Domain Science & Engineering From Computer Science to The Sciences of Informatics Part I of II: The Engineering Part 1

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

2

This is Part I of a two-part paper. The present part first brings an example narrative + formalisation of a domain, that is, of a part of some real, man-made world — in this case a world of pipelines, whether oil or gas. Then we characterise some engineering and societal aspects of domains. In this part 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 con- 3
cepts of **domain science** and **domain engineering** become an indispensable part of
the **science of informatics** and of **software engineering**. And we finally wish to **ad-** 4
vocate (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.). In part II of the paper we explore the possibilities of of establishing a "domain science'.

Contents

1 Introduction 5

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 6 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* 8 *science*, *software* and *software engineering*.

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". Part II 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 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] 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

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, in Part II of this paper, 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 facing page show the planned Nabucco pipeline system. 17

2.1 Pipeline Basics 18

Figure 2 on page 6 conceptualises an example pipeline. Emphasis is on showing a pipeline
19 net consisting of units and connectors (•).
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
20 changes. We omit consideration of "pigs" and "pig"-insertion and "pig"-extraction units.

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

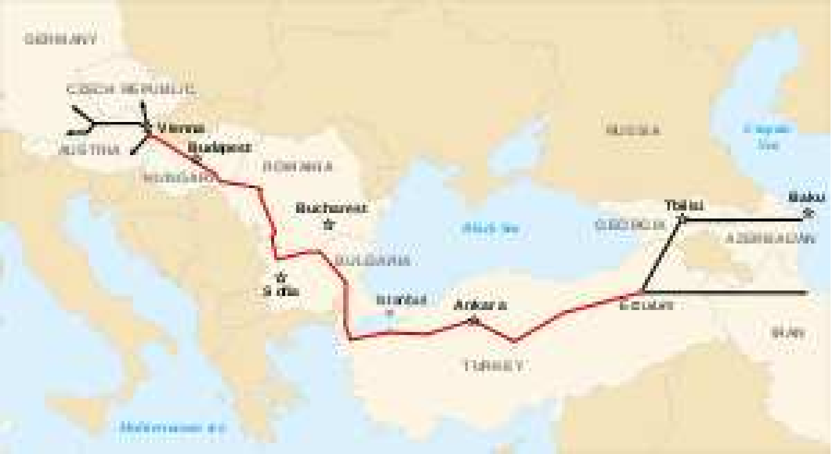

Figure 1: The Planned Nabucco Pipeline: http://en.wikipedia.org/wiki/Nabucco_Pipeline

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)
```

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
```

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.

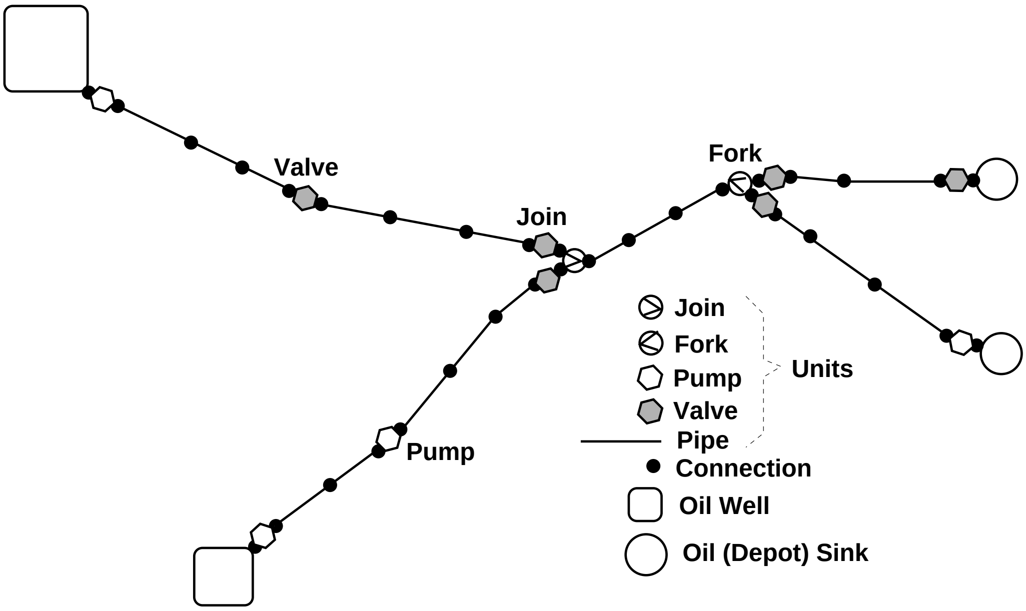

Figure 2: An oil pipeline system

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:

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
```

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
```

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
```

No Circular Routes:

26
28
29

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.

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 →̃ (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

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′)
```

42

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)
```

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 7
52   ∧ acyclic(n): Item 19 on page 8
52   ∧ wfN_SP(n): Item 20 on page 8
52   ∧ satisfies 39 on page 10 and 40 on the preceding page
52a   ∧ ∃ w:W,s:S • {w,s}⊆obs_Us(n)
52b       ⇒ ∃ r:R• ⟨w⟩⌢r⌢⟨s⟩ ∈ routes(n)
```

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′
```

48

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:

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})
```

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, repre-

sented 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′ = 𝒰_i(ui)(u) in unit(ui)(u′) end
77  𝒰_i: 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.

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, Part II of this paper, 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).

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 Mother Nature. 60

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

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 [14] 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.

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 Conclusion 74

4.1 What Have We Done in Part I ?

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 formalisations. 75

•••

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.

4.2 What Shall We Do in Part II ?

In Part II of this paper we shall discuss one of the above mentioned challenges, namely the foundations of description; albeit for a postulated set of description primitives:

- categories,
- observers,
- axioms,
- actions,
- events and
- behaviours.

4.3 Discussion 76

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` [16],
- `ASM` [23],
- `CafeOBJ` [9],
- `CASL` [7],
- `CSP` [13],
- `DC` [27],
- `Event B` [1],
- `Maude` [6, 20, 5],

- MSCs [15],
- Petri Nets [24],
- RSL [10],
- Statecharts [11],
- TLA+ [17],
- VDM [8],
- Z [26],
- etcetera.

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

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

4.4 Acknowledgements 78

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. And I am deeply grateful to Mr. **Evgeni Ivanov** for his work on the translation of this paper into Ukrainian.

5 Bibliographical Notes

Specification languages, techniques and tools, that cover the spectrum of domain and 77 requirements specification, refinement and verification, are dealt with in Alloy: [16], ASM: [23], B/event B: [1], CafeOBJ: [9], CSP [13], DC [27] (Duration Calculus), Live Sequence Charts [12], Message Sequence Charts [15], RAISE [10] (RSL), Petri nets [24], Statecharts [11], Temporal Logic of Reactive Systems [18, 19, 21, 22], TLA+ [17] (Temporal Logic of Actions), VDM [8], and Z [26]. Techniques for integrating "different" formal techniques are covered in [2]. The recent book on Logics of Specification Languages [4] 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] D. Bjørner. On Mereologies in Computing Science. In *Festschrift for Tony Hoare*, History of Computing (ed. Bill Roscoe), London, UK, 2009. Springer.

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

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

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

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

[13] 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).

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

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

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

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

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

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

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

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

[22] 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. .

[23] W. Reisig. *Logics of Specification Languages*, chapter Abstract State Machines for the Classroom, pages 15–46 in [4]. Springer, 2008.

[24] 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.

[25] 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.

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

[27] 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.