Domain Engineering

Dines Bjørner*

Section on Computer Science and Engineering, Department of Informatics and Mathematical Modelling (IMM), Technical University of Denmark (DTU), DK-2800 Kgs. Lyngby, Denmark. bjorner@gmail.com

Before software can be designed we must know its requirements. Before requirements can be expressed we must understand the domain. So it follows, from our dogma, that we must first establish precise descriptions of domains; then, from such descriptions, "derive" at least domain and interface requirements; and from those and machine requirements design the software, or, more generally, the computing systems.

The preceding was an engineering dogma. Now follows a science dogma:

Just as physicists have studied this universe for centuries (and more), and will continue to do so for centuries (or more), so it is about time that we also study such man-made universes as air traffic, the financial service industry, health care, manufacturing, "the market", railways, indeed transportation in general, and so forth. Just in-and-by-themselves. No need to excuse such a study by stating only engineering concerns. To understand is all. And helps engineering.

In the main part of this chapter, Sect. 4, we shall outline what goes into a domain description[2]. We shall not cover other domain stages such as stakeholder identification (etc.), domain acquisition, analysis of domain acquisition units, domain verification, and domain validation. That is: before we can acquire domain knowledge we must know what are suitable structures of domain descriptions. Thus we shall outline ideas of modelling (i) the intrinsics (of a domain), (ii) the support technologies (of ...), (iii) the management and organisation (of ...), (iv) the rules and regulations (including [licence or contract] scripts) (of ...), and (v) the human behaviours (of a domain).

Before delving into the main part we shall, however first overview what we see as basic principles of describing phenomena and concepts of domains.

* Professor Emeritus

[2] We shall use the following three terms: description when we specify what there is (as for domains), prescription when we specify what we would like there to be (as for requirements), and specification when we specify software (or hardware) designs.

At the basis of all modelling work is abstraction. Mathematics is a reliable carrier of abstraction. Hence our domain modelling will be presented both as informal, yet short and precise, that is, concise narratives as well as formal specifications. In this chapter we primarily express the formalisations in the RAISE [26] specification language, RSL [24]. We refer to [5, 6] for a comprehensive coverage of formal abstractions and models.

Two remarks are now in order. Firstly, there are other specification (cum development or design) languages. For example Alloy [41], ASM [13, 63], B [1], CafeOBJ [23, 22, 19], CASL [4, 16, 56], VDM-SL [9, 10, 21] and Z [70, 71, 75, 37]. But, secondly, none of these suffices. Each, including RSL, have their limitations in what they were supposed to express with ease. So one needs to combine, to integrate either of the above formal notations with for example the notations of Duration Calculus [77, 78] (DC), Message [38, 39, 40] or Live Sequence Charts (MSCs and LSCs) [17, 35, 50], Petri Nets [49, 58, 60, 61, 62], Statecharts [31, 32, 34, 36, 33] (SCs), TLA+ [51, 52, 55] etcetera.

Chapters 12–15 of [6] presents an extensive coverage of Petri Nets, MSCs and LSCs, SCs and DC, respectively. The present chapter presents an essence of chapters 5, 6 and 11 of [7].

The forthcoming book 'Logics of Specification Languages' [8] covers the following formal notations ASM, B, CafeOBJ, CASL, DC, RAISE, VDM-SL, TLA+ and Z. [8] represents an extensive revision of the following published papers: [63, 15, 19, 56, 25, 55, 37].

1 Introduction

1.1 Application cum Business Domains

By 'domain' we shall loosely speaking understand the same as by 'application (or business) domain': a 'universe of discourse', 'an area of human and societal activity' for which, eventually some support in the form of computing and (electronic) communication may be desired. Once computing and communication, that is, hardware and software, that is, machine, has been installed then the environment for that machine is the former, possibly business process re-engineered[3], domain, and the new domain includes the machine. The machine interacts with its possibly business process re-engineered (former) domain. But we can speak of a domain without ever thinking, or having to think about computing (etc.) applications.

Examples of domains are: (i) air traffic, (ii) airports, (iii) the financial service industry (clients, banks, brokers, securities exchange, portfolio managers, insurance companies, credit card companies, financial service industry watchdogs, etc.), (iv) freight logistics, (v) health care, (vi) manufactur-

[3] By 'business process re-engineered domain' we mean a domain some of whose facets have been altered without the changes containing computing or communication. The changes will normally involve interfacing to the machine being sought.

ing, and (vii) transportation (air lines, railways, roads (private automobiles, buses, taxis, trucking), shipping). These examples could be said to be “grand scale”, and to reflect infrastructure components of a society. Less ‘grand’ examples of domains are: (viii) the interconnect-cabling and the electronic and electro-mechanical boxes of, for example electronic equipment; or (ix) the interlocking of groups of rail points (switches) of a railway station; or (x) an automobile (with all its mechanical, electro-mechanical and electronic parts and the composition of all these parts, and with the driver and zero, one or more passengers); or (xi) a set of laws (or just rules and regulations) of a business enterprise. In all of these latter cases we usually include the human or technological monitoring and control of these domains.

1.2 Physics, Domains and Engineering

Physics has been studied for millenia. Physicists continue to unravel deeper and deeper understandings of the physically observable universe around us. Classical engineering builds on physics. Every aeronautical & aerospace, or chemical, or civil, or electrical & electronics, or mechanical & control engineer is expected to know all the laws of physics relevant to their field, and much more; and is expected to model, using various branches of mathematics (calculus, statistics, probability theory, graph theory, combinatorics, etc.) phenomena of the domain in which their engineering artifacts are placed (as well as, of course, these artifacts themselves).

Software engineers sometimes know how to model their own artifacts (compilers, operating systems, database management systems, data communication systems, web services, etc.), but they seldom, if ever are expected to model, and they mostly cannot, i.e., do not know how to model, the domain in which their software operates.

1.3 So: What is a Domain ?

We shall use the three terms ‘domain’, ‘domain description’ and ‘domain model’ quite extensively in this chapter.

Above we briefly characterised the term ‘domain’: By a domain we shall loosely speaking understand the same as by an application (or business) domain: a universe of discourse, an area of human and societal activity, etc.

By a domain description we mean a pair of descriptions: an informal, natural, but probably a professional technical language narrative text which describes the domain as it is and a formal, mathematical text which, supposedly hand-in-hand with the narrative, formalises this description.

By a domain model we mean that which is designated by a domain description. Two views can be taken here. Either we may claim that the domain description designates the domain, i.e., an actual universe of discourse “out there”, in reality. Or we may — more modestly — say that the domain description, denotes a possibly infinite, possibly empty set of mathematical structures which (each) satisfy the formal description. The former view is taken when the domain describer is validating the description by asking domain stake holders whether the description corresponds to their view (conceptualisation) of the domain (in which they work). The latter view is taken when the domain describer is building a domain theory, that is, proves theorems that hold of predicates over the mathematical domain model.

1.4 Relation to Previous Work

Considerations of domains in software development has, of course, always been there. Jackson, already in [42] and especially in the work [43, 44, 45, 76, 47] leading up to the seminal [48] has, again and again, emphasized the domain, or as he calls it, the environment aspects. In classical data analysis — in preparation for the design of data-based information systems — we find a strong emphasis on domain analysis. In work on ontology as they relate to software engineering we likewise find this emphasis, notably on the idiosyncratic [69]. We are not claiming that domain analysis, as part of our domain engineering approach is new. What we are claiming is that the emphasis we put on describing, also formally, the domain in isolation from any concern for requirements, let alone software is new. Of course, as one then goes on to develop requirements prescriptions from domain descriptions and software from requirements prescriptions, discrepancies and insufficiences of the base domain description may very well be uncovered — and must be repaired. We appreciate the approach taken in Jackson's Problem Frame approach [48] in alternating between concerns of domain, requirements and software design.

1.5 Structure of Chapter

In Sect. 2 we briefly express the dogma behind the concept of domain engineering and its role in software development. We likewise briefly, outline basic stages of development of a domain description, i.e., of domain engineering. Section 3 outline an ontology of concepts that we claim appear in any interesting model: entities, functions (or relations), events and behaviours. A new think here, perhaps, is our treatment of the modelling of entities: atomic in terms of their attributes and composite in terms of their attributes, their sub-entities and the composition, which we refer to as the mereology of these sub-entities. Section 4 forms the major part of this chapter. We present some high level pragmatic principles for decomposing the task of describing a domain — in terms of what we call domain facets — and we illustrate facet modelling by some, what you might think as as low level descriptions. Section 3 and 4 focused on some aspects of domain abstraction. Section 5 comments on a number of abstraction principles and techniques that are not covered in this chapter. Basically this chapter assumes a number of (other) abstraction principles and techniques which we cover elsewhere — notably in [5, 6] and [7, Chaps. 2–10]. First part of Sect. 6 very briefly indicates how one may meaningfully obtain major parts of a requirements prescription from a domain description. Section 6 also contains a discussion of the differences between domain engineering and requirements engineering. Section 7 takes up the motivation and justification for domain engineering.

• • •

A word of warning: This chapter only covers one aspect of domain engineering, namely that of domain facets. There are a number of other aspects of software engineering which underlie professional domain engineering — such as (i) general principles of abstraction and modelling, (ii) special principles of modelling languages and systems, and (iii) other special principles of modelling domains. These are extensively covered in [5], [6] and [7, Chaps.2–10] respectively.

2 Domain Engineering: The Engineering Dogma

- *Before software can be designed we must know its requirements.*
- *Before requirements can be expressed we must understand the domain.*
- *So it follows, from our dogma, that we must*
  - *first establish precise descriptions of domains;*
  - *then from such descriptions, "derive" at least domain and interface requirements;*
  - *and from those and machine requirements design the software, or, more generally, the computing systems.*

That is, we propose — what we have practised for many years — that the software engineering process be composed — and that it be iterated over, in a carefully monitored and controlled manner — as follows:

- domain engineering,
- requirements engineering, and
- software design.

Each with their many stages and many steps.

We see the domain engineering process as composed from, and iterated over the following stages:[4]

1. identification of and regular interaction with stakeholders
2. domain (knowledge) acquisition
3. domain analysis
4. domain modelling
5. domain verification
6. domain validation

[4] The requirements engineering stages are listed in Sect. 6.1.

7. domain theory formation

In this chapter we shall only look at the principles and techniques of domain modelling, that is, item 4. To pursue items 2.–3. one must know what goes into a domain description, i.e., a domain model.

- *A major part of the domain engineering process is taken up by finding and expressing suitable abstractions, that is, descriptions of the domain.*
- *Principles for identifying, classifying and describing domain phenomena and concepts are therefore needed.*

This chapter focuses on presenting some of these principles and techniques.

3 Entities, Functions, Events and Behaviours

In the domain we observe phenomena. From usually repeated such observations we form (immediate, abstract) concepts. We may then lift such immediate abstract concepts to more general abstract concepts.

Phenomena are manifest. They can be observed by human senses (seen, heard, felt, smelled or tasted) or by physical measuring instruments (mass, length, time, electric current, thermodynamic temperature, amount of substance, luminous intensity). Concepts are defined.

We shall analyse phenomena and concepts according to the following simple, but workable classification: **entities**, **functions** (over entities), **events** (involving changes in entities, possibly as caused by function invocations, i.e., **actions**, and/or possibly causing such), and **behaviours** as (possibly sets of) sequences of actions (i.e., function invocations) and events.

3.1 Entities

- *By an* **entity** *we shall understand something that we can point to, something manifest, or a concept abstracted from such phenomena or concepts.*

Entities are either atomic or composite. The decision as to which entities are considered what is a decision solely taken by the describer.

Atomic Entities

- *By an* **atomic entity** *we intuitively understand an entity which 'cannot be taken apart' (into other, the sub-entities).*

Attributes — Types and Values:

With any entity we can associate one or more attributes.

- *By an* **attribute** *we understand a pair of a* **type** *and a* **value**.

Example 1. **Atomic Entities:**

Entity: **Person**		Entity: **Bank Account**	
Type	**Value**	**Type**	**Value**
Name	Dines Bjørner	number	212 023 361 918
Weight	118 pounds	balance	1,678,123 Yen
Height	179 cm	interest rate	1.5 %
Gender	male	credit limit	400,000 Yen

•

'Removing' attributes from an entity destroys its 'entity-hood', that is, attributes are an essential part of an entity.

Mereology

- *By* **mereology** *we shall understand a theory of part-hood relations. That is, of the relations of part to whole and the relations of part to part within a whole.*

The term mereology seems to have been first used in the sense we are using it by the Polish mathematical logician Stanisław Leśniewski [54, 72].

Composite Entities

- *By a composite entity we intuitively understand an entity (i) which "can be taken apart" into sub-entities, (ii) where the composition of these is described by its* **mereology**, *and (iii) which further possess one or more attributes.*

Example 2. **Transport Net, A Narrative:**

Entity: **Transport Net**		
Subentities:	Segments	
	Junctions	
Mereology:	"set" of one or more s(egment)s and	
	"set" of two or more j(unction)s	
such that each s(egment) is delimited by two j(unctions)		
and such that each j(unction) connects one or more s(egments)		
Attributes		
	Types:	**Values:**
	Multimodal	Rail, Roads
	Transport Net of	Denmark
	Year Surveyed	2006

•

To put the above example of a composite entity in context we give an example of both an informal narrative and a corresponding formal specification:

Example 3. **Transport Net, A Formalisation:** A transport net consists of one or more segments and two or more junctions. With segments [junctions] we can associate the following attributes: segment [junction] identifiers, the identifiers of the two junctions to which segments are connected [the identifiers of the one or more segments connected to the junction], the mode of a segment [the modes of the segments connected to the junction].

type
  N, S, J, Si, Ji, M
value
  obs_Ss: N → S-**set**, obs_Js: N → J-**set**
  obs_Si: S → Si, obs_Ji: J → Ji
  obs_Jis: S → Ji-**set**, obs_Sis: J → Si-**set**
  obs_M: S → M, obs_Ms: J → M-**set**
axiom
  ∀ n:N • **card** obs_Ss(n) ≥ 1 ∧ **card** obs_Js(n) ≥ 2
  ∀ n:N • **card** obs_Ss(n) ≡ **card** {obs_Si(s)|s:S • s ∈ obs_Ss(n)}
  ∀ n:N • **card** obs_Js(n) ≡ **card** {obs_Ji(c)|j:J • j ∈ obs_Js(n)}
  ...
type
  Name, Country, Year
value
  obs_Name: N → Name, obs_Country: N → Country, obs_Year: N → Year

Si, Ji, M, Name, Country and Year are not entities. They are names of attribute types and, as such, designate attribute values. N is composite, S and J are considered atomic.[5] •

States

- *By a domain* **state** *we shall understand a collection of domain entities chosen by the domain engineer.*

The pragmatics of the notion of state is that states are recurrent arguments to functions and are changed by function invocations.

[5] As remarked by a referee: "using cardinality to express injectivity seems obscure, reveals a symptom rather than a cause and is useless in proof". We agree.

3.2 Functions

- *By a* **function** *we shall understand something which when applied to some argument values yield some entities called the result value of the function (application).*
- *By an* **action** *we shall understand the same things as applying a state-changing function to its arguments (including the state).*

Function Signatures

By a function signature we mean the name and type of a function.

type
  A, B, ..., C, X, Y, .., Z
value
  f: A $\times$ B $\times$ ... $\times$ C $\rightarrow$ X $\times$ Y $\times$ ... $\times$ Z

The last line above expresses a schematic function signature.

Function Descriptions

By a function description we mean a function signature and something which describes the relationship between function arguments and function results.

Example 4. **Well Formed Routes:**

```
type
   P = Ji × Si × Ji            /∗ path: triple of identifiers ∗/
   R′ =  P∗                    /∗ route: sequence of connected paths ∗/
   R = {| r:R′ • wf_R(r) |}    /∗ subtype of R′: those r′s satisfying wf_R(r) ∗/
value
   wf_R: R′ → Bool
   wf_R(r) ≡
      ∀ i:Nat•{i,i+1}⊆inds r⇒let (,,ji′)=r(i),(ji″,,)=r(i+1) in ji′=ji″ end
```

•

The last line above describes the route wellformedness predicate. [The meaning of the "(,," and ",,)" is that the omitted path components "play no rôle".]

3.3 Events

- *By an* **event** *we shall understand an instantaneous change of state not directly brought about by some explicitly willed action in the domain, but either by "external" forces. or implicitly as a non-intended result of an explicitly willed action.*

Events may or may not lead to the initiation of explicitly issued operations.

Example 5. **Events:** A 'withdraw' from a positive balance bank account action may leave a negative balance bank account. A bank branch office may have to temporarily stop actions, i.e., close, due to a bank robbery. •

Internal events: The first example above illustrates an internal action. It was caused by an action in the domain, but was not explicitly the main intention of the "withdraw" function.

External events: The second example above illustrates an external action. We assume that we have not explicitly modelled bank robberies!

Formal modelling of events: With every event we can associate an event label. An event label can be thought of as a simple identifier. Two or more event labels may be the same.

3.4 Behaviours

- *By a* **behaviour** *we shall understand a structure of actions (i.e., function invocations) and events. The structure may typically be a set of sequences of actions and events.*

We here refrain from stating whether the "set of sequences" is supposed to model interleaved concurrency, as when we express concurrency in terms of CSP, or is supposed to model "true" concurrency, as when we express concurrency in terms of Petri Nets. Such a statement is required, or implied, however, whenever we present a particular model.

A behaviour is either a simple behaviour, or is a concurrent behaviour, or, if the latter, can be either a communicating behaviour or not.

- *By a* **simple behaviour** *we shall understand a sequence of actions and events.*

Example 6. **Simple Behaviours:** The opening of a bank account, the deposit into that bank account, zero, one or more other such deposits, a withdrawal from the bank account in question, etc. (deposits and withdrawals), ending with a closing of the bank account. Any prefix of such a sequence is also a simple behaviour. Any sequence in which one or more events are interspersed is also a simple behaviour. •

- *By a* **concurrent behaviour** *we shall understand a set of behaviours (simple or otherwise).*

Example 7. **Concurrent Behaviours:** A set of simple behaviours may result from two or more distinct bank clients, each operating their own, distinct, that is, non-shared accounts. •

- *By a* **communicating behaviour** *we shall understand a set of two or more behaviours where otherwise distinct elements (i.e., behaviours) share events.*

The sharing of events can be identified via the event labels.

Example 8. **Communicating Behaviours:** To model that two or more clients can share the same bank account one could model the bank account as one behaviour and each client as a distinct behaviour. Let us assume that only one client can open an account and that only one client can close an account. Let us further assume that sharing is brought about by one client, say the one who opened the account, identifying the sharing clients. Now, in order to make sure that at most one client accesses the shared account at any one time (in any one "smallest" transaction interval) one may model "client access to account" as a pair of events such that during the interval between the first (begin transaction) and the second (end transaction) event no other client can share events with the bank account behaviour. Now the set of behaviours of the bank account and one or more of the client behaviours is an example of a communicating behaviour. •

Formal modelling of behaviours: Communicating behaviours, the only really interesting behaviours, can be modelled in a great variety of ways: from set-oriented models in B, RSL, VDM, or Z, to models using for example CSP (as for example "embedded" in RSL), or, to diagram models using, for example, Petri nets, message or live sequence charts, or Statecharts.

3.5 Discussion

The main aim of Sect. 3 is to ensure that we have a clear understanding of the modelling concepts of entities, functions, events and behaviours. To "reduce" the modelling of phenomena and concepts to these four kinds of phenomena and concepts is, of course, debatable. Our point is that it works, that further classification, as is done in for example John F. Sowa's [69], is not necessary, or rather, is replaced by how we model attributes of for example entities[6] and how we model facets, such as we shall call them. The modelling of facets is the main aim of this chapter.

4 Domain Facets

- *By a domain* **facet** *we shall understand one amongst a finite set of generic ways of analysing a domain: a view of the domain, such that the different facets cover conceptually different views, and such that these views together cover the domain.*

The hedge here is "finite set of generic ways". Thus there is an assumption, a conjecture to be possibly refuted. Namely the postulate that there is a finite number of facets. We shall offer the following facets: intrinsics, support technology, management and organisation, rules and regulations (and scripts), and human behaviour.

[6] For such issues as static and dynamic attributes, dimensionality, tangibility, time and space, etc., we refer to Michael A. Jackson's [46] or [7, Chap. 10].

4.1 Intrinsics

- *By domain* **intrinsics** *we shall understand those phenomena and concepts of a domain which are basic to any of the other facets (listed earlier and treated, in some detail, below), with such domain intrinsics initially covering at least one specific, hence named, stakeholder view.*

Example 9. **Railway Net Intrinsics:** We narrate and formalise three railway net intrinsics.

- From the view of *potential train passengers* a railway net consists of lines, l:L, with names, ln:Ln, stations, s:S, with names sn:Sn, and trains, tn:TN, with names tnm:Tnm. A line connects exactly two distinct stations.
- From the view of *actual train passengers* a railway net — in addition to the above — allows for several lines between any pair of stations and, within stations, provides for one or more platform tracks, tr:Tr, with names, trn:Trn, from which to embark on or alight from a train.
- From the view of *train operating staff* a railway net — in addition to the above — has lines and stations consisting of suitably connected rail units. A rail unit is either a simple (i.e., linear, straight) unit, or is a switch unit, or is a simple crossover unit, or is a switchable crossover unit, etc. Simple units have two connectors. Switch units have three connectors. Simple and switchable crossover units have four connectors. A path, p:P, (through a unit) is a pair of connectors of that unit. A state, $\sigma : \Sigma$, of a unit is the set of paths, in the direction of which a train may travel. A (current) state may be empty: The unit is closed for traffic. A unit can be in any one of a number of states of its state space, $\omega : \Omega$.

A summary formalisation of the three narrated railway net intrinsics could be:

- *Potential train passengers:*

```
scheme N0 =
  class
    type
      N, L, S, Sn, Ln, TN, Tnm
    value
      obs_Ls: N → L-set, obs_Ss: N → S-set
      obs_Ln: L → Ln, obs_Sn: S → Sn
      obs_Sns: L → Sn-set, obs_Lns: S → Ln-set
    axiom
      ...
  end
```

N, L, S, Sn and Ln designate nets, lines, stations, station names and line names. One can observe lines and stations from nets, line and station names from lines and stations, pair sets of station names from lines, and lines names (of lines) into and out from a station from stations. Axioms ensure proper graph properties of these concepts.

- *Actual train passengers:*

  scheme N1 = **extend** N0 **with**
    class
      type
        Tr, Trn
      value
        obs_Trs: S $\rightarrow$ Tr-**set**, obs_Trn: Tr $\rightarrow$ Trn
      axiom
        ...
    end

  The only additions are that of track and track name types, related observer functions and axioms.
- *Train operating staff:*

  scheme N2 = **extend** N1 **with**
    class
      type
        U, C
        $\mathsf{P}' = \mathsf{U} \times (\mathsf{C}\times\mathsf{C})$
        P = {| p:P′ • **let** (u,(c,c′))=p **in** (c,c′)∈ ∪ obs_Ω(u) **end** |}
        Σ = P-**set**
        Ω = Σ-**set**
      value
        obs_Us: (N|L|S) $\rightarrow$ U-**set**
        obs_Cs: U $\rightarrow$ C-**set**
        obs_Σ: U $\rightarrow$ Σ
        obs_Ω: U $\rightarrow$ Ω
      axiom
        ...
    end

Unit and connector types have been added as have concrete types for paths, unit states, unit state spaces and related observer functions, including unit state and unit state space observers. The reader is invited to compare the three narrative descriptions with the three formal descriptions, line by line. •

Different stakeholder perspectives, not only of intrinsics, as here, but of any facet, lead to a number of different models. The name of a phenomenon of one perspective, that is, of one model, may coincide with the name of a "similar" phenomenon of another perspective, that is, of another model, and so on. If the intention is that the "same" names cover comparable phenomena, then the developer must state the comparison relation.

Example 10. **Comparable Intrinsics:** We refer to Example 9. We claim that the concept of nets, lines and stations in the three models of Example 9 must relate. The simplest possible relationships are to let the third model be the common "unifier" and to mandate

- that the model of nets, lines and stations of the *potential train passengers* formalisation is that of nets, lines and stations of the *train operating staff* model; and
- that the model of nets, lines, stations and tracks of the *actual train passengers* formalisation is that of nets, lines, stations of the *train operating staff* model.

Thus the third model is seen as the definitive model for the stakeholder views initially expressed. •

Example 11. **Intrinsics of Switches:** The intrinsic attribute of a rail switch is that it can take on a number of states. A simple switch $({}^{c_|}Y_c^{c_/})$ has three connectors: $\{c, c_|, c_/\}$. c is the connector of the common rail from which one can either "go straight" $c_|$, or "fork" $c_/$ (Fig. 1). So we have that a possible state space of such a switch could be ω_{g_s} :

$$\begin{array}{l}
\{\{\},\\
\{(c,c_|)\},\{(c_|,c)\},\{(c,c_|),(c_|,c)\},\\
\{(c,c_/)\},\{(c_/,c)\},\{(c,c_/),(c_/,c)\},\{(c_/,c),(c_|,c)\},\\
\{(c,c_|),(c_|,c),(c_/,c)\},\{(c,c_/),(c_/,c),(c_|,c)\},\{(c_/,c),(c,c_|)\},\{(c,c_/),(c_|,c)\}\}
\end{array}$$

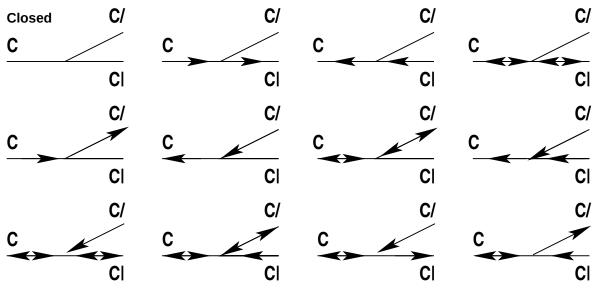

Fig. 1. Possible states of a rail switch

The above models a general switch ideally. Any particular switch ω_{p_s} may have $\omega_{p_s} \subset \omega_{g_s}$. Nothing is said about how a state is determined: who sets and resets it, whether determined solely by the physical position of the switch gear, or also by visible or virtual (i.e., invisible, intangible) signals up or down the rail, away from the switch. •

Conceptual Versus Actual Intrinsics

In order to bring an otherwise seemingly complicated domain across to the reader, one may decide to present it piecemeal:[7] First, one presents the very basics, the fewest number of inescapable entities, functions and behaviours. Then, in a step of enrichment, one adds a few more (intrinsic) entities, functions and behaviours. And so forth. In a final step one adds the last (intrinsic) entities, functions and behaviours. In order to develop what initially may seem to be a complicated domain, one may decide to develop it piecemeal: We basically do as for the presentation steps: Steps of enrichment — from a big lie, via increasingly smaller lies, till one reaches a truth!

On Modelling Intrinsics

Domains can be characterised by intrinsically being entity, or function, or event, or behaviour intensive. Software support for activities in such domains then typically amount to database systems, computation-bound systems, real-time embedded systems, respectively distributed process monitoring and control systems. Modelling the domain intrinsics in respective cases can often be done property-oriented specification languages (like CafeOBJ or CASL), model-oriented specification languages (like B, VDM-SL, RSL, or Z), event-based languages (like Petri nets or CSP), respectively process-based specification languages (like MSCs, LSCs, Statecharts, or CSP),

4.2 Support Technologies

- *By a domain* **support technology** *we shall understand ways and means of implementing certain observed phenomena or certain conceived concepts.*

Example 12. **Railway Support Technology:** We give a rough sketch description of possible rail unit switch technologies.

(i) In "ye olde" days, rail switches were "thrown" by manual labour, i.e., by railway staff assigned to and positioned at switches.

(ii) With the advent of reasonably reliable mechanics, pulleys and levers[8] and steel wires, switches were made to change state by means of "throwing" levers in a cabin tower located centrally at the station (with the lever then connected through wires etc., to the actual switch).

(iii) This partial mechanical technology then emerged into electromechanics, and cabin tower staff was "reduced" to pushing buttons.

---

[7] That seemingly complicated domain may seem very complicated, containing hundreds of entities, functions and behaviours. Instead of presenting all the entities, functions, events and behaviours in one "fell swoop", one presents them in stages: first, around seven such (entities, functions, events and behaviours), then seven more, etc.

(iv) Today, groups of switches, either from a station arrival point to a station track, or from a station track to a station departure point, are set and reset by means also of electronics, by what is known as interlocking (for example, so that two different routes cannot be open in a station if they cross one another). •

It must be stressed that Example 12 is just a rough sketch. In a proper narrative description the software (cum domain) engineer must describe, in detail, the subsystem of electronics, electromechanics and the human operator interface (buttons, lights, sounds, etc.).

An aspect of supporting technology includes recording the state-behaviour in response to external stimuli. We give an example.

Example 13. **Probabilistic Rail Switch Unit State Transitions:** Figure 2 indicates a way of formalising this aspect of a supporting technology. Figure 2 intends to model the probabilistic (erroneous and correct) behaviour of a switch when subjected to settings (to switched (s) state) and re-settings (to direct (d) state). A switch may go to the switched state from the direct state when subjected to a switch setting s with probability psd. •

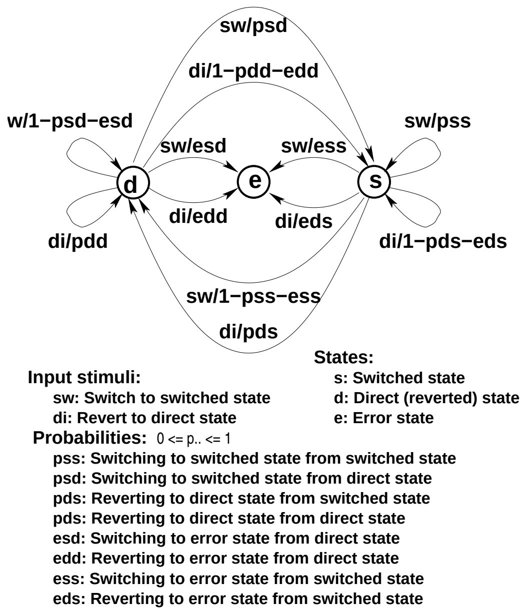

Fig. 2. Probabilistic state switching

Another example shows another aspect of support technology: Namely that the technology must guarantee certain of its own behaviours, so that software

8 http://en.wikipedia.org/wiki/Lever

designed to interface with this technology, together with the technology, meets dependability requirements.

Example 14. **Railway Optical Gates:** Train traffic (itf:iTF), intrinsically, is a total function over some time interval, from time (t:T) to continuously positioned (p:P) trains (tn:TN).

Conventional optical gates sample, at regular intervals, the intrinsic train traffic. The result is a sampled traffic (stf:sTF). Hence the collection of all optical gates, for any given railway, is a partial function from intrinsic to sampled train traffics (stf).

We need to express quality criteria that any optical gate technology should satisfy — relative to a necessary and sufficient description of a closeness predicate. The following axiom does that:

> For all intrinsic traffics, itf, and for all optical gate technologies, og, the following must hold: Let stf be the traffic sampled by the optical gates. For all time points, t, in the sampled traffic, those time points must also be in the intrinsic traffic, and, for all trains, tn, in the intrinsic traffic at that time, the train must be observed by the optical gates, and the actual position of the train and the sampled position must somehow be check-able to be close, or identical to one another.

Since units change state with time, n:N, the railway net, needs to be part of any model of traffic.

type
$\quad \mathsf{T}, \mathsf{TN}$
$\quad \mathsf{P} = \mathsf{U}^*$
$\quad \mathsf{NetTraffic} == \mathsf{net{:}N}\ \ \mathsf{trf{:}(TN} \underset{\overrightarrow{m}}{} \mathsf{P)}$
$\quad \mathsf{iTF} = \mathsf{T} \rightarrow \mathsf{NetTraffic}$
$\quad \mathsf{sTF} = \mathsf{T} \underset{\overrightarrow{m}}{} \mathsf{NetTraffic}$
$\quad \mathsf{oG} = \mathsf{iTF} \stackrel{\sim}{\rightarrow} \mathsf{sTF}$
value
$\quad [\,\mathsf{close}\,]\ \mathsf{c}{:}\ \mathsf{NetTraffic} \times \mathsf{TN} \times \mathsf{NetTraffic} \stackrel{\sim}{\rightarrow} \mathbf{Bool}$
axiom
$\quad \forall\ \mathsf{itt{:}iTF,\ og{:}OG} \bullet \mathbf{let}\ \mathsf{stt} = \mathsf{og(itt)}\ \mathbf{in}$
$\quad\quad \forall\ \mathsf{t{:}T} \bullet \mathsf{t} \in \mathbf{dom}\ \mathsf{stt}\ \Rightarrow$
$\quad\quad\quad \forall\ \mathsf{Tn{:}TN} \bullet \mathsf{tn} \in \mathbf{dom}\ \mathsf{trf(itt(t))}$
$\quad\quad\quad\quad \Rightarrow \mathsf{tn} \in \mathbf{dom}\ \mathsf{trf(stt(t))} \wedge \mathsf{c(itt(t),tn,stt(t))}\ \mathbf{end}$

Check-ability is an issue of testing the optical gates when delivered for conformance to the closeness predicate, i.e., to the axiom. •

On Modelling Support Technologies

Support technologies in their relation to the domain in which they reside typically reflect real-time embeddedness. As such the techniques and languages for modelling support technologies resemble those for modelling event and process intensity, while temporal notions are brought into focus. Hence typical modelling notations include event-based languages (like Petri nets or CSP), respectively process-based specification languages (like MSCs, LSCs, Statecharts, or CSP), as well as temporal languages (like the Duration Calculus and Temporal Logic of Actions, TLA+).

4.3 Management and Organisation

Example 15. **Train Monitoring, I:** In China, as an example, rescheduling of trains occurs at stations and involves telephone negotiations with neighbouring stations ("up and down the lines"). Such rescheduling negotiations, by phone, imply reasonably strict management and organisation (M&O). This kind of M&O reflects the geographical layout of the rail net. •

- *By domain* **management** *we shall understand such people (such decisions) (i) who (which) determine, formulate and thus set standards (cf. rules and regulations, Sect. 4.4) concerning strategic, tactical and operational decisions; (ii) who ensure that these decisions are passed on to (lower) levels of management, and to floor staff; (iii) who make sure that such orders, as they were, are indeed carried out; (iv) who handle undesirable deviations in the carrying out of these orders cum decisions; and (v) who "backstop" complaints from lower management levels and from floor staff.*
- *By domain* **organisation** *we shall understand the structuring of management and non-management staff levels; the allocation of strategic, tactical and operational concerns to within management and non-management staff levels; and hence the "lines of command": who does what, and who reports to whom, administratively and functionally.*

Example 16. **Railway Management and Organisation: Train Monitoring, II:** We single out a rather special case of railway management and organisation. Certain (lowest-level operational and station-located) supervisors are responsible for the day-to-day timely progress of trains within a station and along its incoming and outgoing lines, and according to given timetables. These supervisors and their immediate (middle-level) managers (see below for regional managers) set guidelines (for local station and incoming and outgoing lines) for the monitoring of train traffic, and for controlling trains that are either ahead of or behind their schedules. By an incoming and an outgoing line we mean part of a line between two stations, the remaining part being handled by neighbouring station management. Once it has been decided, by such a manager, that a train is not following its schedule, based on information monitored by non-management staff, then that manager directs that staff: (i) to suggest a new schedule for the train in question, as well as for possibly affected other trains, (ii) to negotiate the new schedule with appropriate neighbouring stations, until a proper reschedule can be decided upon, by the managers at respective stations, (iii) and to enact that new schedule.[9] A (middle-level operations) manager for regional traffic, i.e., train traffic involving several stations and lines, resolves possible disputes and conflicts. •

The above, albeit rough-sketch description, illustrated the following management and organisation issues: (i) There is a set of lowest-level (as here: train traffic scheduling and rescheduling) supervisors and their staff; (ii) they are organised into one such group (as here: per station); (iii) there is a middle-level (as here: regional train traffic scheduling and rescheduling) manager (possibly with some small staff), organised with one such per suitable (as here: railway) region; and (iv) the guidelines issued jointly by local and regional (...) supervisors and managers imply an organisational structuring of lines of information provision and command.

Conceptual Analysis, First Part

People staff enterprises, the components of infrastructures with which we are concerned, i.e., for which we develop software. The larger these enterprises — these infrastructure components — the more need there is for management and organisation. The role of management is roughly, for our purposes, twofold: first, to perform strategic, tactical and operational work, to set strategic, tactical and operational policies — and to see to it that they are followed. The role of management is, second, to react to adverse conditions, that is, to unforeseen situations, and to decide how they should be handled, i.e., conflict resolution.

Policy setting should help non-management staff operate normal situations — those for which no management interference is thus needed. And management "backstops" problems: management takes these problems off the shoulders of non-management staff.

To help management and staff know who's in charge wrt. policy setting and problem handling, a clear conception of the overall organisation is needed. Organisation defines lines of communication within management and staff, and between these. Whenever management and staff has to turn to others for assistance they usually, in a reasonably well-functioning enterprise, follow the command line: the paths of organigrams — the usually hierarchical box and arrow/line diagrams.

Methodological Consequences

The *management and organisation* model of a domain is a partial specification; hence all the usual abstraction and modelling principles, techniques and

[9] That enactment may possibly imply the movement of several trains incident upon several stations: the one at which the manager is located, as well as possibly at neighbouring stations.

tools apply. More specifically, management is a set of predicate functions, or of observer and generator functions These either parametrise other, the operations functions, that is, determine their behaviour, or yield results that become arguments to these other functions.

Organisation is thus a set of constraints on communication behaviours. Hierarchical, rather than linear, and matrix structured organisations can also be modelled as sets (of recursively invoked sets) of equations.

Conceptual Analysis, Second Part

To relate classical organigrams to formal descriptions we first show such an organigram (Fig. 3), and then we show schematic processes which — for a rather simple scenario — model managers and the managed!

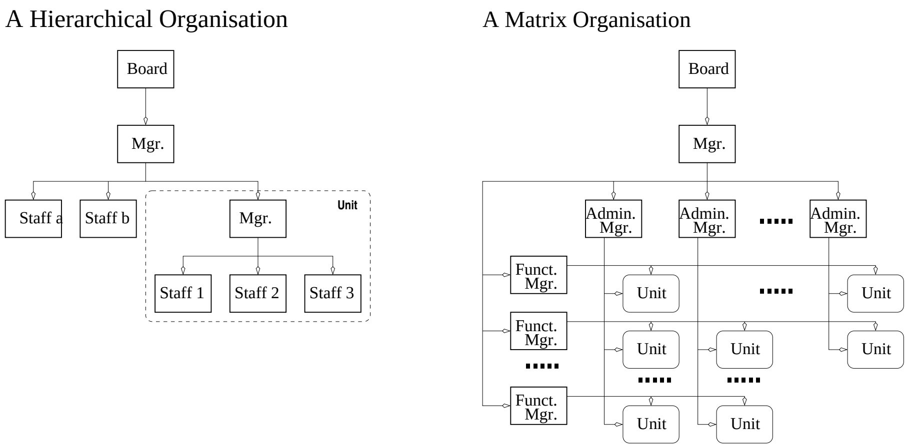

Fig. 3. Organisational structures

Based on such a diagram, and modelling only one neighbouring group of a manager and the staff working for that manager we get a system in which one manager, mgr, and many staff, stf, coexist or work concurrently, i.e., in parallel. The mgr operates in a context and a state modelled by ψ. Each staff, stf(i) operates in a context and a state modelled by sσ(i).

type
  Msg, Ψ, Σ, Sx
  SΣ = Sx $\underset{m}{\rightarrow}$ Σ
channel
  { ms[i]:Msg | i:Sx }
value
  sσ:SΣ, ψ:Ψ

sys: **Unit** $\rightarrow$ **Unit**
sys() $\equiv \parallel$ { stf(i)(sσ(i)) | i:Sx } $\parallel$ mg(ψ)

In this system the manager, mgr, (1) either broadcasts messages, m, to all staff via message channel ms[i]. The manager's concoction, m_out(ψ), of the message, msg, has changed the manager state. Or (2) is willing to receive messages, msg, from whichever staff i the manager sends a message. Receipt of the message changes, m_in(i,m)(ψ), the manager state. In both cases the manager resumes work as from the new state. The manager chooses — in this model — which of thetwo things (1 or 2) to do by a so-called non-deterministic internal choice ($\sqcap$).

mg: $\Psi \rightarrow$ **in**,**out** {ms[i]|i:Sx} **Unit**
mg(ψ) $\equiv$
(1) (**let** (ψ',m)=m_out(ψ) **in** $\parallel${ms[i]!m|i:Sx};mg(ψ')**end**)
$\sqcap$
(2) (**let** ψ'=$\sqcap${**let** m=ms[i]? **in** m_in(i,m)(ψ) **end**|i:Sx} **in** mg(ψ') **end**)

m_out: $\Psi \rightarrow \Psi \times$ MSG,
m_in: Sx $\times$ MSG $\rightarrow \Psi \rightarrow \Psi$

And in this system, staff i, stf(i), (1) either is willing to receive a message, msg, from the manager, and then to change, st_in(msg)(σ), state accordingly, or (2) to concoct, st_out(σ), a message, msg (thus changing state) for the manager, and send it ms[i]!msg. In both cases the staff resumes work as from the new state. The staff member chooses — in this model — which of thetwo "things" (1 or 2) to do by a non-deterministic internal choice ($\sqcap$).

st: i:Sx $\rightarrow \Sigma \rightarrow$ **in**,**out** ms[i] **Unit**
stf(i)(σ) $\equiv$
(1) (**let** m = ms[i]? **in** stf(i)(stf_in(m)(σ)) **end**)
$\sqcap$
(2) (**let** (σ',m) = st_out(σ) **in** ms[i]!m; stf(i)(σ') **end**)

st_in: MSG $\rightarrow \Sigma \rightarrow \Sigma$,
st_out: $\Sigma \rightarrow \Sigma \times$ MSG

Both manager and staff processes recurse (i.e., iterate) over possibly changing states. The management process non-deterministically, internal choice, "alternates" between "broadcast"-issuing orders to staff and receiving individual messages from staff. Staff processes likewise non-deterministically, internal choice, alternate between receiving orders from management and issuing individual messages to management.

The conceptual example also illustrates modelling stakeholder behaviours as interacting (here CSP-like) processes.

On Modelling Management and Organisation

Management and organisation basically spans entity, function, event and behaviour intensities and thus typically require the full spectrum of modelling techniques and notations — summarised in the two "On Modelling ..." paragraphs at the end of the two previous sections.

4.4 Rules and Regulations

- *By a domain* **rule** *we shall understand some text (in the domain) which prescribes how people or equipment are expected to behave when dispatching their duty, respectively when performing their function.*
- *By a domain* **regulation** *we shall understand some text (in the domain) which prescribes what remedial actions are to be taken when it is decided that a rule has not been followed according to its intention.*

Example 17. **Trains at Stations:**

- Rule: In China the arrival and departure of trains at, respectively from, railway stations is subject to the following rule:

  In any three-minute interval at most one train may either arrive to or depart from a railway station.
- Regulation: *If it is discovered that the above rule is not obeyed*, then there is some regulation which prescribes administrative or legal management and/or staff action, as well as some correction to the railway traffic.

•

Example 18. **Trains Along Lines:**

- Rule: In many countries railway lines (between stations) are segmented into blocks or sectors. The purpose is to stipulate that if two or more trains are moving along the line, then:

  There must be at least one free sector (i.e., without a train) between any two trains along a line.
- Regulation: *If it is discovered that the above rule is not obeyed*, then there is some regulation which prescribes administrative or legal management and/or staff action, as well as some correction to the railway traffic.

•

A Meta-characterisation of Rules and Regulations

At a meta-level, i.e., explaining the general framework for describing the syntax and semantics of the human-oriented domain languages for expressing rules and regulations, we can say the following: There are, abstractly speaking, usually three kinds of languages involved wrt. (i.e., when expressing) rules and regulations (respectively when invoking actions that are subject to rules and regulations). Two languages, Rules and Reg, exist for describing rules, respectively regulations; and one, Stimulus, exists for describing the form of the [always current] domain action stimuli.

A syntactic stimulus, sy_sti, denotes a function, se_sti:STI: $\Theta \to \Theta$, from any configuration to a next configuration, where configurations are those of the system being subjected to stimulations. A syntactic rule, sy_rul:Rule, stands for, i.e., has as its semantics, its meaning, rul:RUL, a predicate over current and next configurations, $(\Theta \times \Theta) \to \mathbf{Bool}$, where these next configurations have been brought about, i.e., caused, by the stimuli. These stimuli express: If the predicate holds then the stimulus will result in a valid next configuration.

type
  Stimulus, Rule, Θ
  STI = $\Theta \to \Theta$
  RUL = $(\Theta \times \Theta) \to \mathbf{Bool}$
value
  meaning: Stimulus $\to$ STI
  meaning: Rule $\to$ RUL

  valid: Stimulus $\times$ Rule $\to \Theta \to \mathbf{Bool}$
  valid(sy_sti,sy_rul)(θ) $\equiv$ meaning(sy_rul)(θ,(meaning(sy_sti))(θ))

  valid: Stimulus $\times$ RUL $\to \Theta \to \mathbf{Bool}$
  valid(sy_sti,se_rul)(θ) $\equiv$ se_rul(θ,(meaning(sy_sti))(θ))

A syntactic regulation, sy_reg:Reg (related to a specific rule), stands for, i.e., has as its semantics, its meaning, a semantic regulation, se_reg:REG, which is a pair. This pair consists of a predicate, pre_reg:Pre_REG, where Pre_REG = $(\Theta \times \Theta) \to \mathbf{Bool}$, and a domain configuration-changing function, act_reg:Act_REG, where Act_REG = $\Theta \to \Theta$, that is, both involving current and next domain configurations. The two kinds of functions express: If the predicate holds, then the action can be applied.

The predicate is almost the inverse of the rules functions. The action function serves to undo the stimulus function.

type
  Reg
  Rul_and_Reg = Rule $\times$ Reg
  REG = Pre_REG $\times$ Act_REG
  Pre_REG = $\Theta \times \Theta \to \mathbf{Bool}$
  Act_REG = $\Theta \to \Theta$
value
  interpret: Reg $\to$ REG

The idea is now the following: Any action of the system, i.e., the application of any stimulus, may be an action in accordance with the rules, or it may not.

Rules therefore express whether stimuli are valid or not in the current configuration. And regulations therefore express whether they should be applied, and, if so, with what effort.

More specifically, there is usually, in any current system configuration, given a set of pairs of rules and regulations. Let (sy_rul,sy_reg) be any such pair. Let sy_sti be any possible stimulus. And let θ be the current configuration. Let the stimulus, sy_sti, applied in that configuration result in a next configuration, θ', where $\theta' = (\mathsf{meaning}(\mathsf{sy\_sti}))(\theta)$. Let θ' violate the rule, $\sim\mathsf{valid}(\mathsf{sy\_sti},\mathsf{sy\_rul})(\theta)$, then if predicate part, pre_reg, of the meaning of the regulation, sy_reg, holds in that violating next configuration, $\mathsf{pre\_reg}(\theta,(\mathsf{meaning}(\mathsf{sy\_sti}))(\theta))$, then the action part, act_reg, of the meaning of the regulation, sy_reg, must be applied, $\mathsf{act\_reg}(\theta)$, to remedy the situation.

axiom
  ∀ (sy_rul,sy_reg):Rul_and_Regs •
    let se_rul = meaning(sy_rul),
        (pre_reg,act_reg) = meaning(sy_reg) **in**
    ∀ sy_sti:Stimulus, θ:Θ •
      ∼valid(sy_sti,se_rul)(θ)
        ⇒ pre_reg(θ,(meaning(sy_sti))(θ))
            ⇒ ∃ nθ:Θ • act_reg(θ)=nθ ∧ se_rul(θ,nθ)
    end

It may be that the regulation predicate fails to detect applicability of regulations actions. That is, the interpretation of a rule differs, in that respect, from the interpretation of a regulation. Such is life in the domain, i.e., in actual reality

On Modelling Rules and Regulations

Usually rules (as well as regulations) are expressed in terms of domain entities, including those grouped into "the state", functions, events, and behaviours. Thus the full spectrum of modelling techniques and notations may be needed. Since rules usually express properties one often uses some combination of axioms and wellformedness predicates. Properties sometimes include temporality and hence temporal notations (like Duration Calculus or Temporal Logic of Actions) are used. And since regulations usually express state (restoration) changes one often uses state changing notations (such as found in B, RSL, VDM-SL, and Z). In some cases it may be relevant to model using some constraint satisfaction notation [2] or some Fuzzy Logic notations [65].

4.5 Scripts and Licensing Languages

- *By a domain* **script** *we shall understand the structured, almost, if not outright, formally expressed, wording of a rule or a regulation that has legally binding power, that is, which may be contested in a court of law.*

Example 19. **A Casually Described Bank Script:** We deviate, momentarily, from our line of railway examples, to exemplify one from banking. Our formulation amounts to just a (casual) rough sketch. It is followed by a series of four large examples. Each of these elaborate on the theme of (bank) scripts.

The problem area is that of how repayments of mortgage loans are to be calculated. At any one time a mortgage loan has a balance, a most recent previous date of repayment, an interest rate and a handling fee. When a repayment occurs, then the following calculations shall take place: (i) the interest on the balance of the loan since the most recent repayment, (ii) the handling fee, normally considered fixed, (iii) the effective repayment — being the difference between the repayment and the sum of the interest and the handling fee — and the new balance, being the difference between the old balance and the effective repayment.

We assume repayments to occur from a designated account, say a demand/deposit account. We assume that bank to have designated fee and interest income accounts.

(i) The interest is subtracted from the mortgage holder's demand/deposit account and added to the bank's interest (income) account. (ii) The handling fee is subtracted from the mortgage holder's demand/deposit account and added to the bank's fee (income) account. (iii) The effective repayment is subtracted from the mortgage holder's demand/deposit account and also from the mortgage balance. Finally, one must also describe deviations such as overdue repayments, too large, or too small repayments, and so on. •

Example 20. **A Formally Described Bank Script:**

First we must informally and formally define the bank state:

There are clients (c:C), account numbers (a:A), mortgage numbers (m:M), account yields (ay:AY) and mortgage interest rates (mi:MI). The bank registers, by client, all accounts (ρ:A_Register) and all mortgages (μ:M_Register). To each account number there is a balance (α:Accounts). To each mortgage number there is a loan (ℓ:Loans). To each loan is attached the last date that interest was paid on the loan.

value
  r, r′:**Real axiom** ...
type
  C, A, M, Date
  AY′ = **Real**, AY = {| ay:AY′ • 0<ay≤r |}
  MI′ = **Real**, MI = {| mi:MI′ • 0<mi≤r′ |}
  Bank′ = A_Register × Accounts × M_Register × Loans
  Bank = {| β:Bank′ • wf_Bank(β)|}
  A_Register = C $\overrightarrow{m}$ A-**set**
  Accounts = A $\overrightarrow{m}$ Balance
  M_Register = C $\overrightarrow{m}$ M-**set**
  Loans = M $\overrightarrow{m}$ (Loan × Date)
  Loan,Balance = P
  P = **Nat**

Then we must define well-formedness of the bank state:

```
value
   ay:AY, mi:MI

   wf_Bank: Bank → Bool
   wf_Bank(ρ,α,μ,ℓ) ≡ ∪ rng ρ = dom α ∧ ∪ rng μ = dom ℓ
axiom
   ay<mi [ ∧ ... ]
```

We — perhaps too rigidly — assume that mortgage interest rates are higher than demand/deposit account interest rates: ay<mi.

Operations on banks are denoted by the commands of the bank script language. First the syntax:

```
type
   Cmd = OpA | CloA | Dep | Wdr | OpM | CloM | Pay
   OpA == mkOA(c:C)
   CloA == mkCA(c:C,a:A)
   Dep == mkD(c:C,a:A,p:P)
   Wdr == mkW(c:C,a:A,p:P)
   OpM == mkOM(c:C,p:P)
   Pay == mkPM(c:C,a:A,m:M,p:P,d:Date)
   CloM == mkCM(c:C,m:M,p:P)
   Reply = A | M | P | OkNok
   OkNok == ok | notok
value
   period: Date × Date → Days [for calculating interest]
   before: Date × Date → Bool [first date is earlier than last date]
```

And then the semantics:

```
   int_Cmd(mkPM(c,a,m,p,d))(ρ,α,μ,ℓ) ≡
      let (b,d′) = ℓ(m) in
      if α(a)≥p
         then
            let i = interest(mi,b,period(d,d′)),
                ℓ′ = ℓ † [m↦ℓ(m)−(p−i)]
                α′ = α † [a↦α(a)−p,a_i↦α(a_i)+i] in
            ((ρ,α′,μ,ℓ′),ok) end
         else
            ((ρ,α′,μ,ℓ),nok)
      end end
      pre c ∈ dom μ ∧ a ∈ dom α ∧ m ∈ μ(c)
      post before(d,d′)

      interest: MI × Loan × Days → P
```

•

The idea about scripts is that they can somehow be objectively enforced: that they can be precisely understood and consistently carried out by all stakeholders, eventually leading to computerisation. But they are, at all times, part of the domain.

Licensing Languages

A special form of scripts are increasingly appearing in some domains, notably the domain of electronic, or digital media, where these licences express that the licensor permits the licensee to render (i.e., play) works of proprietary nature CD ROM-like music, DVD-like movies, etc. while obligating the licensee to pay the licensor on behalf of the owners of these, usually artistic works. We refer to [29, 59, 67, 11] for papers and reports on license languages.

On Modelling Scripts

Scripts (as are licenses) are like programs (respectively like prescriptions program executions). Hence the full variety of techniques and notations for modelling programming (or specification) languages apply [18, 30, 64, 68, 73, 74]. [6, Chaps. 6–9] cover pragmatics, semantics and syntax techniques for defining languages.

4.6 Human Behaviour

- *By domain* **human behaviour** *we shall understand any of a quality spectrum of carrying out assigned work: from (i) careful, diligent and accurate, via (ii) sloppy dispatch, and (iii) delinquent work, to (iv) outright criminal pursuit.*

Example 21. **Banking — or Programming — Staff Behaviour:** Let us assume a bank clerk, "in ye olde" days, when calculating, say mortgage repayments (cf. Example 19).

We would characterise such a clerk as being *diligent,* etc., if that person carefully follows the mortgage calculation rules, and checks and double-checks that calculations "tally up", or lets others do so. We would characterise a clerk as being *sloppy* if that person occasionally forgets the checks alluded to above. We would characterise a clerk as being *delinquent* if that person systematically forgets these checks. And we would call such a person a *criminal* if that person intentionally miscalculates in such a way that the bank (and/or the mortgage client) is cheated out of funds which, instead, may be diverted to the cheater.

Let us, instead of a bank clerk, assume a software programmer charged with implementing an automatic routine for effecting mortgage repayments (cf. Example 20).

We would characterise the programmer as being *diligent* if that person carefully follows the mortgage calculation rules, and throughout the development verifies and tests that the calculations are correct with respect to the rules. We would characterise the programmer as being *sloppy* if that person forgets certain checks and tests when otherwise correcting the computing program under development. We would characterise the programmer as being *delinquent* if that person systematically forgets these checks and tests. And we would characterise the programmer as being a *criminal* if that person intentionally provides a program which miscalculates the mortgage interest, etc., in such a way that the bank (and/or the mortgage client) is cheated out of funds. •

Example 22. **A Human Behaviour Mortgage Calculation:**

Example 20 gave a semantics to the mortgage calculation request (i.e., command) as would a diligent bank clerk be expected to perform it. To express, that is, to model, how sloppy, delinquent, or outright criminal persons (staff?) could behave we must modify the int_Cmd(mkPM(c,a,m,p,d′))(ρ,α,μ,ℓ) definition.

$$
\begin{array}{l}
\mathsf{int\_Cmd(mkPM(c,a,m,p,d))}(\rho,\alpha,\mu,\ell) \equiv \\
\quad \textbf{let } \mathsf{(b,d')} = \ell(\mathsf{m}) \textbf{ in} \\
\quad \textbf{if } \mathsf{q}(\alpha(\mathsf{a}),\mathsf{p}) \ /*\ \alpha(\mathsf{a}) \leq \mathsf{p} \vee \alpha(\mathsf{a}) = \mathsf{p} \vee \alpha(\mathsf{a}) \leq \mathsf{p} \vee ...\ */ \\
\quad\quad \textbf{then} \\
\quad\quad\quad \textbf{let } \mathsf{i} = \mathsf{f_1(interest(mi,b,period(d,d')))}, \\
\quad\quad\quad\quad \ell' = \ell \dagger [\,\mathsf{m} \mapsto \mathsf{f_2}(\ell(\mathsf{m}) - (\mathsf{p} - \mathsf{i}))\,] \\
\quad\quad\quad\quad \alpha' = \alpha \dagger [\,\mathsf{a} \mapsto \mathsf{f_3}(\alpha(\mathsf{a}) - \mathsf{p}), a_i \mapsto \mathsf{f_4}(\alpha(a_i) + \mathsf{i}), \\
\quad\quad\quad\quad\quad\quad\quad a_{\text{“staff”}} \mapsto \mathsf{f}_{\text{“staff”}}(\alpha(a_{\text{“staff”}}) + \mathsf{i})\,] \textbf{ in} \\
\quad\quad\quad ((\rho,\alpha',\mu,\ell'),\mathsf{ok}) \textbf{ end} \\
\quad\quad \textbf{else} \\
\quad\quad\quad ((\rho,\alpha',\mu,\ell),\mathsf{nok}) \\
\quad \textbf{end end} \\
\quad \textbf{pre } \mathsf{c} \in \textbf{dom } \mu \wedge \mathsf{m} \in \mu(\mathsf{c})
\end{array}
$$

$$
\begin{array}{l}
\mathsf{q}\colon \mathsf{P} \times \mathsf{P} \stackrel{\sim}{\to} \textbf{Bool} \\
\mathsf{f_1,f_2,f_3,f_4,f_{\text{“staff”}}}\colon \mathsf{P} \stackrel{\sim}{\to} \mathsf{P} \ \ [\,\text{typically: } \mathsf{f}_{\text{“staff”}} = \lambda \mathsf{p}.\mathsf{p}\,]
\end{array}
$$

•

The predicate q and the functions f_1, f_2, f_3, f_4 and $f_{\text{“staff”}}$ of Example 22 are deliberately left undefined. They are being defined by the “staffer” when performing (incl., programming) the mortgage calculation routine.

The point of Example 22 is that one must first define the mortgage calculation script precisely as one would like to see the diligent staff (programmer) to perform (incl., correctly program) it before one can “pinpoint” all the places where lack of diligence may “set in”. The invocations of q, f_1, f_2, f_3, f_4 and $f_{\text{“staff”}}$ designate those places.

The point of Example 22 is also that we must first domain-define, “to the best of our ability” all the places where human behaviour may play other than a desirable role. If we cannot, then we cannot claim that some requirements aim at countering undesirable human behaviour.

A Meta-characterisation of Human Behaviour

Commensurate with the above, humans interpret rules and regulations differently, and not always consistently — in the sense of repeatedly applying the same interpretations.

Our final specification pattern is therefore:

type
  Action = $\Theta \xrightarrow{\sim} \Theta$-**infset**
value
  hum_int: Rule $\rightarrow \Theta \rightarrow$ RUL-**infset**
  action: Stimulus $\rightarrow \Theta \rightarrow \Theta$
  hum_beha: Stimulus $\times$ Rules $\rightarrow$ Action $\rightarrow \Theta \xrightarrow{\sim} \Theta$-**infset**
  hum_beha(sy_sti,sy_rul)$(\alpha)(\theta)$ **as** θset
    post
      θset $= \alpha(\theta) \wedge$ action(sy_sti)$(\theta) \in \theta$set
      $\wedge\ \forall\ \theta'{:}\Theta \bullet \theta' \in \theta$set $\Rightarrow$
        $\exists$ se_rul:RUL•se_rul $\in$ hum_int(sy_rul)$(\theta) \Rightarrow$ se_rul(θ,θ')

The above is, necessarily, sketchy: There is a possibly infinite variety of ways of interpreting some rules. A human, in carrying out an action, interprets applicable rules and chooses one which that person believes suits some (professional, sloppy, delinquent or criminal) intent. "Suits" means that it satisfies the intent, i.e., yields **true** on the pre/post-configuration pair, when the action is performed — whether as intended by the ones who issued the rules and regulations or not. We do not cover the case of whether an appropriate regulation is applied or not.

The above-stated axioms express how it is in the domain, not how we would like it to be. For that we have to establish requirements.

On Modelling Human Behaviour

To model human behaviour is, "initially", much like modelling management and organisation. But only 'initially'. The most significant human behaviour modelling aspect is then that of modelling non-determinism and looseness, even ambiguity. So a specification language which allows specifying non-determinism and looseness (like CafeOBJ and RSL) is to be preferred.

4.7 Completion

Domain acquisition results in typically up to thousands of units of domain descriptions. Domain analysis subsequently also serves to classify which facet any one of these description units primarily characterises. But some such "compartmentalisations" may be difficult, and may be deferred till the step of "completion". It may then be, "at the end of the day", that is, after all of the above facets have been modelled that some description units are left as not having been described, not deliberately, but "circumstantially". It then behoves the domain engineer to fit these "dangling" description units into suitable parts of the domain description. This "slotting in" may be simple, and all is fine. Or it may be difficult. Such difficulty may be a sign that the chosen model, the chosen description, in its selection of entities, functions, events and behaviours to model — in choosing these over other possible selections of phenomena and concepts is not appropriate. Another attempt must be made. Another selection, another abstraction of entities, functions, etc., may need be chosen. Usually however, after having chosen the abstractions of the intrinsic phenomena and concepts, one can start checking whether "dangling" description units can be fitted in "with ease".

4.8 Integrating Formal Descriptions

We have seen that to model the full spectrum of domain facets one needs not one, but several specification languages. No single specification language suffices. It seems highly unlikely and it appears not to be desirable to obtain a single, "universal" specification language capable of "equally" elegantly, suitably abstractly modelling all aspects of a domain. Hence one must conclude that the full modelling of domains shall deploy several formal notations. The issues are then the following which combinations of notations to select, and how to make sure that the combined specification denotes something meaningful. The ongoing series of "Integrating Formal Methods" conferences [3, 27, 14, 12, 66] is a good source for techniques, compositions and meaning.

5 On Modelling

A number of remarks may be in order — especially given the terseness of many of the statements and examples given in the previous two sections.

Abstractions: In abstraction we conscientiously omit some properties deciding instead to focus on other properties. Whenever an abstraction is presented one should carefully discuss the abstraction choice. This has not been done in this chapter.

Models: We model both domain phenomena and domain concepts. The latter are abstractions of phenomena. However, in modelling a phenomenon we "make it into" a concept. (So models of domain concepts could be said to be meta-concepts.) The point is: our domain description designates a model, or, more generally, a class of models each of which satisfies the description. These models are not the domain, only an abstract and only a model of it.

Real-world Constraints: Published models of, for example, railways[10], contain axioms that reflect the physical constraints of the world of physics: trains arrive after they have departed, if a train appears in the traffic at times t and t' then it appears at all time in between these times, etc. We have not dealt with this aspects of modelling here — but see [7, Chap. 10] whose modelling principles and techniques are covered "in depth" in [5, 6].

Type Invariants: We have shown a few examples. Usually as part of axioms that govern type, or as subtype definitions, or as explicitly defined functions over types, say A, then usually named wf_A. Again we have not dwelt on this modelling aspect here — but see [5, Chaps. 13–18] which also covers principles and techniques of modelling type invariants.

Vagaries of Domains: Care has to be taken too model the domain not only as we would prefer the properties of its entities, functions, events and behaviours to be, but as they are. Namely stochastically varying, unpredictable, erroneous, etc. Some modelling aspects of this, notably wrt. human behaviour, has been mentioned — and again we refer to the full book [5, 6, 7] for a more comprehensive treatment.

Monitoring of Domain States: In Sect. 4.6 we gave an example varieties of human behaviour — the second version of int_Cmd(mkPM(c,a,m,p,d)) (ρ,α,μ,ℓ). We did not follow up on any monitoring (audit of bank accounts) wrt. this behaviour. We could, and should, i.e., must model monitoring (and related control), if there is a notion of monitoring (etc.) in the domain. We have here assumed that such a monitoring, viz.: a bank audit, today, 2007, might be a natural task for computing and would hence not describe it as part of the domain, but as part of a domain requirements prescription.

Incompleteness and Inconsistency: It is unavoidable that early iterations of domain description are incomplete, in fact may remain so throughout. It may not be detrimental to the overall objective of the emerging domain description as long as the "lacuna" of incompletenesses are well identified and acceptable. It is also difficult to avoid that early iterations of domain description are inconsistent. Domain description analysis, in the form of dispatching proof obligations raised by the formalisation is one source of discovering inconsistencies. Another, earlier source is that of validation in which two forms of inconsistencies may be identified. One in which an inconsistency has its roots in conflicting statements about the domain from stakeholders supposedly of a same tightly knit group. In that case the domain engineer must resolve it through mediation within such a group. The other source has the inconsistency take its root in conflicting statements about the domain from stakeholders of different groups. Here, not the domain engineer, but stakeholder management must intervene and produce a consistent description.

---

[10] http://www.railwaydomain.org/PDF/tb.pdf

6 From Domain Models to Requirements Models

One rôle for $\mathcal{D}$omain descriptions is to serve as a basis for constructing , $\mathcal{R}$equirements prescriptions. The purpose of constructing $\mathcal{R}$equirements prescriptions is to specify properties (not implementation) of a $\mathcal{M}$achine. The $\mathcal{M}$achine is the hardware (equipment) and the software that together implements the requirements. The implementation relations is

$$\mathcal{D}, \mathcal{M} \models \mathcal{R}$$

The $\mathcal{M}$achine is proven to implement the $\mathcal{R}$equirements in the context of [assumptions about] the $\mathcal{D}$omain. That is, proofs of correctness of the $\mathcal{M}$achine wrt. the $\mathcal{R}$equirements often refer to properties of the $\mathcal{D}$omain.

The $\mathcal{D}, \mathcal{M} \models \mathcal{R}$ formula expresses a context in which the individual phases of software development takes place. Awareness of this contextualisation was the basis for the **ProCoS** project [57, 53] and is also at the basis of the later work by Jackson et al. [76, 28].

6.1 Requirements Engineering Stages

As for domain engineering, requirements engineering is pursued in a number of iterated and sometimes concurrent stages:

1. identification of and interaction with requirements stakeholders,
2. domain requirements development, modelling and analysis,
3. interface requirements development, modelling and analysis,
4. machine requirements development, modelling and analysis,
5. further analysis work,
6. requirements validation, and
7. requirements satisfiability and feasibility study.

In this section we shall only cover the modelling parts of stages 2., 3. and 4.

• • •

In the next three subsections we very briefly sketch the relationships between a domain description and a requirements prescription. We introduce the notions of domain requirements (elsewhere called functional requirements), interface requirements (elsewhere sometimes called user requirements), and machine requirements (elsewhere called non-functional requirements). We find our labels semantically less “coded”.

6.2 Domain Requirements

By a domain requirements we understand a requirements (prescription) which uses only terms from the domain description — and which additionally, in the requirements narrative uses the terms shall or must (and probably not should or may).

First, in a concrete sense, you copy the domain description and call it a requirements prescription. Then that requirements prescription is subjected to a number of operations: (i) removal (projection away) of all those aspects not needed in the requirements; (ii) instantiation of remain aspects to the specifics of the client's domain; (iii) making determinate what is unnecessarily or undesirably non-deterministic in the evolving requirements prescription; (iv) extending it with concepts not feasible in the domain; and (v) fitting these requirements to those of related domains (say monitoring & control of public administration procedures). The result is called a domain requirements.

6.3 Interface Requirements

By an interface requirements we understand a requirements (prescription) which uses terms from both the domain description and the machine terminology — in addition to the terms shall or must. An interface requirements deal with the entities, functions (relations), events and behaviours that are share between the domain and the machine.

From the domain requirements one now constructs the interface requirements: First one selects all phenomena and concepts, entities, functions, event and behaviours shared with the environment of the machine (hardware + software) being requirements specified[11]. Then one requirements prescribe how each shared phenomenon and concept is being initialised and updated: entity initialisation and refreshment, function initialisation and refreshment (interactive monitoring and control of computations), and the physiological man-machine and machine-machine implements.

6.4 Machine Requirements

By a machine requirements we understand a requirements (prescription) which basically only uses terms from the machine terminology — in addition to the terms shall or must. A machine requirements may, however, refer, and then generically, to entities, functions, event and behaviours of the domain.

Finally one deals with machine requirements performance, dependability, maintainability, portability, etc., where dependability addresses such issues as availability, accessability, reliability, safety, security, etc.

[11] A domain phenomenon or concept that is also to "appear" in the machine, in some form or another, is said to be shared.

6.5 Domain Descriptions versus Requirements Prescriptions

On the background of the previous subsections (domain, interface and machine requirements) we can now characterise some differences between domain descriptions and requirements prescriptions.

Indicative versus Putative: A narrative domain description is indicative. It uses such verbs *is* and *are*. A domain description tells *what there is.*

A narrative requirements prescription is putative. It uses such verbs as *shall* and *must.* A prescription tells *what there will be.*

This distinction is not captured by the formalisations that we have shown. One could think of introducing some keywords for that purpose, but mathematically the meaning of such keywords is doubtful.

Computability: A formal domain description may very well denote mathematical entities which cannot be represented computationally, or mathematical functions (or relations) that are not computable. The purpose of a domain and interface requirements construction (from a domain description) is to render possibly in-computable entities and possibly in-computable functions computable (as the objective of software design is to make entity representations and function computations data structure-wise and algorithmically efficient).

Stability: It is often claimed that requirements prescriptions are unstable: that they change repeatedly during software development. We claim, in contrast, that once a domain description has reached a relative maturity, i.e., being relatively stable, then, using the domain and interface requirements "derivation" principles and techniques hinted at here (and covered in more details in [7, Chaps. 17–24]) then the resulting requirements prescriptions will be more, or even far more stable.

So, we have decomposed the "requirements instability" problem into three problems: (i) the "domain instability", (ii) the "domain description instability", and (iii) the "requirements prescription instability" problems. The domain instability problem, (i), has two roots: (i.1) the inherent changes of domain support technologies, management & organisation, rules & regulations facets etc., and (i.2) the inherent inability of any group of people (i.e., the stakeholders and the domain engineers) to express and hence capture all of a domain. As for (i.1), the scope of facet changes is more well-defined and their identification seems easier — so we may now be able to better cope with that part of the (former) "requirements instability" problem. As for (i.2), all there is to say is: "such is life"! The domain description instability problem, (ii), is due to misunderstandings, inconsistencies and incompletenesses of the stakeholders and the domain engineers' understanding of "their" the domain. By having decomposed the overall (former) "requirements instability" problem into smaller, more well defined problem areas, and by decomposing the domain description problem into several facets as well as forcing their formalisation, we can hope to now be better able to deal with this, the (ii), problem.

Finally, the (new, redefined) "requirements instability" problem area, (iii), is now of a size which is potentially much "smaller" than the former "requirements instability" problem area. Any actual "requirements prescription instability" is now due to a wrong decision by the stakeholders and domain engineers as to the desired projection, instantiation, determination, extension or fitting of the domain description at hand.

Overall we claim that the requirements engineering process — as we see it — is now based on a rather dramatically different basis than what is otherwise reported in the literature and certainly different from what is mostly practised in industry. Therefore generic claims of "requirements instability" based on what we obviously consider an antiquated requirements engineering approach, that such claims must be refined to claims of the kind: (i.1), (i.2), (ii) or (iii).

Domain Engineering versus Requirements Engineering Stages: The domain engineering phase involves the stages of (D1.) identification of and regular interaction with stakeholders, (D2.) domain (knowledge) acquisition, (D3.) domain analysis, (D4.) domain modelling, (D5.) domain verification, (D6.) domain validation and (D7.) domain theory formation. The requirements phase involves the stages of (R1.) identification of and regular interaction with stakeholders, (R2.–R4.) domain, interface and machine requirements development, modelling and analysis, (R5.) further analysis work, (R6.) requirements validation and (R7.) requirements satisfiability and feasibility study. Phases (D1.,R1.), (D5.,R5.) and (D6.,R6.) have similar names but their inputs, and hence their procedural steps (principles, techniques and tool) are different. The same is true, of course, for all phases, but we can say that requirements engineering phases (R2.), (R3.) and (R4.) each are "similar" to a combination of domain engineering phases (D2.), (D3.) and (D4.). Space considerations prevent us from further clarifications.

7 Why Domain Engineering?

7.1 Two Reasons for Domain Engineering

We believe that one can identify two almost diametrically opposed reasons for the pursuit of domain descriptions. One is utilitarian, concrete, commercial and engineering goal-oriented. It claims that domain engineering will lead to better software, and to development processes that can be better monitored and controlled. and the other is science-oriented. It claims that establishing domain theories is a necessity, that it must be done, whether we develop software or not.

We basically take the latter, the science, view, while, of course, noting the former, the engineering consequences. We will briefly look at these.

7.2 An Engineering Reason for Domain Modelling

In a recent e-mail, in response, undoubtedly to my steadfast, perhaps conceived as stubborn insistence, on domain engineering, Tony Hoare summed up his reaction to domain engineering as follows, and I quote[12]:

"There are many unique contributions that can be made by domain modelling.

1. The models describe all aspects of the real world that are relevant for any good software design in the area. They describe possible places to define the system boundary for any particular project.
2. They make explicit the preconditions about the real world that have to be made in any embedded software design, especially one that is going to be formally proved.
3. They describe the whole range of possible designs for the software, and the whole range of technologies available for its realisation.
4. They provide a framework for a full analysis of requirements, which is wholly independent of the technology of implementation.
5. They enumerate and analyse the decisions that must be taken earlier or later in any design project, and identify those that are independent and those that conflict. Late discovery of feature interactions can be avoided."

All of these issues are dealt with, one-by-one, and in depth, in Vol. 3 of my three volume book [7].

7.3 On a Science of Domains

Domain Theories

Although not brought out in this chapter the concept of domain theories must now be mentioned.

- *By a* **domain theory** *we shall understand a domain description together with lemmas, propositions and theorems that can be proved about the description — and hence can be claimed to hold in the domain.*

To create a domain theory the specification language must possess a proof system. It appears that an essence of possible theorems of — that is, laws about — domains can be found in laws of physics. For a delightful view of the law-based nature of physics — and hence possibly also of man-made universes we refer to Richard Feynman's Lectures on Physics [20]. But, whereas laws of physics are conjectures and may basically (still) be refuted, laws of man-made domains are less conjectural[13], but sometimes subject to change[14].

---

[12] E-Mail to Dines Bjørner, CC to Robin Milner et al., July 19, 2006

[13] In early versions of a domain description, one that has yet to be validated by domain stake holders, one might very well claim that the model is conjectural.

[14] Typically we would wish the intrinsics part of a domain model to be invariant, to reflect innate laws of the domain. And: typically management and organisation

A Scientific Reason for Domain Modelling

So, inasmuch as the above-listed issues of Sect. 7.2, are of course of utmost engineering importance, it is really, in our mind, the science issues that are foremost: We must first and foremost understand. There is no excuse for not trying to first understand. Whether that understanding can be "translated" into engineering tools and techniques is then another matter. But then, of course, it is nice that clear and elegant understanding also leads to better tools and hence better engineering. It usually does.

8 Conclusion

8.1 Summary

We have introduced the scientific and engineering concept of domain theories and domain engineering; and we have brought but a mere sample of the principles, techniques and tools that can be used in creating domain descriptions.

8.2 Grand Challenges of Informatics

To establish a reasonably trustworthy and believable theory of a domain, say the transportation, or just the railway domain, may take years, possibly 10–15! Similarly for domains such as the financial service industry, the market (of consumers and producers, retailers, wholesaler, distribution cum supply chain), health care, and so forth.

The current author urges younger scientists to get going! It is about time.

8.3 Acknowledgements

The author thank Springer for allowing this chapter — which is basically a rewriting and "condensing-editing" of Chaps. 5 and 11 of [7] to appear also in the present book. The author thanks Paul Boca and Jonathan P. Bowen for inviting him in the first place to give basically this chapter as a BCS FACS seminar in the summer of 2005.[15] The author thanks Tony Hoare for permission to quote his characterisation of domain engineering (Sect. 7.2).

and also rules and regulations are contain or reflect laws that may be valid for a time — only to be replaced by other "laws".

[15] At that time the book [5, 6, 7] was in the hands of the editors of Springer, i.e., not yet published.

9 Bibliographical Notes

To create domain descriptions, or requirements prescriptions, or software designs, properly, at least such as this author sees it, is a joy to behold. The beauty of carefully selected and balanced abstractions, their interplay with other such, the relations between phases, stages and steps, and many more conceptual constructions make software engineering possibly the most challenging intellectual pursuit today. For this and more consult [5, 6, 7].

References

1. J.-R. Abrial: *The B Book: Assigning Programs to Meanings* (Cambridge University Press, Cambridge, England 1996)
2. K.R. Apt: *Principles of Constraint Programming* (Cambridge University Press, August 2003)
3. K. Araki, A. Galloway, K. Taguchi, editors. *IFM 1999: Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, York, UK, June 1999. Springer. Proceedings of 1st Intl. Conf. on IFM.
4. M. Bidoit, P.D. Mosses: CASL *User Manual* (Springer, 2004)
5. D. Bjørner: *Software Engineering, Vol. 1: Abstraction and Modelling* (Springer, 2006)
6. D. Bjørner: *Software Engineering, Vol. 2: Specification of Systems and Languages* (Springer, 2006)
7. D. Bjørner: *Software Engineering, Vol. 3: Domains, Requirements and Software Design* (Springer, 2006)
8. Edited by D. Bjørner, M. Henson: *Logics of Specification Languages* (Springer, 2007)
9. Edited by D. Bjørner, C.B. Jones: *The Vienna Development Method: The Meta-Language*, vol 61 of *LNCS* (Springer–Verlag, 1978)
10. Edited by D. Bjørner, C.B. Jones: *Formal Specification and Software Development* (Prentice-Hall, 1982)
11. D. Bjørner, A. Yasuhito, C. Xiaoyi, X. Jianwen: A Family of License Languages. Technical Report, JAIST, Graduate School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292 (2006)
12. E.A. Boiten, J. Derrick, G. Smith, editors. *IFM 2004: Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, London, England, April 4-7 2004. Springer. Proceedings of 4th Intl. Conf. on IFM. ISBN 3-540-21377-5.
13. E. Börger, R. Stärk: *Abstract State Machines. A Method for High-Level System Design and Analysis* (Springe, 2003) p 452
14. M.J. Butler, L. Petre, K. Sere, editors. *IFM 2002: Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, Turku, Finland, May 15-18 2002. Springer. Proceedings of 3rd Intl. Conf. on IFM. ISBN 3-540-43703-7.
15. D. Cansell, D. Méry: *Logical Foundations of the B Method.* Computing and Informatics **22**, 1–2 (2003)
16. CoFI (The Common Framework Initiative): CASL *Reference Manual*, vol 2960 of *Lecture Notes in Computer Science (IFIP Series)* (Springer–Verlag, 2004)

17. W. Damm, D. Harel: *LSCs: Breathing Life into Message Sequence Charts.* Formal Methods in System Design **19** (2001) pp 45–80
18. J. de Bakker: *Control Flow Semantics* (The MIT Press, Cambridge, Mass., USA, 1995)
19. R. Diaconescu, K. Futatsugi, K. Ogata: *CafeOBJ: Logical Foundations and Methodology.* Computing and Informatics **22**, 1–2 (2003)
20. R. Feynmann, R. Leighton, M. Sands: *The Feynmann Lectures on Physics*, vol Volumes I–II–II (Addison-Wesley, California Institute of Technology 1963)
21. J.S. Fitzgerald, P.G. Larsen: *Developing Software using* `VDM-SL` (Cambridge University Press, The Edinburgh Building, Cambridge CB2 1RU, England 1997)
22. K. Futatsugi, R. Diaconescu: *CafeOBJ Report The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification* (World Scientific Publishing Co. Pte. Ltd., 5 Toh Tuck Link, SINGAPORE 596224. Tel: 65-6466-5775, Fax: 65-6467-7667, E-mail: wspc@wspc.com.sg 1998)
23. K. Futatsugi, A. Nakagawa, 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.
24. C.W. George, P. Haff, K. Havelund et al: *The RAISE Specification Language* (Prentice-Hall, Hemel Hampstead, England 1992)
25. C.W. George, A.E. Haxthausen: *The Logic of the RAISE Specification Language.* Computing and Informatics **22**, 1–2 (2003)
26. C.W. George, A.E. Haxthausen, S. Hughes et al: *The RAISE Method* (Prentice-Hall, Hemel Hampstead, England 1995)
27. W. Grieskamp, T. Santen, B. Stoddart, editors. *IFM 2000: Integrated Formal Methods*, volume of *Lecture Notes in Computer Science*, Schloss Dagstuhl, Germany, November 1-3 2000. Springer. Proceedings of 2nd Intl. Conf. on IFM.
28. C.A. Gunter, E.L. Gunter, M.A. Jackson, P. Zave: *A Reference Model for Requirements and Specifications.* IEEE Software **17**, 3 (2000) pp 37–43
29. C.A. Gunter, S.T. Weeks, A.K. Wright: Models and Languages for Digtial Rights. In: *Proc. of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)* (IEEE Computer Society Press, Maui, Hawaii, USA 2001) pp 4034–4038
30. C. Gunther: *Semantics of Programming Languages* (The MIT Press, Cambridge, Mass., USA, 1992)
31. D. Harel: *Statecharts: A Visual Formalism for Complex Systems.* Science of Computer Programming **8**, 3 (1987) pp 231–274
32. D. Harel: *On Visual Formalisms.* Communications of the ACM **33**, 5 (1988)
33. D. Harel, E. Gery: *Executable Object Modeling with Statecharts.* IEEE Computer **30**, 7 (1997) pp 31–42
34. D. Harel, H. Lachover, A. Naamad et al: *STATEMATE: A Working Environment for the Development of Complex Reactive Systems.* Software Engineering **16**, 4 (1990) pp 403–414
35. D. Harel, R. Marelly: *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine* (Springer-Verlag, 2003)
36. D. Harel, A. Naamad: *The STATEMATE Semantics of Statecharts.* ACM Transactions on Software Engineering and Methodology (TOSEM) **5**, 4 (1996) pp 293–333
37. M.C. Henson, S. Reeves, J.P. Bowen: *Z Logic and its Consequences.* Computing and Informatics **22**, 1–2 (2003)

38. ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992.
39. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1996.
40. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1999.
41. D. Jackson: *Software Abstractions Logic, Language, and Analysis* (The MIT Press, Cambridge, Mass., USA April 2006)
42. M.A. Jackson: *Principles of Program Design* (Academic Press, 1969)
43. M.A. Jackson: *Problems, Methods and Specialisation.* Software Engineering Journal **9**, 6 (1994) pp 249–255
44. M.A. Jackson: Problems and requirements (software development). In: *Second IEEE International Symposium on Requirements Engineering (Cat. No.95TH8040)* (IEEE Comput. Soc. Press, 1995) pp 2–8
45. M.A. Jackson: *Software Requirements & Specifications: a lexicon of practice, principles and prejudices* (Addison-Wesley Publishing Company, Wokingham, nr. Reading, England; E-mail: ipc@awpub.add-wes.co.uk 1995)
46. M.A. Jackson: *Software Requirements & Specifications: a lexicon of practice, principles and prejudices* (Addison-Wesley Publishing Company, Wokingham, nr. Reading, England; E-mail: ipc@awpub.add-wes.co.uk 1995)
47. M.A. Jackson: *The Meaning of Requirements.* Annals of Software Engineering **3** (1997) pp 5–21
48. M.A. Jackson: *Problem Frames — Analyzing and Structuring Software Development Problems* (Addison–Wesley, Edinburgh Gate, Harlow CM20 2JE, England 2001)
49. K. Jensen: *Coloured Petri Nets*, vol 1: Basic Concepts (234 pages + xii), Vol. 2: Analysis Methods (174 pages + x), Vol. 3: Practical Use (265 pages + xi) of *EATCS Monographs in Theoretical Computer Science* (Springer–Verlag, Heidelberg 1985, revised and corrected second version: 1997)
50. J. Klose, H. Wittke: An Automata Based Interpretation of Live Sequence Charts. In: *TACAS 2001*, ed by T. Margaria, W. Yi (Springer-Verlag, 2001) pp 512–527
51. L. Lamport: *The Temporal Logic of Actions.* Transactions on Programming Languages and Systems **16**, 3 (1995) pp 872–923
52. L. Lamport: *Specifying Systems* (Addison–Wesley, Boston, Mass., USA 2002)
53. Edited by H. Langmaack, W.P. de Roever, J. Vytopil: *Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*, vol 863 of *Lecture Notes In Computer Science* (Springer, Heidelberg, Germany 1994)
54. E. Luschei: *The Logical Systems of Lesniewski* (North Holland, Amsterdam, The Netherlands 1962)
55. S. Merz: *On the Logic of TLA+.* Computing and Informatics **22**, 1–2 (2003)
56. T. Mossakowski, A.E. Haxthausen, D. Sanella, A. Tarlecki: *CASL — The Common Algebraic Specification Language: Semantics and Proof Theory.* Computing and Informatics **22**, 1–2 (2003)
57. E.-R. Olderog: *Nets, Terms and Formulas: Three Views of Concurrent Processes and Their Relationship* (Cambridge University Press, 1991 (paperback 2005))
58. C.A. Petri: *Kommunikation mit Automaten* (Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962)
59. R. Pucella, V. Weissman: A Logic for Reasoning about Digital Rights. In: *Proc. of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)* (IEEE Computer Society Press, 2002) pp 282–294
60. W. Reisig: *Petri Nets: An Introduction*, vol 4 of *EATCS Monographs in Theoretical Computer Science* (Springer Verlag, 1985)

61. W. Reisig: *A Primer in Petri Net Design* (Springer Verlag, 1992)
62. W. Reisig: *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets* (Springer Verlag, 1998)
63. W. Reisig: *The Expressive Power of Abstract State Machines.* Computing and Informatics **22**, 1–2 (2003)
64. J.C. Reynolds: *The Semantics of Programming Languages* (Cambridge University Press, 1999)
65. F. Van der Rhee, H. Van Nauta Lemke, J. Dukman: *Knowledge Based Fuzzy Control of Systems.* IEEE Trans. Autom. Control **35**, 2 (1990) pp 148–155
66. J.M. Romijn, G.P. Smith, J.C. van de Pol, editors. *IFM 2005: Integrated Formal Methods*, volume 3771 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands, December 2005. Springer. Proceedings of 5th Intl. Conf. on IFM. ISBN 3-540-30492-4.
67. P. Samuelson: *Digital rights management* {*and, or, vs.*} *the law.* Communications of ACM **46**, 4 (2003) pp 41–45
68. D.A. Schmidt: *Denotational Semantics: a Methodology for Language Development* (Allyn & Bacon, 1986)
69. J.F. Sowa: *Knowledge Representation: Logical, Philosophical, and Computational Foundations* (Pws Pub Co, August 17, 1999)
70. J.M. Spivey: *Understanding Z: A Specification Language and its Formal Semantics*, vol 3 of *Cambridge Tracts in Theoretical Computer Science* (Cambridge University Press, 1988)
71. J.M. Spivey: *The Z Notation: A Reference Manual*, 2nd edn (Prentice Hall International Series in Computer Science, 1992)
72. Edited by J.T.J. Srzednicki, Z. Stachniak: *Lesniewski's lecture notes in logic* (Dordrecht, 1988)
73. R. Tennent: *The Semantics of Programming Languages* (Prentice–Hall Intl., 1997)
74. G. Winskel: *The Formal Semantics of Programming Languages* (The MIT Press, Cambridge, Mass., USA, 1993)
75. J.C.P. Woodcock, J. Davies: *Using Z: Specification, Proof and Refinement* (Prentice Hall International Series in Computer Science, 1996)
76. P. Zave, M.A. Jackson: *Four dark Corners of Requirements Engineering.* ACM Transactions on Software Engineering and Methodology **6**, 1 (1997) pp 1–30
77. C.C. Zhou, M.R. Hansen: *Duration Calculus: A Formal Approach to Real–time Systems* (Springer–Verlag, 2004)
78. C.C. Zhou, C.A.R. Hoare, A.P. Ravn: *A Calculus of Durations.* Information Proc. Letters **40**, 5 (1992)