 oUIp = {|ouip:oUIp′•wf_oUIp(ouip)|}
value
  60 obs_oUIp: K → oUIp
  61 wf_oUIp: oUIp′ → Bool
  61 wf_oUIp(uon,uon′) ≡
  61   uon=nil⇒uon′≠nil
  61  ∨ uon′=nil⇒uon≠nil ∨ uon≠uon′
```

### Connector Adjacency:

62. Under the assumption that a fork unit cannot be adjacent to a join unit
63. we impose the constraint that no two distinct connectors feature the same pair of actual (distinct) unit identifiers.
64. The first proper unit identifier of a pair of “optional” (distinct) unit identifiers must identify a unit of the net.
65. The second proper unit identifier of a pair of “optional” (distinct) unit identifiers must identify a unit of the net.

```
axiom
  62  ∀ n:N,u,u′:U•{u.u′}⊆obs_Us(n)∧adj(u,u′)
                        ⇒ ∼(is_F(u)∧is_J(u′))
63  ∀ k,k′:K•obs_KI(k)≠obs_KI(k′)⇒
          case (obs_oUIp(k),obs_oUIp(k′)) of
            ((nil,ui),(nil,ui′)) → ui≠ui′,
            ((nil,ui),(ui′,nil)) → false,
            ((ui,nil),(nil,ui′)) → false,
            ((ui,nil),(ui′,nil)) → ui≠ui′,
            _ → false
          end
  ∀ n:N,k:K•k ∈ obs_Ks(n) ⇒
      case obs_oUIp(k) of
  64      (ui,nil) → ∃UI(ui)(n)
  65      (nil,ui) → ∃UI(ui)(n)
  64-65   (ui,ui′) → ∃UI(ui)(n)∧∃UI(ui′)(n)
      end
value
  ∃UI: UI → N → Bool
  ∃UI(ui)(n) ≡ ∃ u:U•u ∈ obs_Us(n)∧obs_UI(u)=ui
```

## 2.6 A CSP Model of Pipelines 49

We recapitulate Sect. 2.5 — now adding connectors to our model:

### Connectors: Preparation for Channels:

48

66. From an oil pipeline system one can observe units and connectors.
67. Units are either well, or pipe, or pump, or valve, or join, or fork or sink units.
68. Units and connectors have unique identifiers.
69. From a connector one can observe the ordered pair of the identity of the two from-, respectively to-units that the connector connects.

```
type
66 OPLS, U, K
68 UI, KI
value
66 obs_Us: OPLS → U-set
66 obs_Ks: OPLS → K-set
67 is_WeU, is_PiU, is_PuU, is_VaU,
67   is_JoU, is_FoU, is_SiU: U → Bool [mut. excl.]
68 obs_UI: U → UI, obs_KI: K → KI
69 obs_UIp: K → (UI|{nil}) × (UI|{nil})
```


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

50

Above, we think of the types OPLS, U, K, UI and KI as denoting semantic entities. Below, in the next section, we shall consider exactly the same types as denoting syntactic entities !

51

### CSP Behaviours, Channels, etc.:

70. There is given an oil pipeline system, opls.
71. To every unit we associate a CSP behaviour.
72. Units are indexed by their unique unit identifiers.
73. To every connector we associate a CSP channel. Channels are indexed by their unique "k"onnector identifiers.
74. Unit behaviours are cyclic and over the state of their (static and dynamic) attributes, represented by u.
75. Channels, in this model, have no state.
76. Unit behaviours communicate with neighbouring units — those with which they are connected.
77. Unit functions, $\mathcal{U}_i$, change the unit state.
78. The pipeline system is now the parallel composition of all the unit behaviours.

```
value
70 opls:OPLS
channel
73 {ch[obs_KI(k)]|k:K•k ∈ obs_Ks(opls)} M
value
78 pipeline_system: Unit → Unit
78 pipeline_system() ≡
71   ‖{unit(obs_UI(u))(u)|u:U•u ∈ obs_Us(opls)}
72 unit: ui:UI → U →
76       in,out {ch[obs_KI(k)]|k:K•k ∈ obs_Ks(opls)∧
76       let (ui′,ui″)=obs_UIp(k) in
76       ui ∈{ui′,ui″}\{nil} end} Unit
74 unit(ui)(u) ≡ let u′ = 𝒰ᵢ(ui)(u) in unit(ui)(u′) end
77 𝒰ᵢ: ui:UI → U →
77       in,out {ch[obs_KI(k)]|k:K•k ∈ obs_Ks(opls)∧
77       let (ui′,ui″)=obs_UIp(k) in
77       ui ∈{ui′,ui″}\{nil} end} Unit
```

# 3 Issues of Domains and Software Engineering

53

## 3.1 Domain Description Observations

The domain model of the previous section was supposed to have been read in a hasty manner, one which emphasised what the formulas were intended to model, rather than going into any details on modelling choice and notation.

What can we conclude from such a hastily read example ?

### 3.1.1 Syntax

54

We describe and formalise some of the **syntax** of nets of pipeline units: not the syntactical, physical design of units, but the conceptual “abstract structure” of nets. how units are connected, and notions like routes and special property routes.

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

### 3.1.2 Semantics 55

We hint at and formalise some of the **semantics** of nets of pipeline units, not a "full" semantics, just "bits and pieces": the flow of liquids (oil) or gasses (has), the opening and closing of valves, the pumping or not pumping of pumps, and how all of these opened or closed valves and pumping or not pumping pumps conceptually interact, concurrently, with other units.

### 3.1.3 Domain Laws 56

We also hint at some **laws** that pipelines must satisfy. Laws of physical systems (such as pipelines) are properties that hold irrespectively of how we model these systems. They are, for physical systems, "laws of nature". For financial service systems, such as the branch offices of a bank, a law could be:

> The amount of cash **in** the bank immediately before the branch office opens in the morning (for any day) **minus** the amount of cash withdrawn from the branch during its opening hours (that day) **plus** the amount of cash deposited into the branch during its opening hours (that day) **equals** the amount of cash in the bank immediately after the branch office closes for the day !

This law holds even though the branch office staff steals money from the bank or criminals robs the bank. The law is broken if (someone in) the bank prints money !

### 3.1.4 Description Ontology 57

The pipeline description focuses on **entities** such as the **composite entity**, the pipeline net, formed, as we have treated them in this model, from **atomic entities** such as forks, joins, pipes, pumps, valves and wells; **operations** such as opening and closing valves, setting pumps to pump and resetting them to not pump, etc.; **events**, not illustrated in this model, but otherwise such as a pipe exploding, that is, leaking more than acceptable, etc.; and **behaviours** — which are only hinted at in the CSP model of nets. Where nets were composite so is the net process: composed from "atomic" unit processes, all cyclic, that is, never-ending.

### 3.1.5 Modelling Composite Entities 58

We have **not modelled** pipeline nets **as** the **graphs**, as they are normally seen, using standard mathematical models of graphs. Instead we have made use of the uniqueness of units, hence of unit identifiers, to endow any unit with the observable attributes of the other units to which they are connected. We shall later, in Sect. 4 on page 18 [Mereology], comment on how we utilise the concept of unique identifiers of entities (such as pipeline units) to abstractly model how such system components form parts of wholes (including parts of parts).


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

## 3.2 Domain Modelling 59

Physicists model Mother Nature, that is, such natural science phenomena such as classical mechanics, thermodynamics, relativity and quantum mechanics. And physicists rely on mathematics to express their models and to help them predict or discover properties of
60 Mother Nature.

Physicists research physics, classically, with the sôle intention of understanding, that is, not for the sake of constructing new mechanical, thermodynamical, nuclear, or other
61 gadgets.

Software engineers now study domains, such as air traffic, banking, health care, pipelines, etc. for the sake of creating software requirements from which to create software.

## 3.3 Current and Possible Practices of Software Development 62

### 3.3.1 Todays Common, Commercial Software Development

A vast majority of todays practice lets software development (2) start with UML-like software design specifications, (3) followed by a "miraculous" stage of overall code design, and (4) ending with coding — with basically no serious requirements prescription and no attempts to show that (3) relates to (2) and (4) to (3) ! 40 years of Hoare Logics has had basically no effect. Hoare Logics may be taught at universities, but !?

### 3.3.2 Todays "Capability Maturity Model" Software Development 63

In "a few hundred" software houses software development (1) starts with more proper, still UML-like, but now requirements prescription, (2) continues with more concrete UML-like software design specifications, (3) still followed by a "miraculous" stage of overall code design, (4) and ending with coding — with basically all these (1–4) phases being process assessed and process improved [29] based on rather extensive, cross-correlated documents and more-or-less systematic tests.

### 3.3.3 Todays Professional Software Development 64

In "a few dozen" software houses software development phases and stages within (1–4) above are pursued (a) in a systematic (b) or a rigorous (c) or a formal manner and (a) where specifications of (1–4) are also formalised, where properties of individual stages (b-c) are expressed and (b) sometimes or (c) or always proved or model-checked or formally tested, and where correctness of relations between phases (1↔2, 2↔3 and 3↔4) are likewise expressed etc. (b–c–d) ! Now 40 years of computing science is starting to pay off, but only for such a small fraction of the industry !

## 3.4 Tomorrows Software Development 65

### 3.4.1 The Triptych Dogma

The dogma expresses that before software can be designed we must have a robust understanding of the requirements; and before requirements can be prescribed we must have a robust understanding of the domain.

An "ideal" consequence of the dogma is that software development is pursued in three phases: first (0) one of domain engineering, then (1) one of requirements engineering and finally (2–4) one of software design.

### 3.4.2 Triptych Software Development 66

In **domain engineering** (i) we liaise with clearly identified groups of all relevant **domain stakeholders,** far more groups and far more liaison that you can imagine; (ii) **acquiring** and **analysing** knowledge about the **domain**; (iii) creating a **domain terminology**; (iv) rough-describing the **business processes**; (v) **describing:** narratively and formally, "the" **domain**; (vi) **verifying** (**proving, model checking, formally testing**) **properties** (laws etc.) about the described domain; (vi) **validating** the domain description; and, all along, (vii) creating a **domain theory** — all this in iterative stages and steps 67

In **requirements engineering** we (i) **"derive"**, with clearly identified groups of all relevant requirements stakeholders, domain, interface and machine requirements; (ii) rough-describing the **re-engineered business processes**; (iii) creating a **domain terminology**; (iv) **prescribing:** narratively and formally, "the" **requirements** (based on the "derivations"); (v) **verifying** (**proving, model checking, formally testing**) **properties** (laws etc.) about the prescribed requirements; and thus (vi) establishing the **feasibility** and **satisfiability** of the requirements — all this in iterative stages and steps, sometimes bridging back to domain engineering. 68

In **software design** we **refine**, in stages of increasing concretisation, the requirements prescription into **components** and **modules** — while **model-checking, formally testing** and **proving correctness** of refinements as well as properties of components and modules. 69

Thus **formal specifications**, phases, stages and steps of **refinement**, **formal tests, model checks**, and **proofs** characterise tomorrows software development.

A few companies are doing just this: **Altran Praxis** (UK) — throughout all projects; **Chess Consulting** (NL), — consulting on formal methods; **Clearsy** Systems Engineering (F) — throughout most projects; **CSK Systems** (J) — in some, leading edge projects; **ISPRAS** (RU) — in some projects; and **Microsoft** (US) — in a few projects.

But none of them are, as yet, including **domain engineering.**

### 3.4.3 Justification 70

How can we then argue that domain engineering is a must ? We do so in three ways.


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

### The Right Software and Software That Is Right

First we must make sure that the customers get the right software. A thorough study of the domain and a systematic "derivation" of requirements from the domain description are claimed to lead to software that meets customers' expectations.

71

Then we must make sure that the software is right. We claim that carefully expressed and analysed specifications, of domains, of requirements and of software designs, together with formal verifications, model checks and tests — all based also on formalisations — will result in significantly less error-prone software.

### Professional Engineering

72

Classical engineering is based on the natural sciences and proceeds on the basis of their engineers having a deep grasp of those sciences.

Aeronautical engineers have deep insight into aerodynamics and celestial mechanics and understands and exploits their mathematical models.

73

Mobile radio-telephony engineers understands Maxwell's equations and can "massage" these while designing new Mobile telephony radio towers.

Control engineers designing automation for paper mills, power plants, cement factories, etc., are well-versed in stochastic and adaptive control theories and rely on these to design optimal systems.

Practicing software engineers, in responsible software houses, must now specialise in domain-specific developments — documented domain models become corporate assets — and are increasingly forced to formalise these models.

# 4 A Specification Ontology and Epistemology[3]

## 4.1 A Twofold Problem

The twofold problem is the following. (1) There is a dichotomy between what an informal description, a narrative, and what a formal description, a formalisation, refers to. (2) And which are our means of description?

74

Problem (1) is indicated in Fig. 3 on the facing page. A narrative, which is expressed in some national, that is, informal language, designates some (i.e., "the") domain. A formalisation (supposedly of some (i.e., "the") domain) denotes a mathematical model. One can claim, as we shall, that a formalisation designates a number of narratives, including

[3]Webster Collegiate Dictionary explanation of philosophical terms:

(i–ii) Ontology: (i) a branch of metaphysics concerned with the nature and relations of being; (ii) a particular theory about the nature of being or the kinds of things that have.

(iii) Epistemology: the study or a theory of the nature and grounds of knowledge especially with reference to its limits and validity

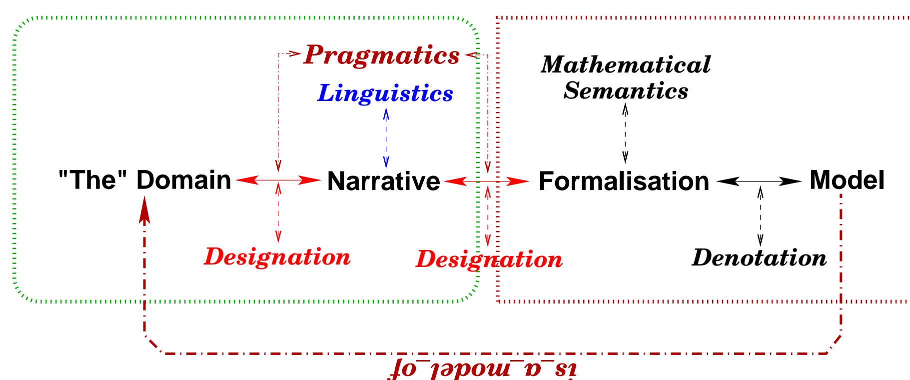


Figure 3: Formal and Informal Relations

"the one" given. One can "narrate" a formalisation. The "beauty" of a formalisation, if any, is its ability to "inspire" designated formalisations that are worth reading, that exhibit clear a clear, didactic understanding of the domain, and is pedagogical, that is, introduces domain phenomena and notions, gently, one-by-one. (1.1) In Sect. 2 on page 3 we presented 75 a pair of descriptions — purportedly — of a domain of (a class of) pipeline systems. In the narrative part of the pair such terms as type, value, entity, function and behaviour were intended to "point to", to refer to phenomena and concepts of the domain, that is, to sets of such phenomena and concepts formed, basically, from these phenomena. (1.2) 76 In the formal part of the pair of domain descriptions the corresponding, formal textual phrases, i.e., the syntactic sentences of the formulas, have a semantics, and that semantics, as in RSL, is usually, for formal specifications, some mathematical structures. Therein, 77 in (1.1–1.2), lies the first problem: The relation between narrative texts and the domain, "the real world", can only be an informal one. The relation between the formal texts and their semantics is a formal one. *"And never the twain shall meet!"*[4] One can not establish a formal relation between the informal world of domains and the formal world of mathematical specifications. **The above is an essence of abstract models.** 78

(2.) Then we outline the "describability" problem. **What, of actual domains, can be modelled ?** That is, what, of a[ny] domain, on one hand, can be narrated, all or some ? and on the other hand, can be formalised, all or some ? This double question is one of ontology and of epistemology. This section shall discuss issues related to Item (i–ii) of Footnote 3 on the facing page.

#### 4.1.1 Russell's Problem 79

Bertrand Russell, in [50] and in several other writings, brought the following example (abbreviated): "The present King of France". At present there is no designation in any

[4]Rudyard Kipling, *Barrack-room Ballads*, 1892: *"Oh, East is East, and West is West, and never the twain shall meet."*


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

domain of such a person. Thus the sentence does not make sense. In a formalisation we would express this as:

**type**
  DOMAIN, DESIGNATION
**value**
  obs_DESIGNATIONs: DOMAIN → DESIGNATION-**infset**
  designation: Sentence → DOMAIN $\stackrel{\sim}{\rightarrow}$ DESIGNATION
  existance: Sentence → DOMANI → **Bool**
  designation(s)($\omega$) ≡ **if** existance(s)($\omega$) **then** ... **else chaos end**

I claim that Russell's "problem" lies in the **green** dashed [rounded edge] box of the left side of Fig. 3 on the previous page. The simpler-minded computer/computing science problem of syntax and semantics is then that of (the **brown** rectangular box of) the right side of Fig. 3 on the preceding page.

### 4.1.2 The Problem of This Paper 80

The problem of this paper should now be a little more clear: It is that of "reconciling" of what is indicated by the two "boxes" of Fig. 3 on the previous page: the classical epistemological universe of discourse of the left side of that figure, and the domain science universe of discourse of the right side of that figure.

Put in different terms: We would somehow like to establish that the horisontal, non-dashed double arrows expresses that the model (to the extreme right) *"is a model"* of the domain (to the extreme left).

## 4.2 What Is An Ontology ? 81

### 4.2.1 Some Remarks

By a specification ontology we shall understand a set of mathematical concepts to be used in specifying "something". By a domain description ontology we shall understand a set of concepts to be used in describing a domain. We shall choose a textual, rather than a diagrammatic (graphical) form for expressing descriptions. The description ontology therefore hinges on a number of textual (i.e., non-diagrammatic) syntactic constructs. These 82 will be covered in Sects. 4.3–4.6 and 4.8–4.9. The pragmatics of description designations (using these syntactic constructs) are, of course, phenomena of the domain. The semantics of description designations are, of course, mathematical constructs. Thus we have a duality here: On one hand we have that the pragmatics, that is, the intention of our use of descriptions, is that they designate phenomena of an actual world, whereas, on the other hand, the semantics, that is, the formal meaning of descriptions, is that they denote mathematical concepts.

Is there a problem here ?

83 And: what can we describe ?

Yes, there is a "slight problem"? We cannot, in fact never, establish a formal relationship between the pragmatics and the semantics.

And: On one hand there is a world, that is, a set of domains, to be described. And, on the other hand, there are some syntactic constructs that can be used in providing such descriptions. Are the latter sufficient to describe all that we wish to describe? Well, we shall never know. So it is a conjecture, not a theory of description ontologies. 84

The above remarks, naturally, influence the rest of this more "theoretical/philosophical" part of the paper.

That is, we thus hint at, but do not present, a more thorough philosophy of science reasoning, arguments that should support our description ontology.

### 4.2.2 Description Ontology Versus Ontology Description 85

According to Wikipedia: *Ontology is the philosophical study of* (i) *the nature of being, existence or reality in general,* (ii) *as well as of the basic categories of being and their relations.*

Section 3.1.4 emphasized the need for describing domain phenomena and concepts. This section puts forward a description ontology — (i) *which "natures of being, existence or reality"* and (ii) *which "categories of being and their relations"* — that we shall apply in the description of domain phenomena and concepts. 86

Yes, we do know that the term 'description ontology' can easily be confused with 'ontology description' — a term used very much in two computing related communities: AI (artificial intelligence) and WWW (World Wide Web). These communities use the term 'ontology' as we use the term 'domain' [3, 13, 15, 22, 23, 24, 25, 52].

By [domain] 'description ontology' we shall mean a set of notions that are used in describing a domain. So the ontology is one of the description language not of the domain that is being described.

## 4.3 Categories, Predicates and Observers for Describing Domains 87

It is not the purpose of this paper to motivate the categories, predicates and observer functions for describing phenomena and concepts. This is done elsewhere [4, 5, 7, 8, 9]. Instead we shall more-or-less postulate one approach to the analysis of domains. We do so by postulating a number of meta-categories, meta-predicates and meta-observer functions. They characterise those non-meta categories, predicates and observer functions that the domain engineer cum researcher is suggested to make use of. There may be other approaches [51, John Sowa, 1999] than the one put forward in this paper.

### 4.3.1 An Aside on Notation 88

In this entire section we shall be using two kinds of notation. Both may look like uses of RSL, but they are not. A notation which involves the use of $\mathbb{THIS\ FONT}$. And a notation which, in some form of mathematics, explain the former.


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

Please note that these meta-functions, those "partially spelled" with $\mathbb{THIS\ FONT}$ are not RSL functions but are mental functions applied by the domain modeller in the analysis of domains.

### 4.3.2 The Hypothetical Nature of Categories, Predicates and Observers 89

In the following we shall postulate some categories of phenomena[5], that is, some meta-types:

**categories**
$\mathbb{ALPHA}, \mathbb{BETA}, \mathbb{GAMMA}$

What such a clause as the above means is that we postulate that there are the categories $\mathbb{ALPHA}, \mathbb{BETA}, \mathbb{GAMMA}$ of "things" (phenomena and concepts) in the world of domains. 90 That is, there is no proof that such "things" exists. It is just our way of modelling domains. If that way is acceptable to other domain science researchers or domain engineers, fine. In the end, which we shall never reach, those aspects of a, or the domain science, may "survive". If not, well, then they will not "survive" !

### 4.3.3 Predicates and Observers 91

With the categories just introduced we then go on to postulate some predicate and observer functions. For example:

**predicate signatures**
is_$\mathbb{ALPHA}$: "Things" $\rightarrow$ **Bool**
is_$\mathbb{BETA}$: "Things" $\rightarrow$ **Bool**
is_$\mathbb{GAMMA}$: "Things" $\rightarrow$ **Bool**
**observer signatures**
obs_$\mathbb{ALPHA}$: "Things" $\stackrel{\sim}{\rightarrow}$ $\mathbb{ALPHA}$
obs_$\mathbb{BETA}$: $\mathbb{ALPHA}$ $\stackrel{\sim}{\rightarrow}$ $\mathbb{BETA}$
obs_$\mathbb{GAMMA}$: $\mathbb{ALPHA}$ $\stackrel{\sim}{\rightarrow}$ $\mathbb{GAMMA}$

92 So we are "fixing" a logic !

The "Things" clause is a reference to the domain under scrutiny. Some 'things' in that domain are of category[6] $\mathbb{ALPHA}$, or $\mathbb{BETA}$, or $\mathbb{GAMMA}$. Some are not. It is then postulated that from such things of category $\mathbb{ALPHA}$ one can observe things of categories $\mathbb{BETA}$ or $\mathbb{GAMMA}$. Whether this is indeed the case, i.e., that one can observe these things is a matter of conjecture, not of proof.

[5]These observable phenomena or abstract concepts are, in general, entities, more specifically either simple entities, actions, events or behaviours.

[6]We use the term 'category' in lieu of either of the term 'type', 'class', 'set'. By here using the term 'category' we do not mean category in the mathematical sense of category theory.

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark 

### 4.3.4 Uncertainty 93

The function signature:

**value**
fct: A $\xrightarrow{\Re}$B

expresses a relation, $\mathsf{fct}_{\Re}$, which one may think of as:

**type**
$\mathrm{FCT}_{\Re} = (\mathrm{A} \times \mathrm{B})$**-infset**.

Applying fct to an argument a, that is, f(a), may then either "always" result in some specific b, in which case fct is a function; or result in **chaos**, that is, fct is not defined for that argument a, that is: $\mathsf{a} \notin \mathsf{fct}_{\Re}$; or sometimes result in some b, sometimes in another b′, etc., that is, $\mathsf{fct}_{\Re} = \{(\mathsf{a},\mathsf{b}), (\mathsf{a},\mathsf{b}'), (\mathsf{a},\mathsf{b}''), \ldots\}$.

### 4.3.5 Meta-Conditions 94

Finally we may sometimes postulate the existence of a meta-axiom:

**meta condition:**
Predicates over $\mathbb{ALPHA}, \mathbb{BETA}$ and $\mathbb{GAMMA}$

Again, the promulgation of such logical meta-expressions are just conjectures, not the expression of "eternal" truths.

### 4.3.6 Discussion 95

So, all in all, we suggest four kinds of meta-notions:

- categories,
- is_Category and obs_Property predicates,
- obs_Category and obs_Attribute observers, and
- meta-conditions, i.e., axioms.

The **category [type]** A, B, ..., is_A, is_B, ... obs_A, obs_B, ... **meta-condition [axiom]** predicate notions derive from McCarthy's *analytic syntax* [35].

**Discussion:** Thus the formal specification and the high level programming languages' use, that is, the software designers' use of **type** clauses, predicate functions and observer (in the form of selector) functions shall be seen in the context of specifications, respectively program code dealing with computable quantities and decomposing and constructing such quantities.


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

The proposal here, of suggesting that the domain engineer cum researcher makes us of **categories, predicates, observers** and **meta-conditions** is different. In domain descriptions an existing "universe of discourse" is being analysed. Perhaps the **categories, predicates, observers** and **meta-conditions** makes sense, perhaps the domain engineer cum researcher can use these descriptional "devices" to "compose" a consistent and relative complete "picture", i.e., description, of the domain under investigation.

Either the software designers' use of formal specification or programming language constructs is right or it is wrong, but the domain engineer cum researchers' use is just an attempt, a conjecture. If the resulting domain description is inconsistent, then it is wrong. But it can never be proven right. Right in the sense that it is **the** right description. As in physics, it is just a conjecture. There may be refutations of domain models.

## 4.4 Entities 96

What we shall describe is what we shall refer to as entities. In other words, there is a category and meta-logical predicate $\mathbb{ENTITY}$, is_$\mathbb{ENTITY}$. The is_$\mathbb{ENTITY}$ predicate applies to "whatever" in the domain, whether an entity or not, and "decides", i.e., is postulated to analyse whether that "thing" is an entity or not:

**predicate signature:**
  is_$\mathbb{ENTITY}$: "Thing" → **Bool**
**meta condition:**
  ∀ e:$\mathbb{ENTITY}$ • is_$\mathbb{ENTITY}$(e)

**Discussion:** When we say "things", or entities, others may say 'individuals', 'objects', or use other terms.

The meta-predicate is_$\mathbb{ENTITY}$ provides a rather "sweeping" notion, namely that someone, the domain engineer, an oracle or other, can decide whether "something" is to be described as a phenomenon or concept of the domain.

97

• • •

By introducing the predicate is_$\mathbb{ENTITY}$ we have put the finger on what this section is all about, namely *"what exists ?"* and *"what can be described ?"* We are postulating a description ontology. It may not be an adequate one. It may have flaws. But, for the purposes of raising some issues of epistemological and ontological nature, it is adequate.

### 4.4.1 Entity Categories 98

We postulate four entity categories:

**category:**
  $\mathbb{SIMPLE_ENTITY}$, $\mathbb{ACTION}$, $\mathbb{EVENT}$, $\mathbb{BEHAVIOUR}$

Some **phenomena** or **concepts** are simple entities. **Simple entity phenomena** are the things we can point to, touch and see. They are manifest. Other phenomena, for example those we can hear, smell, taste, or measure by physics (including chemistry) apparatus are properties (attributes) of simple entity phenomena. **Concepts** are abstractions about phenomena and/or other concepts. 99

A subset of simple domain entities form a **state**. **Actions** are the result of applying functions to simple domain entities and changing the **state**. What is changed are the attribute values of simple (state) entities. Actions are observable through the observation of the occurrence of the 'before' and 'after' states. The functions or relations that relate before and after states are not observable. They are our way of "explaining" the actions. If you wish to consider them as simple entities then they are atomic have no name, but do have a type, the function signature. Actions are caused by domain behaviours. 100

**Events** are **state** changes that satisfy a predicate on the 'before' and 'after states'. Events are observable through their "taking place", that is, by observing, as if they were actions, their 'before' and 'after states'. but also by, somehow, observing that they are not caused by domain behaviours, well, possibly then by behaviours "outside" the domain being considered. The above represents a "narrow" concept of events. A less narrow concept would characterise some domain actions as events; we might call them "interesting" action events. 101

**Behaviours** are sets of sequences (of sets of) actions and events. Behaviours are observable — through the observation of the constituent actions and events.

Below we shall have "much more" to say about these four categories of entities. 102

**category:**

$\mathbb{ENTITY} = \mathbb{SIMPLE_ENTITY} \cup \mathbb{ACTION} \cup \mathbb{EVENT} \cup \mathbb{BEHAVIOUR}$

**Discussion:** The four categories of entities may overlap.

With each of the four categories there is a predicate:

**predicate signature:**

is_$\mathbb{SIMPLE_ENTITY}$ "Thing" $\xrightarrow{\Re}$ **Bool**
is_$\mathbb{ACTION}$ "Thing" $\xrightarrow{\Re}$ **Bool**
is_$\mathbb{EVENT}$ "Thing" $\xrightarrow{\Re}$ **Bool**
is_$\mathbb{BEHAVIOUR}$ "Thing" $\xrightarrow{\Re}$ **Bool**

Each of the above four predicates require that their argument t:"Thing" satisfies:

is_$\mathbb{ENTITY}$(t)

The use of $\xrightarrow{\Re}$ shall illustrate the uncertainty that may befell the domain modeller. We shall henceforth "boldly" postulate functionality, i.e., $\rightarrow$, of the is_$\mathbb{SIMPLE_ENTITY}$, is_$\mathbb{ACTION}$, is_$\mathbb{EVENT}$ and is_$\mathbb{BEHAVIOUR}$ functions. 103

The $\cup$ "union" is inclusive:


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

**meta condition:**
∀ t:"Thing"•is_ENTITY(t) ⇒
is_SIMPLE_ENTITY(t) ∨ is_ACTION(t) ∨ is_EVENT(t) ∨ is_BEHAVIOUR(t)

## 4.5 Simple Entities 104

We postulate that there are **atomic** simple entities, that there are [therefrom distinct] **composite** simple entities, and that a simple entity is indeed either atomic or composite. That atomic simple entities cannot meaningfully be described as consisting of proper other simple entities, but that composite simple entities indeed do consist of proper other simple entities. It is us, the observers, who decide to abstract a simple entity as either being atomic or as being composite.

105

That is:

**category:**
SIMPLE_ENTITY = ATOMIC ∪ COMPOSITE
**observer signature:**
is_ATOMIC: SIMPLE_ENTITY → **Bool**
is_COMPOSITE: SIMPLE_ENTITY → **Bool**
**meta condition:**
ATOMIC ∩ COMPOSITE = {}
∀ s: "Things":SIMPLE_ENTITY •
is_ATOMIC(s) ≡ ∼is_COMPOSITE(s)

**Discussion:** We put in brackets, in the text paragraph before the above formulas, [therefrom distinct]. One may very well discuss this constraint — are there simple entities that are both atomic and composite ? — and that is done by Bertrand Russell in his 'Philosophy of Logical Atomism' [50].

### 4.5.1 Discrete and Continuous Entities 106

We postulate two forms of SIMPLE_ENTITIES: DISCRETE, such as a railroad net, a bank, a pipeline pump, and a securities instrument, and CONTINUOUS, such as oil and gas, coal and iron ore, and beer and wine.

**category:**
SIMPLE_ENTITY = DISCRETE_SIMPLE_ENTITY ∪ CONTINUOUS_SIMPLE_ENTITY
**predicate signatures:**
is_DISCRETE_SIMPLE_ENTITY: SIMPLE_ENTITY → **Bool**
is_CONTINUOUS_SIMPLE_ENTITY: SIMPLE_ENTITY → **Bool**
**meta condition:**
[ is it desirable to impose the following ]
∀ s:SIMPLE_ENTITY •
is_DISCRETE_SIMPLE_ENTITY(s) ≡ ∼CONTINUOUS_SIMPLE_ENTITY(s) ?

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark 

**Discussion:** In the last lines above we raise the question whether it is ontologically possible or desirable to be able to have simple entities which are both discrete and continuous. Maybe we should, instead, express an axiom which dictates that every simple entity is at least of one of these two forms.

### 4.5.2 Attributes 107

Simple entities are characterised by their attributes: attributes have name, are of type and has some value; no two (otherwise distinct) attributes of a simple entity has the same name.

**category:**
  $\mathbb{ATTRIBUTE}, \mathbb{NAME}, \mathbb{TYPE}, \mathbb{VALUE}$
**observer signature:**
  obs_$\mathbb{ATTRIBUTE}$s: $\mathbb{SIMPLE_ENTITY} \rightarrow \mathbb{ATTRIBUTE}$-**set**
  obs_$\mathbb{NAME}$: $\mathbb{ATTRIBUTE} \rightarrow \mathbb{NAME}$
  obs_$\mathbb{TYPE}$: $\mathbb{ATTRIBUTE} \rightarrow \mathbb{TYPE}$
  obs_$\mathbb{VALUE}$: $\mathbb{ATTRIBUTE} \rightarrow \mathbb{VALUE}$
**meta condition:**
  $\forall$ s:$\mathbb{SIMPLE_ENTITY}$ •
    $\forall$ a,a′:$\mathbb{ATTRIBUTE}$ • {a,a′}⊆obs_$\mathbb{ATTRIBUTE}$s(s)
     $\wedge$ a≠a′ $\Rightarrow$ obs_$\mathbb{NAME}$(a)≠obs_$\mathbb{NAME}$(a′)

108

Examples of attributes of atomic simple entities are: (i) A pipeline pump usually has the following attributes: `maximum pumping capacity, current pumping capacity, whether for oil or gas, diameter (of pipes to which the valve connects)`, etc. (ii) Attributes of a person usually includes `name, gender, birth date, central registration number, address, marital state, nationality`, etc. 109

Examples of attributes of composite simple entities are: (iii) A railway system usually has the following attributes: `name of system, name of geographic areas of location of rail nets and stations, whether a public or a private company, whether fully, partly or not electrified`, etc. (iv) Attributes of a bank usually includes: `name of bank, name of geographic areas of location of bank branch offices, whether a commercial portfolio bank or a high street, i.e., demand/deposit bank`, etc.

We do not further define what we mean by attribute names, types and values. Instead we refer to [37, Properties] and [20, Properties, Types and Meaning] for philosophical discourses on attributes.

### 4.5.3 Atomic Simple Entities: Attributes 110

Atomic simple entities are characterised only by their attributes.
**Discussion:** We shall later cover a notion of domain actions, that is functions being applied to entities, including simple entities. We do not, as some do for programming languages, "lump" entities and functions (etc.) into what is there called 'objects'.

 © Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

### 4.5.4 Composite Simple Entities: Attributes, Sub-entities and Mereology

Composite simple entities are characterised by three properties: (i) their **attributes**, (ii) a proper set of one or more **sub-entities** (which are simple entities) and (iii) a **mereology** of these latter, that is, how they relate to one another, i.e., how they are composed.

#### Sub-entities 111

Proper sub-entities, that is simple entities properly contained, as immediate **parts** of a composite simple entity, can be observed (i.e., can be postulated to be observable):

**observer signature:**
obs_SIMPLE_ENTITIES: COMPOSITE → SIMPLE_ENTITY-**set**

#### Mereology 112

Mereology is the theory of **part-hood** relations: of the relations of part to whole and the relations of **part** to **part** within a **whole**. Suffice it to suggest some mereological structures:

- **Set Mereology:** The individual sub-entities of a composite entity are "un-ordered" like elements of a set. The obs_SIMPLE_ENTITIES function yields the set elements.

  **predicate signature:**
  is_SET: COMPOSITE → **Bool**

113

- **Cartesian Mereology:** The individual sub-entities of a composite entity are "ordered" like elements of a Cartesian (grouping). The function obs_ARITY yields the arity, 2 or more, of the simple Cartesian entity. The function obs_CARTESIAN yields the Cartesian composite simple entity.

  **predicate signature:**
  is_CARTESIAN: COMPOSITE → **Bool**
  **observer signatures:**
  obs_ARITY: COMPOSITE $\stackrel{\sim}{\to}$ **Nat**
  **pre**: obs_ARITY(s): is_CARTESIAN(s)
  obs_CARTESIAN: COMPOSITE $\stackrel{\sim}{\to}$
  SIMPLE_ENTITY × ... × SIMPLE_ENTITY
  **pre** obs_CARTESIAN(s): is_CARTESIAN(s)

114

**meta condition:**
∀ c:SIMPLE_ENTITY•
is_COMPOSITE(c)∧is_CARTESIAN(c) ⇒
obs_SIMPLE_ENTITIES(c) = **elements of** obs_CARTESIAN(c)
∧ **cardinality of** obs_SIMPLE_ENTITIES(c) = obs_ARITY(c)

We just postulate the **elements of** and the **cardinality of** meta-functions. Although one may have that distinct(ly positioned) elements of a Cartesian to be of the same type and even the same value, they will be distinct — with distinctness "deriving" from their distinct positions. 115

- **List Mereology:** The individual sub-entities of a composite entity are "ordered" like elements of a list (i.e., a sequence). Where Cartesians are fixed arity sequences, lists are variable length sequences.

**predicate signature:**
is_LIST: COMPOSITE → **Bool**
**observer signatures:**
obs_LIST: COMPOSITE $\stackrel{\sim}{\to}$ **list of** SIMPLE_ENTITY
**pre** obs_LIST(s): is_LIST(s)
obs_LENGTH: COMPOSITE $\stackrel{\sim}{\to}$ **Nat**
**pre** obs_LENGTH(s): is_LIST(s)

116

**meta condition:** ∧
∀ s:SIMPLE_ENTITY•
is_COMPOSITE(s)∧is_LIST(s) ⇒
obs_SIMPLE_ENTITIES(s) = **elements of** obs_LIST(s) ∧
**cardinality of elements of** obs_LIST(s) = LENGTH(s) ∧
∀ e,e′ • {e,e′} ⊆ **elements of** ⇒
obs_LIST(s) ⇒ obs_TYPE(e)=obs_TYPE(e′)

We also just postulate the **list of, elements of** and the **cardinality of** meta-functions. Again, as for Cartesians, we shall postulate that although distinct elements of a list (which are all of the same type) may have the same value — they are distinct due to their distinct position in the list, that is, their adjacency to a possible immediately previous and to a possibly immediately following element. 117

- **Graph Mereology:** The individual sub-entities of a composite entity are "ordered" like elements of a graph, i.e., a net, of elements. Trees, lattices, cycles and other structures are just special cases of graphs. Any (immediate) sub-entity of a composite simple entity of GRAPH mereology may be related to any number


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

of (not necessarily other) (immediate) sub-entities of that same composite simple entity GRAPH in a number of ways:it may immediately PRECEDE, or immediate SUCCEED or be BIDIRECTIONALLY_LINKED with these (immediate) sub-entities of that same composite simple entity. In the latter case some sub-entities PRECEDE a SIMPLE_ENTITY of the GRAPH, some sub-entities SUCCEED a SIMPLE_ENTITY of the GRAPH, some both.

118

```
predicate signature:
    is_GRAPH: COMPOSITE → Bool
observer signatures:
    obs_GRAPH: COMPOSITE ~→ GRAPH
              pre obs_GRAPH(g): is_GRAPH(g)
    obs_PRECEEDING_SIMPLE_ENTITIES:
          COMPOSITE × SIMPLE_ENTITY → SIMPLE_ENTITY-set
              pre obs_PRECEEDING_SIMPLE_ENTITIES(c,s):
                       is_GRAPH(c) ∧ s ∈ obs_SIMPLE_ENTITIES(c)
    obs_SUCCEEDING_SIMPLE_ENTITIES:
          COMPOSITE × SIMPLE_ENTITY → SIMPLE_ENTITY-set
              pre obs_PRECEEDING_SIMPLE_ENTITIES(c,s):
                       is_GRAPH(c) ∧ s ∈ obs_SIMPLE_ENTITIES(c)
meta condition:
    ∀ c:SIMPLE_ENTITY • is_COMPOSITE(c) ∧ is_GRAPH(c)
          ⇒ let ss = SIMPLE_ENTITIES(c) in
              ∀ s′:SIMPLE_ENTITY • s′ ∈ ss
                 ⇒ obs_PRECEEDING_SIMPLE_ENTITIES(c)(s′) ⊆ ss
                   ∧ obs_SUCCEEDING_SIMPLE_ENTITIES(c)(s′) ⊆ ss
              end
```

119

### 4.5.5 Discussion

Given a "thing", $s$, which satisfies is_SIMPLE_ENTITY(s), the domain engineer can now systematically analyse this "thing" using any of the is_ATOMIC(s), is_COMPOSITE(s), is_SET(s), is_CARTESIAN(s), is_LIST(s), is_GRAPH(s), etcetera. predicates and using also the observer functions sketched above.

120

Given any SIMPLE_ENTITY the domain engineer can now analyse it to find out whether it is an ATOMIC or a COMPOSITE entity. An, in either case, the domain engineer can analyse it to find out about its ATTRIBUTES. If the SIMPLE_ENTITY is COMPOSITE then its SIMPLE_ENTITIES and their MEREOLOGY can be additionally ascertained. In summery: If ATOMIC then ATTRIBUTES can be analysed. If COMPOSITE then ATTRIBUTES, SIMPLE_ENTITIES and MEREOLOGY can be analysed.

121

Please note that these meta-functions, those "partially spelled" with THIS FONT, are not RSL functions but are mental functions applied, conceptually, i.e., "by the brain" of the domain modeller in the analysis of domains.

### 4.5.6 Practice 122

How do we interpret this section, Sect. 4.5, on simple entities? We practice it by analysing the domain according to the principles laid down in this section, Sect. 4.5, by discovering simple entities of the domain, by discovering and writing down, for example in RSL, their sorts (i.e., abstract types) or their concrete types, by possibly also discovering (postulating, conjecturing) and writing down constraints, that is axioms or well-formedness predicates over these sorts and types. That is, we are not using these "funny" names, such as SIMPLE_ENTITY, ATOMIC, 123 COMPOSITE, DISCRETE_SIMPLE_ENTITY, CONTINUOUS_SIMPLE_ENTITY, ATTRIBUTE, NAME, TYPE, VALUE, CARTESIAN, ARITY, LIST, LENGTH, GRAPH, PRECEDING_SIMPLE_E SUCCEEDING_SIMPLE_ENTITIES, etc., nor their is_ or obs_ forms. The example of Sect. 2 has given several examples of sort and type definitions as well as of constraint definitions.

## 4.6 Actions 124

### 4.6.1 Definition

By a STATE we mean a set of one or more SIMPLE_ENTITIES. By an ACTION we shall understand the application of a *FUNCTION* to (a set of, including the state of) SIMPLE_ENTITIES such that a STATE change occurs.

### 4.6.2 Non-Observables 125

The mathematical concept of a function, explained as "that thing which when applied to something(s) called its arguments yields (i.e., results) in something called its results that concept is an elusive one. No-one has ever "seen", "touched", "heard" or otherwise sensed a function. Functions are purely a mathematical construction, possessing a number of properties and introduced here in order to explain what is going on in a(ny) domain. We shall not be able to observe functions. To highlight this point we use this *SPELLING OF FUNCTIONS*.

### 4.6.3 Observers &c. 126

We postulate that the domain engineer can indeed decide, that is, conjecture, whether a "thing", which is an ENTITY is an ACTION.

**category:**
ACTION, *FUNCTION*, STATE
**predicate signature:**
is_ACTION: ENTITY → **Bool**


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

127 Given an ENTITY of category ACTION one can observe, i.e., conjecture the *FUNCTION* (being applied), the ARGUMENT CARTESIAN of SIMPLE_ENTITIES to which the *FUNCTION* is being applied, and the resulting change STATE change. Not all elements 128 of the CARTESIAN ARGUMENT are SIMPLE STATE ENTITIES.

**category:**
STATE = SIMPLE_ENTITY
*FUNCTION* = SIMPLE_ENTITY × STATE → STATE
ARGUMENT = {|s:SIMPLE_ENTITY•is_CARTESIAN(s)|}
**observer signatures:**
obs_ACTION: ENTITY → ACTION
obs_ARGUMENT: ACTION → ARGUMENT
obs_INPUT_STATE: ACTION → STATE
obs_RESULT_STATE: ACTION → STATE

### 4.6.4 Practice 129

How do we interpret this section, Sect. 4.6, on actions? We practice it by analysing the domain according to the principles laid down in this section, Sect. 4.6, by discovering actions of the domain, by discovering and writing down, for example in RSL, their signatures, by possibly also discovering (postulating, conjecturing) and writing down their definitions. That is, we are not using these "funny" names, such as ACTION, STATE, ARGUMENT, INPUT_STATE, $\mathrm{RESULT}_\mathrm{S}\mathrm{TATE}$ nor their is_ or obs_ forms. The earlier example of Sect. 2 has given several examples of action signatures and definitions.

## 4.7 "Half-way" Discussion[7] 130

Some pretty definite assertions were made above: We postulate that the domain engineer can indeed decide whether a "thing", which is an ENTITY is an ACTION And that one can observe the *FUNCTION*, the ARGUMENT and the RESULT of an ACTION. We do not really have to phrase it that deterministically. It is enough to say: One can speak of actions, functions, their arguments and their results. Ontologically we can do so. Whether, for any specific simple entity we can decide whether it is an actions is, in a sense, immaterial: we can always postulate that it is an action and then our analysis can be based on that hypothesis. This discussion applies `inter alia` to all of the entities being introduced here, together with their properties.

The domain engineer cum researcher can make such decisions as to whether an entity is a simple one, or an action, or an event or a behaviour. And from such a decision that domain engineer cum researcher can go on to make decisions as to whether a simple entity is discrete or continuous, and atomic or composite, and then onto a mereology for

[7] "Halfway": after simple entities and actions and before events and behaviours.

the composite simple entities. Similarly the domain engineer cum researcher can make decisions as to the function, arguments and results of an action. All these decisions does not necessarily represent the “truth”. They hopefully are not “falsities”. At best they are abstractions and, as such, they are approximations.

## 4.8 Events 131

Like we did for simple entities we distinguish between atomic composite events

### 4.8.1 Definition of Atomic Events 132

By an EVENT we shall understand A pair, $(\sigma, \sigma')$, of STATEs, a STIMULUS, $s$, (which is like a $\mathcal{FUNCTION}$ of an ACTION), and an EVENT PREDICATE, $p : \mathcal{P}$, such that $p(\sigma, \sigma')(s)$, yields **true**.

The difference between an ACTION and an EVENT is two things: the EVENT ACTION need not originate within the analysed DOMAIN, and the EVENT PREDICATE is trivially satisfied by most ACTIONs which originate within the analysed DOMAIN.

### 4.8.2 Examples of Atomic Events 133

Examples of atomic events, that is, of predicates are: a bank goes “bust” (e.g., looses all its monies, i.e., bankruptcy), a bank account becomes negative, (unexpected) stop of gas flow and iron ore mine depleted. Respective stimuli of these events could be: (massive) loan defaults, a bank client account is overdrawn, pipeline breakage, respectively over-mining.

### 4.8.3 Composite Events 134

#### Definition

A Composite event is composed from a sequence of two or more events: The ‘after’ state of one event becomes the ‘before’ event of the immediately subsequent event We

#### Examples of Composite Events 135

### 4.8.4 Observers &c. 136

We postulate that the domain engineer from an EVENT can observe the STIMULUS, the BEFORE_STATE, the AFTER_STATE and the EVENT_PREDICATE. As said before: the domain engineer cum researcher can decide on these abstractions, these approximations. 137

**category:**
  STIMULUS = SIMPLE_ENTITY $\times$ STATE $\rightarrow$ STATE
  $\mathcal{P}$ = STATE $\times$ STATE $\rightarrow$ **Bool**
**observer signatures:**
  obs_STIMULUS: EVENT $\rightarrow$ STIMULUS


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

obs_BEFORE_STATE: EVENT → STATE
obs_AFTER_STATE: EVENT → STATE
obs_EVENT_PREDICATE: EVENT → $\mathcal{P}$
**meta condition:**
∀ e:EVENT •
∃ s:STIMULUS •
INPUT_STATE(e) = BEFORE_STATE(s) ∧
RESULT_STATE(e) = AFTER_STATE(s) ∧
∃ $p$:$\mathcal{P}$ •$p$(s)(INPUT_STATE(e),RESULT_STATE(e))

### 4.8.5 Practice 138

How do we interpret this section, Sect. 4.8, on events? We practice it by analysing the domain according to the principles laid down in this section, Sect. 4.8, by discovering events of the domain, that is, by discovering and writing down, for example in RSL, the "defining" pre/post condition predicates, by discovering and writing down, for example in RSL, their signatures (as if they were domain actions), by possibly also discovering (postulating, conjecturing) and writing down their definitions (as if they were domain actions). That is, we are not using these "funny" names, such as EVENT, STIMULUS, EVENT_PREDICATE, BEFORE_-STATE AFTER_STATE nor their is_ or obs_ forms. The example of Sect. 2 has not given very many examples.

139

So we give some narrative examples, that is, without formalisations: an oil well runs dry (atomic event); a valve gets stuck in closed position, causing further events; a gas pipe breaks causing a fire causing gas to flow, etc. (composite event); etcetera.

## 4.9 Behaviours 140

### 4.9.1 A Loose Characterisation

By a BEHAVIOUR we shall understand a set of sequences of ACTIONs and EVENTs one sequence of which designates the behaviour of the environment the remaining sequences designate behaviours of different "overlapping" or "disjoint" parts of the domain. It may now be so that some EVENTs in two or more such sequences have their STATEs and PREDICATEs express, for example, mutually exclusive synchronisation and communication EVENTs between these sequences which are each to be considered as simple SEQUENTIAL_BEHAVIOURs. Other forms than mutually exclusive synchronisation and communication EVENTs, that "somehow link" two or more behaviours, can be identified.

141

### 4.9.2 Observers &c. 142

We abstract from the orderly example of synchronisation and communication given above and introduce a further un-explained notion of behaviour (synchronisation and communication) BEHAVIOUR INTERACTION LABELs and allow BEHAVIOURs to now just be

sets of sequences of ACTIONs and BEHAVIOUR INTERACTION LABELs. such that any one simple sequence has unique labels. 143

We can classify some BEHAVIOURs.

(i) SIMPLE SEQUENTIAL BEHAVIOURs are sequences of ACTIONs.

(ii) SIMPLE_CONCURRENT_BEHAVIOURs are sets of SIMPLE_SEQUENTIAL_BEHAVIOURs.

(iii) COMMUNICATING_CONCURRENT_BEHAVIOURs are sets of sequences of ACTIONs and BEHAVIOUR_INTERACTION_LABELs. We say that two or more such COMMUNICATING_CONCURRENT_BEHAVIOURs SYNCHRONISE_&_COMMUNICATE when all distinct BEHAVIOURs "sharing" a (same) label have all reached that label. 144

Many other composite behaviours can be observed. For our purposes it suffice with having just identified the above.

SIMPLE_ENTITIES, ACTIONs and EVENTs can be described without reference to time. BEHAVIOURs, in a sense, take place over time.[8] It will bring us into a rather long 145 discourse if we are to present some predicates, observer functions and axioms concerning behaviours — along the lines such predicates, observer functions and axioms were present, above, for SIMPLE_ENTITIES, ACTIONs and EVENTs. We refer instead to Johan van Benthem's seminal work on the `The Logic of Time` [53]. In addition, more generally, we refer to A.N. Prior's [43, 44, 45, 46, 42] and McTaggart's works [36, 17, 49]. The paper by Wayne D. Blizard [11] proposes an axiom system for time-space.

### 4.9.3 Practice 146

How do we interpret this section, Sect. 4.9, on behaviours? We practice it by analysing the domain according to the principles laid down in this section, Sect. 4.9, by discovering behaviours of the domain, that is, by discovering and writing down, for example in RSL, the signatures of these behaviours (e.g., as CSP processes), by possibly also discovering (postulating, conjecturing) and writing down their definitions (as sequences of domain actions and events — the latter modelled as CSP communications between behaviours). That is, we are 147 not using all these "funny" names such as: BEHAVIOUR, SEQUENTIAL_BEHAVIOUR, SIMPLE_SEQUENTIAL_BEHAVIOUR, CONCURRENT_BEHAVIOUR, COMMUNICATING_CONCURRENT_BEHAVIOUR, BEHAVIOUR_INTERACTION_LABELs, etcetera.

## 4.10 Mereology 148

Simple entities — when composite — are said to exhibit a mereology. Thus composition of simple entities imply a mereology. We discussed mereologies of behaviours: simple sequential, simple concurrent, communicating concurrent, etc. Above we did not treat actions and events as potentially being composite. But we now relax that seeming constraint. There

[8] If it is important that ACTIONs take place over time, that is, are not instantaneous, then we can just consider ACTIONs as very simple SEQUENTIAL_BEHAVIOURs not involving EVENTs.


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

149 is, in principle, nothing that prevents actions and events from exhibiting mereologies. An action, still instantaneous, can, for example, “fork” into a number of concurrent actions, all instantaneous, on “disjoint” parts of a state; or an instantaneous action can “dribble” (not little-by-little, but one-after-the-other. still instantaneously) into several actions as if 150 a simple sequential behaviour, but instantaneous. Two or more events can occur simultaneously: two or more (up to four, usually) people become grandparents when a daughter of theirs give birth to their first grandchild; or an event can — again a “dribble” (not little-by-little, but instantaneously) — “rapidly” sequence through a number of instantaneous sub-events (with no intervening time intervals): A bankruptcy events immediately causes the bankruptcy of several enterprises which again causes the immediate bankruptcy 151 of several employes, etcetera.

The problems of compositionality of entities, whether simple, actions, events or behaviours, is was studied, initially, in [9, Bjørner and Eir, 2008]

### 4.11 Impossibility of Definite Mereological Analysis of Seemingly Composite Entities 152

It would be nice if there was a more-or-less obvious way of “deciphering” the mereology of an entity. In the many • (bulleted) items above (cf. Set, Cartesian, List, Map, Graph) we may have left the impression with the reader that is a more-or-less systematic way of uncovering the mereology of a composite entity. That is not the case: there is no such obvious way. It is a matter of both discovery and choice between seemingly alternative mereologies, and it is also a matter of choice of abstraction.

### 4.12 What Exists and What Can Be Described ? 153

In the previous section we have suggested a number of *categories*[9] of entities, a number of *predicate*[10] and *observer*[11] functions and a number of *meta conditions* (i.e., axioms). These concepts and their relations to one-another, suggest an ontology for describing domains. It is now very important that we understand these categories, predicates, observers and axioms properly.

## 5 Description Versus Specification Languages 154

Footnotes 9–11 (Page 36) summarised a number of main concepts of an ontology for describing domains. The categories and predicate and observer function signatures are not

[9] Some categories: ENTITY, SIMPLE_ENTITY, ACTION, EVENT, BEHAVIOUR, ATOMIC, COMPOSITE, DISCRETE, CONTINUOUS, ATTRIBUTE, NAME, TYPE, VALUE, SET, CARTESIAN, LIST, MAP, GRAPH, *FUNCTION*, STATE, ARGUMENT, STIMULUS, EVENT_PREDICATE, BEFORE_STATE, AFTER_STATE, SEQUENTIAL_BEHAVIOUR, BEHAVIOUR_INTERACTION_LABEL, SIMPLE_SEQUENTIAL_BEHAVIOUR, SIMPLE_CONCURRENT_BEHAVIOUR, COMMUNICATING_CONCURRENT_BEHAVIOUR, etc.

[10] Some predicates: is_ENTITY, is_SIMPLE_ENTITY, is_ACTION, is_EVENT, is_BEHAVIOUR, is_ATOMIC, is_COMPOSITE, is_DISCRETE_SIMPLE_ENTITY, is_CONTINUOUS_SIMPLE_ENTITY, is_SET, is_CARTESIAN, is_LIST, is_MAP, is_GRAPH, etc.

[11] Some observers: obs_SIMPLE_ENTITY, obs_ACTION, obs_EVENT, obs_BEHAVIOUR, obs_ATTRIBUTE, obs_NAME, obs_TYPE, obs_VALUE, obs_SET, obs_CARTESIAN, obs_ARITY, obs_LIST, obs_LENGTH, obs_DEFINITION_SET, obs_RANGE, obs_IMAGE, obs_GRAPH, obs_PRECEDING_SIMPLE_ENTITIES, obs_SUCCEEDING_SIMPLE_ENTITIES, obs_MEREOLOGY, obs_INPUT_STATE, obs_ARGUMENT, obs_RESULT_STATE, obs_STIMULUS, obs_EVENT_PREDICATE, obs_BEFORE_STATE, obs_AFTER_STATE, etc.

part of a formal language for descriptions. The identifiers used for these categories are intended to denote the real thing, classes of entities of a domain. In a philosophical discourse about describability of domains one refers to the real things. That alone prevents us from devising a formal specification language for giving (syntax and) semantics to a specification, in that language, of what these (Footnote 9–11) identifiers mean.

## 5.1 Formal Specification of Specific Domains 155

Once we have decided to describe a specific domain then we can avail ourselves of using one or more of a set of formal specification languages. But such a formal specification does not give meaning to identifiers of the categories and predicate and observer functions; they give meaning to very specific subsets of such categories and predicate and observer functions. And the domain specification now ascribes, not the real thing, but usually some form of mathematical structures as models of the specified domain.

## 5.2 Formal Domain Specification Languages 156

There are, today, 2010, a large number of formal specification languages. Some or textual, some are diagrammatic. The textual specification languages are like mathematical expressions, that is: linear text, often couched in an abstract "programming language" notation. The diagrammatic specification languages provide for the specifier to draw two-dimensional figures composed from primitives. Both forms of specification languages have precise mathematical meanings, but the linear textual ones additionally provide for proof rules. 157

Examples of textual, formal specification languages are

- `Alloy [31]`: model-oriented,
- `B, Event-B [1]`: model-oriented,
- `CafeOBJ [19]`: property-oriented (algebraic),
- `CASL [16]`: property-oriented (algebraic),
- `CSP [28]`: communicating sequential processes,
- `DC (Duration Calculus) [55]`: temporal logic,
- `Maude [14, 38, 12]`: property-oriented (algebraic),
- `RAISE, RSL [21]`: property and model-oriented,
- `TLA+ [32]`: temporal logic and sets,
- `VDM, VDM-SL [18]`: model-oriented and
- `Z [54]`: model-oriented.


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

DC and TLA+ are often used in connection with either a model-oriented specification languages or just plain old discrete mathematics notation ! 158

But the model-oriented specification languages mentioned above do not succinctly express concurrency. CSP does. The diagrammatic, formal specification languages, listed below, all do that:

- Petri Nets [48],
- Message Sequence Charts (MSC) [30],
- Live Sequence Charts (LSC) [27] and
- Statecharts [26].

### 5.3 Discussion: "Take-it-or-leave-it !" 159

With the formal specification languages, not just those listed above, but with any conceivable formal specification language, the issue is: you can basically only describe using that language what it was originally intended to specify, and that, usually, was to specify software ! If, in the real domain you find phenomena or concepts, which it is somewhat clumsy and certainly not very abstract or, for you, outright impossible, to describe, then, well, then you cannot formalise them !

## 6 Conclusion 160

### 6.1 What Have We done ?

We have emphasised the crucial rôles that computing science plays in software engineering and that formalisation plays in software devdlopment. We have focused on domain engineering as a set of activities preceding those of requirements engineering and hence those of software design. We have given a concise description of pipeline systems emphasising the close, but "forever" informal relations between narrative, informal, but concise descriptions and formalisions. 161

The example pipeline systems description was primarly, in this paper intended to illustrate

- that one can indeed describe non-trivial aspects of domains and the challenges that domain descriptions pose
  - to software engineering,
  - to computing science and
  - to computer science.

We have discussed one of these challenges the foundations of description; albeit for a postulated set of description primitives:

categories (sorts and types), observers, axioms, actions, events and behaviours.

### 6.2 Discussion 162

The chosen description primitives are not necessarily computable, but then domains appears to be characterised also by such, incomputable phenomena and concepts.

The, by now "classical", formal specification languages

Alloy, ASM, CSP, DC, Event Petri Nets, RSL, VDM, Z, etcetera.
CafeOBJ, CASL, B, Maude, MSCs, Statecharts, TLA+,

need be further explored, formal interfaces of *satisfaction* established, and new, formal, or at least mathematical specification languages be developed. 163

**Domain engineering** gives rise to a number of exciting computer and comouting science as well as software engineering research problems.

### 6.3 Acknowledgements 164

In response to my paper on *Mereologies in Computing Science* [6] for **Tony Hoare**'s 75 anniversary Festschrift April 2009 (Springer Series on Hisotry of Computing) where I focused on mereologies and a relation between mereologies and CSP [28], Tony brought up, in private communication, Bertrand Russell's concept of *Logical Atomism* [50]. I am Tony very grateful for bringing this topic into my considerations. I am also grateful to **Stephen Linton** of the University of St. Andrews, Scotland for inviting me to present a paper from which the present one is quite a revision.[12] I am likewise grateful to **Alan Bundy** of the University of Edinburgh, Scotland, for a six week distinguished CISA guest Rresearcher invitation where I had time to write the first version of this paper. 165

Finally I am grateful to Profs. **Alexander Letichevsky** and **Nikolaj Nikitchenko** of Glushkov Institute of Cybernetics, Institute of Program Systems, for inviting me to this workshop and to Ukraine.

## 7 Bibliographical Notes

Specification languages, techniques and tools, that cover the spectrum of domain and requirements specification, refinement and verification, are dealt with in Alloy: [31], ASM: [47], B/event B: [1], CafeOBJ: [19], CSP [28], DC [55] (Duration Calculus), Live Sequence Charts [27], Message Sequence Charts [30], RAISE [21] (RSL), Petri nets [48], Statecharts [26], Temporal Logic of Reactive Systems [33, 34, 39, 40], TLA+ [32] (Temporal Logic of Actions), VDM [18], and Z [54]. Techniques for integrating "different" formal techniques are covered in [2].

[12] I expect the present paper to be further revised between its submisson, early march 2010, and its presentation, mid May 2010.

 © Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

The recent book on Logics of Specification Languages [10] covers `ASM, B/event B, CafeObj, CASL, DC, RAISE, TLA+, VDM` and `Z`.

# References


[1] J.-R. Abrial. The B Book: Assigning Programs to Meanings *and* Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge, England, 1996 and 2009.

[2] K. Araki et al., editors. *IFM 1999–2009: Integrated Formal Methods*, volume 1945, 2335, 2999, 3771, 4591, 5423 (only some are listed) of *Lecture Notes in Computer Science*. Springer, 1999–2009.

[3] P. Bernus and L. Nemes, editors. *Modelling and Methodologies for Enterprise Integration*, International Federation for Information Processing, London, UK, 1996 1995. IFIP TC5, Chapman & Hall. Working Conference, Queensland, Australia, November 1995.

[4] D. Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. Republished in English by Qinghua Univ. Press, Peking, China, 2009 – and translated into Chinese (by Dr. Liu Bo Chao), and also published by Qinghua Univ. Press, 2010.

[5] D. Bjørner. Domain Theory: Practice and Theories, Discussion of Possible Research Topics. In *ICTAC'2007*, volume 4701 of *Lecture Notes in Computer Science (eds. J.C.P. Woodcock et al.)*, pages 1–17, Heidelberg, September 2007. Springer.

[6] D. Bjørner. On Mereologies in Computing Science. In *Festschrift for Tony Hoare*, History of Computing (ed. Bill Roscoe), London, UK, 2009. Springer.

[7] D. Bjørner. Domain Engineering. In *BCS FACS Seminars*, Lecture Notes in Computer Science, the BCS FAC Series (eds. Paul Boca and Jonathan Bowen), pages 1–42, London, UK, 2010. Springer.

[8] D. Bjørner. *Domain Engineering: Technology Management, Research and Engineering*. JAIST Press, March 2009. JAIST Research Monograph #4, 536 pages: http://www2.imm.dtu.dk/~db/jaistmono.pdf.

[9] D. Bjørner and A. Eir. Compositionality: Ontology and Mereology of Domains. Some Clarifying Observations in the Context of Software Engineering in July 2008, eds. Martin Steffen, Dennis Dams and Ulrich Hannemann. In *Festschrift for Prof. Willem Paul de Roever Concurrency, Compositionality, and Correctness*, volume 5930 of *Lecture Notes in Computer Science*, pages 22–59, Heidelberg, July 2010. Springer.

[10] D. Bjørner and M. C. Henson, editors. *Logics of Specification Languages*. EATCS Series, Monograph in Theoretical Computer Science. Springer, Heidelberg, Germany, 2008.

[11] W. D. Blizard. A Formal Theory of Objects, Space and Time. *The Journal of Symbolic Logic*, 55(1):74–89, March 1990.

[12] R. Bruni and J. Meseguer. Generalized Rewrite Theories. In Jos C. M. Baeten and Jan Karel Lenstra and Joachim Parrow and Gerhard J. Woeginger, editor, *Automata, Languages and Programming. 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, volume 2719 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, 2003.

[13] W. Clancey. The knowledge–level reinterpreted: modeling socio–technical systems. *International Journal of Intelligent Systems*, 8:33–49, 1993.

[14] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, June 2003.

[15] N. Cocchiarella. Formal Ontology. In H. Burkhardt and B. Smith, editors, *Handbook in Metaphysics and Ontology*, pages 640–647. Philosophia Verlag, Munich, Germany, 1991.

[16] CoFI (The Common Framework Initiative). CASL *Reference Manual*, volume 2960 of *Lecture Notes in Computer Science (IFIP Series)*. Springer–Verlag, 2004.

[17] D. J. Farmer. *Being in time: The nature of time in light of McTaggart's paradox*. University Press of America, Lanham, Maryland, 1990. 223 pages.

[18] J. Fitzgerald and P. G. Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, Cambridge, UK, Second edition, 2009.

[19] K. Futatsugi, A. Nakagawa, and T. Tamai, editors. *CAFE: An Industrial–Strength Algebraic Formal Method*, Sara Burgerhartstraat 25, P.O. Box 211, NL–1000 AE Amsterdam, The Netherlands, 2000. Elsevier. Proceedings from an April 1998 Symposium, Numazu, Japan.

[20] B. H. P. Gennaro Chierchia and R. Turner, editors. *Properties, Types and Meaning*. Kluwer Academic, 15 December 1988. Vol. I: Foundational Issues, Vol. II: Semantic Issues.

[21] C. W. George, A. E. Haxthausen, S. Hughes, R. Milne, S. Prehn, and J. S. Pedersen. *The RAISE Development Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.

[22] T. R. Gruber and G. R. Olsen. An Ontology for Engineering Mathematics. In J. Doyle, P. Torasso, and E. Sandewall, editors, *Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann, 1994. Fourth International Conference. Gustav Stresemann Institut, Bonn, Germany.[13].

[13] http://www-ksl.stanford.edu/knowledge-sharing/papers/engmath.html


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

[23] M. Gruninger and M. Fox. The Logic of Enterprise Modelling. In *Modelling and Methodologies for Enterprise Integration, see [3]*, pages 141–157, November 1995.

[24] N. Guarino. Formal Ontology, Conceptual Analysis and Knowledge Representation. *Intl. Journal of Human–Computer Studies*, 43:625–640, 1995.

[25] N. Guarino. Some Organising Principles for a Unified Top–level Ontology. Int.rept., Italian National Research Council (CNR), LADSEB–CNR, Corso Stati Uniti 4, I–35127 Padova, Italy. guarino@ladseb.pd.cnr.it, 1997.

[26] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[27] D. Harel and R. Marelly. *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.

[28] T. Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985. Published electronically: http://www.usingcsp.com/-cspbook.pdf (2004).

[29] W. Humphrey. *Managing The Software Process*. Addison-Wesley, 1989. ISBN 0201180952.

[30] ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992, 1996, 1999.

[31] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.

[32] L. Lamport. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA, 2002.

[33] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive Systems: Specifications*. Addison Wesley, 1991.

[34] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive Systems: Safety*. Addison Wesley, 1995.

[35] J. McCarthy. Towards a Mathematical Science of Computation. In C. Popplewell, editor, *IFIP World Congress Proceedings*, pages 21–28, 1962.

[36] J. M. E. McTaggart. The Unreality of Time. *Mind*, 18(68):457–84, October 1908. New Series. See also: [41].

[37] D. H. Mellor and A. Oliver. *Properties*. Oxford Readings in Philosophy. Oxford Univ Press, , May 1997. ISBN: 0198751761, 320 pages.

[38] J. Meseguer. Software Specification and Verification in Rewriting Logic. NATO Advanced Study Institute, 2003.

[39] B. C. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, Cambridge, England, 1986.

[40] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, IEEE CS FoCS, pages 46–57. Providence, Rhode Island, IEEE CS, 1977. .

[41] R. L. Poidevin and M. MacBeath, editors. *The Philosophy of Time*. Oxford University Press, 1993.

[42] A. Prior. *Changes in Events and Changes in Things*, chapter in [41]. Oxford University Press, 1993.

[43] A. N. Prior. *Logic and the Basis of Ethics*. Clarendon Press, Oxford, UK, 1949.

[44] A. N. Prior. *Time and Modality*. Oxford University Press, Oxford, UK, 1957.

[45] A. N. Prior. *Past, Present and Future*. Clarendon Press, Oxford, UK, 1967.

[46] A. N. Prior. *Papers on Time and Tense*. Clarendon Press, Oxford, UK, 1968.

[47] W. Reisig. *Logics of Specification Languages*, chapter Abstract State Machines for the Classroom, pages 15–46 in [10]. Springer, 2008.

[48] W. Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Institut für Informatik, Humboldt Universität zu Berlin, Unter den Linden 6, 10099 Berlin, Germany, 1 Oktober 2009. 276 pages. http://www2.informatik.hu-berlin.de/top/pnene_buch/pnene_buch.pdf.

[49] G. Rochelle. *Behind time: The incoherence of time and McTaggart's atemporal replacement*. Avebury series in philosophy. Ashgate, Brookfield, Vt., USA, 1998. vii + 221 pages.

[50] B. Russell. The Philosophy of Logical Atomism. *The Monist: An International Quarterly Journal of General Philosophical Inquiry,*, xxxviii–xxix:495–527, 32–63, 190–222, 345–380, 1918–1919.

[51] J. F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Pws Pub Co, August 17, 1999. ISBN: 0534949657, 512 pages, Amazon price: US $ 70.95.

[52] S. Staab and R. Stuber, editors. *Handbook on Ontologies*. International Handbooks on Information Systems. Springer, Heidelberg, 2004.

[53] J. van Benthem. *The Logic of Time*, volume 156 of *Synthese Library: Studies in Epistemology, Logic, Methhodology, and Philosophy of Science (Editor: Jaakko Hintika)*. Kluwer Academic Publishers, P.O.Box 17, NL 3300 AA Dordrecht, The Netherlands, second edition, 1983, 1991.

 © Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

[54] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.

[55] C. C. Zhou and M. R. Hansen. *Duration Calculus: A Formal Approach to Real–time Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer–Verlag, 2004.