Abstract

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 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 infrastructure 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”.

1 Introduction
1.1 Some Definitions of Informatics Topics ........................................ 2
1.2 The Triptych Dogma ......................................................................... 4
1.3 Structure of This Paper ................................................................. 4

Contents
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

Let us first delineate our field of study. It first focuses on computer science, computing 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 computer programming and other “such” languages), complexity theory, type theory, etc.

Some may take exception to the term ‘things’ 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.” 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.

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, parallel programming, etc. To these we shall add a few in this paper.

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.

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

¹and also to the term ‘exist’.
²Cf. 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

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.

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

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

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.

2.1 Pipeline Basics

Figure 2 on page 6 conceptualises an example pipeline. Emphasis is on showing a pipeline 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 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, \( \pi : I \), 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

![Map of Nabucco Pipeline]

of PIpes, VAlves, PUmps, FORks, JOins, WElls and SKs.

```plaintext
type
1 N, PI, VA, PU, FO, JO, WE, SK
2 U = Π | V | P | F | J | S | W
2 II == mkII(pi:PI)
```

**Unique Identifiers:**

4. We associate with each unit a unique identifier, $u_i : UI$.
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.

```plaintext
type
4 UI
value
```

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
11. With a sink we associate exactly one input connection and zero output connections.

9 \text{obs}_{\text{outCs}}: J \rightarrow \{|1: \text{Nat}|\}
10 \text{obs}_{\text{inCs}}: W \rightarrow \{|0: \text{Nat}|\}
10 \text{obs}_{\text{outCs}}: W \rightarrow \{|1: \text{Nat}|\}
11 \text{obs}_{\text{inCs}}: S \rightarrow \{|1: \text{Nat}|\}

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 \(n\), \(2 \leq n \leq \text{obs}_{\text{Cs}}(f)\) output units.
e) Joins, \( j \), can be connected to zero or \( n \), 
\[ 2 \leq n \leq \text{obs}_\text{Cs}(j) \] 
input units and zero 
or one output units.

value

12 \( \text{obs}_\text{Us} : N \to \text{U-set} \)
13 \( \text{obs}_\text{cUIs} : U \to \text{UI-set} \times \text{UI-set} \)
\( \text{wf}_\text{Conn} : U \to \text{Bool} \)
\( \text{wf}_\text{Conn}(u) \equiv \)
let \( (\text{iuis,ouis}) = \text{obs}_\text{cUIs}(u) \) in 
\( \text{iuis} \cap \text{ouis} = \{\} \)
\( \text{case } u \text{ of } \)
13a \( \text{mkW}(\_\_) \to \)
\( \text{card iuis} \in \{0\} \land \text{card ouis} \in \{0,1\} \)
13b \( \text{mkS}(\_\_) \to \)

Wellformedness:

14. The unit identifiers observed by the \( \text{obs}_\text{cUIs} \) 
observer must be identifiers of units of the net.

axiom

14 \( \forall n : N, u : U \land u \in \text{obs}_\text{Us}(n) \Rightarrow \)

2.2 Routes

Routes:

15. By a route we shall understand a sequence of 
units.

16. Units form routes of the net.

type

15 \( R = \text{UI}^\omega \)

value

16 routes : \( N \to \text{R-infset} \)
16 routes\( (n) \equiv \)
16 \( \text{let } \text{us} = \text{obs}_\text{Us}(n) \text{ in} \)
16 \( \text{let } \text{rs} = \{\langle u \rangle \mid u : U \land u \in \text{us}\} \cup \{\langle r \rangle, \langle r' \rangle \mid r, r' : R \land \{r, r'\} \subseteq \text{rs} \land \text{adj}(r, r')\} \text{ in} \)
16 \( \text{rs } \text{end } \text{end} \)

Adjacent Routes:

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

value

17 \( \text{adj} : R \times R \to \text{Bool} \)
17 \( \text{adj}(fr,lr) \equiv \)
17 \( \text{let } (lu, fu) = (fr(\text{len } fr), \text{hd } lr) \text{ in} \)
17 \( \text{let } (lui, fui) = (\text{obs}_\text{UL}(lu), \text{obs}_\text{UL}(fu)) \text{ in} \)
17 \( \text{let } ((\_\_luis), (\_\_fuis, \_\_)) = \)
17 \( \text{(obs}_\text{cULs}(lu), \text{obs}_\text{cULs}(fu)) \text{ in} \)
18 \( \text{lui } \in \text{fuis } \land \text{fui } \in \text{luis } \text{end } \text{end} \)

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

value
19 acyclic: $\mathbb{N} \rightarrow \mathbb{Bool}$

Wellformed Nets, Special Pairs, $\text{wfN}_{SP}$:

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 $\text{wfN}_{SP}: \mathbb{N} \rightarrow \mathbb{Bool}$
20 $\text{wfN}_{SP}(n) \equiv$
20 $\forall r: R \cdot r \in \text{routes}(n) \Rightarrow$
20 $\forall i: \mathbb{Nat} \cdot \{i,i+1\} \subseteq \text{inds} r \Rightarrow$
20 case $r(i)$ of

<table>
<thead>
<tr>
<th>Case</th>
<th>Condition</th>
<th>Clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>20a</td>
<td>$\text{mkF}() \rightarrow \forall u: U \cdot \text{adj}(\langle r(i) \rangle, \langle u \rangle)$</td>
<td>$\Rightarrow \text{is}_V(u)$, $\rightarrow \text{true}$ end $\land$</td>
</tr>
<tr>
<td>20b</td>
<td>$\text{mkJ}() \rightarrow \forall u: U \cdot \text{adj}(\langle u \rangle, \langle r(i) \rangle)$</td>
<td>$\Rightarrow \text{is}_V(u)$, $\rightarrow \text{true}$ end $\land$</td>
</tr>
<tr>
<td>20c</td>
<td>$\text{mkW}(\langle \rangle) \rightarrow \text{is}_P(r(2))$,</td>
<td>$\rightarrow \text{true}$ end $\land$</td>
</tr>
<tr>
<td>20d</td>
<td>$\text{mkS}(\langle \rangle) \rightarrow \text{is}_P(r(\text{len } r - 1))$,</td>
<td>$\lor \text{is}_V(r(\text{len } r - 1))$,</td>
</tr>
<tr>
<td></td>
<td>$\lor \text{is}_V(r(\text{len } r - 1))$,</td>
<td>$\rightarrow \text{true}$ end</td>
</tr>
</tbody>
</table>

The true clauses may be negated by other case distinctions’ is$_V$ or is$_V$ clauses.

2.2.1 Special Routes, I

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.
value

21-28 \( ppr, sppr, pvr, spvr, svpr, svvr: R \rightarrow \text{Bool} \)

\[ \text{pre} \{ ppr, sppr, pvr, spvr, svpr, svvr \} (u): \ n \geq 2 \]

21 \( ppr(r; \langle fu \rangle \hat{\ell} \langle lu \rangle) \equiv \text{is}_P(fu) \land \text{is}_P(lu) \land \text{is}_\pi fjr(\ell) \)

22 \( sppr(r; \langle fu \rangle \hat{\ell} \langle lu \rangle) \equiv ppr(r) \land \text{is}_\pi(\ell) \)

23 \( pvr(r; \langle fu \rangle \hat{\ell} \langle lu \rangle) \equiv \text{is}_P(fu) \land \text{is}_V(r(\text{len } r)) \land \text{is}_\pi fjr(\ell) \)

24 \( spvr(r; \langle fu \rangle \hat{\ell} \langle lu \rangle) \equiv ppr(r) \land \text{is}_\pi(\ell) \)

25 \( vpr(r; \langle fu \rangle \hat{\ell} \langle lu \rangle) \equiv \text{is}_P(fu) \land \text{is}_V(lu) \land \text{is}_\pi fjr(\ell) \)

26 \( svpr(r; \langle fu \rangle \hat{\ell} \langle lu \rangle) \equiv ppr(r) \land \text{is}_\pi(\ell) \)

27 \( vvr(r; \langle fu \rangle \hat{\ell} \langle lu \rangle) \equiv \text{is}_V(fu) \land \text{is}_V(lu) \land \text{is}_\pi fjr(\ell) \)

28 \( svvr(r; \langle fu \rangle \hat{\ell} \langle lu \rangle) \equiv ppr(r) \land \text{is}_\pi(\ell) \)

\( \text{is}_\pi fjr, \text{is}_\pi: R \rightarrow \text{Bool} \)

\( \text{is}_\pi fjr(r) \equiv \forall u: U \cdot u \in \text{elems } r \Rightarrow \text{is}_\Pi(u) \lor \text{is}_F(u) \lor \text{is}_J(u) \)

\( \text{is}_\pi(r) \equiv \forall u: U \cdot u \in \text{elems } r \Rightarrow \text{is}_\Pi(u) \)

2.2.2 Special Routes, II

Given a unit of a route,

29. if they exist (\( \exists \)),

30. find the nearest pump or valve unit,

31. “upstream” and

32. “downstream” from the given unit.

value

29 \( \exists \text{UpPoV}: U \times R \rightarrow \text{Bool} \)

29 \( \exists \text{DoPoV}: U \times R \rightarrow \text{Bool} \)

31 find\_UpPoV: U \times R \rightsquigarrow (P | V), \text{pre} \text{find\_UpPoV}(u,r): \exists \text{UpPoV}(u,r)

32 find\_DoPoV: U \times R \rightsquigarrow (P | V), \text{pre} \text{find\_DoPoV}(u,r): \exists \text{DoPoV}(u,r)

29 \( \exists \text{UpPoV}(u,r) \equiv \)

29 \( \exists i,j: \text{Nat} \cdot \{ i,j \} \subseteq \text{inds } r \land i \leq j \land \{ \text{is}_V \mid \text{is}_P \} (r(i)) \land u = r(j) \)

29 \( \exists \text{DoPoV}(u,r) \equiv \)

29 \( \exists i,j: \text{Nat} \cdot \{ i,j \} \subseteq \text{inds } r \land i \leq j \land u = r(i) \land \{ \text{is}_V \mid \text{is}_P \} (r(j)) \)

31 \( \text{find\_UpPoV}(u,r) \equiv \)

31 \( \text{let } i,j: \text{Nat} \cdot \{ i,j \} \subseteq \text{inds } r \land i \leq j \land u = r(i) \land \{ \text{is}_V \mid \text{is}_P \} (r(j)) \)

32 \( \text{find\_DoPoV}(u,r) \equiv \)

32 \( \text{let } i,j: \text{Nat} \cdot \{ i,j \} \subseteq \text{inds } r \land i \leq j \land u = r(i) \land \{ \text{is}_V \mid \text{is}_P \} (r(j)) \)

32 \( \text{in } r(j) \) end
2.3 State Attributes of Pipeline Units

By a state attribute of a unit we mean either of the following three kinds: (i) the \textit{open}/\textit{close} states of valves and the \textit{pumping}/\textit{not pumping} states of pumps; (ii) the maximum (laminar) oil flow characteristics of all units; and (iii) the current \textit{oil flow} and current \textit{oil leak} states of all units.

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).

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 $\text{obs}_{\text{Curr}}(u)$ then we mean that $\text{obs}_{\text{Curr}}(u) - \text{obs}_{\text{Leak}}(u)$ represents the flow of oil from its outputs.

Flow Laws (II):

39. 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.

Flow Laws (II):

39 \begin{align*}
\forall u: U & \cdot \text{obs}_{\text{in}}(u) = \text{obs}_{\text{out}}(u) - \text{obs}_{\text{Leak}}(u)
\end{align*}
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
   
   are immediately connected

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

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_Ul(p)=obs_Ul(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_Ul(p)=obs_Ul(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)
42 post let v:\∧obs_Ul(v)=obs_Ul(v') in
42 obs_VΣ(v')=open
42 ∧ else_equal(n,n')(v,v') end
42 pre v ∈ obs_Us(n)
42 post let v:\∧obs_Ul(v)=obs_Ul(v') in
42 obs_VΣ(v')=close
42 ∧ else_equal(n,n')(v,v') end
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_Ul, is_V, is_P, is_W}(u''|u''')
40 ∧ adj((u), (u'))
40 ∧ is_Ul(u)∧is_V(u)∧is_P(u)∧is_W(u)
40 ∧ is_Ul(u')∧is_V(u')∧is_P(u')∧is_W(u')
41 ⇒ obs_Us²Φ(u)=obs_Us²Φ'(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, and if there exists a route, \( r \), of length two or more of the net such that 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 \textit{simple desirable property}, \( \text{sprr}(r) \), does not hold for the route, then we have a possibly undesirable event; and

48. if the route “spans” between \( p \) and \( v \) and the \textit{simple desirable property}, \( \text{spvr}(r) \), does not hold for the route, then we have a possibly undesirable event; and

49. if the route “spans” between \( v \) and \( p \) and the \textit{simple desirable property}, \( \text{svpr}(r) \), 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 \textit{simple desirable property}, \( \text{svvr}(r) \), does not hold for the route, then we have a possibly undesirable event.

Wellformed Operational Nets:

51. A well-formed operational net is a well-formed net

52. a) with at least one well, \( w \), and at least one sink, \( s \),

53. and such that there is a route in the net between \( w \) and \( s \).

54. Its pump and valve units are in the following states

a) all pumps are \textit{not} pumping, and

b) all valves are \textit{closed}.

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 \textit{not} pumping, and

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
sprr, spvr, svpr and svvr properties. Finally, setting
the pump adjacent to the (preceding) well starts the
system.

2.5 Connectors

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.

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
   and hence one can extract all its connector identifiers.

59. From a connector one can observe a pair of “optional” (distinct) unit identifiers:
   a) An optional unit identifier is
      either a unit identifier of some unit of the
   c) or a “nil” “identifier”.

60. 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

61. \( \text{type} \ oUlp = \{(Ulp ∣ \{nil\}) × (Ulp ∣ \{nil\})\} \)

62. \( \text{value} \ oUlp = \{(ouip × wfuIp ∣ ouip)\} \)
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

\[ \forall n,N,u,u':U \cdot \{u,u'\} \subseteq \text{obs}_{Us}(n) \land \text{adj}(u,u') \Rightarrow \neg (\text{is}_F(u) \land \text{is}_J(u')) \]

2.6 A CSP Model of Pipelines

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.

\[
\begin{align*}
\text{type} & \quad \text{OPLS, U, K} \\
\text{value} & \quad \text{obs}_{Us}: \text{OPLS} \to U\text{-set} \\
& \quad \text{obs}_{Ks}: \text{OPLS} \to K\text{-set} \\
& \quad \text{is}_{W,U}, \text{is}_{P,U}, \text{is}_{Pu,U}, \text{is}_{Va,U}, \\
& \quad \text{is}_{Jo,U}, \text{is}_{Fo,U}, \text{is}_{Si,U}: U \to \text{Bool} \quad \text{[mut. excl.]} \\
& \quad \text{obs}_{UI}: U \to \text{UI, obs}_{KI}: K \to K\text{-set} \\
& \quad \text{obs}_{Ul,p}: K \to (\text{UI} \cup \{\text{nil}\}) \times (\text{UI} \cup \{\text{nil}\})
\end{align*}
\]

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!

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” connector identifiers.
74. Unit behaviours are cyclic and over the state of their (static and dynamic) attributes, repre-
3 Issues of Domains and Software Engineering

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

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

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 (gas), 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

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:

3.1 Issues of Domains and Software Engineering
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

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

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

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.

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

3.3.1 Today's Common, Commercial Software Development

A vast majority of today's 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 Today's "Capability Maturity Model" Software Development

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 Today's Professional Software Development

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

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

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.

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.

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.

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; Clearsys 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

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.

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

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.
Mobile radio-telephony engineers understand 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

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 development. We have focused on domain engi-
neering 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.

The example pipeline systems description was primarily, 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,
- axioms,
- observers,
- actions,
- events and
- behaviours.

4.3 Discussion

The chosen description primitives are not necessarily computable, but then domains ap-
pears 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],
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

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 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


