Opening

From Domains to Requirements

The Triptych Approach to Software Engineering

A 15-18 Day Lecture Series

Dines Bjørner

Welcome

- These lectures are different !
- In these lectures
  - requirements engineering will now have
  - a solid basis in domain engineering:
    * we will focus more than half the course on domain engineeging,
    * and you will see how requirements can be "smoothly derived" from domain models.
- In other words: a new beginning for software engineering providing
  - formal foundations and
  - strong ability to informally describe domains and requirements.

Aims & Objectives

Aims

- We aim at covering crucial aspects of both domain and requirements engineering:
  - stakeholder identification and liaison;
  - domain and requirements acquisition;
  - business process engineering and reengineering;
  - analysis and terminologisation;
  - description and prescription; $\surd$
  - verification and validation;
  - theory formation; and
  - requirements feasibility and satisfiability;
- some lightly, some in more depth ($\surd$).

Objectives

- Awareness of crucial structure of software engineering
  - stakeholders
  - acquisition
  - business processes
  - analysis
  - terminology
  - **description**
  - **prescription**
  - verification
  - validation
  - theory formation
  - feasibility
  - satisfiability
- Ability to **describe** and **prescribe**
- Understanding of description facets
  - intrinsics
  - support technologies
  - mgt. & org.
  - rules & regulation
  - scripts and contracts
  - human behaviour
- Understanding of preescription facets
  - projection
  - instantiation
  - determination
  - extension
  - fitting
  - consolidation

Lecture Overview

Lectures 1–5

Lectures 6–10

Domain Engineering

Lectures 11–15

Requirements Engineering

Lectures 16–18

Course Project

- Lectures go hand-in-hand with a course project:
  - lectures in the mornings,
  - project tutoring in the afternoons.
- Project topics (choose one for entire class):
  - Airports
  - Air Traffic
  - Banks
  - Commodities Exchanges
  - Container Lines
  - Distribution Chains
  - The Internet
  - Logistics
  - The Market
  - Manufacturing
  - (Oil and Gas) Pipelines
  - The Web
- Class to work in 3 to 4 three person groups each on a part of the project topic.
- Each project group to work out, over 4 weeks, a project report.

Course Evaluation

Project

- Project reports to be evaluated separately from exam.

Exam

- A two hour written exam with 24 questions
- evaluated separately from project report.

Consolidated Evaluation

- Possibly a suitably weighted sum of the two evaluations.

Course Material

- Complete lecture support:
  Notes: **http://www2.imm.dtu.dk/~db/de+re-p.pdf**
  Slides: **http://www2.imm.dtu.dk/~db/de+re-s.pdf**
- Extensive refs. to Examples:
  Search lecturer's Web page: **http://www2.imm.dtu.dk/~db**

Faculty Seminars

- A number of faculty seminars are offered during the 3-4-5 week stay:
  1. **Mereologies in Computing Science**
     http://www2.imm.dtu.dk/˜db/bjorner-hoare75-p.pdf
  2. **An Emerging Domain Science**
     A Rôle for Stanisław Leśniewski's Mereology
     and Bertrand Russell's Philosophy of Logical Atomism
     http://www2.imm.dtu.dk/˜db/domain or **/landin.pdf**
  3. **Rôle of Domain Engineering in Software Development**
     Why Current Requirements Engineering is Flawed !
  4. **IT Security; the ISO Recommendation — an Analysis**
     http://www2.imm.dtu.dk/˜db/5lectures/it-system-security-ISO.pdf
  5. **Script and Contract Languages**
     http://www2.imm.dtu.dk/˜db/5lectures/license-languages.pdf

End of Opening

Lecture 0

These Lectures are Different !

0. Preface

0.1. These Lectures are Different !

- This textbook is different in a number of ways:

  1. **The Triptych Dogma** : The dogma "says":
     - Before software can be designed one must understand the requirements.
     - Before requirements can be prescribed one must understand the domain.

     This dogma carries the two main parts of the book:
     - Part8: Domains and
     - Part9: Requirements.

     No other 'Software Engineering' textbook
     - *(other than [TheSEBook3]*
       - *of the approximately 2400 page [TheSEBook1,TheSEBook2,TheSEBook3])*
     - *propagates this dogma.*

2. Domain Engineering :

- This is a new phase of software development.
- It is thoroughly treated in Part 8.
- It is explained and motivated in Parts 2–4.
  - *No other 'Software Engineering' textbook*
    - *(other than [TheSEBook3])*
    - *covers 'Domain Engineering' —*
    - *and the present volume*
  - *covers that topic in a novel (read: "improved") way.*

3. Derivation of Requirements from Domain Models :

- Requirements development is here presented
  * in a way which differs fundamentally and significantly
  * from how it has been presented by past textbooks on 'Software Requirements Engineering'.
- This novel and simpler approach, as based on careful domain descriptions, both in narrative and in formal form is thoroughly treated in Part 9.
  * *No other 'Software Engineering' textbook*
    · *(other than [TheSEBook3])*
  * *covers Requirements Engineering in this novel and logical way*
  * *and the current treatment significantly improves that of [TheSEBook3].*

4. **Proper Conceptualisation** (Parts 5–7):
   - Software development is a highly intellectual process.
   - Among the constituent sets of theories, principles and techniques of software development are those of
     * ‘Abstraction & Modelling’,
     * ‘Semiotics’ and
     * ‘Specification Ontology’.
   - These are treated in separate parts of these lectures.
   - *No other ‘Software Engineering’ textbook*
     * *(other than [TheSEBook1,TheSEBook2,TheSEBook3])*
   - *covers these three concepts,*
     * *‘Abstraction & Modelling’,*
     * *‘Semiotics’ and*
     * *‘Specification Ontology’,*
   - *in this simplified way.*

– We shall very briefly explain these three concepts.

4.1 Abstraction & Modelling Part 5:

– **Abstraction** relates to
  * conquering complexity of description through the judicious use of abstraction,
  * where abstraction, briefly,
    · is the act and result of omitting consideration
    · of (what would then be called) details while,
    · instead, focusing on
    · (what would therefore be called) important aspects.

- **Modelling** relates to choosing between
  * (i) property- and model-oriented specification;
  * (ii) a suitable balance between analogic, analytic and iconic modelling;
  * (iii) descriptive and prescriptive modelling
    · as for domain modelling,
    · respectively requirements modelling;
  * and (iv) extensional versus intentional models.
- Modelling also has to decide "for which purposes" a model shall serve:
  * to gain understanding,
  * to get inspiration and to inspire,
  * to present, educate and train,
  * to assert and predict and
  * to implement requirements derived from domain models.

4.2 Semiotics Part 6:

- Semiotics deal with
  - ∗ the form, i.e., the **syntax**, in which we express concepts;
  - ∗ the meaning, i.e., the **semantics**, of what is being expressed; and
  - ∗ the reason, i.e., the **pragmatics**, of why we express something and the chosen form of expression.
- Since all we really ever do
  - ∗ when expressing
    - · domains,
    - · requirements and
    - · software
  - ∗ is to produce textual documents
- it is of utmost importance that we command these three facets of semiotics.

4.3 Specification Ontology [1] Part 7:

- How do we present descriptions ?
- The technical means of expressing the phenomena and concepts of domains form a meta-ontology.
- And the description itself is an ontology of the domain.
- In Part 7 we advance three "faces" of ontological nature:
  * (i) the simple entity, operation, event and behaviour approach to description;
  * (ii) the mereology of simple entities; and
  * (iii) the laws of description !

Our treatment of 'Abstraction & Modelling', 'Semiotics' and 'Specification Ontology', above are quite novel and constitute, in our opinion, quite a significant improvement of [TheSEBook3].

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

5. Examples :

- The book carries more that 140 substantial
- both informal and formal examples.
- Almost half are several pages long.
- *No other 'Software Engineering' textbook*
  - *(not even [TheSEBook1,TheSEBook2,TheSEBook3])*

  carries so many informal&formal examples,
- *examples that are substantial —*
- *and the present volume ties the many examples more strongly together.*

6. Projects :

- In book Appendix A there is a list of annotated course project proposals.
- A course — based on this book — is proposed to consist of
  * both 'formal' class lectures — covering this book —
  * and 'informal' tutoring sessions —
    · advising students on how to proceed using the book in engineering
    · both a domain description
    · and a requirements prescription
    · for one of the projects listed in book Appendix A.
  * That appendix will give some hints, to both
    · lecturers (course project tutors) and
    · students.

  ∗ Hints to lecturers on how to use this book in the ‘formal’ class lectures is given in a separate booklet that is (i.e., will be) available on the Internet.

– We cannot overemphasise the pedagogical and didactical need to

  ∗ both give the ‘formal’ class lectures

  ∗ and the course project ‘informal’ tutoring sessions:

    • “*learn by doing*”

    • “*but on a science-based foundation*”.

0.2. Course Overview

- (Chapter 1) We start by providing a background for this study.

- (Chapter 2) We introduce the concepts of domains, that is, potential or actual application domains for software.

- (Chapter 3) We then motivate the study of domains where such studies aim at creating both precise informal and formal descriptions of domains
- – (and) where formal descriptions are limited to what we can today mathematically formalise.

- (Chapter 4) Abstraction and modelling are keywords in specifications
- and we shall therefore very briefly summarise a few key concepts –
  - including property- and model-oriented abstractions.
  - We shall also, likewise very briefly, overview a tool for formal abstraction:
    - the main specification language. `RSL`, of these lectures
    - and its use in achieving abstractions.

- (Chapter 5) We take a very brief look at issues of semiotics: pragmatics, semantics and syntax.
  - The prime goal of software engineering work is description, prescription and specification,
  - that is: producing documents, that is, informal and formal texts.
  - Texts have syntax —
  - what we write has meaning (i.e., semantics),
  - and the reason we wrote it down is motivated, i.e., is pragmatics.

- (Chapter 6) What is it that we are
  - describing (as for domains),
  - prescribing (as for requirements) and
  - specifying (as for software designs)?

  We shall suggest that the descriptions (etc.) focus on

  - entities and behaviours,
  - functions and events –

  and shall therefore

  - briefly summarise these concepts (and likewise briefly exemplify their abstract modelling)
  - before deploying this "specification ontology" in domain and in requirements engineering.

- (Chapter 7) Domain engineering is then outlined in terms of its many stages:

  i information document creation,

  ii identification of domain stake-holders,

  iii business process rough sketching,

  iv domain acquisition,

  v domain analysis and concept formation,

  vi domain terminologisation,

  vii domain modelling – the major stage –

  viii domain model verification (checking, testing),

  ix domain description validation, and

  x domain theory creation.

Emphasis is put on business process description (Sect.) and on the six sub-stages of domain modelling:

- (a) intrinsics,
- (b) support technologies,
- (c) management and organisation,
- (d) rules and regulations,
- (e) scripts and
- (f) human behaviour.

- A final section summarises the opening and closing stages of domain engineering: stakeholder identification and liaison, acquisition, business processes, terminoligisation, respectively verification, model checking, testing, validation and domain theory issues.

- (Chapter 8) It is finally outlined, in some detail, how major parts of requirements can be systematically "derived" from domain descriptions: in three major sub-stages:
  - [A] domain requirements,
  - [B] interface requirements and
  - [C] machine requirements – where our contribution is sôlely placed in sub-stages [A–B].

In this part it is briefly argued why current requirements engineering appears to be based on a flawed foundation.

- Emphasis is put on the pivotal steps of domain requirements in which
  - (a) business processes are re-engineering;
  - (b) domain requirements are projected, instantiated, made more deterministic, extended and fitted;
  - (c) interface requirements are "created" while considering the simple entities, functions, events and behaviours shared that are (to be) shared between the domain and the machine; and
  - (d) machine requirements are laboriously enumerated and instantiated.

- A final section summarises the opening and closing stages of requirements engineering: stakeholder identification and liaison, acquisition, business process re-engineering, terminoligisation, respectively verification, model checking, testing, validation, satisfiability & feasibility and requirements theory issues.

End of Lecture 0

These Lectures are Different !

Lecture 1

Background

1. Background

- This book is written on the background of three more-or-less independent lines of
  - (a) more than 40 years of speculations, by our community, about, proposals for, and, obviously, practice of software engineering;
  - (b) about 40 years of progress in program verification, and
  - (c) of almost 50 years of formal specification of first programming language semantics, then software designs, then requirements, and now, finally domains.

- This book is also written on the background of many efforts that seek to merge these lines — as witnessed in strand (c) above:
  - (d) notably there is the effort to express abstractions and their refinement, for example,
    - (e) such as these abstractions and refinements, with respect abstract data structures and abstract operations, were (first) facilitated in `VDM`, and,
    - (g) with respect to process abstractions, such as they were facilitated in `CSP`.

(1. **Background**)

- Over 30+ years of determined efforts in the areas of
  - formal specification languages and
  - refinement;
- and of their deployment in many industrial projects,
- this line of research and experimental development has been manifested in at least three notable forms:
  - (i) the systematic-to-rigorous development of an `Ada` compiler using `VDM`;
  - (ii) the commercialisation of an industry-strength tool set for the `VDM` `S`pecification `L`anguage, `VDM-SL`, by the Japanese software house CSK[2]; and
  - (iii) the publication of my three volume book on SE.

[2]http://www.csk.com/support_e/vdm/index.html

(1. **Background**)

- All this research and development,
  - (1) 35+ years of doing advanced type experimental, explorative and actually overseeing real, industry-strength commercial software developments,
  - (2) 30+ years of teaching the underlying approaches, semantics, formal specification, and refinement in a software engineering setting, and
  - (3) putting students on the road to found and direct some eight software houses (now with some 600 former students) — based on student MSc and PhD projects — and survive in the business,
- makes me conclude that the basic elements to be included in a proper software engineering education and to be regularly practised by the graduating software engineers include the following concepts:

(1. Background)

- (a) a firm grasp *on the simple use* of discrete mathematics: sets, Cartesians, sequences, maps, functions (including the λ-Calculus), and simple universal algebras;
- (b) a firm grasp *on the simple use* of mathematical logic;
- (c) a firm grasp *on the simple use* of abstract and concrete types and their values, sub-types and derived types (such as found in several formal specification languages);
- (d) a firm grasp *on the simple use* of the semiotics concepts of syntax, semantics and pragmatics, including the formalisation of syntax and semantics — in various forms: “classical” operational semantics, denotational semantics, structural operational semantics, etc.;

(1. **Background**)

- (e) a firm grasp *on the simple use* of property- as well as model-oriented abstractions as facilitated by such formal specification languages as `Alloy`, `B` and `Event B`, `RAISE`, `VDM` or `Z`;
- (f) a firm grasp *on the simple use* of the diagrammatic specification approaches provided by `finite state automata` and `finite state machines` (any reasonable textbook on formal languages and automata theory should do), `MSC` (message sequence charts), `Petri nets` and `Statecharts`;

- (g) a firm grasp *on the use* some temporal logic approach to specify time dependent behaviours, `DC` (duration calculus), `ITL` (interval temporal logic), the Pnueli/Manna approach, or `TLA+`; and
- (h) a firm grasp *on the simple use* of `CSP` (communicating sequential processes).

- This book will overview some, we think crucial, aspects of software engineering on this background. (We shall not cover Items (f–g).)

End of Lecture 1

Background

Lecture 2

What are Domains ?

2. What are Domains?

2.1. Delineation

Definition 1 – Domain:

- *By a domain, or, more precisely an application domain, we shall understand*
  - *(i) a suitably delineated area of human activity, that is,*
  - *(ii) a universe of discourse, something for which we have what we will call a domain-specific terminology,*
  - *(iii) such that this domain has reasonably clear interfaces to other such domains.*

■

Definition 2 – Domain Description:

- *By a domain description we shall understand*
  - *(i) a set of pairs of informal, for ex., English language, and formal, say mathematical, texts,*
  - *(ii) which are commensurate, that is, the English text "reads" the formulas, and*
  - *(iii) which describe the simple entities, operations, events and behaviours of a domain in a reasonably comprehensive manner.*

■

2.1.1. Elements, Aims and Objectives of Domain Science(I)

- What will emerge from this book are the contours of 'domain science':
  - the study and knowledge of domains.
- We shall here start the sketching of these contours.
- In order to better understand what domain engineering is about we contrast it to physics.
- But first we must make a distinction between the terms 'phenomenon' (phenomena) and 'concept' (concepts).

Definition 3 – **Phenomenon:** *By a phenomenon we understand*

- *an observable fact, that is,*
  - *a temporal or spatio/temporal individual (particular, "thing")*
  - *of sensory experience*
  - *as distinguished from a noumenon*[3]*,*
  - *that is a fact of scientific interest susceptible to scientific description and explanation.*

■

[3]Noumenon: a posited object or event as it appears in itself independent of perception by the senses.

Definition 4 – Concept: *By a concept we understand*

- *something conceived in the mind,*
- *a thought,*
- *an abstract or generic idea generalized from particular instances.*

■

2.1.2. Physics versus Domain Science

2.1.2.1. General

- Physicists study 'mother nature':

  "Physics (Greek: physis $\phi\upsilon\sigma\iota\varsigma$ meaning 'nature'), a natural science, is the study of matter and its motion through space-time and all that derives from these, such as energy and force. More broadly, it is the general analysis of nature, conducted in order to understand how the world and universe behave." [Wikipedia]

- Domain scientists and engineers study 'domains':
  - *"Domains are here seen as predominantly man-made universes,*
  - *that is, as areas of human activity, where the emphasis is on*
    - *the structures (entities) conceived and built by humans (the domain owners, managers, designers, domain enterprise workers, etc.),*
    - *and the operations that are initially requested, or triggered, by humans (the domain users)."*

- In physics (as characterised above) the physicists,
  - in principle, **do not include** human actions and behaviour in their study.[4]
- In domain science and engineering the scientists and engineers,
  - in principle, **do include** human actions and behaviours in their study.

[4] The claimed possibility that humans are the origin, through their use of fossil energy sources, of the depletion of the ozon layer, does not mean that the physicists, in their model include human actions and behaviour: if physicists do consider humans as the "culprits", then that still does not enter into their models !

- In physics physicists model
  - **usually continuous** state values of the chosen sub-universe, that is,
  - the dynamics of observable or postulated state component values,
  - and their principle tools are those of
    - differential equations,
    - integral calculi,
    - statistics, etc.
  - Space and time plays a core rôle.

- In domain science and engineering the scientists and engineers model
  - (i) algebraic[5] structures of the chosen sub-universe
    - (in addition to their **usually discrete** “state” values and operations),
  - (ii) how simple entities are composed,
    - (not only just their atomic but also composite values),
  - (iii) how these structures may expand or retract, that is,
    - operations on structures, not just on values.
  - Space and time normally plays only a secondary rôle.

---

[5] Recall that an algebra is
* a usually finite set
* of possibly infinite classes (i.e., types)
* of usually discrete entities
* and a usually finite set of operations
* whose signature ranges of the entity types.

- The tools of domain scientists and engineers are those of
  - careful, precise informal (i.e., narrative) natural language and
  - likewise careful abstractions
    - expressed in some formal specification languages
    - emphasising the algebraic nature of entities and their operations.
  - That is, tools that originate with computer and computing scientists.

- Why not use the same tools as physicists do?
  - Well, they are simply not suited for the problems at hand.
    - ∗ Firstly the states of physics typically vary continuously,
    - ∗ whereas those of domains typically vary in discrete steps.
    - ∗ Secondly the number of state variables of physics do usually not vary,
    - ∗ whereas those of domain do — whole structures “collapse” or “expand” (sometimes “wholesale”, sometimes “en detail”).
    - ∗ Thirdly the models of physics, by comparison to those of domains, contain “only a few types” of oftentimes thousands of state variables — almost all modelled as reals, or vectors, matrices, tensors, etc., of reals,
    - ∗ whereas those of domains contain very many, quite different types — sometimes atomic, sometimes composite, but rarely modelled in matrix form.

- Models of physics, as already mentioned, express continuous phenomena.
- Models of domains, as also already mentioned, express logic properties of discrete, algebraic structures.
- For those and several other reasons
  - the tools of physicists
  - are quite different from
  - the tools of domain scientists and engineers.

- In theoretical physics there is no real concern for computability.
  - Mathematical models themselves provide the answers.
- For domain engineering there is a real concern for computability.
  - The mathematical models often serve as a basis for requirements for software, that is, for computing.
- Hence it was natural that the tools of domain science and engineering originated with the formal specification languages that were and are used for specifying software.

2.1.2.2. Spatial Attributes of Phenomena and Concepts

- Some phenomena (p:P)
- enjoy the meta-linguistic $\mathcal{L}$ property, $\mathcal{L}$ for $\mathcal{L}$ocation.
- Let any phenomenon be subject to the meta-linguistic predicate, has_$\mathcal{L}$, and function, obs_$\mathcal{L}$:

type
  P,C
value
  has_$\mathcal{L}$: P $\rightarrow$ **Bool**
  obs_$\mathcal{L}$: P $\xrightarrow{\sim}$ $\mathcal{L}$, **pre** obs_$\mathcal{L}$(e): has_$\mathcal{L}$(e)
axiom
  $\forall$ p,p′:P ·
    has_$\mathcal{L}$(p)$\wedge$has_$\mathcal{L}$(p′) $\wedge$ p$\neq$p′ $\Rightarrow$ obs_$\mathcal{L}$(p) $\neq$ obs_$\mathcal{L}$(p′)

- We consider $\mathcal{L}$ (for $\mathcal{L}$ocations) to be an attribute of those phenomena which satisfy the has_$\mathcal{L}$ property.

2.1.2.3. Simple Entities versus Attributes

- We make a distinction between
  - simple entities and
  - attributes.
- Simple entities are phenomena or concepts
  - that may be separable parts of other simple entities;
  - that may be composed into other simple entities; and
  - that possess one or more attributes.

- Attributes are properties of simple entity phenomena
  - that together form
    - atomic simple entities,
    - or characterise composite entities apart from their sub-entities;
  - that cannot be "removed" from a simple entity possessing such attributes; and
  - that may be modelled as values of simple or composite types.

2.1.3. Constituent Sciences of Domain Science

2.1.3.1. Knowledge Engineering

"Knowledge engineering is an engineering discipline that involves integrating knowledge into computer systems in order to solve complex problems normally requiring a high level of human expertise." [Wikipedia]

- Knowledge (science and) engineering,
  - what humans know and believe,
  - promise and commit,
  - what is necessary, probable and/or possible,
  - is a proper part of domain science —
  - but we omit any treatment of this fascinating topic.

2.1.3.2. Computer Science

Definition 5 – Computer Science: *Computer science*

- *is the study and knowledge*
- *about the “things” that may exist inside computers.*

■

2.1.3.3. Computing Science

Definition 6 – Computing Science:

- *is the study and knowledge*
- *how to construct the “things” that may exist inside computers.*

■

2.1.4. Elements, Aims and Objectives of Domain Science (II)

- So
  - computing science and
  - knowledge (science and) engineering

  are both part of domain science.
- Computer science,
  - notably with its emphasis on algebraic structures and
  - mathematical (modal) logics
  - provide some of the foundations for the studies of
    - computing science and
    - knowledge (science and) engineering

2.2. Informal Examples

- We will give several informal examples.
  - For each of these examples we shall very briefly mention
    * observable simple entity, operation, event and behaviour phenomena
    * as well as concepts.

Example 1 – **Air Traffic (I):** The domain-specific terminology includes such entities as:

- aircraft,
- ground, terminal, area and continental control towers and centers,
- air-lanes, etc.

The modelled atomic and composite structures and operations include

- airspace as consisting of air-lanes, airports and various control towers;
- traffic modelled as function from time to aircraft positions in airspace;
- operations of aircraft take-off, guidance and landing;
- events of communication between pilots and control towers;
- et cetera.

■

Example 2 – **Banking:** The domain-specific terminology includes such terms as:

- *clients*;
- *banks*
  - with *demand/deposit* accounts with *yield* and *interest* rates and *credit limits* and
  - with *open, deposit, withdraw, transfer, statement* and *close* operations; and
  - with *mortgage* accounts and *loan approval*, *payment installation*, *loan defaulting*, etc.; and
  - *bankruptcy*, *payment due*, *credit limit exceeded*, etc. events; et cetera;
- et cetera.

■

Example 3 – **Container Line Industry:** The domain-specific terminology includes such terms as:

- container,
- container line,
- container vessel (bay, row, stack, etc.),
- container terminal port (quay, crane, stack/stacking, etc.),
- sea lane, etc,
- container stowage,
- et cetera.

■

Example 4 – Health Care: The domain-specific terminology includes such terms as:

- citizen cum patient, medical staff,
- hospital, ward, bed,
- operating theatre,
- patient medical journal,
- anamnese[6], analysis, diagnostics, treatment, etc.,
- hospitalisation plan,
- et cetera.

■

[6] Anamnese: the patients' history of illness, including the most present.

Example 5 – "The Market": The domain-specific terminology includes such terms as:

- consumer, retailer, wholesaler and producer;
- merchandise, order, price, quantity, in-store, back-order, etc.;
- supply chain;
- inquire, order, inspect delivered goods, accept goods, pay;
- failure of delivery, default on payments, etc.;
- et cetera.

■

Example 6 – **Oil Industry:** The domain-specific terminology includes such terms as:

- oil field, pump and platform;
- oil pipeline, pipe, flow pump, valve, etc.;
- oil refinery;
- oil tanker, harbour, etc.

- $\boxed{\text{MORE TO COME}}$

■

Example 7 – Public Government: The domain-specific terminology includes such terms as:

- citizens, lawmakers, administrators, judges, etc.,
- law-making, law-enforcing (central and local government administration) and law-judging ("the judiciary"),
- documents: law drafts, laws, public administration templates, forms and letters, verdicts, etc.,
- document creation, editing, reading, copying, distribution and shredding, etc.

- MORE TO COME

■

Example 8 – Railways: The domain-specific terminology includes such terms as:

- railway net with track units such as linear, simple switches, simple crossover, crossover switches, signals, etc;
- trains;
- passengers, tickets, reservations;
- timetable and train traffic;
- train schedules, train rostering, train maintenance plan, etc.

- MORE TO COME

■

Example 9 – **Road System:** The domain-specific terminology includes such terms as:

- hubs (intersections) and links (road segments),
- open and close hub and link traversal directions,
- hub semaphores, etc.

• $\boxed{\text{MORE TO COME}}$

■

2.3. An Initial Domain Description Example

- Before we delve into pragmatic and methodological issues of domain engineering we need an example which show the both informal and formal form in which we express a domain description.
- The example is that of describing a transport net.

Example 10 – Transport Net (I):

1. There are hubs and links.
2. There are nets, and a net consists of a set of two or more hubs and one or more links.

type

1 H, L,

2 N = H-**set** × L-**set**

axiom

2 ∀ (hs,ls):N · **card** hs≥2 ∧ **card** ks≥1

3. There are hub and link identifiers.
4. Each hub (and each link) has an own, unique hub (respectively link) identifiers (which can be observed from the hub [respectively link]).

type

3 HI, LI

value

4a obs_HI: H $\rightarrow$ HI, obs_LI: L $\rightarrow$ LI

axiom

4b $\forall$ h,h′:H, l,l′:L $\cdot$ h$\neq$h′ $\Rightarrow$

obs_HI(h)$\neq$obs_HI(h′) $\wedge$ l$\neq$l′$\Rightarrow$obs_LI(l)$\neq$obs_LI(l′)

In order to model the physical (i.e., domain) fact that links are delimited by two hubs and that one or more links emanate from and are, at the same time incident upon a hub we express the following:

5. From any link of a net one can observe the two hubs to which the link is connected.
   (a) We take this 'observing' to mean the following: From any link of a net one can observe the two distinct identifiers of these hubs.
6. From any hub of a net one can observe the one or more links to which are connected to the hub.
   (a) Again: by observing their distinct link identifiers.
7. Extending Item 5: the observed hub identifiers must be identifiers of hubs of the net to which the link belongs.
8. Extending Item 6: the observed link identifiers must be identifiers of links of the net to which the hub belongs

We used, above, the concept of 'identifiers of hubs' and 'identifiers of links' of nets. We define, below, functions (iohs, iols) which calculate these sets.

value

5a obs_HIs: L → HI-**set**,

6a obs_LIs: H → LI-**set**,

axiom

5b $\forall$ l:L • **card** obs_HIs(l)=2 $\wedge$

6b $\forall$ h:H • **card** obs_LIs(h)$\geq$1 $\wedge$

$\forall$ (hs,ls):N •

5(a)) $\forall$ h:H • h $\in$ hs $\Rightarrow$ $\forall$ li:LI • li $\in$ obs_LIs(h) $\Rightarrow$

$\exists$ l′:L • l′ $\in$ ls $\wedge$ li=obs_LI(l′) $\wedge$ obs_HI(h) $\in$ obs_HIs(l′) $\wedge$

6(a)) $\forall$ l:L • l $\in$ ls $\Rightarrow$

$\exists$ h′,h″:H • {h′,h″}$\subseteq$hs $\wedge$ obs_HIs(l)={obs_HI(h′),obs_HI(h″)}

7 $\forall$ h:H • h $\in$ hs $\Rightarrow$ obs_LIs(h) $\subseteq$ iols(ls)

8 $\forall$ l:L • l $\in$ ls $\Rightarrow$ obs_HIs(h) $\subseteq$ iohs(hs)

value

iohs: H-**set** → HI-**set**, iols: L-**set** → LI-**set**

iohs(hs) ≡ {obs_HI(h)|h:H•h $\in$ hs}

iols(ls) ≡ {obs_LI(l)|l:L•l $\in$ ls}

- In the above extensive example we have focused on just five entities: nets, hubs, links and their identifiers.
- The nets, hubs and links can be seen as separable phenomena.
- The hub and link identifiers are conceptual models of the fact that hubs and links are connected
  - — so the identifiers are abstract models of 'connection',
  - or, as we shall later discuss it, the mereology of nets,
  - that is, of how nets are composed.
- These identifiers are attributes of entities.
- Links and hubs have been modelled to possess link and hub identifiers.
  - A link's "own" link identifier enables us to refer to the link,
  - A link's two hub identifiers enables us to refer to the connected hubs.
  - Similarly for the hub and link identifiers of hubs.

First we treat the syntax of operation designators ("commands").

9. To a net one can insert a new link in either of three ways:
   (a) Either the link is connected to two existing hubs — and the insert operation must therefore specify the new link and the identifiers of two existing hubs;
   (b) or the link is connected to one existing hub and to a new hub — and the insert operation must therefore specify the new link, the identifier of an existing hub, and a new hub;
   (c) or the link is connected to two new hubs — and the insert operation must therefore specify the new link and two new hubs.
   (d) From the inserted link one must be able to observe identifier of respective hubs.

10. From a net one can remove a link. The removal command specifies a link identifier.

```
type
  9 Insert == Ins(s_ins:Ins)
  9     Ins = 2xHubs | 1x1nH | 2nHs
  9(a))   2xHubs == 2oldH(s_hi1:HI,s_l:L,s_hi2:HI)
  9(b))   1x1nH == 1oldH1newH(s_hi:HI,s_l:L,s_h:H)
  9(c))   2nHs == 2newH(s_h1:H,s_l:L,s_h2:H)
axiom
  9(d)) ∀ 2oldH(hi′,l,hi″):Ins • hi′≠hi″ ∧ obs_LIs(l)={hi′,hi″} ∧
    ∀ 1old1newH(hi,l,h):Ins • obs_LIs(l)={hi,obs_HI(h)} ∧
    ∀ 2newH(h′,l,h″):Ins • obs_LIs(l)={obs_HI(h′),obs_HI(h″)}
type
  10 Remove == Rmv(s_li:LI)
```

Then we consider the meaning of the Insert operation designators.

11. The insert operation takes an Insert command and a net and yields either a new net or **chaos** for the case where the insertion command "is at odds" with, that is, is not semantically well-formed with respect to the net.

12. We characterise the "is not at odds", i.e., is semantically well-formed, that is:

    - pre_int_Insert(op)(hs,ls),

    as follows: it is a propositional function which applies to Insert actions, op, and nets, (hs.ls), and yields a truth value if the below relation between the command arguments and the net is satisfied. Let (hs,ls) be a value of type N.

13. If the command is of the form 2oldH(hi′,l,hi′) then

    $\star 1$ hi′ must be the identifier of a hub in hs,

    $\star s2$ l must not be in ls and its identifier must (also) not be observable in ls, and

    $\star 3$ hi″ must be the identifier of a(nother) hub in hs.

14. If the command is of the form 1oldH1newH(hi,l,h) then
    - ⋆1 hi must be the identifier of a hub in hs,
    - ⋆2 l must not be in ls and its identifier must (also) not be observable in ls, and
    - ⋆3 h must not be in hs and its identifier must (also) not be observable in hs.

15. If the command is of the form 2newH(h′,l,h″) then
    - ⋆1 h′ — left to the reader as an exercise (see formalisation !),
    - ⋆2 l — left to the reader as an exercise (see formalisation !), and
    - ⋆3 h″ — left to the reader as an exercise (see formalisation !).

Conditions concerning the new link (second ⋆s, ⋆2, in the above three cases) can be expressed independent of the insert command category.

value

11 $\text{int\_Insert: Insert} \rightarrow \text{N} \xrightarrow{\sim} \text{N}$

12′ $\text{pre\_int\_Insert: Ins} \rightarrow \text{N} \rightarrow$ **Bool**

12″ $\text{pre\_int\_Insert(Ins(op))(hs,ls)} \equiv$

⋆2 $\text{s\_l(op)} \notin \text{ls} \wedge \text{obs\_LI(s\_l(op))} \notin \text{iols(ls)} \wedge$

case op **of**

13) $\text{2oldH(hi}',\text{l,hi}'') \rightarrow \{\text{hi}',\text{hi}''\} \in \text{iohs(hs)},$

14) $\text{1oldH1newH(hi,l,h)} \rightarrow$

$\text{hi} \in \text{iohs(hs)} \wedge \text{h} \notin \text{hs} \wedge \text{obs\_HI(h)} \notin \text{iohs(hs)},$

15) $\text{2newH(h}',\text{l,h}'') \rightarrow$

$\{\text{h}',\text{h}''\} \cap \text{hs}=\{\} \wedge \{\text{obs\_HI(h}'),\text{obs\_HI(h}'')\} \cap \text{iohs(hs)}=\{\}$

end

16. Given a net, (hs,ls), and given a hub identifier, (hi), which can be observed from some hub in the net, xtr_H(hi)(hs,ls) extracts the hub with that identifier.
17. Given a net, (hs,ls), and given a link identifier, (li), which can be observed from some link in the net, xtr_L(li)(hs,ls) extracts the hub with that identifier.

value

16: $\mathsf{xtr\_H}: \mathsf{HI} \rightarrow \mathsf{N} \xrightarrow{\sim} \mathsf{H}$

16: $\mathsf{xtr\_H(hi)(hs,\_)} \equiv \mathbf{let}\ \mathsf{h{:}H{\cdot}h} \in \mathsf{hs} \wedge \mathsf{obs\_HI(h)=hi}\ \mathbf{in}\ \mathsf{h}\ \mathbf{end}$

$\qquad\qquad \mathbf{pre}\ \mathsf{hi} \in \mathsf{iohs(hs)}$

17: $\mathsf{xtr\_L}: \mathsf{HI} \rightarrow \mathsf{N} \xrightarrow{\sim} \mathsf{H}$

17: $\mathsf{xtr\_L(li)(\_,ls)} \equiv \mathbf{let}\ \mathsf{l{:}L{\cdot}l} \in \mathsf{ls} \wedge \mathsf{obs\_LI(l)=li}\ \mathbf{in}\ \mathsf{l}\ \mathbf{end}$

$\qquad\qquad \mathbf{pre}\ \mathsf{li} \in \mathsf{iols(ls)}$

18. When a new link is joined to an existing hub then the observable link identifiers of that hub must be updated to reflect the link identifier of the new link.
19. When an existing link is removed from a remaining hub then the observable link identifiers of that hub must be updated to reflect the removed link (identifier).

value

aLI: H $\times$ LI $\rightarrow$ H, rLI: H $\times$ LI $\xrightarrow{\sim}$ H

18: aLI(h,li) **as** h′

pre li $\notin$ obs_LIs(h)

post obs_LIs(h′) = {li} $\cup$ obs_LIs(h) $\wedge$ non_l_eq(h,h′)

19: rLI(h′,li) **as** h

pre li $\in$ obs_LIs(h′) $\wedge$ **card** obs_LIs(h′)$\geq$2

post obs_LIs(h) = obs_LIs(h′) \ {li} $\wedge$ non_l_eq(h,h′)

20. If the Insert command is of kind 2newH(h',l,h") then the updated net of hubs and links, has
    - the hubs hs joined, ∪, by the set {h′,h″} and
    - the links ls joined by the singleton set of {l}.
21. If the Insert command is of kind 1oldH1newH(hi,l,h) then the updated net of hubs and links, has

21.1 : the hub identified by hi updated, hi′, to reflect the link connected to that hub.

21.2 : The set of hubs has the hub identified by hi replaced by the updated hub hi′ and the new hub.

21.2 : The set of links augmented by the new link.

22. If the Insert command is of kind 2oldH(hi',l,hi") then

22.1–.2 : the two connecting hubs are updated to reflect the new link,

22.3 : and the resulting sets of hubs and links updated.

int_Insert(op)(hs,ls) $\equiv$
$\star_i$ **case** op **of**
20 2newH(h′,l,h″) $\longrightarrow$ (hs $\cup$ {h′,h″},ls $\cup$ {l}),
21 1oldH1newH(hi,l,h) $\longrightarrow$
21.1 **let** h′ = aLI(xtr_H(hi,hs),obs_LI(l)) **in**
21.2 (hs$\setminus${xtr_H(hi,hs)}$\cup${h,h′},ls $\cup${l}) **end**,
22 2oldH(hi′,l,hi″) $\longrightarrow$
22.1 **let** hsδ = {aLI(xtr_H(hi′,hs),obs_LI(l)),
22.2 aLI(xtr_H(hi″,hs),obs_LI(l))} **in**
22.3 (hs$\setminus${xtr_H(hi′,hs),xtr_H(hi″,hs)}$\cup$ hsδ,ls $\cup${l}) **end**
$\star_j$ **end**
$\star_k$ **pre** pre_int_Insert(op)(hs,ls)

23. The remove command is of the form Rmv(li) for some li.

24. We now sketch the meaning of removing a link:

   (a) The link identifier, li, is, by the pre_int_Remove pre-condition, that of a link, l, in the net.

   (b) That link connects to two hubs, let us refer to them as h′ and h′.

   (c) For each of these two hubs, say h, the following holds wrt. removal of their connecting link:

      i. If l is the only link connected to h then hub h is removed. This may mean that
         - either one
         - or two hubs

         are also removed when the link is removed.

      ii. If l is not the only link connected to h then the hub h is modified to reflect that it is no longer connected to l.

   (d) The resulting net is that of the pair of adjusted set of hubs and links.

value

23 int_Remove: Rmv $\rightarrow$ N $\stackrel{\sim}{\rightarrow}$ N

24 int_Remove(Rmv(li))(hs,ls) $\equiv$

24(a)) **let** l = xtr_L(li)(ls), {hi′,hi″} = obs_HIs(l) **in**

24(b)) **let** {h′,h″} = {xtr_H(hi′,hs),xtr_H(hi″,hs)} **in**

24(c)) **let** hs′ = cond_rmv(h′,hs) $\cup$ cond_rmv_H(h″,hs) **in**

24(d)) (hs\{h′,h″} $\cup$ hs′,ls\{l}) **end end end**

24(a)) **pre** li $\in$ iols(ls)

cond_rmv: LI $\times$ H $\times$ H-**set** $\rightarrow$ H-**set**

cond_rmv(li,h,hs) $\equiv$

24((c))i) **if** obs_HIs(h)={li} **then** {}

24((c))ii) **else** {sLI(li,h)} **end**

pre li $\in$ obs_HIs(h)

This ends Example 10 ■

2.4. Preliminary Summary

- So domain descriptions are both informal and formal descriptions:
- narratives and formalisations of the domain
  - as it is;
  - no references are to be made to requirements
  - let alone to software (being required).
- Domain descriptions can never be normative.

- We should be able to foresee a time, say 10 years from now, ideally,
  - where there are a number of text- and reference-book like domain descriptions for a large variety of domains:
    * air traffic,
    * airports,
    * financial services institutions (banks, brokers, stock and other commodities [metal, crops, oil, etc.] exchanges, portfolio management, credit cards, insurance, etc.),
    * transportation (container lines, airlines, railways, commuter bus transports, etc.),
    * assembly manufacturing, gas and oil pipelines, health care, etc.

- But although these domain descriptions
  - should represent quite extensive and detailed models
  - they are only indicative.
- Any one software house which specialises in software
  - (or, in general IT systems) within one (or more) of these domains will,
  - when doing their requirements engineering tasks
  - do so most likely on instantiated, i.e., modified, such domain descriptions.
- We will never be able to describe a domain completely.

2.5. Structure of Lectures

- Motivation for Domain Descriptions
- Abstraction and Modelling
- Six Facets of Domain Descriptions
- Deriving Requirements from Domain Descriptions
- Domain Demos
- Innovation

End of Lecture 2

What are Domains ?

Lecture 3

Motivation for Domain Engineering

3. Motivation for Domain Descriptions

- There are two basic reasons for creating domain descriptions.
  - One is general and is related to the understanding of the world around and, to some extent, within us;
  - the other is in relation to the development of IT systems, notably software.

(3. **Motivation for Domain Descriptions**)

3.1. Domain Descriptions of Infrastructure Components

- We use here a term, 'infrastructure', that we ought first define.
- according to the World Bank,
  - *'infrastructure'*[7].
  - *is an umbrella term for many activities referred to as 'social overhead capital' by some development economists,*
  - *and encompasses activities that share technical and economic features (such as economies of scale and spill-overs from users to non-users).*[8]

[7] Winston Churchill is quoted as having said, in the House of Commons, in 1946: *. . . the young Labourite speaker, that we just heard, obviously wishes to impress his constituency with the fact that he has attended Eton and Oxford when he uses such modern terms as 'infrastructure' . . .*

[8] I thank Jan Goossenarts for bringing the text of this paragraph to my attention.

- We take a more technical view,
  - and see infrastructures as concerned with supporting other systems or activities.
- A first reason for pursuing the research and experimental engineering of domain descriptions —
  - both informal narratives and formal specifications —
  - is to achieve understanding, insight and, eventually, theories of domains being thus described.

- A second reason for domain engineering is
  - to create, not necessarily normative models,
  - but models which can be instantiated
    * to fit a current constellation of a number of these institutions
    * with the aim of studying possible business process re-engineering proposals,
    * yes even to generate such proposals,
    * or with the aim of software development.

- A third reason for domain engineering is
  - to create (again not necessarily normative) descriptions
  - whose narrative parts can be used in company training and in school education,

3.2. Domain Descriptions for Software Development

- The reasons given above are independent of whether one aims at developing software for a segment of the described domain or not.
- But, a reason for pursuing the research and experimental engineering of domain descriptions can, nevertheless be
  - that one wishes to develop software support for
    * entities,
    * operations,
    * events and
    * behaviours

    and in the
    * supporting technologies,
    * mgt. and org.,
    * rules and regulations,
    * scripts and
    * human behaviours

- So here is the dogma that guides us:

Dogma 1 – **The $\mathcal{D},\mathcal{S} \models \mathcal{R}$ Dogma:**

- *Before $\mathcal{S}$oftware can be designed*
- *we must understand the $\mathcal{R}$equirements,*
- *and before $\mathcal{R}$equirements can be expressed*
- *we must understand the $\mathcal{D}$omain.*

■

- This dogma entails that we decompose software development into three phases and their attendant *stages:*

  Domain Engineering:

  - *identification of domain stake-holders,*
  - *domain acquisition,*
  - *rough sketch of business processes,*
  - *domain analysis and concept formation,*
  - *domain 'terminologisation',*
  - *the main stages of domain modelling (intertwined with domain model verification),*
  - *domain validation* and
  - *domain theory formation.*

Requirements Engineering:

- *identification of requirements stake-holders,*
- *requirements acquisition,*
- *rough sketch of business process re-engineering,*
- *requirements analysis and concept formation,*
- *requirements 'terminologisation',*
- *the main stages of requirements modelling (intertwined with requirements model verification),*
- *requirements validation,*
- *requirements feasibility and satisfiability* and
- *requirements theory formation.*

Software Design: etcetera.

We shall later show how the requirements acquisition stage is basically a rough sketch version of the the main stages of requirements modelling.

3.3. Discussion

- The dogma as enunciated above is not “dogmatic”.
- Engineers of the classical engineering disciplines are all rather deeply educated and trained in the domains of their subject: electronic chip designers are well-versed in plasma physics; aeronautical engineers are well-versed in aerodynamics, and celestial mechanics; mobile phone antenna designers, whether emitters or receivers, are well-versed in applying (“massaging” and calculating over) Maxwell’s equations; et cetera.

- No pharmaceutical company would hire a person into their research and development of new medical drugs unless that person had a serious, professional education and training in the scientific, i.e., in the domain disciplines of pharmaceutics. Likewise for structural engineers hired to design suspension or other forms of road and rail bridges: certainly they must be well-versed in structural engineering.

- For a software engineer — to be deployed in the development of software for transportation, or for financial service institutions, or for health care, etc. — to be well-versed in the theories of automata and formal languages, semantics of programming and specification languages, operating systems, compilers, database management systems, etc., is accepted — but what is also needed is an ability to either read existing or to develop new domain descriptions for the fields of respectively transportation, financial service institutions, health care, or similarly.

End of Lecture 3

Motivation for Domain Engineering

Lecture 4

Abstraction & Modelling

4. Abstraction & Modelling

4.1. Abstraction

- Abstraction relates to
  - conquering complexity of description through the judicious use of abstraction,
  - where abstraction, briefly, is the act and result of omitting consideration of (what would then be called) details while, instead, focusing on (what would therefore be called) important aspects.

4.1.1. From Phenomena to Concepts

- Phenomena are “things” that we can point to.
- They are often referred to as ‘individuals’
  - since what is pointed to is a single specimen
  - of possibly many “similar” instances of phenomena.
- We can then, when “figuratively pointing to” an individual (a phenomenon),
  - either keep “talking about” just that one individual,
  - or we can ‘abstract’ to the class of all ‘similar’ phenomena.
- When we do the latter
  - then we have abstracted
  - from a phenomenon, that is, a specific value,
  - to a concept, i.e., to the type of all such values.

4.1.2. From Narratives to Formalisations

- We describe domains
  - both informally, in terms of concise natural language narratives,
  - and formally, using one or more formal specification languages.
- The terms of the natural language narrative designate concepts
  - nouns typically denoting types and values of simple entities;
  - verbs typically denoting operations over entities;
  - etcetera.
- These terms are chosen carefully to correspond,
  - as far as is reasonable in order to achieve a readable natural language text,
  - to names of types, values, operations, etc., of the formal specification.

- Thus there is, in fact, a "two-way relation"
  - between the choice of mathematical abstractions of the formal specification
  - and the terms of the narrative;
  - the objective is to bring "an as close as possible" relation between the narrative and the formalisation.

4.1.3. Examples of Abstraction

- Example 10 illustrated two forms of abstraction:
  - (i) model-oriented abstraction and
  - (ii) property-oriented abstraction.
- The model-oriented abstraction of Example 10 is illustrated by the modelling of nets as pairs of sets of hubs and links, cf. Item 2 on Slide 62: $\mathsf{N} = \mathsf{H}\textsf{-}\mathbf{set} \times \mathsf{L}\textsf{-}\mathbf{set}$, as well as by the concrete type syntax types of link insertion and remove commands and their semantics, Items 9–24(d) (Slides 67–77).
- The property-oriented abstraction of Example 10 is illustrated by the sorts and observers relating to hubs and links, cf. Items 1 on Slide 62, 3–8 (Slides 63–65).
- In this section we shall give some small examples of abstractions.

Example 11 – Model-oriented Directory:

25. Terminal directory entries are files and files are further undefined.
26. A directory consists of a finite set of uniquely (directory identifier) distinguished entries.
27. A directory is either a file or is a directory.

type

25. FILE, DId
26. $\mathrm{DIR} = \mathrm{Did} \underset{m}{\rightarrow} \mathrm{Entry}$
27. $\mathrm{Entry} = \mathrm{FILE} \mid \mathrm{DIR}$

- Directories are modelled as maps.
- The specification abstracts from representation of directory identifiers and files.

■

Example 12 – Networked Social Structures:

- People live in communities.
- People of communities may network with people of distinct other communities.
- And people of such network may network with people of distinct other networks.
- We formulate this in a narrative and we formalise the narrative.

28. People are at the heart of any social structure.
29. A region consists of a finite set of one or more communities and a finite set of zero, one or more social networks.
30. A community consists of a non-empty, finite set of people.
31. A social network consists of a non-empty, finite set of two or more people, such that
    (a) all people of a network belong to distinct communities of the region, (i.e., no two people of a net belong to the same community),
    (b) and, if they also are members of other networks, then they all belong to distinct other networks (i.e., no two people of a network belong to the same other networks),

type

28. P

29. $R' = C\text{-}\mathbf{set} \times N\text{-}\mathbf{set}$, $\quad R = \{|(cs,ns):R' \cdot cs \neq \{\}|\}$

30. $C' = P\text{-}\mathbf{set}$, $\quad C = \{|c:C' \cdot c \neq \{\}|\}$

31. $N' == mkN(sn:P\text{-}\mathbf{set})$, $\quad N = \{|mkN(ps):N' \cdot \mathbf{card}\ ps \geq 2|\}$

axiom

$\forall\ (cs,ns):R\ \cdot$

31(a). $\forall\ n:N \cdot n \in ns \Rightarrow$

$\quad \mathbf{card}\ n = \mathbf{card}\{c|c:C \cdot c \in cs \wedge n \cap c \neq \{\}\} \wedge$

31(b). $\quad \exists\ p:P \cdot p \in n \wedge$

$\quad\quad \exists\ n':N \cdot n' \in ns \wedge n \neq n' \wedge p \in n' \Rightarrow$

$\quad\quad\quad \forall\ p:P \cdot p \in n \Rightarrow$

$\quad\quad\quad\quad \exists\ n'':N \cdot n'' \in ns \wedge n \neq n'' \wedge p \in n'' \wedge$

$\quad\quad\quad\quad\quad \mathbf{card}\ n' = \mathbf{card}\{n''|n'':N \cdot n'' \in ns \wedge n'' \cap n' \neq \{\}\}$

- Formula line $\mathbf{card}\,n=\mathbf{card}\{c|c:C\cdot c\in cs\wedge n\cap c\neq\{\}\}$, the first of the two lines starting with **card**, expresses
  - that the number of persons in the network
  - is the same as the number of the communities to which these persons belong.
- The fact that $n\cap c\neq\{\}$ can be proven to be the same as $\mathbf{card}(n\cap c)=1$ is left as an exercise.
- Formula line $\mathbf{card}\,n'=\mathbf{card}\{n''|n'':N\cdot n''\in ns\wedge n''\cap n'\neq\{\}\}$, the second of the two lines starting with **card**, expresses
  - that the number of persons in the network
  - for the case that at least one of the persons in the network is a member of some other network,
  - is the same as the number of the networks to which all other persons of the n must belong.
- The fact that $n''\cap n'\neq\{\}$ can be proven to be the same as $\mathbf{card}(n''\cap n')=1$ is left as an exercise.

■

Example 13 – Railway Nets:

- We bring a variant of Example 10.

32. A railway net consists of one or more lines and two or more stations.

type
  32. RN, LI, ST
value
  32. obs_LIs: RN $\rightarrow$ LI-**set**
  32. obs_STs: RN $\rightarrow$ ST-**set**
axiom
  32. $\forall$ n:RN $\cdot$ **card** obs_LIs(n) $\geq 1 \wedge$ **card** obs_STs(n) ≥ 2

33. A railway net consists of rail units.

type
33. U
value
33. obs_Us: RN $\rightarrow$ U-**set**

34. A line is a linear sequence of one or more linear rail units.

axiom
34. $\forall$ n:RN, l:LI $\cdot$ l $\in$ obs_LIs(n) $\Rightarrow$ lin_seq(l)

35. The rail units of a line must be rail units of the railway net of the line.

value
34. obs_Us: LI → U-**set**
axiom
35. ∀ n:RN, l:LI · l ∈ obs_LIs(n) ⇒ obs_Us(l) ⊆ obs_Us(n)

36. A station is a set of one or more rail units.

value
36. obs_Us: ST → U-**set**
axiom
36. ∀ n:RN, s:ST · s ∈ obs_STs(n) ⇒ **card** obs_Us(s) ≥ 1

37. The rail units of a station must be rail units of the railway net of the station.

axiom

37. $\forall$ n:RN, s:ST $\cdot$ s $\in$ obs_STs(n) $\Rightarrow$ obs_Us(s) $\subseteq$ obs_Us(n)

38. No two distinct lines and/or stations of a railway net share rail units.

axiom

38. $\forall$ n:RN,l,l′:LI$\cdot$\{l,l′\}$\subseteq$obs_LIs(n)$\wedge$l$\neq$l′$\Rightarrow$obs_Us(l)$\cap$ obs_Us(l′)=\{\}
38. $\forall$ n:RN,l:LI,s:ST$\cdot$l $\in$ obs_LIs(n)$\wedge$s $\in$ obs_STs(n)$\Rightarrow$obs_Us(l)$\cap$ obs_U
38. $\forall$ n:RN,s,s′:ST$\cdot$\{s,s′\}$\subseteq$obs_STs(n)$\wedge$s$\neq$s′$\Rightarrow$obs_Us(s)$\cap$ obs_Us(s′)=\{\}

39. A station consists of one or more tracks.

type

39. Tr

value

39. $\mathsf{obs\_Trs}: \mathsf{ST} \rightarrow \mathsf{Tr}\text{-}\mathbf{set}$

axiom

39. $\forall\ \mathsf{s}:\mathsf{ST}\cdot\mathbf{card}\ \mathsf{obs\_Trs(s)}\geq 1$

40. A track is a linear sequence of one or more linear rail units.

axiom

40. $\forall\ \mathsf{n}:\mathsf{RN},\mathsf{s}:\mathsf{ST},\mathsf{t}:\mathsf{Tr}\cdot \mathsf{s} \in \mathsf{obs\_STs(n)} \wedge \mathsf{t} \in \mathsf{obs\_Trs(s)} \Rightarrow \mathsf{lin\_seq(t)}$

41. No two distinct tracks share rail units.

axiom

41. $\forall$ n:RN,s:ST,t,t′:Tr•s $\in$ obs_STs(n)$\wedge$\{t,t′\}$\subseteq$obs_Trs(s)$\wedge$t$\neq$t′$\Rightarrow$obs_Us

42. The rail units of a track must be rail units of the station (of that track).

value

42. obs_Us: Tr $\rightarrow$ U-**set**

axiom

42. $\forall$ rn:RN,st:ST,tr:TR $\cdot$
st $\in$ obs_STs(rn)$\wedge$tr $\in$ obs_Trs(st)$\Rightarrow$obs_Us(tr)$\subseteq$obs_Us(st)

43. A rail unit is either a linear, or is a switch, or a is simple crossover, or is a switchable crossover, etc., rail unit.

value

43. is_Linear: U → **Bool**
43. is_Switch: U → **Bool**
43. is_Simple_Crossover: U → **Bool**
43. is_Switchable_Crossover: U → **Bool**

44. A rail unit has one or more connectors.

type

44. K

value

44. obs_Ks: U → K-**set**

45. A linear rail unit has two distinct connectors. A switch (a point) rail unit has three distinct connectors. Crossover rail units have four distinct connectors (whether simple or switchable), etc.

axiom

$\forall$ u:U $\cdot$

is_Linear(u) $\Rightarrow$ **card** obs_Ks(u)=2$\wedge$

is_Switch(u) $\Rightarrow$ **card** obs_Ks(u)=3$\wedge$

is_Simple_Crossover(u) $\Rightarrow$ **card** obs_Ks(u)=4$\wedge$

is_Switchable_Crossover(u) $\Rightarrow$ **card** obs_Ks(u)=4

46. For every connector there are at most two rail units which have that connector in common.

axiom

46. $\forall$ n:RN $\cdot$ $\forall$ k:K $\cdot$ k $\in$ $\cup${obs_Ks(u)|u:U$\cdot$u $\in$ obs_Us(n)}

$\Rightarrow$**card**{u|u:U$\cdot$u $\in$ obs_Us(n)$\wedge$k $\in$ obs_Ks(u)}$\leq$2

47. Every line of a railway net is connected to exactly two distinct stations of that railway net.

axiom

$$
\begin{array}{l}
47.\ \ \forall\ \mathsf{n:RN,l:LI} \cdot\ \mathsf{l} \in \mathsf{obs\_LIs(n)} \Rightarrow \\
\quad \exists\ \mathsf{s,s':ST} \cdot \{\mathsf{s,s'}\} \subseteq \mathsf{obs\_STs(n)} \wedge \mathsf{s{\neq}s'} \Rightarrow \\
\quad\quad \textbf{let}\ \mathsf{sus{=}obs\_Us(s),sus'{=}obs\_Us(s'),lus{=}obs\_Us(l)}\ \textbf{in} \\
\quad\quad \exists\ \mathsf{u,u',u'',u''':U} \cdot \mathsf{u} \in \mathsf{sus} \wedge \\
\quad\quad\quad \mathsf{u'} \in \mathsf{sus'} \wedge \{\mathsf{u'',u'''}\} \subseteq \mathsf{lus} \Rightarrow \\
\quad\quad\quad \textbf{let}\ \mathsf{sks} = \mathsf{obs\_Ks(u)},\ \mathsf{sks'} = \mathsf{obs\_Ks(u')}, \\
\quad\quad\quad\quad \mathsf{lks} = \mathsf{obs\_Ks(u'')},\ \mathsf{lks'} = \mathsf{obs\_Ks(u''')}\ \textbf{in} \\
\quad\quad\quad \exists!\mathsf{k,k':K}{\cdot}\mathsf{k{\neq}k'}{\wedge}\mathsf{sks} \cap \mathsf{lks}{=}\{\mathsf{k}\}{\wedge}\mathsf{sks'} \cap \mathsf{lks'}{=}\{\mathsf{k'}\} \\
\quad\quad \textbf{end end}
\end{array}
$$

48. A linear sequence of (linear) rail units is an acyclic sequence of linear units such that neighbouring units share connectors.

value

48. $\text{lin\_seq: U-}\mathbf{set} \to \mathbf{Bool}$

$$
\begin{array}{l}
\text{lin\_seq(us)} \equiv \\
\quad \forall\ \text{u:U} \cdot \text{u} \in \text{us} \Rightarrow \text{is\_Linear(u)} \wedge \\
\quad\quad \exists\ \text{q:U}^* \cdot \mathbf{len}\ \text{q} = \mathbf{card}\ \text{us} \wedge \mathbf{elems}\ \text{q} = \text{us} \wedge \\
\quad\quad\quad \forall\ \text{i:}\mathbf{Nat} \cdot \{\text{i,i+1}\} \subseteq \mathbf{inds}\ \text{q} \Rightarrow \exists\ \text{k:K} \cdot \text{obs\_Ks(q(i))} \cap \text{obs\_Ks(q(i} \\
\quad\quad\quad \mathbf{len}\ \text{q} > 1 \Rightarrow \text{obs\_Ks(q(i))} \cap \text{obs\_Ks(q(}\mathbf{len}\ \text{q))} = \{\}
\end{array}
$$

This ends Example 13 ■

Example 14 – A Telephone Exchange:

We start the informal description by presenting a *synopsis* and its immediate *analysis:*

- **Synopsis:**
  - The simple telephone exchange system
  - serves to efficiently honour
  - requests for conference calls
  - amongst any number of subscribers,
  - whether immediately connectable,
  - whereby they become actual,
  - or being queued, i.e., deferred (or pending) for later connection.

- **Analysis:** The concepts of subscribers and calls are central:
  - In this example we do not further analyse the concept of subscribers.
  - A call is either
    - an actual call, involving two or more subscribers not involved in any other actual calls,
    - or a call is a deferred call, i.e., a requested call that is not actual, because one or more of the subscribers of the deferred call is already involved in actual calls.
  - We shall presently pursue the concepts of requested, respectively actual calls, and only indirectly with deferred calls.

The structure of the types of interest are first described. We informally describe first the basis types, then their composition.

- **Subscribers:** There is a class (S) of further undefined subscribers.
- **Connections:** There is a class (C) of connections. A connection involves one subscriber, the ‘caller’, and any number of one or more other subscribers, the ‘called’.

- **Exchange:** At any time an exchange reflects (i.e., is in a state which records) a number of requested connections and a number of actual connections
  - such that no two actual connections share any subscribers,
  - such that all actual connections are also requested connections, and
  - such that there are no requested calls that are not actual and share no subscribers in common with any other actual connection.

- **Requested connections:** The set of all requested connections for a given exchange forms a set of connections.
- **Actual connections:** The set of all actual connections, for a given exchange, forms a subset of its requested connections such that no two actual connections share subscribers.
- In this example we shall also be able to refer to the exchange, later to be named X, as 'the state' (of the telephone exchange system).

We shall later have a great deal more to say about the concept of state.

4.1.3.0.1. • *Formalisation of Property-oriented State*•

type
  S, C, X
value
  obs_Caller: C → S
  obs_Called: C → S-**set**
  obs_Requests: X → C-**set**
  obs_Actual: X → C-**set**

  subs: C → S-**set**
  subs(c) ≡ obs_Caller(c) ∪ obs_Called(c)

  subs: C-**set** → S-**set**
  subs(cs) ≡ ∪ { subs(c) | c:C • c ∈ cs }

- The overloaded function name **subs** stands for two different functions.
  - One observes ("extracts") the set of all subscribers said to be **engaged** in a **connection**.
  - The other likewise observes the set of all subscribers **engaged** in any set of **connections**.
- We shall often find it useful to introduce such *auxiliary functions*.

axiom

[1] $\forall$ c:C, $\exists$ s:S ·
[2] s = obs_Caller(c) $\Rightarrow$ s $\notin$ obs_Called(c),

[3] $\forall$ x:X ·
[4] **let** rcs = obs_Requests(x),
[5] acs = obs_Actual(x) **in**
[6] acs $\subseteq$ rcs $\wedge$
[7] $\forall$ c,c′:C · c $\neq$ c′ $\wedge$ {c,c′} $\subseteq$ acs $\Rightarrow$
[8] obs_Caller(c) $\neq$ obs_Caller(c′) $\wedge$
[9] obs_Called(c) $\cap$ obs_Called(c′) = {} $\wedge$
[10] $\sim\exists$ c:C · c $\in$ rcs \ acs ·
[11] subs(c) $\cap$ subs(acs) = {} **end**

- The last two lines above express the efficiency criterion mentioned earlier.
- We can express a law that holds about the kind of exchanges that we are describing:

theorem

$\forall$ x:X $\cdot$

obs_Actual(x)={} $\equiv$ obs_Requests(x)={}

- The *law* expresses that there cannot be a non-empty set of deferred calls if there are no actual calls. That is, at least one deferred call can be established should a situation arise in which a last actual call is terminated and there is at least one deferred call.
- The *law* is a *theorem* that can be *proved* on the basis of the telephone exchange system *axioms* and a *proof system* for *sets.*

4.1.3.0.2. • *Operation Signatures*•

The following operations, involving telephone exchanges, can be performed:

- **Request:** A caller indicates, to the exchange, the set of one or more other subscribers with which a connection (i.e., a call) is requested. If the connection can be effected then it is immediately made actual, else it is deferred and (the connection) will be made actual once all called subscribers are not engaged in any actual call.
- **Caller_Hang:** A caller, engaged in a requested call, whether actual or not, can hang up, i.e., terminate, if actual, and then on behalf of all called subscribers also, or can cancel the requested (but not yet actual) call.

- **Called_Hang:** Any called subscriber engaged in some actual call can leave that call individually. If that called subscriber is the only called subscriber ("left in the call"), then the call is terminated, also on behalf of the caller.
- **is_Busy:** Any subscriber can inquire as to whether any other subscriber is already engaged in an actual call.
- **is_Called:** Any subscriber can inquire as to the identities of all those (zero, one or more) callers who has requested a call with the inquiring subscriber.

First the signatures:

value

newX: **Unit** $\rightarrow$ X
request: S $\times$ S-**set** $\rightarrow$ X $\rightarrow$ X
caller_hang: S $\rightarrow$ X $\stackrel{\sim}{\rightarrow}$ X
called_hang: S $\rightarrow$ X $\stackrel{\sim}{\rightarrow}$ X
is_busy: S $\rightarrow$ X $\rightarrow$ **Bool**
is_called: S $\rightarrow$ X $\rightarrow$ **Bool**

- The *generator function* newX is an *auxiliary function.*
- It is needed only to make the *axioms* cover all *states* of the telephone exchange system.
- In a sense it *generates* an *empty*, that is, an *initial state.*
- Usually such *empty state generator functions* are “paired” with a similar *test for empty state observer function.*

Then we get the axioms:

axiom

$\forall$ x:X $\cdot$ obs_Requests(x)={} $\equiv$ x=newX(),
$\forall$ x:X,s,s′:S,ss:S-**set** $\cdot$
  $\sim$is_busy(s,newX()) $\wedge$
  s$\neq$s′ $\Rightarrow$
    s $\in$ ss $\Rightarrow$ is_busy(s)(request(s′,ss)(x)) $\wedge$
    s $\notin$ ss $\Rightarrow$ is_busy(s)(request(s′,ss)(x)) $\equiv$ is_busy(s)(x),
    ... etcetera ...

- We leave the axiom incomplete.
- Our job was to illustrate the informal and formal parts of a property-oriented specification,
- not to do it completely.

4.1.3.0.3. • *Model-oriented State*•

type

$$\begin{array}{l}
\mathrm{S} \\
\mathrm{C} = \{|\ \mathrm{ss}\ |\ \mathrm{ss{:}S}\text{-}\mathbf{set} \cdot \mathbf{card}\ \mathrm{ss} \geq 2\ |\} \\
\mathrm{R} = \mathrm{C}\text{-}\mathbf{set} \\
\mathrm{A} = \mathrm{C}\text{-}\mathbf{set} \\
\mathrm{X} = \{|\ (\mathrm{r,a})\ |\ (\mathrm{r,a}){:}\mathrm{R}{\times}\mathrm{A} \cdot \mathrm{a} \subseteq \mathrm{r} \wedge \bigcap \mathrm{a} = \{\}\ |\}
\end{array}$$

4.1.3.0.4. • *Efficient States*•

- There is a notion of telephone exchange system efficiency,
  - a constraint that governs its operation,
  - hence the state, at any one time.
- The efficiency criterion says that
  - all requested calls that can actually be connected
  - are indeed connected:

value

$\text{eff\_X}: X \xrightarrow{\sim} \textbf{Bool}$

$\text{eff\_X}(r,a) \equiv \sim\exists\, a':A \cdot a \subset a' \wedge (r,a') \in X$

4.1.3.0.5. • *Formalisation of Action Types*•

type

$\mathrm{Cmd} = \mathrm{Call} \mid \mathrm{Hang} \mid \mathrm{Busy}$

$\mathrm{Call}' == \mathrm{mk\_Call}(\mathrm{p:S,cs:C})$

$\mathrm{Call} = \{\mid \mathrm{c:Call}' \cdot \textbf{card}\ \mathrm{cs(c)} \geq 1 \mid\}$

$\mathrm{Hang} == \mathrm{mk\_Hang}(\mathrm{s:S})$

$\mathrm{Busy} == \mathrm{mk\_Busy}(\mathrm{s:S})$

4.1.3.0.6. • *Pre/Post and Direct Operation Definitions*•

- We shall, for each operation, define its meaning
  - both in terms of pre/post conditions
  - and in terms of a direct “abstract data type algorithm”.

4.1.3.0.7. • *Multi-party Call*•

- A multi-party call involves a (primary, s) caller and one or more (secondary, ss) callees.
- Enacting such a call makes the desired connection a requested connection.
- If none of the callers are already engaged in an actual connection then the call can be actualised.
- A multi-party call cannot be made by a caller who has already requested other calls.

value

$$\text{int\_Call: Call} \xrightarrow{\sim} \text{X} \xrightarrow{\sim} \text{X}$$

$$\text{int\_Call(mk\_Call(p,cs))(r,)}\ \textbf{as}\ \text{(r}',\text{a}')$$

$$\textbf{pre}\ \text{p} \notin \bigcup \text{r}$$

$$\textbf{post}\ \text{r}' = \text{r} \cup \{\{\text{p}\} \cup \text{cs}\} \wedge \text{eff\_X(r}',\text{a}')$$

$$\text{int\_Call(mk\_Call(p,cs))(r,a)} \equiv$$

$$\quad \textbf{let}\ \text{r}' = \text{r} \cup \{\{\text{p}\} \cup \text{cs}\},$$

$$\quad\quad \text{a}' = \text{a} \cup \textbf{if}\ (\{\{\text{p}\} \cup \text{cs}\} \cap \bigcup \text{a}) = \{\}$$

$$\quad\quad\quad\quad \textbf{then}\ \{\{\text{p}\} \cup \text{cs}\}\ \textbf{else}\ \{\}\ \textbf{end in}$$

$$\quad (\text{r}',\text{a}')\ \textbf{end}$$

$$\quad \textbf{pre}\ \text{p} \notin \bigcup \text{r}$$

The above **pre/post**-definition (of int_Call) illustrates the power of this style of definition. No algorithm is specified, instead all the work is expressed by appealing to the invariant!

4.1.3.0.8. • *Call Termination*•

It takes one person, one subscriber, to terminate a call.

value

$$
\begin{array}{l}
\quad \text{int\_Hang: Hang} \rightarrow \text{X} \stackrel{\sim}{\rightarrow} \text{X} \\
\quad \text{int\_Hang(mk\_Hang(p))(r,a)}\ \textbf{as}\ (\text{r}',\text{a}') \\
\qquad \textbf{pre}\ \text{existS c:C} \cdot \text{c} \in \text{a} \wedge \text{p} \in \text{a} \\
\qquad \textbf{post}\ \text{r}' = \text{r} \setminus \{\text{c}|\text{c:C} \cdot \text{c} \in \text{r} \wedge \text{p} \in \text{c}\} \wedge \text{eff\_X}(\text{r}',\text{a}') \\
\\
\quad \text{int\_Hang(mk\_Hang(p))(r,a)} \equiv \\
\qquad \textbf{let}\ \text{r}' = \text{r} \setminus \{\ \text{c} \mid \text{c:C} \cdot \text{c} \in \text{a} \wedge \text{p} \in \text{c}\ \}, \\
\qquad\quad \text{a}' = \text{a} \setminus \{\ \text{c} \mid \text{c:C} \cdot \text{c} \in \text{r} \wedge \text{p} \in \text{c}\ \}\ \textbf{in} \\
\qquad \textbf{let}\ \text{a}'' = \text{a}' \cup \{\ \text{c} \mid \text{c:C} \cdot \text{c} \in \text{r}' \wedge \text{c} \bigcap \text{a}' = \{\}\ \}\ \textbf{in} \\
\qquad (\text{r}',\text{a}'')\ \textbf{end end} \\
\qquad \textbf{pre}\ \text{existS c:C} \cdot \text{c} \in \text{a} \wedge \text{p} \in \text{a}
\end{array}
$$

The two ways of defining the above **int_Hang** function again demonstrate the strong abstractional feature of defining by means of **pre/post**-conditions.

4.1.3.0.9. • *Subscriber Busy*•

A line (that is, a subscriber) is only 'busy' if it (the person) is engaged in an actual call.

value
$\quad \text{int\_Busy: S} \rightarrow \text{X} \xrightarrow{\sim}$ **Bool**
$\quad \text{int\_Busy(mk\_Busy(p))(\_,a)}$ **as** b
$\qquad$ **pre true**
$\qquad$ **post if** b **then** $p \in \bigcup a$ **else** $p \notin \bigcup a$ **end**

$$\text{int\_Busy(mk\_Busy(p))(\_,a)} \equiv p \in \bigcup a$$

Here, perhaps not so surprisingly, we find that the explicit function definition is the most straightforward. This ends Example 14 ■

4.1.4. Mathematics and Formal Specification Languages

- Using mathematical concepts has shown to be the most powerful way of expressing abstractions.
- The discrete mathematical concepts of
  - sets,
  - Cartesians,
  - sequences,
  - maps and
  - functions,
- as well as
  - mathematical logic and
  - algebras
- has served mathematicians well for quite some time
- and will serve professional software engineers well.

- Formal specification languages, like
  - `Alloy`,
  - `Event B`,
  - `RSL`,
  - `VDM`,
  - `Z`
  - and others,
- embody the above-mentioned mathematical concepts in quite readable forms.
- The current book favours the `RAISE` specification language RSL.

4.2. Modelling

Definition 7 ***Model:***

- *A model is the mathematical meaning of*
  - *a description of a domain,*
  - *or a prescription of requirements,*
  - *or a specification of software,*

  i.e.,
  - *is the meaning of a specification*
  - *of some universe of discourse*

. ■

Definition 8 ***Modelling:*** Modelling

- *is the act (or process) of identifying appropriate phenomena and concepts*
- *and of choosing appropriate abstractions*
- *in order to construct a model (or a set of models)*

. ■

4.2.1. Property-oriented Modelling

Definition 9 ***Property-oriented Modelling:*** *By property-oriented modelling we shall understand a modelling*

- *which emphasises the properties of what is being modelled,*
- *through suitable use of abstract types, that is, sorts,*
- *of postulated observer (***obs_***), generator (***mk_***) and type checking (***is_***) functions,*
- *and axioms over these*

. ■

4.2.2. Model-oriented Modelling

Definition 10 ***Model-oriented Modelling:*** *By abstract, but model-oriented modelling we shall understand a modelling*

- *which expresses the properties of what is being modelled,*
- *through suitable use of mathematical concepts such as*
- *sets, Cartesians, sequences, maps (finite domain, enumerable functions), and functions (in the sense of λ-Calculus functions)*

. ■

4.3. Model Attributes

- Specifications achieve their intended purpose by emphasising one or more attributes.
  - Either:
    - (i.1) analogic,
    - (i.2) analytic and/or
    - (i.3) iconic;
  - and then either:
    - (ii.1) descriptive or
    - (ii.2) prescriptive;
  - and finally either:
    - (iii.1) extensional or
    - (iii.2) intensional.

- That is, a model may, at the same time (although time has nothing to do with this aspect of models), be one or more of
  - analogic, analytic and iconic;
  - expressed either only descriptive, or mostly descriptive (with some prescriptive aspects), or only prescriptive, or mostly prescriptive (etc.); and
  - expressed either only extensional, or mostly extensional (with some aspects), or only intensional, or mostly intensional (etc.).
- We may claim that a good model blends the above consciously and judiciously — including featuring exactly (or primarily) one attribute from each of the three categorisations.
- We next take a look at these model attributes.

4.3.1. Analogic, Analytics and Iconic Models

Definition 11 ***Analogic Model:***

- *An* analogic model *resembles some other universe than the universe of discourse purported to be modelled*

. ■

Definition 12 ***Analytic Model:***

- *An* analytic model *is a mathematical specification: It allows analysis of the universe of discourse being modelled*

. ■

Definition 13 ***Iconic Model:***

- *An* iconic model *is an "image" of the universe of discourse that is the target of our attention*

. ■

Example 15 – **Analogic, Analytic and Iconic Models:** We lump three kinds of examples into one larger example:

- *Analogic models:*
  - (1) The symbol, on the visual display screen of your computer, of a `trash can`.
  - (2) A four-pole, electric circuit network of resistors, inductances, capacitors and current or voltage supplies can be used to analogically model some aspects of the behaviour of certain mechanical vibration and/or spring dampening aggregations.
  - (3) A tomographic image of, say the brain, with its colour-enhanced "blots" is an analogic model of a cross section of that brain!

- *Analytic models:*
  - (4) The differential equations whose variables model spatial x, y, z coordinates and the temporal t dimension, and whose constant, m, model the mass of a stone, may be an analytic model of the dynamics of the throwing of such a stone in a vacuum.
  - (5) A description, in RSL, involving quantities that purport to model bank accounts, their balance, time, etc., may be an analytic model of a banking system — in the real world — provided the model reflects at least "some of the things that can go wrong" in actual life.
  - (6) A graph with labelled nodes and weighted arcs may be used as a model of a road net with cities and distances between these, and can be used for the computation of shortest distances, etc.

- *Iconic models:* Typical iconic models are certain advisory or judicially binding traffic signs:
  - (7) The roadside sign showing an `Elk`;
  - (8) the roadside sign showing an `automobile` (from behind) `"underlined with two crossing` S `curves`; and
  - (9) the roadside sign showing a `crossed-out horn`.

Observe that a model may possess characteristics of more than one of the above attributes. ■

4.3.2. Descriptive and Prescriptive Models

Definition 14 ***Descriptive Model:***

- *A descriptive model describes something already existing*

. ∎

Definition 15 ***Prescriptive Model:***

- *A prescriptive model models something as yet to be implemented*

. ∎

- Thus domain specifications are descriptive,
- while requirements specifications are prescriptive.
- A requirements specification prescribes properties that the intended software (cum computing system) shall satisfy.
- A software specification prescribes certain kinds of computations.

- We remind the reader that we use the terms model and specification near synonymously.
  - A specification defines a set of zero, one or more, possibly even an infinity, of models.
  - But we use the term the model in connection with a given specification to stand for the general member of the set of models.
- Hence when we use the term model below, please read specification.

Example 16 – Descriptive and Prescriptive Models:

- *A descriptive model:*
  - A railway net *consists* of two or more distinct stations
  - and one or more distinct railway lines.
  - A railway line *consists* of a linear sequence of one or more linear rail units.
  - Any railway line *connects* exactly two distinct stations.
  - A route *is* a sequence of one or more, and if more, then connected railway lines.
  - Two railway lines *are* connected if they have the connecting station in common.

- *A prescriptive model:*
  - The train timetable *shall*, for each train journey, list all station visits.
  - A train timetable station visit *shall* list
    - the name of the station visited,
    - the time of arrival of the train,
    - the time of departure of the train.
  - No train timetable train journey entry lists the same station twice.
  - Times of train departures and train arrivals *shall* be compatible
    - with reasonable stops at stations
    - and with the distance between stations visited.
  - Two immediately time-consecutive train timetable station visits *must* be compatible with the railway net: It shall be possible to route a train between such consecutive stations.

■

- Notice in the descriptive model the unhedged use of the verbs *consists*, *connects*, *is* and *are*.
- A description is *indicative:* It tells *what there is*.
- Likewise notice in the prescriptive model the use of the (compelling) verbs *shall* and *must*.
- A prescription is *putative:* It tells *what there will be*.

4.3.3. Extensional and Intensional Models

Definition 16 ***Extensional Model:***

- *An extensional model (black, opaque box) presentation models something as if observed by someone external to the universe of discourse*

. ■

Definition 17 ***Intensional Model:***

- *An intensional model in logic, correlative words that indicate the reference of a term or concept. Intension indicates the internal content of a term or concept that constitutes its formal definition.*

. ■

- *Intensional versus extensional meaning:* (i) intensional meaning: consists of the qualities or attributes the term *connotes* (the attributes of class membership); (ii) extensional meaning: consists of the qualities or attributes the term denotes (the class members themselves).
- *Connotation:* the suggesting of a meaning by a word apart from the thing it explicitly names or describes.
- *Denotation:* a direct specific meaning as distinct from an implied or associated idea. (glass (or white), transparent box) presentation models the internal structure of the *universe of discourse*

- An *extensional model* presents, i.e., reflects, the behaviour as seen from an outside.
- In that sense one may claim, but the claim cannot be justified from extensionality alone, that an extensional model focuses on properties, on what the thing that is being modelled offers an outside world, i.e., users of that thing.
- If a model is expressed in a property-oriented style, then we can claim the converse: that the model is extensional!
- An *intensional model* presents the internal mechanisms of what is being modelled in a way that may explain why it has the extension that it might have.

Example 17 – Extensional Model Presentations:

- (1) To explain the square root function, $\sqrt{n} = r$, by explaining that $r \times r = n \wedge\ r \geq 0$ is to give an extensional definition, hence model.
- (2) To explain a stack extensionally we may define (a) the stack sorts for elements and stacks, (b) the signatures of the empty, pop, top and push functions, and (c) the axioms which relate sorts and operations.

■

Example 18 – Intensional Model Presentations:

- (1) To explain the square root function, $\sqrt{n}$, by presenting, e.g., the Newton–Raphson algorithm, is to give an intensional definition, hence model.
- (2) An intensional model of stacks may model stacks as lists of (extensionally modelled) elements, and define the (i) empty, (ii) pop, (iii) top, and (iv) push functions in terms of (i) constructing the empty list, of (ii) yielding the tail of a list, of (iii) yielding the head element of a list, and (iv) of concatenating a supplied element to the front of the list.

■

4.4. Rôles of Models

We pursue modelling for one or more reasons:

- (i) *To gain understanding:* in the process of modelling we are forced to come to grips with many issues of the universe of discourse.
- (ii) *To get inspiration and to inspire:* abstraction often invites such generalisations that induce, in the writer, or in the reader, desires of change.
- (iii) *To present, educate and train:* a model can serve as the basis for presentations to others for the purposes of awareness, education or training.

- (iv) *To assert and predict:* a mathematical, including a formal model, usually allows abstract interpretation — in the "vernacular": calculations, computations — that simulates, estimates or otherwise expresses potential properties of the universe of discourse.
- (v) *To implement:* two kinds of implementations can be suggested:
  - in business process re-engineering we propose the re-engineeringof some domain on the basis of a model and
  - in *computing systems design* we base the development of requirements on a domain specification and
  - we base software design on requirements.

End of Lecture 4

Abstraction & Modelling

Lecture 5

Semiotics

5. Semiotics

5.1. An Overview

Definition 18 – **Semiotics:** *Semiotics is the study of and knowledge about the structure of all 'sign systems'.*

- *We divide this study (and our knowledge) into three parts:*
  - *syntax,*
  - *semantics and*
  - *pragmatics.*

■

Definition 19 – **Syntax:** *Syntax is the study of and knowledge about how signs (words) can be put together to form correct sentences and of how sentence-signs relate to one another.* ■

- We shall understand signs (words) and sentences in a wide sense.
  - Programs in a programing languages and specifications in a formal specification language
    - will here be considered to be sentences.
    - and
      - variable and function identifiers (a, ab, id, fct, etc.);
      - constants (0, 1, 2, ..., **true, false, chaos**, etc.);
      - expressions and statements;

· statement and expression symbols (such as

· value operators ($+$, $-$, $/$, $*$, $\times$, $\mapsto$, etc.), and
· **dom, rng, elems, len, card**) etc.;
· type operators (`Boolean`, `integer`, `real`, `char`, `string`, etc.,
· and **-set**, **-infset**, *, $^\omega$, $\rightarrow$, $\xrightarrow{\sim}$, $\xrightarrow[m]{}$, $\times$);

· parentheses ((,), {, }, [,] etc.);
· comma (,);
· semicolon (;);
· assignment symbols (:=, =, $\leftarrow$);
· definition symbols ($\equiv$, ::=)

etc.)

∗ and literals (such as

· **begin, end, let, in, cases, of, while, do, type, value, axiom**,

etc.)

will here be considered words.

– But also
  * diagrams, say technical drawings and
  * actual layout of, for example, buildings and railway tracks,

  will be considered sentences, and
  * the boxes and lines of diagrams,
  * and the various visual (proper) sub-components of actual physical phenomena

  will be considered words.

- That is, we consider phenomena such as
  - geographical and geodetic maps;
  - buildings, and their "accompanying" architectural and engineering drawings;
  - railway tracks (lines and stations) and their "accompanying" engineering drawings;
  - cities and city plans;
  - etc.,

  as languages.
- GIS and CAD/CAM systems
  - thus translate descriptions of such phenomena into database structures
  - and GIS and CAD/CAM system user commands are compiled and executed as language programs in the context of these databases.

- We define syntaxes in terms of either BNF grammars or `RSL` types.
- We distinguish between syntactic types and semantic types.
- The syntactic types designate sentences and words.
- The semantic types designate meanings (of sentences and words).

Definition 20 – **Semantics:** *Semantics is the study of and knowledge about the meaning of words, sentences, and structures of sentences.* ■

- Let

  type
  SynType, SemType

- designate syntactic, respectively semantic types.
- Then

  value
  M: SynType $\rightarrow$ SemType

- presents the signature of a semantic function M.
  - It assumes all syntactic inputs to be well-formed.
  - If the syntax of SynType is such as to allow ill-formed sentences, then we must define a well-formedness function:

    value
    Wf_SynType: SynType $\rightarrow$ **Bool**

- and the signature of M must be sharpened:

  value
  M: SynType $\stackrel{\sim}{\rightarrow}$ SemType

Definition 21 – **Pragmatics:** *Pragmatics is the study of and knowledge about the use of words, sentences and structures of sentences, and of how contexts affect the meanings of words, sentences, etc.* ■

5.2. Syntax

- We recall our definition:
  - syntax is the study of and knowledge about how signs (words) can be put together to form correct sentences
  - and of how sentence-signs relate to one another.
- We shall divide our presentation of syntax into three parts:
  - (i) `BNF` grammars,
  - (ii) concrete type syntax and
  - (iii) abstract type syntax.
- These three are just three increasingly more abstract ways of dealing with syntax.

– The subject of syntax goes well beyond our software engineering treatment.

– The computer science topic of *formal languages and automata theory* studies far wider consequences of grammars and syntax than we cover.

– The computing science topic of *regular expression recognizers* and *context free language parsers* likewise goes well beyond our coverage — and their study is important for the software engineer to implement efficient software for language handling.

- BNF grammars define sets of strings of characters;
- concrete type syntaxes define sets of mathematical structures (numbers, Booleans, sets, Cartesians, maps and functions over concrete type values); and
- abstract type syntaxes define properties of simple phenomena and concept entities.

- We say that BNF grammars and concrete type syntaxes define simple phenomena and concept entities in a *model-oriented* fashion
- whereas abstract type syntaxes define simple phenomena and concept entities in a *property-oriented* fashion.

5.2.1. BNF Grammars

- BNF stands for Backus Naur Form.
- BNF grammars, as we shall see, stand for sets of finite length strings of characters, including blanks and punctuation marks.

Definition 22 – Character: *A character is a symbol that can be displayed (on paper, on a computer screen, or otherwise).* ■

Example 19 – Characters: Our example is the conventional example of characters from an English/American computer keyboard:

- a, A,
- b, B,
- c, C,
- ...,
- z, Z,
- 0, 1,
- 2, 3,
- 4, 5,
- 6, 7,
- 8, 9,
- !, @,
- #, $,
- %, &,
- *, ~,
- (,),
- {, },
- [,],
- -, +,
- ’, ”,
- <, >,
- ., ,,
- :, ;,
- ?, /,
- |, [blank],
- etc.

■

Definition 23 – **Alphabet:** *An alphabet is a finite set of characters.* ■

Example 20 – **Alphabet:** Three examples, $\mathcal{A}_i, \mathcal{A}_j, \mathcal{A}_k$, of subsets of the above characters:

- $\mathcal{A}_i$: {a,b,[blank],O,|}
- $\mathcal{A}_j$: {a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z,[blank]}
- $\mathcal{A}_k$: {0,1,2,3,4,5,6,7,8,9,[blank],a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z}

■

Definition 24 – **Terminal:** *By a terminal we understand a sequence of one or more characters of some given alphabet.* ■

Example 21 – **Terminals:**

- a, b, c, 0, 1;
- a, aa, aaa, abc, 0, 1, 00, 01, 001;
- open, deposit, withdraw, close, account, client, number.

■

Definition 25 – **Non-terminal:** *By a non-terminal we understand a specially highlighted sequence of one or more characters, not necessarily from the alphabet of a given set of terminals.* ■

Example 22 – **Non-terminals:**

- $\nprec$Command$\nsucc$,
- $\nprec$Open$\nsucc$, $\nprec$Deposit$\nsucc$, $\nprec$Withdraw$\nsucc$, $\nprec$Close$\nsucc$
- $\nprec$ClientName$\nsucc$, $\nprec$AccountNumber$\nsucc$,
- $\nprec$Cash$\nsucc$, $\nprec$Amount$\nsucc$

■

Definition 26 – **BNF Rules:** *By a BNF rule we understand a triple:*

- *($\mathcal{L}$, ::=, $\mathcal{A}s$)*
- *where $\mathcal{L}$ is a non-terminal*
- *and $\mathcal{A}s$ is a finite set of zero, one or more alternatives*
  - *where an alternative is a finite sequence of zero, one or more non-terminals and/or terminals.*
- *::= is the definition symbol.*

■

Example 23 – **BNF Rules:**

- ≺Command≻ ::= ≺Open≻ | ≺Close≻
- ≺Open≻ ::= client ≺ClientName≻ opens account
- ≺Withdraw≻ ::= client ≺ClientName≻ withdraws ≺Amount≻
  from account ≺AccountNumber≻

■

Definition 27 – **BNF Grammar:** *By a* ***BNF*** *grammar we understand a quadruple*

$$(\mathcal{N},\mathcal{T},\mathcal{R},\mathcal{G})$$

- $\mathcal{N}$ *is an alphabet of non-terminals,*
- $\mathcal{T}$ *is an alphabet of terminals,*
- $\mathcal{R}$ *is a set of rules,*
- $\mathcal{G}$ *is a non-terminal,*

such that

- $\mathcal{G}$ *is in* $\mathcal{N}$*;*
- *all the left hand sides of rules in* $\mathcal{R}$ *are in* $\mathcal{N}$*;*
- *all the non-terminals of right hand sides of rules in* $\mathcal{R}$ *are distinct and together form* $\mathcal{N}$*; and*
- *all the terminals of right hand sides of rules in* $\mathcal{R}$ *are in* $\mathcal{T}$*.*

■

Example 24 – **BNF Grammar: Banks:** The ... (*Identifiers,Alphanumerics,Numerals*) are not part of the syntax.

$\mathcal{N}$: {≺Command≻, ≺Open≻, ≺Deposit≻, ≺Withdraw≻, ≺Close≻, ≺ClientName≻,

≺AccountNumber≻, ≺Amount≻}

$\mathcal{T}$: {client,opens account,deposits,into account,withdraws,from account,closes account} . . .

... ∪ *Identifiers* ∪ *Alphanumerics* ∪ *Numerals*

$\mathcal{R}$: ≺Command≻ ::= ≺Open≻ | ≺Deposit≻ | ≺Withdraw≻ | ≺Close≻

≺Open≻ ::= client ≺ClientName≻ opens account

≺Deposit≻ ::= client ≺ClientName≻ deposits ≺Amount≻ into account ≺AccountNumber≻

≺Withdraw≻ ::= client ≺ClientName≻ withdraws ≺Amount≻ from account ≺AccountNumber≻

≺Close≻ ::= client ≺ClientName≻ closes account ≺AccountNumber≻

≺ClientName≻ ::= ... *Identifiers*

≺AccountNumber≻ ::= ... *Alphanumerics*

≺Amount≻ ::= ... *Numerals*

$\mathcal{G}$: ≺Command≻

■

- The "... ∪ *Identifiers* ∪ *Alphanumerics* ∪ *Numerals*" which is not part of the syntax, ought be fully defined by a somewhat longer BNF grammar.
- The example showed one form of BNF grammars.
- In the below definition of the meaning of BNF grammars we abstract from the above forms of rules for BNF grammars.
- Examples 26 on Slide 191 and 28 on Slide 209 follow up on this example of a BNF grammar by presenting
  - a concrete type syntax, respectively
  - an abstract type syntax

  for "supposedly" the same command language.

Definition 28 – **Meaning of a BNF Grammar:** *The meaning of a BNF grammar is a language, that is, a possibly infinite set of finite length strings over the terminal alphabet of the BNF grammar. To properly define this language, for any BNF grammar we shall proceed, formally, as follows:*

- *Let N and T denote the alphabets of non-terminals and terminals.*
- *Let $r : (n, \{\ell_1, \ell_2, \ldots, \ell_m\})$ designate a rule, r, that is: $n : N$ and $\ell_i : (N|T)^*$, for $m \geq 0, 1 \leq i \leq m$ where $(N|T)^*$ denotes the possibly infinite set of finite length strings over non-terminal and terminal characters.*
- *Let $G : (N, T, R, n_0)$ where G names the grammar, N the alphabet of non-terminals, T the alphabet of terminals, R the finite set of rules (over N and T), and n_0 a distinguished non-terminal of N.*
- *G is constrained as follows:*
  - *no two distinct rules, (n, ls) and (n', ls') in G have $n = n'$,*
  - *that is: all left hand side non-terminals are distinct and together they form N, and*
  - *there is a rule $r : (n, \{\ell_1, \ell_2, \ldots, \ell_m\})$ in R such that $n = n_0$.*

- *Let* $s_i \widehat{\ } s_j$ *denote the concatenation of strings* s_i *and* s_j.
- *Let* $s \widehat{\ } U \widehat{\ } s'$ *be a string over* $(N|T)^*$.
- *Let* $(U, \{\ell_1, \ell_2, \ldots, \ell_i, \ldots, \ell_m\})$ *be a rule in* R.
- *Then* $s \widehat{\ } U \widehat{\ } s' \ \rightarrow_G \ s \widehat{\ } \ell_i \widehat{\ } s'$ *means: from* $s \widehat{\ } U \widehat{\ } s'$, *by means of rule* $(U, \{\ell_1, \ell_2, \ldots, \ell_m\})$ *of* G, *we derive,* $\rightarrow_G$, $s \widehat{\ } \ell_i \widehat{\ } s'$.
- *If, in some rule* $(U, \{\ell_1, \ell_2, \ldots, \ell_i, \ldots, \ell_m\}), m = 0$, *that is, the rule is* $(U, \{\})$, *then* $s \widehat{\ } U \widehat{\ } s' \ \rightarrow_G \ s \widehat{\ } s'$.
- *If* $s_p \rightarrow_G s_q$, $s_q \rightarrow_G s_r$, ..., *and* $s_v \rightarrow_G s_w$, *then* $s_p \rightarrow_G{}^* s_w$ *(and thus* $s_p \rightarrow_G{}^* s_r$, $s_p \rightarrow_G{}^* s_w$, *etc. — assuming* $s_p \rightarrow_G{}^* s_v$*).*
- *The meaning of* $\rightarrow_G$ *is specific to the given Grammar.*
- *Now the meaning,* $\mathcal{L}(G)$ *is defined as follows:*

$$\mathcal{L}_G = \{s \mid \ n_0 \rightarrow_G{}^* s \wedge \ s \in T^*\}$$

■

- Some BNF grammars are such that $\mathcal{L}_G$ is empty:
  - no derivation, $\rightarrow_G$, and hence $\rightarrow_G{}^*$,
  - results in terminal strings.

Example 25 – **Meaning of a BNF Grammar:** First we show a form of BNF grammar which is more in line with the above definition.

$$\mathcal{N} = \{E, C, V, P, I, B\},$$

$$\mathcal{T} = \{0, 1, 2, 3, 4, 5, \ldots, a, b, c, \ldots, z, +, -, *, /,$$

$$\mathcal{R} =$$

$$\begin{array}{ll}
0 \{ & E = C \mid V \mid P \mid I \mid B, \\
1 & C = 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \ldots, \\
2 & V = a \mid b \mid c \mid \ldots \mid z, \\
3 & P = -E, \\
4 & I = E\ O\ E, \\
5 & O = + \mid - \mid * \mid /, \\
6 & B = (\ E\ )\ \}
\end{array}$$

$$n_0 = E$$

Then we show a derivation of the expression $5 + (a/3) + -c$ from E:

$E \rightarrow$	0	$5 + (E\ O\ E)\ O\ E \rightarrow$	0
$I \rightarrow$	4	$5 + (V\ O\ E)\ O\ E \rightarrow$	2
$E\ O\ E \rightarrow$	0	$5 + (a\ O\ E)\ O\ E \rightarrow$	5
$C\ O\ E \rightarrow$	1	$5 + (a / E)\ O\ E \rightarrow$	0
$5\ O\ E \rightarrow$	5	$5 + (a / C)\ O\ E \rightarrow$	1
$5 + E \rightarrow$	0	$5 + (a / 3)\ O\ E \rightarrow$	5
$5 + I \rightarrow$	4	$5 + (a / 3) + E \rightarrow$	0
$5 + E\ O\ E \rightarrow$	6	$5 + (a / 3) + P \rightarrow$	3
$5 + B\ O\ E \rightarrow$	4	$5 + (a / 3) + -E \rightarrow$	0
$5 + (E)\ O\ E \rightarrow$	0	$5 + (a / 3) + - V \rightarrow$	2
$5 + (I)\ O\ E \rightarrow$	4	$5 + (a / 3) + - c$	

- Please disregard that we have, in the above derivation, always replaced the leftmost non-terminal. That is of no consequence.
- The fact that the BNF grammar is ambiguous,
  - that is, allows entirely distinct derivation sequences to
  - lead to the same final string
  - also should be disregarded.
- It is, at most, perhaps, an unfortunate choice of grammar !

■

5.2.2. Concrete Type Syntax

Definition 29 – Concrete Type Syntax:

- *By a concrete type syntax we shall understand*
- *the definition of a set of mathematical structures*
- *such as sets, Cartesians, lists, maps and functions.*

■

Example 26 – A Concrete Type Syntax: Banks:

49. There are clients, c:C, account numbers a:A and money, m:M.
50. A bank record client accounts and account balances.
51. Client accounts map client names to a finite number of zero, one or more account numbers.
52. Account balances map account numbers into money balances.
53. All client accounts are recorded by the account balances, and the account balances record only accounts listed by one or more clients.

type
49. C, A, Money
50. Bank = Clients $\times$ Accounts
51. Clients = C $\xrightarrow[m]{}$ A-**set**
52. Accounts = A $\xrightarrow[m]{}$ Money

axiom
53. $\forall$ (cs,acs):Bank · $\cup$ **rng** cs = **dom** acs

- The two sentences of Item 53
  - (53.1) All client accounts are recorded by the account balances, and
  - (53.2) the account balances record all accounts listed by clients.
- correspond to:

axiom

53. $\forall$ (cs,acs):Bank $\cdot$

53.1 $\cup$ **rng** cs $\subseteq$ **dom** acs

53.2 **dom** acs $\subseteq$ $\cup$ **rng** cs

- Hence formula line 53.

- We then give a concrete type syntax for a bank/client comand language first hinted at in Example 24 on Slide 182.

54. To the syntactic types we include client identifications, account numbers, money (i.e., cash) and amounts of such.
55. There are open, deposit, withdraw and close commands.
56. Open commands identify the client.
57. Deposit commands identify the client, the account number and the monies to be deposited.
58. Withdraw commands identify the client, the account number and the amount of monies to be withdrawn.
59. Close commands identify the client and the account to be closed.

54. C, A, Money, Amount
55. Command = Open | Deposit | Withdraw | Close
56. Open == mkO(c:C)
57. Deposit == mkD(c:C,a:A,m:Money)
58. Withdraw == mkW(c:C,a:A,amount:Amount)
59. Close == mkC(c:C,a:A)

- This example will be followed up by Examples 28 on Slide 209 and 29 on Slide 213.

■

Definition 30 – Meaning of Concrete Type Syntax:

- *We explain both the syntactic* RSL *type definitions,*
- *expressions and*
- *their meaning. First the syntax.*

60. There are two kinds of type definitions:
    (a) simple type definitions which have a left hand side type name and a right hand side type expression (separated by an equal sign: '='),
    (b) record type definitions which have a left hand side type name and a right hand side pair of a record constructor name and a parenthesized list of pairs of distinct selector and not necessarily distinct type names (where the left and the right is separated by a double equal sign: '==').
61. Type, record constructor and selector names are identifiers.

62. A type expression is either

(a) a Boolean (**Bool**) or
(b) an integer number (**Intg**) or
(c) a natural number (**Nat**) or
(d) a real number (**Real**) type name, or is
(e) a set (A-**set**) or
(f) a Cartesian ($A \times B \times \ldots \times C$) or
(g) a list (A^*) or
(h) a map ($A \xrightarrow[m]{} B$) or
(i) a partial ($A \stackrel{\sim}{\rightarrow} B$) or
(j) a total function ($A \rightarrow B$) type expression, or is
(k) a set of (alternative, |) type expressions.

- The below only shows how such type definitions and expressions may look like when we (otherwise) write them.
- That is, the below type definitions and expressions are not type definitions and proper type expressions.

Type Definition Examples:

60(a). TN = TE

60(b). TN == RN(s1:TN1,s2:TN2,...,sn:TNm)

Type Expression Examples:

62. TE =

62(a). **Bool**

62(b). **Int**

62(c). **Nat**

62(d). **Real**

62(e). TE-**set**

62(f). TE1×TE2×...×TEm

62(g). TE*

62(h). TEi $\overrightarrow{m}$ TEj

62(i). TEi $\stackrel{\sim}{\rightarrow}$ TEj

62(j). TEi → TEj

62(k). TE1 | TE2 | ... | TEm

where: m≥2

- In an concrete type syntax of two or more type definitions
  - all left hand side type names are distinct,
  - all type names occurring in right hand side record constructor and type expressions are defined by an abstract or concrete type syntax, and
  - no set (62(e).) or function (62(h).,62(i).) type is defined recursively.

- Now to the meaning of a concrete type syntax.
- The meaning of a type name is the meaning of the right hand side
  - type expression TE
  - or record constructor expression RN(s1:TN1,s2:TN2,...,sn:TNm).
- We shall use a concept of the meanings being "sets" of values.
  - The practicing software engineer may consider these "sets" just as normal set.
  - But, for reasons not explained here, but based in a proper definition of a mathematical semantics for RSL, they are not sets in the usual sense of mathematics.

- The meaning of a type expression depends on its form:
  - **Bool**: the “set” {**false,true,chaos**};
  - **Intg**: the “set” of all integers: {…,-3,-2,-1,0,1,2,3,…};
  - **Nat**: the “set” of all natural numbers {0,1,2,3,…};
  - **Real**: the “set” of all real number $\{m/n \mid m,n:$**Nat**$, n \neq 0\}$;
  - TE-**set**: the “set” of all finite sets of zero, one or more elements, e_i, of TE: $\{\ldots\{e_i, e_j, \ldots, e_k\}\ldots\}$;
  - TE1×TE2×...×TEm: the “set” of all Cartesians $\{\ \ldots,\ (e_{TE1_i}, e_{TE2_i}, \ldots, e_{TEm_n}), \ldots\}$;
  - TE*: the “set” of all finite length sequences (or lists) of zero, one or more elements, e_i, of TE: $\{\ldots,\langle e_i, e_j, \ldots, e_k\rangle,\ldots\}$;
  - TEi $\overrightarrow{m}$ TEj: the “set” of all finite maps (that is, finite definition set discrete functions) from elements, e_{i_k}, of TEi to elements, e_{j_ℓ}, of TEj: $\{\ldots,[\ldots,e_{i_k}\mapsto e_{j_\ell},\ldots]\ldots\}$;

- TEi$\stackrel{\sim}{\to}$TEj: the "set" of all partial functions from some, but not all elements, e_{i_k}, of TEi, to elements, e_{j_ℓ}, of TEj: λe_i : TEi $\bullet\ \mathcal{E}_{\mathsf{TEj}}$;[9]
- TEi$\to$TEj: the "set" of all total functions from elements e_{i_k}, of type TEi, to elements, e_{j_ℓ}, of TEj: $\lambda e_i \bullet \mathcal{E}_j$; and
- TE1|TE2|...|TEm: the "set" which is a "union" of the "sets" denoted by the TEi for i=1,2,...,m.

- The meaning of a record constructor type definition, Tn == RN(s1:TN1, s2:TN2, ..., sn:TNm), is the "set" of all records, { . . . , RN(te1,te2,. . . ,tem), . . . }, where tei is any value of type TEi for all i.

■

[9] The expression $\lambda e : T_i \bullet \mathcal{E}_{T_j}$ denotes the function which when applied to elements v, of type T_i, yields a value (of type T_j) of expression $\mathcal{E}_{T_j}$ where all free occurrences of e are replaced by v.

- Where the meaning of a BNF grammar is a possibly infinite set of strings over a terminal alphabet,
- the meaning of a concrete type syntax is a possibly infinite "set" of mathematical values:
  - Booleans, numbers, sets, Cartesians, lists, maps, partial and total functions,
  - where the elements of sets, Cartesians, lists and maps,
  - and where the function argument and results values are any of the values of any of these mathematical values.
- The `RSL` type constructs also allow infinite sets, TE-**infset**, and infinite length lists, TE^{ω}.

5.2.3. Abstract Type Syntax

Definition 31 – Abstract Type Syntax Definition: *By an abstract type syntax definition we mean*

- *a set of one or more sorts, that is, type names,*
- *a set of one or more observer, of zero, one or more selector and zero, and one or more constructor ('make') function signatures (function names and argument and result types over these sorts) and*
- *a set of axioms which which range over the sorts and defines the observer, selector and constructor ('make') functions.*

■

Definition 32 – **Abstract Type Syntax:** *By an abstract type syntax we mean*

- *a set of sort values of named type*
- *with observer, selector and constructor ('make') functions*
- *where the sort values and the functions satisfy the axioms.*

■

Example 27 – An Abstract Type Syntax: Arithmetic Expressions:

- First we treat the notion of analytic grammar,
- then that of synthetic grammar.

⊕ *Analytic Grammars: Observers and Selectors* ⊕

- For a "small" language of arithmetic expressions
- we focus just on constants, variables, and infix sum and product terms:

type
  A, Term
value
  is_term: A → **Bool**
  is_const: Term → **Bool**
  is_var: Term → **Bool**
  is_sum: Term → **Bool**
  is_prod: Term → **Bool**
  s_addend: Term → Term
  s_augend: Term → Term
  s_mplier: Term → Term
  s_mpcand: Term → Term

axiom

$$
\begin{array}{l}
\quad \forall\ \mathrm{t}{:}\mathrm{Term} \cdot \\
\qquad (\mathrm{is\_const}(\mathrm{t}) \wedge \sim (\mathrm{is\_var}(\mathrm{t}) \vee \mathrm{is\_sum}(\mathrm{t}) \vee \mathrm{is\_prod}(\mathrm{t}))) \wedge \\
\qquad (\mathrm{is\_var}(\mathrm{t}) \wedge \sim (\mathrm{is\_const}(\mathrm{t}) \vee \mathrm{is\_sum}(\mathrm{t}) \vee \mathrm{is\_prod}(\mathrm{t}))) \wedge \\
\qquad (\mathrm{is\_sum}(\mathrm{t}) \wedge \sim (\mathrm{is\_const}(\mathrm{t}) \vee \mathrm{is\_var}(\mathrm{t}) \vee \mathrm{is\_prod}(\mathrm{t}))) \wedge \\
\qquad (\mathrm{is\_prod}(\mathrm{t}) \wedge \sim (\mathrm{isc\_const}(\mathrm{t}) \vee \mathrm{isv\_ar}(\mathrm{t}) \vee \mathrm{is\_sum}(\mathrm{t}))), \\
\quad \forall\ \mathrm{t}{:}\mathrm{A} \cdot \mathrm{is\_term}(\mathrm{t}) \equiv \\
\qquad (\mathrm{is\_var}(\mathrm{t}) \vee \mathrm{is\_const}(\mathrm{t}) \vee \mathrm{is\_sum}(\mathrm{t}) \vee \mathrm{is\_prod}(\mathrm{t})) \wedge \\
\qquad (\mathrm{is\_sum}(\mathrm{t}) \equiv \mathrm{is\_term}(\mathrm{s\_addend}(\mathrm{t})) \wedge \mathrm{is\_term}(\mathrm{s\_augend}(\mathrm{t}))) \wedge \\
\qquad (\mathrm{is\_prod}(\mathrm{t}) \equiv \mathrm{is\_term}(\mathrm{s\_mplier}(\mathrm{t})) \wedge \mathrm{is\_term}(\mathrm{s\_mpcand}(\mathrm{t})))
\end{array}
$$

- A is a universe of "things":
  - some are terms,
  - some are not!
- The terms are restricted, in this example,
  - to constants,
  - variables,
  - two argument sums
  - and two argument products.
- How a sum is represented one way or another is immaterial to the above.
  - Thus one could think of the following external, written representations:
  - $a + b$,
  - $+ab$,
  - (PLUS A B), or
  - $7^a \times 11^b$.

⊕ *Synthetic Grammars: Generators* ⊕

A synthetic abstract syntax introduces generators of sort values, i.e., as here, of terms:

value

$$\text{mk\_sum: Term} \times \text{Term} \to \text{Term}$$
$$\text{mk\_prod: Term} \times \text{Term} \to \text{Term}$$

axiom

$$\forall\ \text{u,v:Term} \cdot$$
$$\text{is\_sum(mk\_sum(u,v))} \wedge \text{is\_prod(mk\_prod(u,v))} \wedge$$
$$\text{s\_addend(mk\_sum(u,v))} \equiv \text{u} \wedge\ \text{s\_augend(mk\_sum(u,v))} \equiv \text{v} \wedge$$
$$\text{s\_mplier(mk\_prod(u,v))} \equiv \text{u} \wedge\ \text{s\_apcand(mk\_prod(u,v))} \equiv \text{v} \wedge$$
$$\text{is\_sum(t)} \Rightarrow \text{mk\_sum(s\_addend(t),s\_augend(t))} \equiv \text{t} \wedge$$
$$\text{is\_prod(t)} \Rightarrow \text{mk\_prod(s\_mplier(t),s\_mpcand(t))} \equiv \text{t}$$

■

- The previous example illustrated the expression of
  - an abstract type syntax for
  - a *syntactic type* of arithmetic expressions.
- The next example illustrates the expression of
  - an abstract type syntax for
  - a *semantic type* of banks
  - as well as expression of an abstract type syntax
  - for a *syntactic type* of client commands.
- The example "pairs" with Example 26 on Slide 191.

Example 28 – An Abstract Type Syntax: Banks:

- We refer (back) to Example 26 on Slide 191.

⊕ ***Abstract Syntax of Semantic Types*** ⊕

63. There are banks (BANK) and clients (C), and client have accounts (A) with amounts (Amount) of money (M).
64. From a bank one can observe its set of clients (by their client identifications, C),
65. and its set of accounts (by their account numbers, A).
66. From a bank one can observe the account numbers of a client.
67. For every bank client there is at least one account.
68. From a bank one can observe the money of an account of a client.

type

63 BANK, C, A, Amount, M

value

64 obs_Cs: BANK $\rightarrow$ C-**set**

65 obs_As: BANK $\rightarrow$ A-**set**

66 obs_As: BANK $\times$ C $\rightarrow$ A-**set**

pre obs_As(bank,c): c $\in$ obs_Cs(bank)

axiom

$\forall$ bank:BANK $\cdot$

67 $\forall$ c:C $\cdot$ c $\in$ obs_Cs(bank) $\Rightarrow$ obs_As(bank,c)$\subseteq$obs_As(bank)

type

68 obs_M: BANK $\times$ C $\times$ A $\rightarrow$ M

pre obs_M(bank,c,a): c $\in$ obs_Cs(bank)$\wedge$a $\in$ obs_As(bank,c)

⊕ *Abstract Syntax of Syntactic Types* ⊕

69. There are bank transaction commands (Command) and these are either open (Open), deposit (Deposit), withdraw (Withdraw) or close (Close) commands.
70. One can observe whether a command is an open, or a deposit, or a withdraw, or a close command.
71. From any command one can observe the identity of the client issuing the command.
72. From other that open commands one can observe the number of the account "against" which the client is directing the transaction.
73. From a deposit command one can observe the cash money being deposited.
74. From a withdraw command one can observe the amount of cash money to be withdrawn.

type

69 cmd:Command, Open, Deposit, Withdraw, Amount, Close

value

70 is_Open, is_Deposit, is_Withdraw, is_Close: Command $\rightarrow$ **Bool**

71 obs_C: Command $\rightarrow$ C

72 obs_A: Command $\xrightarrow{\sim}$ A

pre obs_A(cmd): ∼is_Open(cmd)

73 obs_M: Command $\xrightarrow{\sim}$ M

pre obs_M(cmd): is_Deposit(cmd)

74 obs_Amount: Command $\xrightarrow{\sim}$ Amount

pre obs_Amount(cmd): is_Withdraw(cmd)

■

5.2.4. Abstract Versus Concrete Type Syntax

Example 29 – **Comparison: Abstract and Concrete Banks:**

- We refer (back) to Examples 26 on Slide 191 and 28 on Slide 209.
- The former presented a concrete type syntax of both semantic and syntactic types related to banking.
- The latter presented an abstract type syntax of both semantic and syntactic types related to banking.
- Supposedly the two notion of banks are the same !
- We formulate this as follows:
  - The meaning of the model-oriented definition of Example 26
  - is a model of the meaning of the property-oriented definition of Example 28.
- The properties expressed by Example 28 are satisfied by the meaning of Example 26.
- Usually a property-oriented definition has many, usually an infinite set of models.

- We now show three "other" model-oriented definitions of the semantic types of banks.

type
$\mathsf{BANK}_1 = (\mathsf{C} \xrightarrow[m]{} \mathsf{A}\text{-}\mathbf{set}) \times (\mathsf{A} \xrightarrow[m]{} \mathsf{M})$

type
$\mathsf{BANK}_2 = \mathsf{A} \xrightarrow[m]{} (\mathsf{C}\text{-}\mathbf{set} \times \mathsf{M})$

type
$\mathsf{BANK}_3 = (\mathsf{C} \times \mathsf{A} \times \mathsf{M})\text{-}\mathbf{set}$

- The first model is that of Example 26 with its invariant as expressed in Item 53 on Slide 191.
- The second model requires the following invariant

axiom
$$\forall\ \mathsf{bank}_2:\mathsf{BANK}_2 \cdot \forall\ (\mathsf{cs},\mathsf{m}):(\mathsf{C}\text{-}\mathbf{set} \times \mathsf{M}) \cdot (\mathsf{cs},\mathsf{m}) \in \mathbf{rng}\ \mathsf{bank}_2 \Rightarrow \mathsf{cs} \neq \{\}$$

- The third model is like a relational database-oriented model.
  - Each Cartesian (c,a,m) in any bank_3 is like a relation tuple.
  - But two different Cartesians with same account number, (c,a,m), (c′,a,m′) must have same cash balance:

type
  $\text{BANK}_3 = (\text{C}\times\text{A}\times\text{M})$-**set**
axiom
  $\forall\ \text{bank}_3:\text{BANK}_3 \cdot$
    $\forall\ (c,a,m),(c',a',m'):(\text{C}\times\text{A}\times\text{M}) \cdot$
      $\{(c,a,m),(c',a',m')\}\subseteq\text{bank}_3 \wedge a=a' \Rightarrow m=m'.$

- For each of the four models:
  - Example 26 and the three above,
  - one can define the observer observer functions of Example 28
  - and prove its axioms.

■

5.3. Semantics

- We recall our definition of semantics:
  - semantics is the study of and knowledge about the meaning of words, sentences, and structures of sentences.
- We consider two forms of semantics definition styles:
  - denotational and behavioural.
  - Both will be briefly characterised
  - and both will be "amply" exemplified.
- There are many (other) semantics definition styles:
  - but we shall leave it to other textbooks to fill you in on those,
  - and even our presentation of the two "announced" styles need a deeper treatment than the present software engineering coverage.

5.3.1. Denotational Semantics

Definition 33 – Denotational Semantics:

- *By a denotational semantics we understand a semantics which*
- *to simple sentences ascribe a mathematical function and*
- *to composite sentences ascribe a semantics which is a homomorphic composition of the meaning of the simpler parts.*

■

Example 30 – A Denotational Language Semantics: Banks:

- We continue Example 26.
- We augment the simple sentences of commands with a 'command' which is a list of simple commands:

type

$\mathsf{Command}' = \mathsf{Command} \mid \mathsf{CmdList}$
$\mathsf{ComdList} == \mathsf{mkCL(cl{:}Command^*)}$
$\mathsf{Response} == \mathsf{ok} \mid \mathsf{nok}_d \mid \mathsf{nok}_w \mid \mathsf{nok}_c \mid \mathsf{mkM(m{:}M)}$

- The meaning of a command is a bank to bank state change and a response value.

value

$\mathcal{M}\colon \mathsf{Command}' \to \mathsf{BANK} \xrightarrow{\sim} \mathsf{BANK} \times \mathsf{Response}$

- The nok_d, nok_w, and nok_c responses "signal" the client that command arguments were erroneous.
- Opening an account is always possible, but for the other simple commands
  - the client must be known by the bank and
  - the account must be an account of that client.
- For the withdraw command
  - the amount to be withdrawn must be less than or equal to the account balance.
- The response value serves to record these conditions for a successful transaction as well as "containing" the "returned" monies in the case of the withdraw and close commands.

value

$\mathcal{M}$: Command$'$ $\to$ BANK $\stackrel{\sim}{\to}$ BANK $\times$ Response

$\mathcal{M}$(cmd)(bank) $\equiv$

case cmd **of**

mkO(c) $\to$ Open(c)(bank),

mkD(c,a) $\to$ Deposit(c,a)(bank),

mkW(c,m) $\to$ Withdraw(c,a,am)(bank),

mkC(c,a) $\to$ Close(c,a)(bank),

mkCL(cl) $\to$ Compose(cl)(bank)(ok)

end

- Next we define the five auxiliary semantic functions, Open, Deposit, Withdraw, Close and Compose
- The auxiliary functions Open, Deposit, Withdraw and Closefunction definitions show the denotational principle of ascribing simple functions, in BANK$\stackrel{\sim}{\to}$BANK, to simple commands.
- The latter, Compose, is defined first.
- It shows the denotational principle of homomorphic composition.

value

Compose: Command* $\rightarrow$ BANK $\xrightarrow{\sim}$ BANK $\times$ Response
Compose(cl)(bank)(r) $\equiv$
if cl=$\langle\rangle$
then (bank,r)
else let (r′,bank′) = $\mathcal{M}$(**hd** cl)(bank) **in** Compose(**tl** cl)(bank′)(r′) **end**
end

- The homomorphic composition is that of function composition:
  - Compose(**tl** cl) being applied to the bank part, (**bank′**), of the result of $\mathcal{M}$(**hd** cl)(bank).
  - The "continuation" Response argument, r, of Compose is there to "clean" up, by "removing", the intermediate response results.

value
m_0:M
Open: C → BANK → BANK × Response
Open(c)(bank) ≡
let a:A · a ∉ **dom** acs **in**
let cs′ = **if** c∉ **dom** cs **then** [c↦{a}] **else** [c↦cs(c)∪{a}] **end**,
acs′ = acs ∪ [a↦m_0] **in**
((cs′,acs′),ok) **end end**

- The lecturer "reads" the function definition.

value
Deposit: $\mathsf{C} \times \mathsf{A} \times \mathsf{M} \rightarrow \mathsf{BANK} \stackrel{\sim}{\rightarrow} \mathsf{BANK} \times \mathsf{Response}$
Deposit(c,a,m)(cs,acs) $\equiv$
  if c $\in$ **dom** cs $\wedge$ a $\in$ cs(c)
    then ((cs,acs$\dagger$[a$\mapsto$AddM(acs(a),m)]),ok)
    else ((cs,acs),nok$_d$)
  end
AddM: $\mathsf{M} \times \mathsf{M} \rightarrow \mathsf{M}$

- The lecturer “reads” the function definition.

- **value**

  Withdraw: C $\times$ A $\times$ Amount $\to$ BANK $\xrightarrow{\sim}$ BANK $\times$ Response
  Withdraw(c,a,am)(cs,acs) $\equiv$
    if c $\in$ **dom** cs $\land$ a $\in$ cs(c) $\land$ LessEqM(am,ConvM(acs(a)))
      then ((cs,acs†[a$\mapsto$SubM(am,acs(a))]),mkM(ConvM(am)))
      else ((cs,acs),nok$_w$)
    end
  SubM: M $\times$ M $\xrightarrow{\sim}$ M
  ConvM: (M $\to$ Amount)|(Amount $\to$ M)
  LessEqM: Amount $\times$ Amount $\to$ **Bool**

- The lecturer "reads" the function definition.

- **value**

$$
\begin{array}{l}
\mathsf{Close}\colon \mathsf{C} \times \mathsf{A} \rightarrow \mathsf{BANK} \xrightarrow{\sim} \mathsf{BANK} \times (\mathsf{nok}|\mathsf{M})\\
\mathsf{Close(c,a)(cs,acs)} \equiv\\
\quad \textbf{if}\ \mathsf{c} \in \textbf{dom}\ \mathsf{cs} \wedge \mathsf{a} \in \mathsf{cs(c)}\\
\quad\quad \textbf{then let}\ \mathsf{cs}' = \mathsf{cs}\|[\,\mathsf{c}'\mapsto\mathsf{cs}(\mathsf{c}')\backslash\{\mathsf{a}\}|\mathsf{c}'\mathsf{:C}\bullet\mathsf{c}' \in \textbf{dom}\ \mathsf{cs}\wedge\mathsf{a} \in \mathsf{cs}(\mathsf{c}')\,],\\
\quad\quad\quad\quad\quad \mathsf{acs}' = \mathsf{acs}\backslash\{\mathsf{a}\}\ \textbf{in}\\
\quad\quad\quad\quad\quad ((\mathsf{cs}',\mathsf{acs}'),\mathsf{acs(a)})\ \textbf{end}\\
\quad\quad \textbf{else}\ ((\mathsf{cs,acs}),\mathsf{nok}_c)\\
\quad \textbf{end}
\end{array}
$$

- The lecturer "reads" the function definition.

- The nok_d, nok_w, and nok_c responses could, in a requirements prescriptions be detailed, for example as follows:
  - nok_d: `"client or account deposit arguments were wrong"`,
  - nok_w: `"client or account deposit arguments were wrong or amount to be withdrawn was too large"`, and
  - nok_c: `"client or account deposit arguments were wrong"`;
- And, of course, even these more informative "diagnostics" can be sharpened to reflect the conjunction of the **if** predicates.

■

5.3.2. Behavioural Semantics

Definition 34 – Behavioural Semantics: *By a behavioural semantics we shall here understand*

- *a semantics which emphasises*
- *concurrency properties*
- *of the language being modelled.*

■

-
-

Example 31 – A Behavioural Semantics: We continue Example 30.

75. There are a number of clients, each is considered a distinct cyclic behaviour
76. indexed by a (well: the) unique Client index.

value

75. client: C ... → **Unit**

- The **Unit** designates a "never ending" client behaviour.
- The . . . will now be "filled in".

77. Each client communicates with one bank (with communication modelled in terms of **channel** input/output (c?, c!cmd)).
78. There is one cyclic bank behaviour.

77. **channel** $\{ch[\,c\,]|c{:}C\}$ Command|Response
75. client: $c{:}C \times C\Sigma \to$ **out**,**in** $\{cb[\,c'\,]|c'{:}C\backslash\{c\}\}$ **Unit**
78. bank: BANK $\to$ **in**,**out** $\{ch[\,c\,]|c{:}C\}$ **Unit**

79. Client behaviours (over some internal state, cσ [not explained]) alternate
    (a) between doing nothing, **skip**, in relation to the bank, and
    (b) arbitrarily issuing, based on some property of its local state,
    (c) a client/banking command to the bank
    (d) and waiting for a response from the bank —
    (e) based on which the client updates its local state and continues.
80. We do not detail the predicate over choice of commands and the local client state nor the local client state update.

type
79. CΣ
value
79. client(c,cσ) $\equiv$
79(a). (**skip** ; client(c,cσ))
$\lceil\rceil$
79(b). (**let** cmd:Command$'$ $\cdot$ $\mathcal{P}$(cmd,cσ) **in**
79(c). ch[c]!cmd;
79(d). **let** r = ch[c]? **in**
79(e). client(c,client_state_update(cmd,r,cσ)) **end end**)

80. $\mathcal{P}$: Command$'$ $\times$ CΣ $\rightarrow$ **Bool**
80. client_state_update: Command$'$ $\times$ Response $\times$ CΣ $\rightarrow$ CΣ

81. The bank alternates between serving any of its customers.
82. Sooner or later, if ever a client, c, issues a command, cmd, that command and its origin is received.
83. The command interpretation results in a new bank and a response.
84. The response is communicated to the issuing client.
85. And the bank continues in the possibly new bank state.

value

81. $\mathsf{bank}(\beta) \equiv$
82. $\quad$ **let** $(\mathsf{c},\mathsf{cmd}) = \,[\!]\{\mathsf{ch}[\,\mathsf{c}\,]?|\mathsf{c}{:}\mathsf{C}\}$ **in**
83. $\quad$ **let** $(\beta',\mathsf{r}) = \mathcal{M}(\mathsf{cmd})(\beta)$ **in**
84. $\quad$ $\mathsf{ch}[\,\mathsf{c}\,]!\mathsf{r};$
85. $\quad$ $\mathsf{bank}(\beta')$ **end end**

■

5.3.3. Axiomatic Semantics

Definition 35 – Axiomatic Semantics:

- *By an axiomatic semantics we understand*
  - *a pair of abstract type presentations*
    - *of syntactic*
    - *and semantic types*
  - *and a set of axioms which express the*
    - *meaning of some syntactic values*
    - *in terms of semantic values.*

■

Example 32 – **An Axiomatic Semantics: Banks:** We continue Example 28. We now give an axiomatics semantics of the simple commands of Example 28. We start by recalling semantic and syntactic types and observer functions. First the semantic types:

type
63 BANK, C, A, Amount, M
value
64 obs_Cs: BANK $\rightarrow$ C-**set**
65 obs_As: BANK $\rightarrow$ A-**set**
66 obs_As: BANK $\times$ C $\rightarrow$ A-**set**
pre obs_As(bank,c): c $\in$ obs_Cs(bank)
axiom
$\forall$ bank:BANK $\cdot$
67 $\forall$ c:C $\cdot$ c $\in$ obs_Cs(bank) $\Rightarrow$ obs_As(bank,c)$\subseteq$obs_As(bank)
type
68 obs_M: BANK $\times$ C $\times$ A $\rightarrow$ M
pre obs_M(bank,c,a): c $\in$ obs_Cs(bank)$\wedge$a $\in$ obs_As(bank,c)

- Then the syntactic types:

type

69 cmd:Command, Open, Deposit, Withdraw, Amount, Close

value

70 is_Open, is_Deposit, is_Withdraw, is_Close: Command $\rightarrow$ **Bool**

71 obs_C: Command $\rightarrow$ C

72 obs_A: Command $\overset{\sim}{\rightarrow}$ A

  pre obs_A(cmd): $\sim$is_Open(cmd)

73 obs_M: Command $\overset{\sim}{\rightarrow}$ M

  pre obs_M(cmd): is_Deposit(cmd)

74 obs_Amount: Command $\overset{\sim}{\rightarrow}$ Amount

  pre obs_Amount(cmd): is_Withdraw(cmd)

- The semantic function signatures are:

value

open: Open $\rightarrow$ BANK $\rightarrow$ BANK $\times$ A
deposit: Deposit $\rightarrow$ BANK $\rightarrow$ BANK $\times$ (ok|nok)
withdraw: Withdraw $\rightarrow$ BANK $\rightarrow$ BANK $\times$ (mkM(m:M)|nok)
close: Close $\rightarrow$ BANK $\rightarrow$ BANK $\times$ (mkM(m:M)|nok)

- We shall illustrate an axiomatic semantics of just Open commands.

```
value
  m₀:M
axiom
  ∀ bank:Bank
    ∀ op:Open ·
      let c = obs_C(op),
         (bank′,a) = open(op)(bank),
         cs = obs_Cs(bank),        cs′ = obs_Cs(bank′),
         acs = obs_As(bank),       acs′ = obs_As(bank′),
         cacs = if c ∈ obs_Cs(bank) then obs_As(bank,c) else {} end,
         cacs′ = obs_As(bank′,c) in
      cs\{c} = cs′\{c} ∧ c ∈ cs′ ∧ a ∉ acs ∧ a ∉ cacs′ ∧
      acs′ = acs ∪{a} ∧ cacs′ = cacs ∪{a} ∧ m₀=obs_M(bank′,c,a) ∧
      ∀ c′:C · c′ ∈ cs\{c} ⇒ obs_As(bank,c′)=obs_As(bank′,c′) ∧
        ∀ a:A · a ∈ obs_As(bank,c′) ⇒  obs_M(bank,c′,a)=obs_M(bank′,c′,a)
      end
```

- The student is encouraged to formulate the axiomatic semantics for the Deposit, Withdraw and Close commands.

This ends Example 32 ■

5.4. Pragmatics

- We recall our definition of pragmatics:
  - pragmatics is the study of and knowledge about the use of words, sentences and structures of sentences,
  - and of how contexts affect the meanings of words, sentences, etc.
- Recall that we "extended" the notion of sentences and words to include
  - building drawings,
  - city plans,
  - machine drawings,
  - production floor machinery,
  - radio circuit diagrams,
  - railway track layouts,
  - enterprise organisation charts,
  - et cetera,
- We think of these two or three dimensional artefacts as designating systems.

- Rather than dwelling on how, for example bank clients may use the client/banking language of command, we shall, in our example we therefore emphasise
  - mostly the pragmatics of both **what** and **how**
    - we choose to domain model (describe) and
    - requirements prescribe

    and
  - to some extent also the pragmatics of **why** these systems are endowed with certain structurings.
- We shall emphasise
  - *"the use of words, sentences and structures of sentences,"*
  - and not say much about
  - *"how contexts affect the meanings of words, sentences, etc."*

Example 33 – Pragmatics: Banks:

- The pragmatics of **what** we describe of banks
  - is determined by the pedagogics of giving as simple, yet as “convincing” examples
  - of syntactic and semantic types
  - and both denotational (albeit a rather “simplistic example of that)
  - without embellishing the example with too many kinds of banking services (for example, intra-bank account transfers, mortgages, statement requests, etc.).

- The pragmatics of **how** we describe banks
  - is determined by the didactics of
  - covering both concrete type syntaxes and abstract type syntaxes of syntactic types, and
  - covering both denotational and behavioural semantics definitions.

- The pragmatics of **why** we describe banks
  - is determined by our wish to convince the student
  - that it is not a difficult software engineering task
  - to give easy and realistic domain descriptions
  - of important, seemingly "large" infrastructure components
  - (such as banks).

- The pragmatics related to *"how contexts affect the meanings"* includes
  - that we do not, in Examples 26, 28, 29, and 30–31,
  - describe other financial institutions such as portfolio (wealth and investment) management, insurance companies, credit card companies, brokers, trader, commodity and stock exchanges,
  - let alone include the modelling of several banks.
  - These other institutions and banks form one possible context of our model and hence our model limits the meaning of client/banking commands.
  - Another possible context is provided by the personal diligent or casual or delinquent or sloppy, etc., behaviour of client. The human behaviours are not modelled, but must eventually be modelled (cf. Sect. 's Examples 68 on Slide 634 and 69 on Slide 636).

- Pragmatics is not about empirical aspects of software engineering.
- The pragmatics
  - that we refer to, in the above definition,
  - is that of staff and users of banks.
- The pragmatics
  - that we covered in the example
  - is that of the pedagogics and didactics
  - of presenting a methodology for software engineering.

- The aspects of software engineering that we cover,
  - namely that of domain and requirements engineering,
  - are not empirical sciences, or, more precisely
  - the methodologies of domain and requirements engineering
  - are not based on studies of the behaviour
  - of neither domain nor requirements engineers.

- The aspects of software engineering that we put forward in these lectures
  - are based on computing science[10]
  - and computing science, like mathematics, upon which it is based,
  - is not an empirical science.[11]

[10] Software engineering is applied computing science.

[11] Although some may reasonably claim that Mathematics is what Mathematicians do, that is not, in our opinion, the same as saying: let us therefore study how all those people who claim they are mathematicians are doing call what they mathematics and let the result of such an empirical study determine what mathematics is!

- The pragmatics of the kind of domains
  - in the context of the way in which we wish to describe these domains
  - and prescribe requirements for computing systems to serve in these domains

  is far from studied.
- We,
  - but this is only a personal remark
  - and not a scientific conjecture,
- venture to claim that
  - perhaps one cannot formalise pragmatics,
    - that is, that pragmatics is what cannot be formalised.
  - But this is just a "hunch" !

5.5. Discussion

- We summarise this lecture on semiotics by first recalling our definition:
  - Semiotics is the study of and knowledge about the structure of all 'sign systems'.
    - In accordance with some practice we have divided our presentation into three parts:
      - syntax,
      - semantics and
      - pragmatics.

- BNF grammars were first[12] made known (in the late 1950s) in connection with the work on defining the first block structured programming language, `Algol 60`.
- So BNF grammars were for defining the one-dimensional, i.e., textual layout of programming languages.
- In Sect. we enlarge the scope of syntax to also embody the definition of the structure of 'systems' (that is, domains) such as mentioned there (Slide 168).
- The "language" of systems is the possibly infinite set of utterings that staff and users, i.e., system stake holders express when working with (or in) the system. We exemplified this only briefly and in terms of client/banking commands.

[12] Dines: Find reference to Don Knuth's "paper" on ancient Indian's knowing of "BNF".

- We make, in these lectures , a distinction between using syntax definitions to define syntactic types versus using syntax definitions to define semantic types.

MORE TO COME

- We encourage students to embark on studies of the (albeit informal) pragmatics of domains.

End of Lecture 5

Semiotics

Lecture 6

A Specification Ontology

6. A Specification Ontology

The point of philosophy is to start with something so simple
as not to seem worth stating,
and to end with something so paradoxical
that no one will believe it.

Bertrand Russell
The Philosophy of Logical Atomism
The Monist[13], The Open Court Publ.Co., Chicago, USA
Vol. XXVIII 1918: pp 495-527
Vol. XXIX 1919: pp 32-63, 190-222, 345-380

[13] See http://www.archive.org/search.php?query=title%3A(the monist) AND creator%3A(Hegeler Institute)

- The basic approach to description (prescription and specification) is to describe algebras.
- We take a somewhat “novel” approach to this:
  - We describe simple entities and functions (i.e., operations) over these, as for any algebra;
  - and then we focus on behaviours of simple entities as sequences of function invocations and events, where events are the results of usually external function invocations.
- In addition we describe (prescribe and specify)
  - both informally, by means of precise narratives
  - and formally — here in the `RAISE` specification language `RSL`.

Example 34 – **Transport Net (II):**

- In Example 10 nets, hubs and links are examples of simple entities of (Pages 62–66).
- (Hub and link identifiers are not simple entities, they are entity attributes.)
- The link insert and delete operations (Slides 66–77) of that example are examples of operations.
- The situation that a link suddenly “disappears” (a road segment is covered by a mudslide, or a bridge collapses) are examples of events (that can be “mimicked” by the remove link operation).
- The sequence of many insert, some remove and a few link disappearances form a behaviour.

6.1. Russel's Logical Atomism

6.1.1. Metaphysics and Methodology

- Russell's **metaphysical** view can be expressed as follows:
  - *the world consists of a plurality of independent existing particulars (phenomena, things, entities, individuals*[14]*)*
  - *exhibiting qualities and standing in relations.*
- Russell's **methodology** for doing philosophy was
  - *follow a process of analysis,*
  - *whereby one attempts to define or construct*
  - *more complex notions or vocabularies*
  - *in terms of simpler ones.*

[14] We consider the terms 'particulars', 'phenomena', 'things', 'entities' and 'individuals' to be synonymous.

- Russell's idea of **logical atomism** can be expressed as consisting of
  - both
    - the *metaphysics* and
    - the *methodology*
  - as basically outlined above.
- We shall later in this chapter take up Russell's line of inquiry.

6.1.2. The Particulars [Phenomena - Things - Entities - Individuals]

- So which are the particulars, that is, the phenomena that we are to describe?
- Well, they are the particulars of the domain.
- How do we describe them?
- Well, we shall now introduce our description ontology.
  - By 'ontology' is meant

    the philosophical study of the nature of being, existence or reality in general, as well as of the basic categories of being and their relations. Traditionally listed as a part of the major branch of philosophy known as metaphysics, ontology deals with questions concerning what entities exist or can be said to exist, and how such entities can be grouped, related within a hierarchy, and subdivided according to similarities and differences.[15]

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

- We can speak both of
  - a domain ontology
  - and a description ontology.
- First, in this section, we shall cover the notion of description ontology, or, as we shall generalise it, specification ontology.

- The purpose of having a firm understanding of, hopefully a good specification ontology is to be better able to produce good domain ontologies.
- Later in following chapters we shall outline how to construct pleasing domain ontologies.

- Our specification (description, prescription) ontology emphasises, as also mentioned in the above indented and *slanted* quote,
  - *the basic categories of being and their relations* and
  - *how such entities can be grouped, related within a hierarchy, and subdivided according to similarities and differences*.
- The *basic* description *categories*,
  that is, the *grouping, hierarchy, subdivision* of means of description are these:
  - simple entities[16],
  - operations (over entities),
  - events (involving entities) and
  - behaviours.

[16] We shall consider all four categories of description items as entities, but single out simple entities as a category of its own.

- We should here bring a reasoned argument,
  - of philosophical nature,
  - in order to motivate this subdivision of specification means.
- Instead we postulate this subdivision
  - and hope that the reader, after having read this chapter,
  - will accept the subdivision.

6.2. Entities

- We have used and we shall be using the term 'entity' extensively in this book.
- Other, synonymous terms are 'particular' and 'individual'.

Definition 36 – Entity: *By an entity we shall understand a phenomenon or a concept which is*

- *either inert (in which case we shall call it a 'simple entity'),*
- *or "like" a function,*
- *or an event,*
- *or a behaviour.*

■

6.3. Simple Entities and Behaviours

- We lump two of the description categories: simple entities and behaviours in this section.
- The reason is that we wish to highlight a duality:
  - simple entities as exhibiting behaviours, and
  - behaviours are evolving around simple entities.
- In the vernacular one often refers to a phenomenon using a name that both covers that phenomenon as a simple entity and as a behaviour.
- **Example:** A bank as a simple, in this case composite entity with demand/deposit accounts, mortgage accounts, and clients; and a bank as a behaviour with clients opening and closing accounts, depositing into and withdrawing from accounts, etc., and with events such a interest rate change, attempts at withdrawing below the credit limits, etc. □

6.3.1. Simple Entities

Definition 37 – Simple Entity:

- *By a simple entity we shall here understand*
  - *a phenomenon that we can designate, viz.*
  - *see, touch, hear, smell or taste, or*
  - *measure by some instrument (of physics, incl. chemistry).*

■

Example 35 – Simple Entities:

- (i) Air traffic: aircraft, terminal control towers, ground control towers, regional control centers and continental control centers.
- (ii) Financial service industry: money (cash), securities instruments (like stocks, bonds, a transacted credit card slip, etc.), banks, brokers, traders, stock exchanges, commodities exchanges, bank and mortgage accounts, etc.
- (iii) Health care: citizens and potential patients, medical staff, wards, beds, medicine and operating theatres.
- (iv) Railway systems: train stations and rail tracks, their constituent (linear, switch, crossover, etc.) rail units, trains, train wagons, tickets, passengers, timetables, station and train staff.

■

- We model simple entities by stating their types.

type
  A, B, C, D, E, F, G. H, J
  aT
  $cT = A \times B\text{-}\mathbf{set} \times C^* \times (D \underset{m}{\rightarrow} E) \times (F \rightarrow G) \times (H \overset{\sim}{\rightarrow} J)$
value
  obs_A: aT → A
  obs_Bs: aT → B-**set**
  obs_Dl: aT → C*
  obs_mEF: aT → $D \underset{m}{\rightarrow} E$
  obs–tfGH: aT → F→G
  obs_pfJK: aT → $H \overset{\sim}{\rightarrow} J$

- Our specification language, here `RSL`, allows us to define
  - Cartesian,
  - set,
  - list,
  - map,
  - partial function and
  - total function

  types.
- It also allows us to define sorts and observer functions.
- These possibilities permit us to model composite entities as follows:
  - unordered collections of sub-entities as sets,
  - ordered collections of sub-entities as lists,
  - finite sets of uniquely "marked" sub-entities as maps, and
  - infinite or indefinite sets of uniquely "marked" sub-entities as functions, partial or total.

- A simple entity has properties[17].
- A simple entity is
  - either continuous
  - or is discrete, and then it is
    - either atomic
    - or composite.

[17] We shall refrain from a deeper, more ontological discussion of what is meant by properties. Suffice it here to state that properties are what we can model in terms of types, values (including functions) and axioms.

- By an attribute we mean a property of an entity
  - *a simple entity has properties* $p_i, p_j, \ldots, p_k$.
- Typically we express attributes by a pair of
  - a type designator: *the attribute is of type* V, and
  - a value: *the attribute has value* v (of type V, i.e., $v : V$).
- A simple entity may have many properties.

Example 36 – **Attributes:**

- A continuous simple entity, like 'oil', may have the following attributes:
  - class: *mineral*,
  - kind: *Brent-crude*, amount: *6 barrels*,
  - price: *45 US $/barrel*.

type
Oil, Barrel, Price
Class == mineral|organic
Kind == brent_crude|brent_sweet_light_crude|oseberg|ecofisk|forties
value
obs_Class: Oil → Class
obs_Kind: Oil → Kind
obs_No_of_Barrels: Oil → **Nat**
obs_Price: Oil → Price

- An atomic simple entity, like a ‘person’, may have the following attributes:
  - gender : *male*, name: *Dines Bjørner*,
  - age: *(“oh well, too old anyway”)*,
  - height: *178cm*, weight: *(“oh well, too much anyway”)*.

```
type
  Person, Age, Height
  Gender == female|male
value
  obs_Gender: Person → Gender
  obs_Age: Person → Age
  obs_Height: Person → Height
```

- A composite simple entity, like a railway system, may have the following attributes:
  - country: *Denmark*,
  - name: *DSB*,
  - electrified: *partly*,
  - owner : *independent public enterprise owned by Danish Ministry of Transport.*

type
  RS, Owner, Name, Owner,
  Country == denmark!norway|sweden|...
  Electrified == no|partly|yes
value
  obs_Country: RS $\rightarrow$ Country
  obs_Name: RS $\rightarrow$ Name
  obs_Electrified: RS $\rightarrow$ Electrified
  obs_Owner: RS $\rightarrow$ Owner

The above informal and formal descriptions are just rough sketches. ■

- A simple entity is said to be continuous
  - if it can be arbitrarily decomposed into smaller parts
  - each of which still remain simple continuous entities
  - of the same simple entity kind.

Example 37 – **Continuous Entities:**

- Examples of continuous entities are:
  - oil, i.e., any fluid,
  - air, i.e., any gas,
  - time period and
  - a measure of fabric.

■

- A simple entity is said to be discrete if its immediate structure is not continuous.
  - A simple discrete entity may, however, contain continuous sub-entities.

Example 38 – **Discrete Entities:** Examples of discrete entities are:

- persons,
- rail units,
- oil pipes,
- a group of persons,
- a railway line (of one or more rail units) and
- an oil pipeline (of one or more oil pipes, pumps and valves).

■

- A simple entity is said to be atomic
  - if it cannot be meaningfully decomposed into parts
  - where these parts have a useful "value" in the context in which the simple entity is viewed and
  - while still remaining an instantiation of that entity.

Example 39 – Atomic Entities:

- Thus a 'physically able person', which we consider atomic,
  - can, from the point of physical ability,
  - not be decomposed into meaningful parts: a leg, an arm, a head, etc.
- Other atomic entities could be a rail unit, an oil pipe, or a hospital bed.

■

- The only thing characterising an atomic entity is its attributes.
- A simple entity, c, is said to be composite
  - if it can be meaningfully decomposed into sub-entities
  - that have separate meaning in the context in which c is viewed.

Example 40 – Composite Entities (1):

- A *railway net* (of a railway system) can be decomposed into
  - a set of one or more *train lines* and
  - a set of two or more *train stations*.
- Lines and stations are themselves composite entities.

```
type
  RS, RN, Line, Station
value
  obs_RN: RS → RN
  obs_Lines: RN → Line-set
  obs_Stations: RN → Station-set
axiom
  ∀ rs:RS,rn:RN · let rn = obs_RN(rs) in
    card obs_Lines(rn)≥2 ∧ card obs_Stations(rn)≥1 end
```

■

Example 41 – Composite Entities (2):

- An *Oil industry* whose decomposition include:
  - one or more *oil fields*,
  - one or more *pipeline systems*,
  - one or more *oil refineries* and
  - one or more *one or more oil product distribution systems*.
- Each of these sub-entities are also composite.

type
  Oil_Industry, Oil_Field, Pipeline_System, Refinery, Distrib_System
value
  obs_Oil_Field: Oil_Industry → Oil_Field-**set**
  obs_Pipeline_System: Oil_Industry → Pipeline_System-**set**
  obs_Refineries: Oil_Industry → Refinery-**set**
  obs_Distrib_Systems: Oil_Industry → Distrib_System-**set**
axiom
  [all observed sets are non−empty]

■

- Composite simple entities are thus characterisable by
  - their **attributes**,
  - their **sub-entities**, and
  - the **mereology** of how these sub-entities are put together.

Definition 38 – **Mereology:**

- *Mereology is the theory of parthood relations:*
- *of the relations of part to whole*
- *and the relations of part to part within a whole.*

■

We shall exemplify the above in the following, abstract example.

Example 42 – Mereology: Parts and Wholes (1):

- We speak of systems as assemblies.
- From an assembly we can immediately observe a set of parts.
- Parts are either assemblies or units.
- For the time being we do not further define what units are.

type
 S = A, A, U, P = A | U
value
 obs_Ps: A → P-**set**

- Parts observed from an assembly are said to be immediately embedded in that assembly.

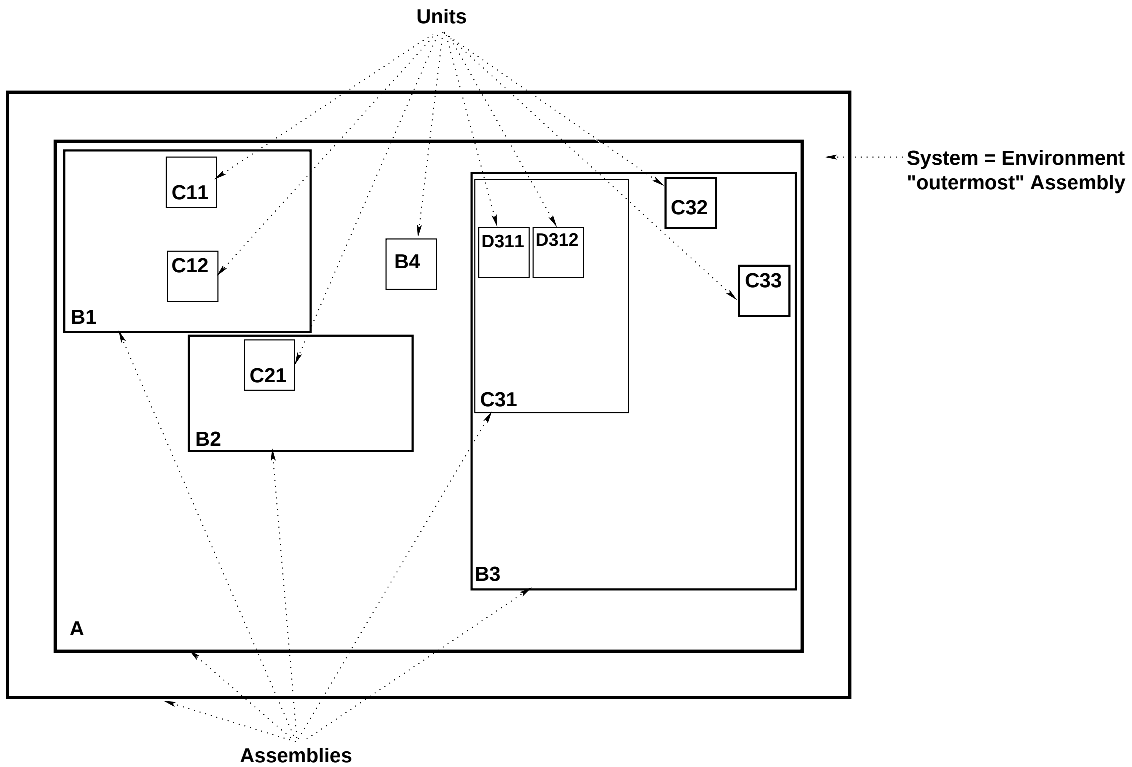


Figure 1: Assemblies and Units "embedded" in an Environment

- For the time being we omit any reference to an environment.
- Embeddedness generalises to a transitive relation.
- All parts thus observable from a system are distinct.
- Given obs_Ps we can define a function, xtr_Ps,
  - which applies to an assembly, a, and
  - which extracts all parts embedded in a.
- The functions obs_Ps and xtr_Ps define the meaning of embeddedness.

value

xtr_Ps: A → P-**set**

xtr_Ps(a) ≡

let ps = obs_Ps(a) **in** ps ∪ ∪{xtr_Ps(a′)|a′:A•a′ ∈ ps} **end**

- Parts have unique identifiers.

type
  AUI
value
  obs_AUI: P $\rightarrow$ AUI
axiom
  $\forall$ a:A $\cdot$
    let ps = obs_Ps(a) **in**
    $\forall$ p′,p″:P $\cdot$ {p′,p″}$\subseteq$ps $\wedge$ p′$\neq$p″ $\Rightarrow$ obs_AUI(p′)$\neq$obs_AUI(p″) $\wedge$
    $\forall$ a′,a″:A $\cdot$ {a′,a″}$\subseteq$ps $\wedge$ a′$\neq$a″ $\Rightarrow$ xtr_Ps(a′)$\cap$ xtr_Ps(a″)={} **end**

- We shall now add to this a rather general notion of parts being otherwise related.
- That notion is one of connectors.
- Connectors may, and usually do provide for connections — between parts.
- A connector is an ability be be connected.
- A connection is the actual fulfillment of that ability.
- Connections are relations between two parts.
- Connections "cut across" the "classical"
  - *parts being part of the (or a) whole* and
  - *parts being related by embeddedness or adjacency*.

6.**A Specification Ontology** 3.**Simple Entities and Behaviours** 1.**Simple Entities** 0. 0

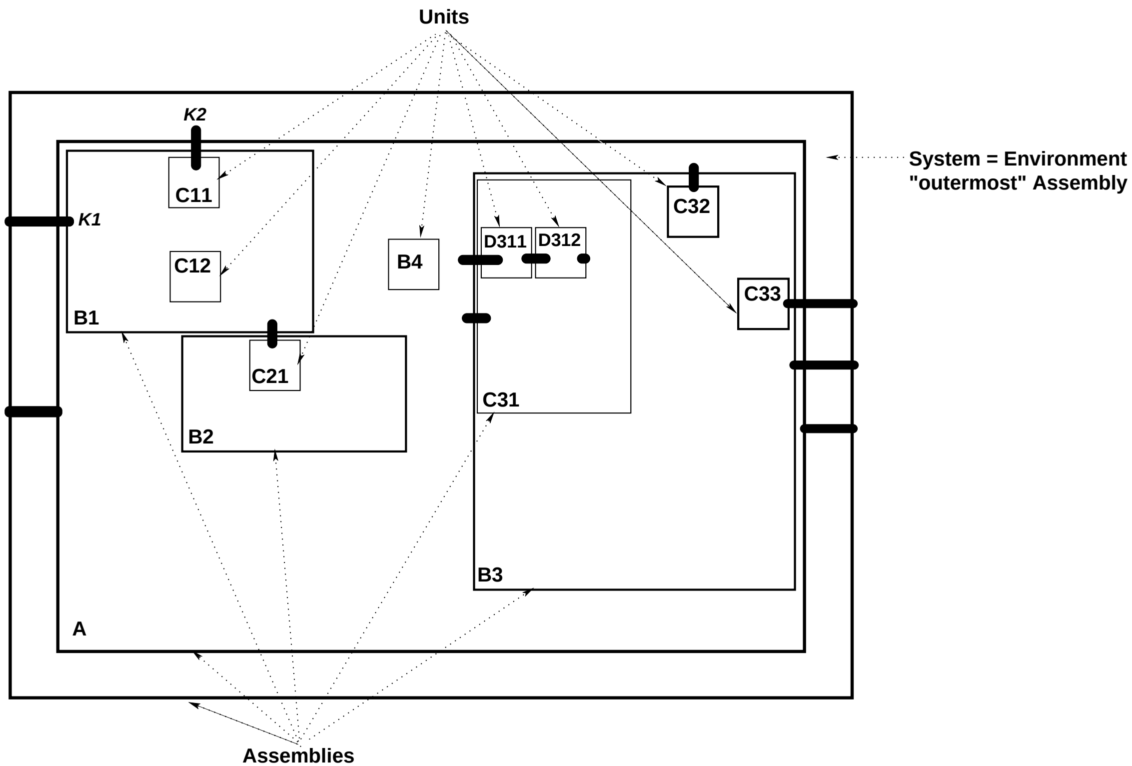


Figure 2: Assembly and Unit Connectors: Internal and External

- Figure 2 on the preceding slide "repeats" Fig. 1 on Slide 281 but "adds" connectors.
- The idea is that connectors
  - allow an assembly to be connected to any embedded part, and
  - allow two adjacent parts to be connected.
- In Fig. 2 on the preceding slide
  - assembly A is connected, by *K2*, (without, as we shall later see, interfering with assembly B1), to part C11;
  - the "external world" is connected, by K1 to B1,
  - etcetera.
- Thus we make, to begin with, a distinction between
  - internal connectors that connect two identified parts, and
  - external connectors that connect an identified part with an external world.
- Later we shall discuss more general forms of connectors.

- From a system we can observe all its connectors.
- From a connector we can observe
  - its unique connector identifier and
  - the set of part identifiers of the parts that the connector connects,
    - two if it is an internal connectors,
    - one if it is an external connector.
- All part identifiers of system connectors identify parts of the system.
- All observable connector identifiers of parts identify connectors of the system.

type
  K
value
  obs_Ks: S → K-**set**
  obs_KI: K → KI
  obs_Is: K → AUI-**set**
  obs_KIs: P → KI-**set**
axiom
  $\forall$ k:K $\cdot$ 1$\leq$**card** obs_Is(k)$\leq$2,
  $\forall$ s:S,k:K $\cdot$ k $\in$ obs_Ks(s) $\Rightarrow$ $\exists$ p:P $\cdot$ p $\in$ xtr_Ps(s) $\Rightarrow$ obs_AUI(p) $\in$ obs_Is(k),
  $\forall$ s:S,p:P $\cdot$ $\forall$ ki:KI $\cdot$ ki $\in$ obs_KIs(p) $\Rightarrow$ $\exists$! k:K $\cdot$ k $\in$ obs_Ks(s) $\wedge$ ki=obs_KI(k)

- This model allows for a rather "free-wheeling" notion of connectors
  - one that allows internal connectors to "cut across" embedded and adjacent parts;
  - and one that allows external connectors to "penetrate" from an outside to any embedded part.

- For Example 44 on Slide 305 we need define an auxiliary function.
  - xtr∀KIs(p) applies to a system
  - and yields all its connector identifiers.

value

xtr∀KIs: S → KI-**set**

xtr∀Ks(s) ≡ {obs_KI(k)|k:K·k ∈ obs_Ks(s)}

- This ends our first model of a concept of mereology.
- The parts are those of assemblies and units.
- The relations between parts and the whole are,
  - on one hand, those of
    - embeddedness and
    - adjacency,

    and
  - on the other hand, those expressed by connectors: relations
    - between arbitrary parts and
    - between arbitrary parts and the exterior.

- A number of extensions are possible:
  - one can add "mobile" parts and "free" connectors, and
  - one can further add operations that allow such mobile parts to move from one assembly to another along routes of connectors.
- Free connectors and mobility assumes static versus dynamic parts and connectors:
  - a free connector is one which allows a mobile part to be connected to another part, fixed or mobile; and
  - the potentiality of a move of a mobile part introduces a further dimension of dynamics of a mereology.

6.**A Specification Ontology** 3.**Simple Entities and Behaviours** 1.**Simple Entities** 0. 0

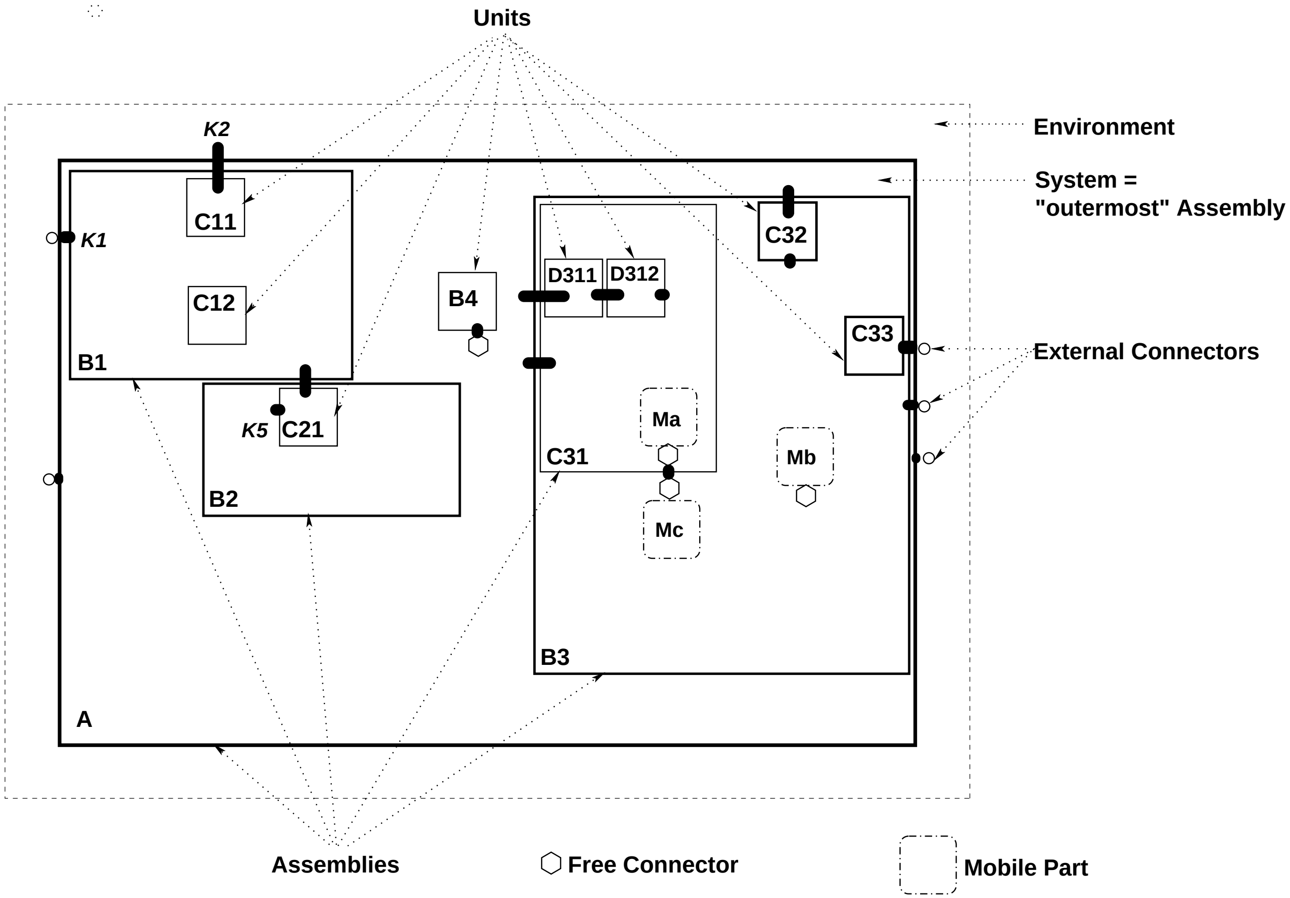


Figure 3: Mobile Parts and Free Connectors

- We shall leave the modelling of free connectors and mobile parts to another time.
- Suffice it now to indicate that the mereology model given so far is relevant:
  - that it applies to a somewhat wide range of application domain structures, and
  - that it thus affords a uniform treatment of proper formal models of these application domain structures.

This ends Example 42 ■

- Summarising we find, for discrete simple entities, that
  - atomic entities are characterisable by their attributes (as well as by the operations that apply to entity arguments); and that
  - composite entities are characterisable by their attributes, the sub-entities from which they are made up and the mereology, i.e., the part-whole relations between these sub-entities (as well as by the operations that apply to entity arguments).
- Continuous entities we treat almost as we treat atomic entities
  - except that we can speak of, i.e., define, functions that decompose a continuous entity of kind $\mathcal{C}$ into an arbitrary number of continuous entities of the same kind $\mathcal{C}$, and
  - vice-versa: compose a continuous entity of kind $\mathcal{C}$ from an arbitrary number of continuous entities of the same kind $\mathcal{C}$.

6.3.2. Behaviours

- Behaviours can be
  - simple, sequential, and
  - behaviours can be highly composite.
- To define behaviours we need notions of states and actions:
  - By a **domain state** we mean any collection of simple entities — so designated by the domain engineer.
  - By a **domain action** we mean the invocation of an operation which "changes" the state.[18]

[18] Since we shall be expressing our formalisations in a pure, functional language, state changes are expressed by functions whose signature include state entity types both as arguments and as results.

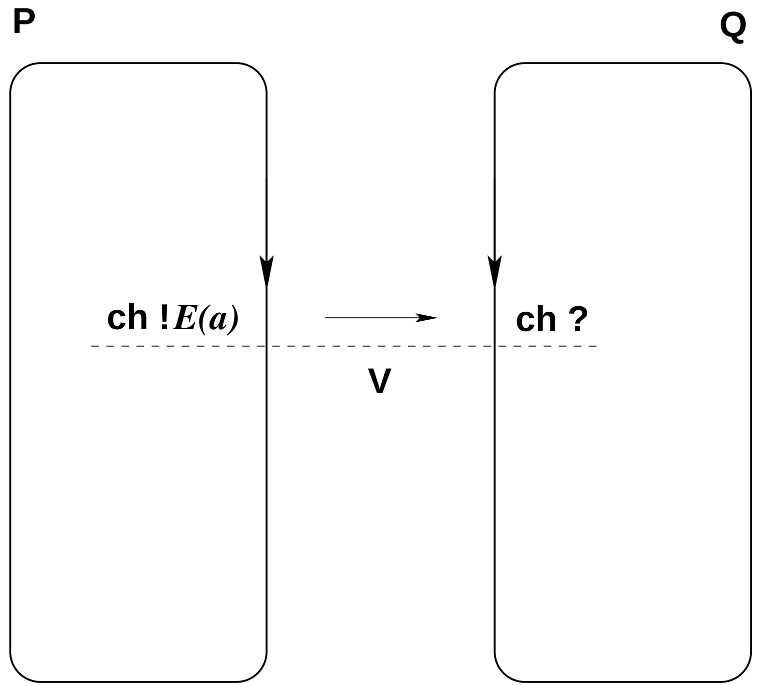
Definition 39 – Behaviours:

- *(i) By a* **behaviour** *we shall understand either a simple, sequential behaviour, or a simple parallel (or concurrent) behaviour, or a communicating behaviour.*
- *(ii) By a* **simple, sequential behaviour** *we shall understand a sequence of actions and events (the latter to be defined shortly).*
- *(iii) By a* **simple parallel (or concurrent) behaviour** *we shall understand a set of simple, sequential behaviours.*
- *(iv) By a* **communicating behaviour** *we shall understand a set of simple parallel behaviours which in addition (to being simple parallel behaviours) communicate messages between one-another.*

■

- We shall base our formal specification of behaviours on the use of `CSP` (Hoare's Communicating Sequential Processes).
- Other concurrency formalisms can, of course, be used:
  - Petri nets,
  - Message Sequence Charts (MSCs,
  - Statecharts,
  - or other.
- Communication, between two behaviours (`CSP` processes),
  - `P` and `Q` is in `CSP` expressed by
  - the `CSP` output
  - and input clauses: `ch!e`, respectively `ch?`
  - where `ch` designates a `CSP channel`.

type
 A, B, M
channel
 ch:M
value
 av:A, bv:B
 S = P(av) ∥ Q(bv)
 P: A → **out** ch **Unit**
 P(a) ≡ ... ch!$\mathcal{E}(a)$ ... P(a′)
 Q: B → **in** ch **Unit**
 Q(b) ≡ ... **let** v = ch? **in** ... Q(b′) **end**



where a, a' and b, b' designate process P, respectively process Q state values provided to process invocations (with a' and b' resulting from "calculations" not shown in the bodies of the definitions of P and Q. av and bv are initial entities. S is the overall system behaviour of two communicating behaviours operating in parallel ($\|$). M designates the type of the messages sent over channel ch.

We shall now show a duality between entities and behaviours.

- The background for this duality is the following:
  - In everyday parlance we speak of some domain phenomena
  - both as being entities
  - and as embodying behaviours.

Example 43 – Entities and Behaviours:

- A train is the composite entity of one or more engines (i.e., locomotives) and one or more passenger and/or freight cars.

type
  Train, Engine, PassCar, FreightCar
  Car = PassCar|FreightCar
value
  obs_Engine: Train $\rightarrow$ Engine
  obs_Carl: Train $\rightarrow$ Car*
axiom
  $\forall$ tr:Train $\cdot$ **card** obs_Carl(tr)>0

- A train is also the behaviour whose state include the time dependent train location and the states of these engines and cars and whose sequence of actions comprise the arrival and stop of the train at stations, the unloading and loading of passengers and freight at stations, the start-up and departure of trains from stations and the continuous movement, initially at accelerated speeds, then constant speed, finally at decelerating speeds along the rail track between stations — occasionally allowing for stops at track segment blocking signals. A train behaviour event could be that a cow presence of the track causes interrupt of scheduled train behaviour. Et cetera.

type
  T, Loc,
  TrainBehaviour = T $\overrightarrow{m}$ Train
channel
  net_channel (is_at_station|**Bool**)
value
  obs_Loc: Train → Loc
  train: Train → T → **out**,**in** net_channel → **Unit**
  train(tr)(t) ≡
    if is_at_Station(obs_Loc(tr))
      then
        let (tr′,t′) = stop_train(tr)(t);
        let (tr″,t″) = load_unload_passengers_and_freight(tr′)(t′) **in**
        let (tr‴,t‴) = move_train(tr″)(t″) **in**
            [**assert:** ∼is_at_Station(obs_Loc(tr‴))]
        train(eσ″,(el′,pfl′),loc′)(t′) **end end end**
      else
        let (tr′,t′) = move_train(tr)(t) **in** train(tr′)(t′) **end**
    end

- Here we leave undefined a number of auxiliary functions:

value
  is_at_Station: Loc → **out**,**in** net_channel **Bool**
  stop_train: Train → T → Train × T
  load_unload_passengers_and_freight: Train → T → Train × T
  move_train: Train → T → Train × T

- The above "model" of a train behaviour is really not of the kind (of models) that we shall eventually seek.
- The predicate is_at_Station communicates with the net behaviour (not shown).
- The function load_unload_passengers_and_freight communicates with the net behaviour (platforms, marshalling yards, etc., not shown).
- It is a rough sketch meant only to illustrate the process behaviour.

■

- Example 42 illustrated a very general class of mereologies.
- The next example,
  - Example 44
  - will show how the duality between entities and behaviours
  - can be "drawn" to an ultimate conclusion !

Example 44 – Mereology: Parts and Wholes (2):

- The model of mereology (Slides 280–293) given earlier focused on the following simple entities
  - the assemblies,
  - the units and
  - the connectors.
- To assemblies and units we associate CSP processes, and
- to connectors we associate CSP channels,
- one-by-one.

- The connectors form the mereological attributes of the model.
- To each connection we associate a CSP channel,
  - it is “anchored” in two parts:
  - if a part is a unit then in “its corresponding” unit process, and
  - if a part is an assembly then in “its corresponding” assembly process.
- From a system assembly we can extract all connector identifiers.
- They become indexes into an array of channels.
  - Each of the connector channel identifiers is mentioned
  - in exactly one unit or one assembly process.

- From a system which is an assembly,
  - we can extract all the connector identifiers
  - as well as all the internal connector identifiers.
- They become indexes into an array of channels.
  - Each of the external connector channels is mentioned in exactly one unit or one assembly process;
  - and each of these internal connection channels is a mentioned in exactly two unit or assembly processes.
- The xtr∀KIs(s) below was defined in Example 42 (Slide 289).

value
  s:S
  kis:KI-**set** = xtr∀KIs(s)
type
  ChMap = AUI $\overrightarrow{m}$ KI-**set**
value
  cm:ChMap = [obs_AUI(p)↦obs_KIs(p)|p:P•p ∈ xtr_Ps(s)]
channel
  ch[i|i:KI•i ∈ kis] MSG

value
  system: S → **Process**
  system(s) ≡ assembly(s)

value

$$\text{assembly: a:A}\rightarrow\textbf{in,out}\ \{\text{ch}[\,\text{cm(i)}\,]|\text{i:KI}\cdot\text{i} \in \text{cm(obs\_AUI(a))}\}\ \textbf{process}$$
$$\text{assembly(a)} \equiv$$
$$\quad \mathcal{M}_{\mathcal{A}}\text{(a)(obs\_A}\Sigma\text{(a))} \parallel$$
$$\quad \parallel \{\text{assembly(a}'\text{)}|\text{a}'\text{:A}\cdot\text{a}' \in \text{obs\_Ps(a)}\} \parallel$$
$$\quad \parallel \{\text{unit(u)}|\text{u:U}\cdot\text{u} \in \text{obs\_Ps(a)}\}$$
$$\text{obs\_A}\Sigma\text{: A} \rightarrow \text{A}\Sigma$$

$$\mathcal{M}_{\mathcal{A}}\text{: a:A}\rightarrow\text{A}\Sigma\rightarrow\textbf{in,out}\ \{\text{ch}[\,\text{cm(i)}\,]|\text{i:KI}\cdot\text{i} \in \text{cm(obs\_AUI(a))}\}\ \textbf{process}$$
$$\mathcal{M}_{\mathcal{A}}\text{(a)(a}\sigma\text{)} \equiv \mathcal{M}_{\mathcal{A}}\text{(a)(}\mathcal{A}\mathcal{F}\text{(a)(a}\sigma\text{))}$$

$$\text{A}\mathcal{F}\text{: a:A} \rightarrow \text{A}\Sigma \rightarrow \textbf{in,out}\ \{\text{ch}[\,\text{em(i)}\,]|\text{i:KI}\cdot\text{i} \in \text{cm(obs\_AUI(a))}\}\times\text{A}\Sigma$$

- The unit process
  - is defined in terms of the recursive meaning function $\mathcal{M}_{\mathcal{U}}$ function
  - which requires access to all the same channels as the unit process.

value

unit: u:U $\rightarrow$ **in**,**out** $\{$ch$[$cm(i)$]|$i:KI•i $\in$ cm(obs_UI(u))$\}$ **process**
unit(u) $\equiv$ $\mathcal{M}_{\mathcal{U}}$(u)(obs_U$\Sigma$(u))
obs_UΣ: U $\rightarrow$ UΣ

$\mathcal{M}_{\mathcal{U}}$: u:U $\rightarrow$ UΣ $\rightarrow$ **in**,**out** $\{$ch$[$cm(i)$]|$i:KI•i $\in$ cm(obs_UI(u))$\}$ **process**
$\mathcal{M}_{\mathcal{U}}$(u)(u$\sigma$) $\equiv$ $\mathcal{M}_{\mathcal{U}}$(u)($U\mathcal{F}$(u)(u$\sigma$))

U$\mathcal{F}$: U $\rightarrow$ UΣ $\rightarrow$ **in**,**out** $\{$ch$[$em(i)$]|$i:KI $\cdot$ i $\in$ cm(obs_AUI(u))$\}$ UΣ

- The meaning processes
  - $\mathcal{M}_{\mathcal{U}}$ and
  - $\mathcal{M}_{\mathcal{A}}$

  are generic.
  - Their sôle purpose is to provide a never ending recursions.
  - "In-between" they "makes use" of
    * assembly, respectively
    * unit

    specific functions
  - here symbolised by
    * $A\mathcal{F}$ and
    * $U\mathcal{F}$.

- The assembly function "first" "functions" as a compiler.
- The 'compiler' translates an assembly structure into three process expressions:
  - the $\mathcal{M}_{\mathcal{A}}(a)(a\sigma, a\rho)$ invocation,
  - the parallel composition of assembly processes, a', one for each sub-assembly of a, and
  - the parallel composition of unit processes, one for each unit of assembly a —
  - with these three process expressions "being put in parallel".
  - The recursion in assembly ends when a sub-...-assembly consists of no sub-sub-...-assemblies.
- Then the compiling task ends and the many generated $\mathcal{M}_{\mathcal{A}}(a)(a\sigma, a\rho)$ and $\mathcal{M}_{\mathcal{U}}(u)(u\sigma, u\rho)$ process expressions are invoked.

- We can refine the meaning of connectors.
  - Each connector, so far, was modelled by a CSP channel.
    * CSP channels serve both as a synchronisation and as a communication medium.
  - We now suggest to model it by a process.
    * A channel process can be thought of as having four channels and a buffering process.
    * Connector, κ:K, may connect parts π_i, π_j.
    * The four channels could be thought of as indexed by (κ, π_i), (π_i, κ), $(\kappa, \pi_j$ and (π_j, κ).
    * The process buffer could, depending on parts p_i, p_j, be either queues, sets, bags, stacks, or other.

This ends Example 44 ■

- The duality between simple entities and behaviours
  - has the attributes of atomic as well as of composite entities
  - become the state in which the entity behaviours evolve.
- Whereas — in principle — the mereology of how sub-entities compose into entities
  - are modelled as in Example 42, namely in terms of sorts, observer functions and axioms over unique identifiers of simple entities,
  - their attributes are usually modelled in a more model-oriented way, in terms of mathematical sets, Cartesians, sequences and maps.

6.4. Functions and Events

We shall consider events to be special cases of function invocations.

6.4.1. Functions

Definition 40 – **Function:** *By a* ***function*** *we shall understand something (a* ***functional entity****) which when* ***applied*** *to an entity, which we shall call an* ***argument*** *of the function,* ***yields*** *a* ***result*** *which is also an entity.* ■

- We shall refer to the application of a function to an argument as an **invocation**.
- Functions are characterised by the **function signature** or just **signature** and by their **function definition**.
- We show a number of function (and process) signatures:

type
  A, B, M1, M2

channel
  chi:MI, cho:MO

The above type definitions and channel declarations are used below:

value
  f0: A → B
  f1: **Unit** → B
  f2: **Unit** → **Unit**
  f3: A → **Unit**
  f4: A → **in** chi B
  f5: A → **in** chi **out** cho B
  f6: A → **out** cho B
  f7: A → **in** chi **Unit**
  f8: A → **in** chi **out** cho **Unit**
  f9: A → **out** cho **Unit**

- f0 designates a ("pure, applicative") function from A into B.
- f1 designates a constant function: takes no argument, i.e., invocation is expressed by f1(), but yields a (constant) value in B.
- f2 designates a process (i.e., a process function, one that never terminates). Invocation is expressed by f2(),
- f3 designates a process, accepting arguments in A and otherwise never terminating.
- f4 designates a function, accepting arguments in A and inputs, of type MI, on channel chi and otherwise terminating yielding a value of type B.
- f5 designates a function, accepting arguments in A, and inputs, of type MI, on channel chi, offers outputs, of type MO, on channel cho, and otherwise terminating yielding a value of type B.

- f6 designates a function, accepting arguments in A, offers outputs, of type MO, on channel cho, and otherwise terminating yielding a value of type B.
- f7 designates a function, accepting arguments in A, and inputs, of type MI, on channel chi and otherwise never terminating.
- f8 f7 designates a function, accepting arguments in A, inputs, of type MI, on channel chi, offers outputs, of type MO, on channel cho, and otherwise never terminating.
- f9 designates a function, accepting arguments in A, offers outputs, of type MO, on channel cho, and otherwise never terminating.

- For functions f4–f9 you may replace A by **Unit** to obtain further signatures.
- Thus the literal **Unit**,
  - to the left of the → designates that no input is to be provided,
  - and to the right of the → designates a never ending process.

- We show a number of function signatures exemplifying "Currying":

type
  A, B, C, D
value
  $f': A \times B \times C \to D$
  $f'': A \times B \to C \to D$
  $f''': A \to B \to C \to D$

invocation examples:
  $f'(a,b,c)$
  $f''(a,b)(c)$
  $f'''(a)(b)(c)$

We show two forms of function definitions:

type
  A, B
value
  f: $A \stackrel{\sim}{\to} B$
  f(a) **as** b
  pre $\mathcal{P}$(a)
  post $\mathcal{Q}$(a,b)

type
  A, B
value
  g: $A \to B \,[A \stackrel{\sim}{\to} B]$
  g(a) $\equiv \mathcal{E}$(a)
  [**pre** $\mathcal{P}$(a)]

- f is defined by a pair of **pre/post** conditions expressed by the predicates $\mathcal{P}$(a) and $\mathcal{Q}$(a,b) respectively.
- The clause '**as** b' expresses that the result is named b g and allows $\mathcal{Q}$ to refer to the result.
- g is defined by an explicit ("abstract algorithmic") expression $\mathcal{E}$(a).
- To avoid cluttering $\mathcal{E}$(a) with basically a test on $\mathcal{P}$(a) (should g not be total on A, that test is brought as a **pre** condition.

6.4.2. Events

Definition 41 – Event:

- *By an event we shall generally understand a state change that satisfies a given predicate:*

type
  Σ
value
  $event_i$: $\Sigma \times \Sigma \rightarrow$ **Bool**

- *Given two states:* σ, σ',
- *if* $\mathsf{event}_i(\sigma,\sigma')$ *holds*
- *then we say that event* event_i *has occurred.*

■

- This definition of an event is much too general.
- Of course, the domain (or requirements or software design) engineer is the one who decides which **events** to describe.
- But we shall accept it on formal grounds.
- More pragmatically we shall introduces the notions of
  - **internal event** and
  - **external event**.
- Most actions cause events — and they are all internal events.
- And most of these internal events are (usually) "uninteresting".

- A few internal events are interesting,
  - that is, cause state changes
  - "over-and-above" those primarily intended by the action.

Example 45 – **Interesting Internal Events:** Examples of what we would term interesting internal events are:

- *Banking:* A bank changes its interest rates;
- *Train Traffic:* a train is cancelled, etc.;
- *Oil Pipeline:* a pipeline runs dry of oil (due, for example, to valve and pump settings); and
- *Health Care:* a patient is give a wrong medicine (a form of medical malpractice).

■

- External events are events cause by "functions" beyond "our" control.
- That is, we postulate that some, maybe we could call it "demonic" function caused an event.

Example 46 – **External Events:** Examples of external events are:

- *Banking:* a major debitor defaults on a loan;
- *Train Traffic:* a train runes off the tracks;
- *Oil Pipeline:* a pipeline bursts; and
- *Health Care:* a patient dies.

■

- Internal events are typically modelled by providing the usual function, that is, action definitions with a suitable case distinction based on the **event**$_i$ predicate.

value
  function: A → Σ → Σ × B
  function(a)(σ) ≡
    let (σ′,b) = action(a)(σ) **in**
    if event$_i$(σ,σ′)
      then cope_with_internal_event(a)(σ,σ′)
      else (σ′,b) **end end**
  action: A → Σ → Σ × B
  cope_with_internal_event: A → (Σ×Σ) → Σ × B

- We may model external events as inputs on channels that can thus be said to "originate" in the environments — but for which functions definitions set aside an alternative choice of accepting such inputs from the environment.

type
  A, B, Σ, Event
channel
  x_ch:Event
value
  function: $A \rightarrow \Sigma \rightarrow$ **in** x_ch Σ B
  function(a)(σ) $\equiv$
    action(a)(σ)
   $\bigsqcap$
    let event = x_ch ? **in** cope_with_external_event(a)(event)(σ) **end**
  action: $A \rightarrow \Sigma \rightarrow \Sigma \times B$
  cope_with_external_event: $A \rightarrow Event \rightarrow \Sigma \rightarrow \Sigma \times B$

6.5. On Descriptions

- The discussion of this section amounts to establishing
  - a meta-theory of domains,
  - that is, a theory of the abstract, conceptual laws of describing domains
  - in contrast to a theory of any one specific domain,
  - that is, a theory of the concrete, physical and human laws of the described domain.
- In our discussion we will rely on understanding specifically referenced examples.
  - This understanding might very well be improved
  - as a result of understanding the message of this section.

6.5.1. What Is It that We Describe ?

- What is it that our descriptions denote ?[19]
- The answer is: the "things" that the nouns of our description "points to" are the actual things, "out there", in the domain.
  - The denoted individuals are not "figs of our imagination"[20].
  - They are 'real', 'actual', can be pointed to, seen, heard, touched, smelled, or otherwise measured by physical, including chemical/-physical apparatus(es) !
- Therefore there can be no "identical" copies.
  - If two sensed or measured phenomena are "equal"
  - then they are the same phenomenon.

[19] This question is just another way of expressing the question of the title of this subsection (i.e., Sect.).

[20] I apologize to more philosophically inclined readers: ours is not a discourse on ontology in the philosophical sense: *What may exists ?* etcetera. Our setting is computing science and software engineering — so we have no qualms about postulating that what I can sense, every person in full control of all her senses can sense and in an identical way !

6.5.2. Phenomena Identification

- We have in various examples,
  - as from Example 10 on Slide 62
  - introduced the abstract concept of unique identifiers:
    - of hubs and links (Example 10 on Slide 62),
    - of parts (assemblies and units) (Example 42 on Slide 280),
    - and in many later examples.
- These unique identifications are, in a sense, a mere technicality.
  - We need the unique identifications when we wish to express mereological properties such a
    - *"part of"*,
    - *"next to"*,
    - *"connected to"*,
    - etcetera.

- Therefore,
  - if two sensed or measured and described phenomena are “equal”,
  - except for their postulated unique identification,
  - then they are still the same phenomenon,
  - and there is a problem of description.
- We next turn to such ’problems of description’.

6.5.3. Problems of Description

- We can illustrate a number of 'problems of description'.
  - (i) Unique identifications:
    - two hubs that are claimed to have distinct hub identifiers,
    - but have identical values for the attribute of spatial location
    - must be the same hub, i.e., have identical hub identifier;
  - (ii) Observability:
    - if from a hub, we can observe a link,
    - and, vice versa, from a link we can observe its connected hubs,
    - and not merely their identifiers, but the 'real' phenomena,
    - then we can argue that
      - we can observe, from any hub all hubs and all links,
      - and that is counter to our intuition of how we observe.

6.5.4. Observability

- That is: we reject the dogma:
  - *"The Universe in a Single Atom"*[21],
  - that is, that all can be observed from a single "position".
- But this rejection begs the issue
  - *"What Do We Mean by Observability ?"*
- In the following we shall treat observability of
  - simple entities and their attributes
  - on par, that is, not make a distinction.
- Later we shall make a distinction between
  - observing simple entities and observing attributes.

[21] Also the title of a book by HH The 14th Dalai Lama: 'The Universe in a Single Atom': Reason and Faith

6.5.4.1. Simple Observability

- The simple part of an answer to this question,
  - and, mind you, it is an answer that is based on our computing science and software engineering viewpoint,
  - concerns that which can be physically observed of any domain phenomenon "itself",
  - that is, of the phenomenon observed in isolation.
- That part of the answer goes like this:
  - of a physically manifested phenomenon
  - we can observe all that can be physically sensed:
    - seen,
    - heard,
    - smelled,
    - tasted, and
    - touched;

    as well as
  - measured by physical/chemical apparatus(es).

6.5.4.2. Not-so-Simple, Simple Entity Observability

- The not-so-simple part of an answer
  - focuses on simple entities and
  - concerns that which can be
    - physically observed
    - of the “immediate” mereology
    - of the simple entity “itself”,
  - that is,
    - of the parts
    - to which that simple entity
    - is connected.

- Here the answer is: One can observe immediate phenomenological and conceptual connections, that is,
  - the simple entity parts that are connected to the entity under review,
    - and by reference to their identity —
    - hence the need for the identity concept;

    and
  - similarly for operations, event and behaviours: which operations
    - directly invoke other operations,
    - directly cause events, and, in general,
    - directly participate in behaviours;
  - which events "trigger"
    - operations,
    - other events, and, in general,
    - directly participate in behaviours;
  - and which behaviours
    - synchronise and/or communicate with other behaviours.

6.5.5. On Denoting

- Yes, we do know that Bertrand Russel wrote a famous paper with this title [Rus05].
- But our intention here is less 'lofty', and, perhaps not !
- When, above, we write:
  - *the denoted individuals are not "figs of our imagination"* and
  - *they are 'real', 'actual', can be pointed to, seen, heard, touched, smelled, or otherwise measured by physical, including chemical/physical apparatus(es),*
  - then it is our intention that we express.
- We can make that claim as far as the informal narrative description is concerned.
- But when our description is formalised, then what ?
  - Our formal description language has a semantics.
  - That semantics ascribes to our formalisation some mathematical values, structures.
  - That is, our narrative is of the 'real thing',
  - and our formalisation is a model of the real thing.

- So there are two notions of 'denoting' at play here.
  - an informal one: the relation between the narrative description and the physical (incl. human) phenomena
  - and a formal one: the relation between the syntax of our formal description and its semantics, i.e., 'the model'.
- The two notions relate, but only informally:
  - enumerated lines of the narrative has been "syntactically",
  - that is informally, related to
  - "identically" numbered formula lines,
  - with the informal claim that
    - a numbered narrative line
    - "means" the same as the same-numbered formula line !

6.5.6. A Dichotomy

- Example 42 (Slides 280–293), we now claim,
  - has the narrative denote classes of real phenomena and
  - has the formalisation model the syntax and syntactic well-formedness of a large class of such real phenomena.
- Later, in Example 44, (Slides 305–313), we now claim,
  - has the (the same) narrative (as in Example 42) indicate conceptual semantic models of
  - with the formalisation explicitly designating classes of such semantics models.

- So the transition between the two examples, Example 42 and Example 44, signal a reasonably profound “shift”
  - from
    - (informally) designating actual phenomena and
    - (formally) denoting their algebraic structures,
  - to,
    - informally and formally,
    - referring to semantic models in terms of behaviours and states.
- There is no dichotomy here, just a shift of abstraction.

6.5.7. Suppression of Unique Identification

- When comparing, for example, two simple entities
  - one is comparing not only their attributes
  - but also, when the entities are composite, their sub-entities.
- Concerning unique identifiers of simple entities we have this to say:
  - We can decide to either include unique identifiers as an entity attribute,
  - or we can decide that such identifiers form a third kind of observable property of a simple entity
    - the two others being ("other") attributes — as we see fit to define and
    - the possible sub-entities of composite entities.

- Either way, we need to introduce a meta-linguistic operator[22], say

$$\mathcal{S}_I\text{: Simple\_observable\_entity\_value} \rightarrow \text{Anonymous\_simple\_entity\_value}$$

- The concept of an anonymous value is also meta-linguistic.
- The anonymous value is basically
  - "the same, i.e., "identical" value
  - as is the simple entity value (from which, through $\mathcal{S}_I$[23], it derives)
  - with the single exception that the simple entity value "possesses"
    * the unique identifier of the observable entity value and
    * the anonymous entity value does not.

[22] The operator $\mathcal{S}_I$ is meta-linguistic with respect to RSL: it is not part of RSL, but applies to RSL values.
[23] The $\mathcal{S}$ stands for "suppress" and the $\mathcal{I}$ for the suppressed unique identifier.

6.5.8. Laws of Domain Descriptions

6.5.8.1. Preliminaries

- When we wish to distinguish one simple entity phenomenon from another
- then we say that the two ("the one and the other") are distinct.
- To be distinct to us means that the two phenomena have distinct, that is, unique identifiers.
- Being simple entity phenomena, separately observable in the domain, means that their spatial (positional) properties are distinct.
- That is their anonymous values are distinct.
- Meta-linguistically, that is, going outside the RSL framework[24], we can "formalise" this:

[24]but staying within a proper mathematical framework — once we have understood the mathematical properties of S_I and proper RSL values and 'anonymous' values (which, by the way, are also RSL values)

type

A [A models a type of simple entity phenomena]

I[25] [I models the type of unique A identifiers]

value

$\mathrm{obs\_}I$: A $\rightarrow$ I

axiom

$\forall$ a,a′:A $\cdot$ $\mathrm{obs\_}I(\mathrm{a}) \neq \mathrm{obs\_}I(\mathrm{a}') \Rightarrow \mathcal{S}_I(\mathrm{a}) \neq \mathcal{S}_I(\mathrm{a}')$

[25] We have here emphasized I, the type name of the type of unique A identifiers. Elsewhere in this book we treat types of unique identifiers of different types of observable simple entities as "ordinary" RSL types. Perhaps we should have "singled" such unique identifier type names out with a special font ? Well, we'll leave it as is !

- The above applies to any kind of observable simple entity *phenomenon* A.
- It does not necessarily apply to simple entity *concepts*.
  - **Example:**
    - *Two uniquely identified timetables*
    - *may have their anonymous values*
    - *be the exact same value.* □

- Simple entity phenomena, in our ontology,
  - are closely tied to space/time "co-ordinates" —
  - with no two simple entity phenomena sharing overlapping space.
- Concepts are, in our ontology,
  - not so constrained,
  - that is, we allow "copies"
  - although uniquely named !
- That is, two seemingly distinct concepts
  - may be the same
  - when "stripped" of their unique names !

6.5.8.2. Some Domain Description Laws

- We shall just bring a few domain description laws here.
- Enough, we hope, to spur further research into 'laws of description'.

Domain Description Law 1 – Unique Identifiers:

- If two observable simple entities have the same unique identifier
- then they are the same simple entity.

■

- Any domain description must satisfy this law.
- The domain describer must, typically through axioms, secure that the domain description satisfy this law.
- Thus there is a *proof obligation* to be *dispensed*, namely that the unique identifier law holds of a domain description.

Domain Description Law 2 – Unique Phenomena:

- If two observable simple entities have different unique identifiers
- then their values, "stripped" of their unique identifiers are different.

■

- Any domain description must satisfy this law.
- The domain describer must, typically through axioms, secure that the domain description satisfy this law.
- Thus there is a *proof obligation* to be *dispensed*, namely that the unique phenomena law holds of a domain description.

Domain Description Law 3 – Space Phenomena Consistency:

- Two otherwise unique, and hence distinctly observable phenomena
- can, spatially, not overlap.

■

- We can express the *Space/Time Phenomena Consistency Law*
  - meta-linguistically,
  - yet in a proper mathematical manner:

type

E [E is the type name of a class of observable simple entity phenomena]

I [I is the type name of unique E identifiers]

$\mathcal{L}$ [$\mathcal{L}$ is the type name of E locations]

value

obs_I: E $\rightarrow$ I

obs_$\mathcal{L}$: E $\rightarrow$ $\mathcal{L}$

axiom

$\forall$ e,e′:E $\cdot$ obs_I(e)$\neq$obs_I(e′) $\Rightarrow$ obs_$\mathcal{L}$(e) $\sqcap$ obs_$\mathcal{L}$(e′) = $\emptyset$

- We can assume that this law always holds for
- otherwise unique, and hence distinctly observable phenomena.

Domain Description Law 4 – Space/Time Phenomena Consistency:

- If a simple entity (that has the location property),
- at time t is at location ℓ, and
- at time t' (larger than t) is at location ℓ' (different from ℓ),
- then it moves **monotonically** from ℓ to ℓ' during the interval (t, t').

■

- Specialisations of this law are, for example, that
- if the movement is of two simple entities, like two trains, along a single rail track and in the same direction,
- then where train s_i is in front of train s_j at time t,
- train s_j cannot be in front of train s_i at time t' (where $t' - t$ is some small time interval).

6.5.8.3. Discussion

- There are more domain description laws.
- And there are most likely laws that have yet to be "discovered" !
- Any set of laws must be proven consistent.
- And any domain description must be proven to adhere to these (and "the" other) laws.
- We decided to bring this selection of laws because they are a part of the emerging 'domain science'.
- Laws 3 on Slide 351 and 4 on the preceding slide are also mentioned, in some other form, in [Rus18-19].
- Are these domain description laws laws of the domain or of their descriptions, that is, are they domain laws ?
- We leave the reader to ponder on this !

End of Lecture 6

A Specification Ontology

Lecture 7

Domain Engineering: Opening Stages and Intrinsics

7. Domain Engineering

7.1. The Core Stages of Domain Engineering

- The core stages of domain engineering are those of modelling the following **domain facets**:
  - intrinsics,
  - support technologies,
  - management and organisation,
  - rules and regulations,
  - scripts (contracts and licenses) and
  - human behaviour

  of the domain.

- An important stage of domain engineering is that of
  - rough sketching
  - the business processes.
- This stage is "sandwiched" in-between the opening stages of
  - domain acquisition and
  - domain analysis.

- The decomposition of these core stages
  - into exactly these facet description stages is one of pragmatics.
  - Experience has shown that this decomposition into modelling stages leads to a suitable base for a final model.
- That is, the domain engineers may follow, more-or-less strictly
  - the facet stage sequence hinted at above
  - but the domain engineers may, very well, in the end,
  - present the final domain description
  - without clear delineations, in the description, between these facets.
- In other words,
  - the decomposition and the principles of each individual facet stage,
  - we think, provides a good set of guidelines for the domain engineers
  - on how to proceed.

- These core stages are
  - preceded by a number of opening stages and
  - succeeded by a number of closing stages.
- The opening and closing stages, except for the business process sketching stage,
  - are here considered less germane
  - to the proper understanding of the domain concept.

7.2. Business Processes

- The rough-sketching of business processes
  - shall serve as an "easiest", informal way of starting
  - the more systematic domain acquisition process.

Definition 42 – Business Process: *By a business process we understand*

- *the procedurally describable aspects, of one or more of the ways in which a business, an enterprise, a factory, etc.,*
- *conducts its yearly, quarterly, monthly, weekly and daily processes, that is, regularly occurring chores.*

- *The business processes may include strategic, tactical and operational management and work-flow planning and decision activities; and*
- *the administrative, and where applicable, the marketing, the research and development, the production planning and execution, the sales and the service (work-flow) activities — to name some.*

■

7.2.1. General Remarks

- A domain is often known to its stakeholders by the various actions they play in that domain.
- That is, the domain is known by the various sequences of entities, functions and events the stakeholders are exposed to, are performing and are influenced by.
- Such sequences are what we shall here understand as business processes.

- In our ongoing example, that of railway systems, informal examples of business processes are:
  - for a potential passenger to plan, buy tickets for, and undergo a journey.
  - For the driver of the locomotive the sequence of undergoing a briefing of the train journey plan, taking possession of the train, checking some basic properties of that train, negotiating its start, driving it down the line, obeying signals and the plan, and, finally entering the next station, stopping at a platform, and concluding a trip of the train journey — all that constitutes a business process.
  - For a train dispatcher, the monitoring and control of trains and signals during a work shift constitutes a business process.

- Describing domain intrinsics focuses on the very essentials of a domain.
- It can sometimes be a bit hard for a domain engineer, in collaboration with stakeholders, to decide which are the domain intrinsics.
- It can often help (the process of identifying the domain intrinsics) if one alternatively, or hand in hand analyses and describes what is known as the business processes.
- From a description of business processes one can then analyse which parts of such a description designate, i.e., are about or relate to, which facets.

7.2.2. Rough Sketching

- Initially the domain engineer proceeds by sketching.
- We use the term *rough sketching*[26] to emphasise that a rough sketch is just a preparatory document.
- A roughly sketched business process appears easier to make, that is, gets one started more easily.
- A rough sketch business process does not have to conform to specific principles about what to describe first,
  - whether to first describe phenomena or concepts;
  - whether to first describe discrete facts or continuous;
  - whether to first describe atomic facts or composite;
  - whether to first describe informally or formally;
  - etcetera.

[26] To both say 'rough' and 'sketching' may, perhaps be saying the same thing twice: sketches usually are rough.

Principle 1 – Describing Domain Business Process Facets:

- *As part of understanding any (at least human-made) domain it is important to delineate and describe its business processes.*
- *Initially that should preferably be done in the form of rough sketches.*
- *These rough sketches should — again initially — focus on identifiable simple entities, functions, events and behaviours.*
- *Naturally, being business processes, identification of behaviours comes first.*
- *Then be prepared to rework these descriptions as the modelling of domain facets starts in earnest.*

■

- Roughly sketched business processes help the domain engineer in the more general domain acquisition effort.
  - Domain stakeholders can be asked to sketch the business processes they are part of.
  - The domain engineer, interacting with the domain stakeholder can clarify open points about a sketched business process.
  - And the domain engineer can elicit facts about the domain as inspired by someone else's sketch.

7.2.3. Examples (I)

Example 47 – A Business Plan Business Process:

- The board of any company instructs its chief executive officer (CEO) to formulate revised business plans.[27]
- Briefly, a business plan is a plan for how the company — strategically, tactically and, to some extent, operationally — wishes to conduct its business: what it strives for, product-wise, image-wise, market-share-wise, financially, etc.
- The CEO develops a business plan in consultation with executive layers of (i.e., with strategic) management.
- Strategic management (in-between) discusses the plan (which the CEO wishes to submit to the Board) with tactical management, etc.
- Once generally agreed upon, the CEO submits the plan to the Board.

■

[27] A business plan is not the same as a description of the business' processes.

Example 48 – A Purchase Regulation Business Process:

- In our "example company", purchase of equipment must adhere to the following — roughly sketched — process:
- Once the need for acquisition of one or more units of a certain equipment, or a related set of equipment, has been identified,
- the staff most relevant to take responsibility for the use of this equipment issues a purchase inquiry request.
- The purchase inquiry request is sent to the purchasing department.
- The purchasing department investigates the market and reports back with a purchase inquiry report containing facts about possible equipment choices, prices, and their purchase (i.e., payment), delivery, service and guarantee conditions.

- The person who issued the purchase inquiry request may now proceed to issue a purchase request order, attach the purchase inquiry report and
- send this to the relevant budget controlling manager for acceptance.
- If purchase is approved then the purchasing department is instructed to issue, to the chosen supplier, a purchase request order.
- Once the supplier delivers the ordered equipment, the purchasing department inspects the delivery and issues an equipment inspection report.
- An invoice from the supplier for the above-mentioned equipment is only paid if the equipment inspection report recommends to do so.
- Otherwise the delivered equipment is returned to the supplier.

■

Example 49 – A Comprehensive Set of Administrative Business Processes:

- The University of California at Irvine (UCI), had their Administrative and Business Services department suggest, as a learning example, the description of a number of business processes.
- The "learning" had to do, actually, with business process re-engineering (BPR).
- So we really should bring the below example into the future section on BPR!
- We quote from their home Web page:

1. *Human Resources:*

- "Examine the hiring business process of the University, including the applicant process.
- Special emphasis should be given to simplifying the process, identifying those parts where there is no value added — i.e., where those parts of the process which one considers *simplifying "away"* add no value.
- Increase speed of response to applicant and units, and reduce process costs while achieving high quality."

2. *Renovation:*

- "Review the campus' remodelling and alterations business process,
- and develop recommendations to improve Facilities Management services to other departments for small projects (under $50,000) and minor capital projects (up to $250,000).
- Special emphasis should be given to
  - simplifying the process;
  - identifying those parts where there is no value added to the customer's product;
  - to increase speed and flexibility of response;
  - and to reduce process costs while achieving high quality."

3. *Procurement:*

- "Review the campus procurement business process and develop recommendations/solutions for process improvement.
- The redesigned process should provide
  - "hassle-free" purchasing,
  - give a quick response time to the purchaser,
  - be economical in terms of all costs,
  - be reasonably error-free and
  - be compliant with (US) Federal procurement standards."

4. *Travel:*

- “Study the travel business process
  - from the stage when a staff member identifies the need to travel
  - to the time when reimbursement is received.
- Analyze and redesign the process through a six step program based on the following business process improvement (BPI) principles:
  - (i) simplify the process,
  - (ii) identify those parts where there is no value added to the customer, increase
  - (iii) speed and
  - (iv) flexibility of response,
  - (v) improve clarity for responsibilities and
  - (vi) reduce process costs while meeting customer expectations from travel services.
- The redesign should reflect
  - customer needs,
  - service,
  - economy of operation and
  - be in compliance with applicable regulations.”

5. *Accounts payable:*

- “Redesign the accounts payable business process to meet the following functional objectives (in addition to BPI measures):
  - Payment for goods and services must assure that vendors receive remittance in a timely manner for all goods and services provided to the company.
  - Significantly improve the operation’s ability to serve company customers while maintaining financial solvency and adequate internal controls.”

6. *Parking:*
   - "Review how parking permits
     - are sold to customers and staff
     - with the intent of omitting unnecessary steps and redundant data collection.
   - The redesigned process should achieve
     - a dramatic reduction in time spent by people standing in line to purchase a permit, and
     - reduce administrative time (and cost) in recording and tracking permit sales."

- Please observe that the above examples illustrate requests for possible business process re-engineering —
- but that they also give rough-sketch glimpses of underlying business processes.

■

7.2.4. Methodology

Definition 43 – **Business Process Engineering:** *By business process engineering we understand*

- *the identification of which business processes should be subject to precise description,*
- *describing these and securing their general adoption (acceptance) in the business, and*
- *enacting these business process descriptions*

■

Principle 2 – Business Processes:

- *Human-made universes of discourse*
- *entail the concept of business processes.*
- *The principle of business processes states that the description of business processes is indispensable in any description of a human-made universe of discourse.*
- *The principle of business processes also states that describing these is not sufficient: all facets must be described*

■

Principle 3 – Describing Domain Business Process Facets:

- *As part of understanding any (at least human-made) domain it is important to delineate and describe its business processes.*
- *Initially that should preferably be done in the form of rough sketches.*
- *These rough sketches should — again initially — focus on identifiable entities, functions, events and behaviours.*
- *Naturally, being business processes, identification of behaviours comes first.*
- *Then be prepared to rework these descriptions as other facets are being described in depth*

Technique 1 – Business Processes: *The basic technique of describing a human-made universe of discourse involves:*

- *identification and description of a suitably comprehensive set of behaviours: the behaviours of interest and the environment;*
- *identification and description, for each behaviour, of the entities characteristic of this behaviour;*
- *identification and description, for each entity, of the functions that apply to entities, or from which entities are yielded;*
- *identification and description, for each behaviour, of the events that it shares — either with other specifically identified behaviours of interest, or with a further, abstract, environment*

■

Tool 1 – Business Processes: *Further techniques and the basic tools for describing business processes include:*

- *RSL/CSP definition of processes,*
  - *where one suitably defines their input/output signatures,*
  - *associated channel names and types,*
  - *and their process definition bodies;*

- *Petri nets;*
- *message and live sequence charts for the definition of interaction between behaviours;*
- *statecharts for the definition of highly complex, typically interwoven behaviours;*
- *and the usual, full complement of* RSL*'s type, function* value, *and* axiom *constructs and their abstract techniques for modelling entities and functions*

■

7.2.5. Examples (II)

- We rough-sketch a number of examples.
- In each example we start, according to the principles and techniques enunciated above, with
  - identifying behaviours, events, and hence
  - channels and the
  - type of entities communicated over channels, i.e. participating in events.
- Hence we shall emphasise, in these examples, the behaviour, or process diagrams.
- We leave it to other examples to present other aspects, so that their totality yields the principles, the techniques and the tools of domain description.

Example 50 – Air Traffic Business Processes:

- The main business process behaviours of an air traffic system are the following:
  - the aircraft,
  - the ground control towers,
  - the terminal control towers,
  - the area control centres and
  - the continental control centres
- (Fig. 4 on next to slide).

7.Domain Engineering 2.Business Processes 5.Examples (II) 0. 0

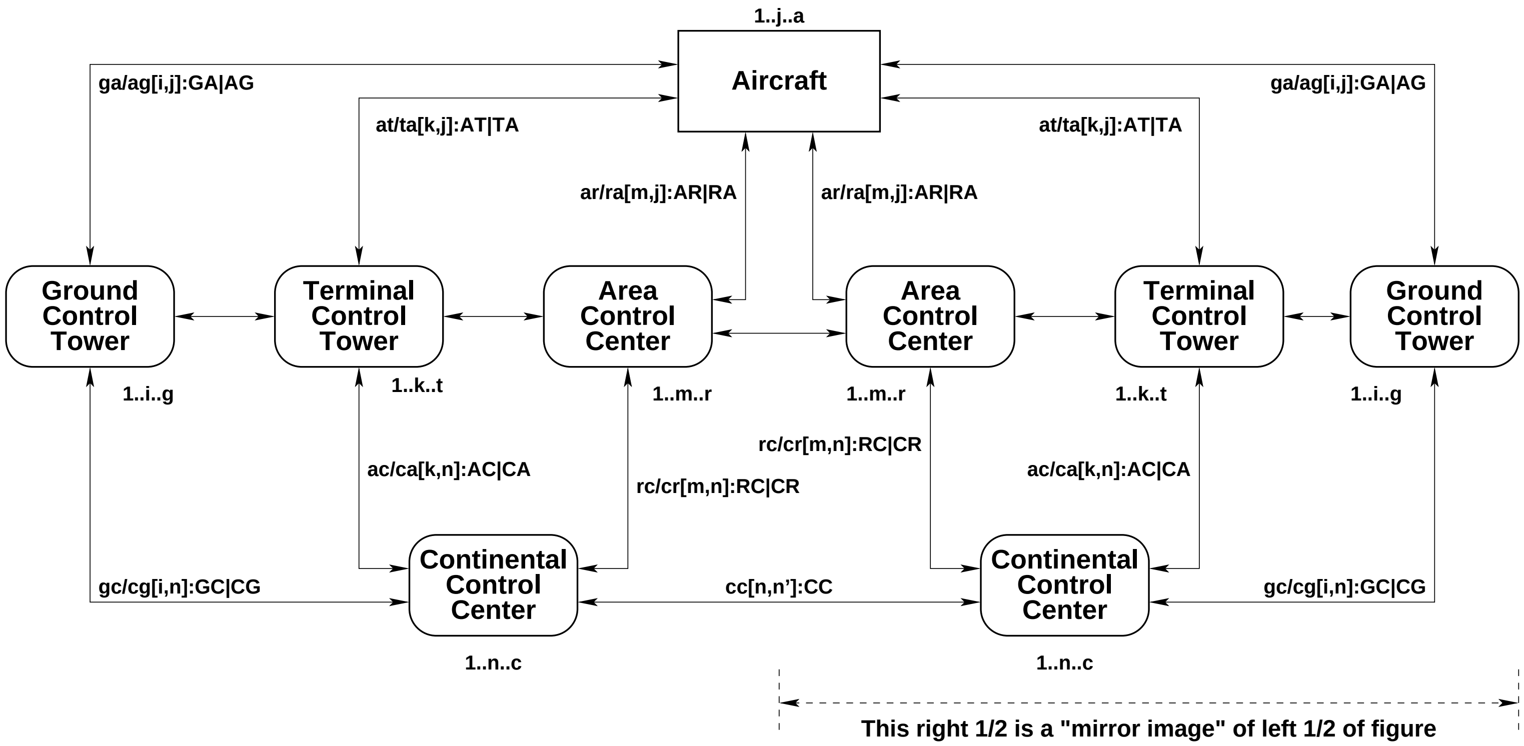


Figure 4: An air traffic behavioural system abstraction

- We describe each of these behaviours separately:
  - *Aircraft*
    - get permission from ground control towers to depart;
    - proceed to fly according to a flight plan (an entity);
    - keep in contact with area control centres along the route,
    - (upon approach) contacting terminal control towers from which they, simplifying, get permission to land; and
    - upon touchdown, changing over from terminal control tower to ground control tower guidance.

– The ground control towers,
  * on one hand, take over monitoring and control of landing aircraft from terminal control towers;
  * and, on the other hand, hand over monitoring and control of departing aircraft to area control centres.
  * Ground control towers, on behalf of a requesting aircraft, negotiate with destination ground control tower and (simplifying) with continental control centres when a departing aircraft can actually start in order to satisfy certain “slot” rules and regulations (as one business process).
  * Ground control towers, on behalf of the associated airport, assign gates to landing aircraft, and guide them from the spot of touchdown to that gate, etc. (as another business process).

– The terminal control towers
  - ∗ play their major rôle in handling aircraft approaching airports with intention to land.
  - ∗ They may direct these to temporarily wait in a holding area.
  - ∗ They — eventually — guide the aircraft down, usually "stringing" them into an ordered landing queue.
  - ∗ In doing this the terminal control towers take over the monitoring and control of landing aircraft from regional control centres,
  - ∗ and pass their monitoring and control on to the ground control towers.

– The area control centres handle aircraft flying over their territory:
  - ∗ taking over their monitoring and control
    - · either from ground control towers,
    - · or from neighbouring area control centres.
  - ∗ Area control centres shall help ensure smooth flight,
    - · that aircraft are allotted to appropriate air corridors, if and when needed (as one business process),
    - · and are otherwise kept informed of "neighbouring" aircraft and weather conditions en route (other business processes).
  - ∗ Area control centres hand over aircraft
    - · either to terminal control towers (as yet another business process),
    - · or to neighbouring area control centres (as yet another business process).

- The continental control centres
  - ∗ monitor and control, in collaboration with
    - · regional and ground control centres,
  - ∗ overall traffic in an area comprising several regional control centres (as a major business process),
  - ∗ and can thus monitor and control whether contracted (landing) slot allocations and schedules can be honoured,
  - ∗ and, if not, reschedule these (landing) slots (as another major business process).

- From the above rough sketches of behaviours the domain engineer then goes on to describe
  - types of messages (i.e., entities) between behaviours,
  - types of entities specific to the behaviours, and
  - the functions that apply to or yield those entities.

■

Example 51 – Freight Logistics Business Processes:

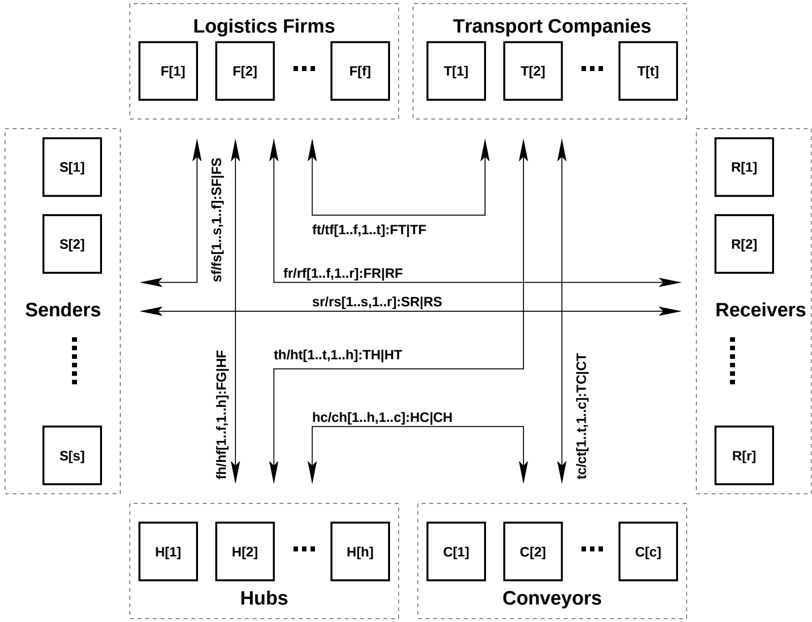


Figure 5: A freight logistics behavioural system abstraction

- The main business process behaviours of a freight logistics system are the following:
  - the senders of freight,
  - the logistics firms which plan and coordinate freight transport,
  - the transport companies on whose conveyors freight is being transported,
  - the hubs between which freight conveyors "ply their trade",
  - the conveyors themselves and
  - the receivers of freight
- A detailed description for each of the freight logistics business process behaviours listed above should now follow.
- We leave this as an exercise to the reader to complete.

Example 52 – Harbour Business Processes:

- The main business process behaviours of a harbour system are the following:
  - the ships who seek harbour to unload and load cargo at a harbour quay,
  - the harbour-master who allocates and schedules ships to quays,
  - the quays at which ships berth and unload and load cargo (to and from a container area) and
  - the container area which temporarily stores ("houses") containers.

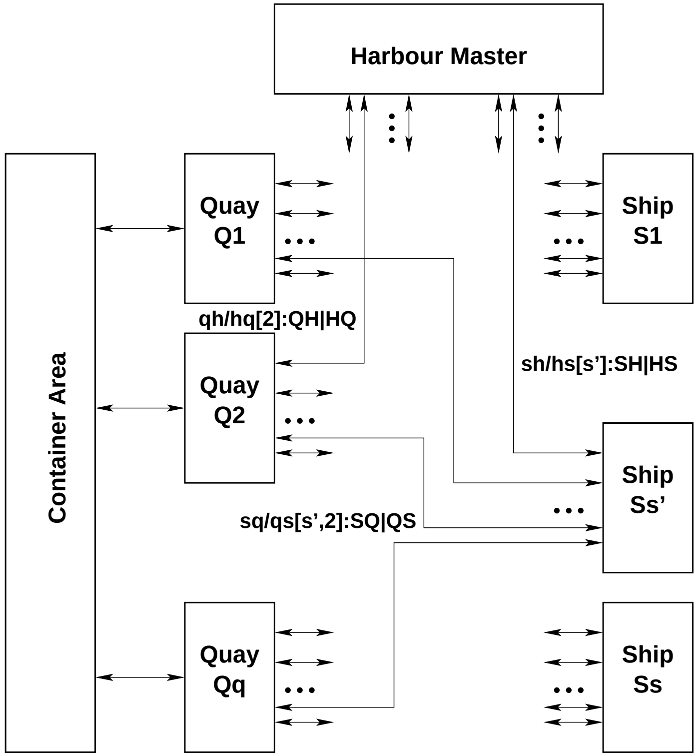


Figure 6: A harbour behavioural system abstraction

- There may be other parts of a harbour:
  - a holding area for ships
    - to wait before being allowed to properly enter the harbour and be berthed at a buoy or a quay,
    - or for ships to rest before proceeding; as well as
  - buoys at which ships may be anchored while
    - unloading and loading.
- We shall assume that the reader can properly complete an appropriate, realistic harbour domain.
- A detailed description for each of the harbour business process behaviours listed above should now follow.
- We leave this as an exercise to the reader to complete.

■

Example 53 – Financial Service Industry Business Processes:

- The main business process behaviours of a financial service system are the following:
  - clients,
  - banks,
  - securities instrument brokers and traders,
  - portfolio managers,
  - (the, or a, or several) stock exchange(s),
  - stock incorporated enterprises and
  - the financial service industry “watchdog”.
- We rough-sketch the behaviour of a number of business processes of the financial service industry.

7.Domain Engineering 2.Business Processes 5.Examples (II) 0. 0

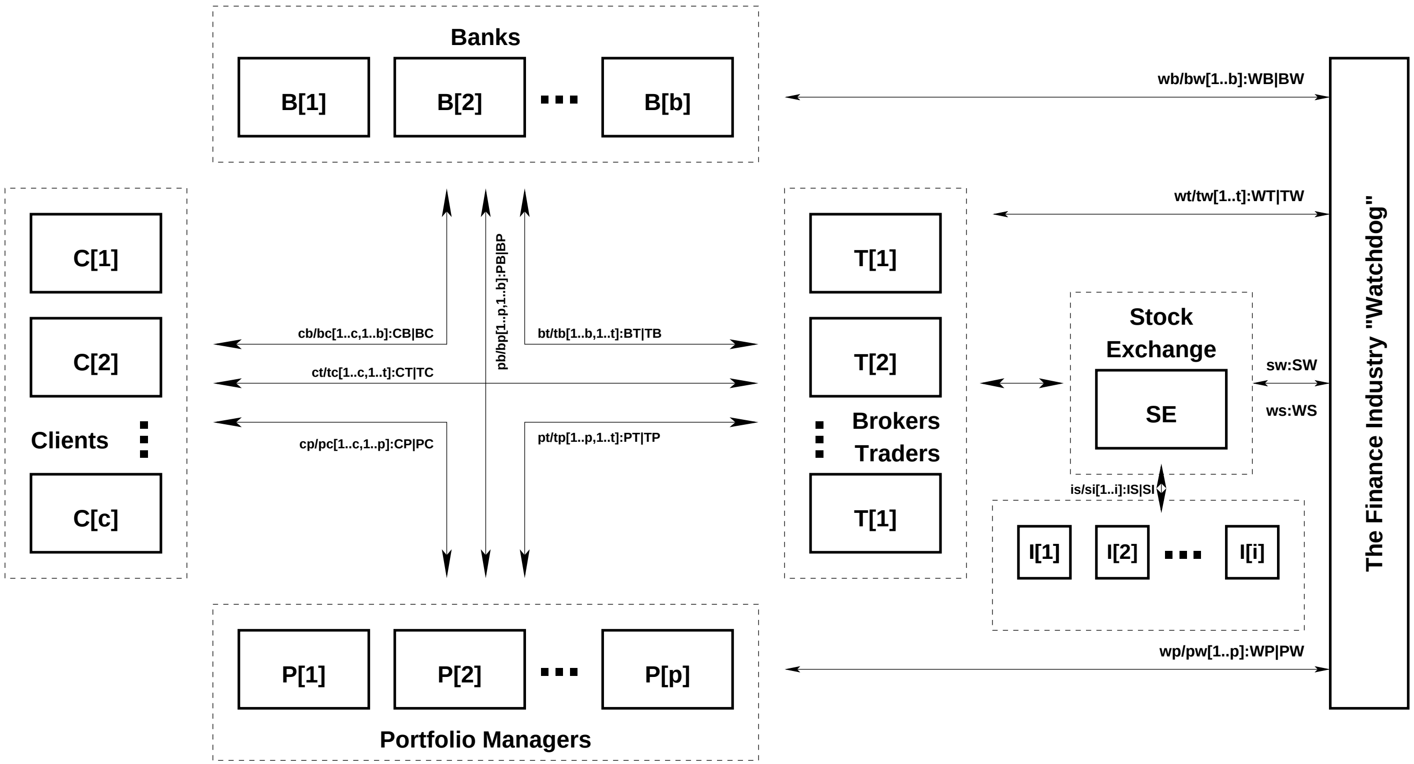


Figure 7: A financial behavioural system abstraction

– Clients engage in a number of business processes:
  * they open, deposit into, withdraw from, obtain statements about, transfer sums between and close demand/deposit, mortgage and other accounts;
  * they request brokers to buy or sell, or to withdraw buy/sell orders for securities instruments (bonds, stocks, futures, etc.); and
  * they arrange with portfolio managers to look after their bank and securities instrument assets, and occasionally they re-instruct portfolio managers in those respects.

– Banks engage with clients, portfolio managers, and brokers and traders in exchanges related to client transactions with banks, portfolio managers, and brokers and traders, as well as with these on their own behalf, as clients.
– Securities instrument brokers and traders engage with clients, portfolio managers and the stock exchange(s) in exchanges related to client transactions with brokers and traders, and, for traders, as well as with the stock exchange(s) on their own behalf, as clients.

– Portfolio managers engage with clients, banks, and brokers and traders in exchanges related to client portfolios.
– Stock exchanges engage with the financial service industry watchdog, with brokers and traders, and with the stock listed enterprises, reinforcing trading practices, possibly suspending trading of stocks of enterprises, etc.
– Stock incorporated enterprises engage with the stock exchange: They send reports, according to law, of possible major acquisitions, business developments, and quarterly and annual stockholder and other reports.
– The financial industry watchdog engages with banks, portfolio managers, brokers and traders and with the stock exchanges.

■

Example 54 – Railway and Train Business Processes:

- This example emphasises the simple entities that enable specific business processes.
  - The net of lines and stations, cf. Fig. 8 on following slide[A], made up from simple units, cf. Fig. 8 on next to slide[B], enable train traffic.
  - And train traffic gives rise to a number of business processes:
    * train journies (say, according to a timetable);
    * the selling of train tickets including reservation of seats;
    * the controlling of signals such that trains can move in and out of stations and along tracks between stations;
    * track and train maintenance;
    * staff rostering;
    * et cetera.

7.Domain Engineering 2.Business Processes 5.Examples (II) 0. 0

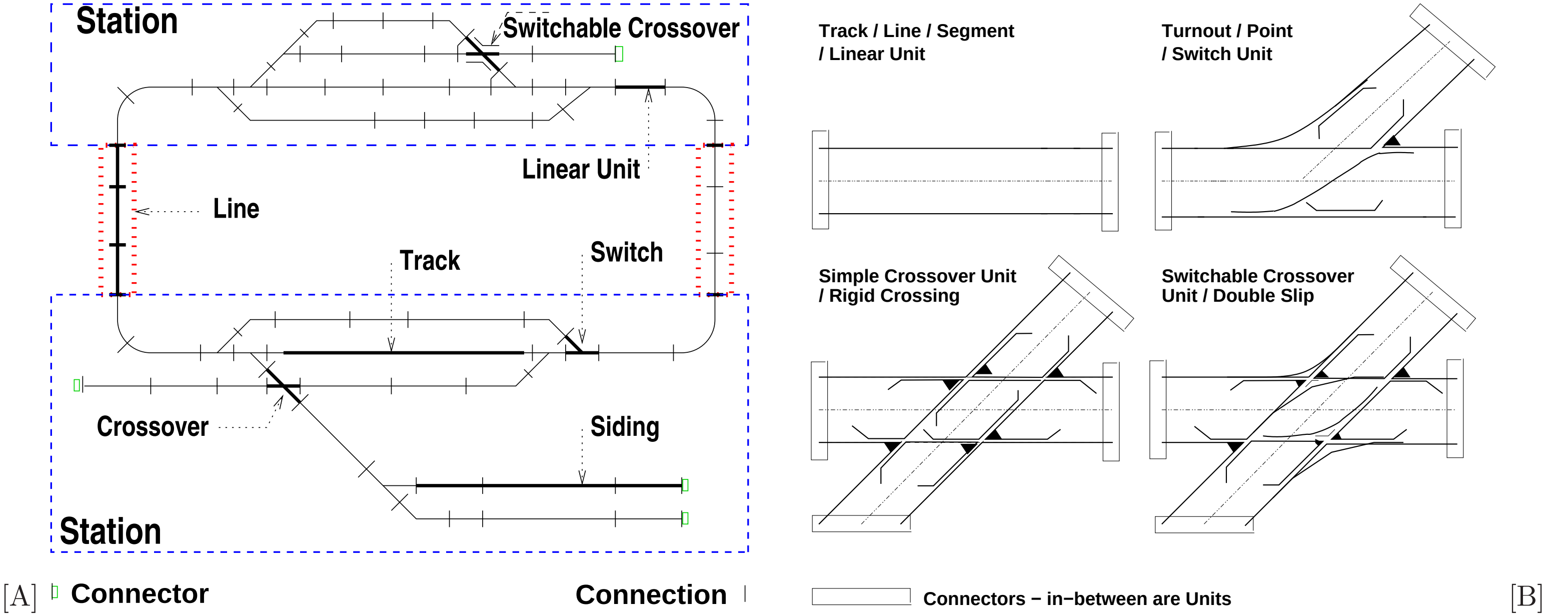


Figure 8: [A] A "model" railway net. An Assembly of four Assemblies:
Two stations and two lines; Lines here consist of linear rail units;
stations of all the kinds of units shown in Fig. 8[B].
There were 66 connections at last count and three "dangling" connectors

■

7.2.6. Discussion

- We shall take up the concept of business processes in in a later lecture where we introduce the important topic of 'business process re-engineering'.

7.3. Domain Intrinsics

Definition 44 – Domain Intrinsics: *By domain intrinsics we shall understand the very basics upon which a domain is based, the very essence of that domain, the simple entities, operations, events and behaviours without which none of the other facets of the domain can be described.* ■

The choice as to which simple entities, operations, events and behaviours "belong" to intrinsics is a pragmatic choice. It is taken, by the domain engineers, based on those persons' choice of abstraction and modelling techniques and tools. It is a choice that requires quite some experience, quite some years of training, including studying other persons' domain descriptions of similar or other domains.

Example 55 – An Oil Pipeline System: Statics of Pipelines

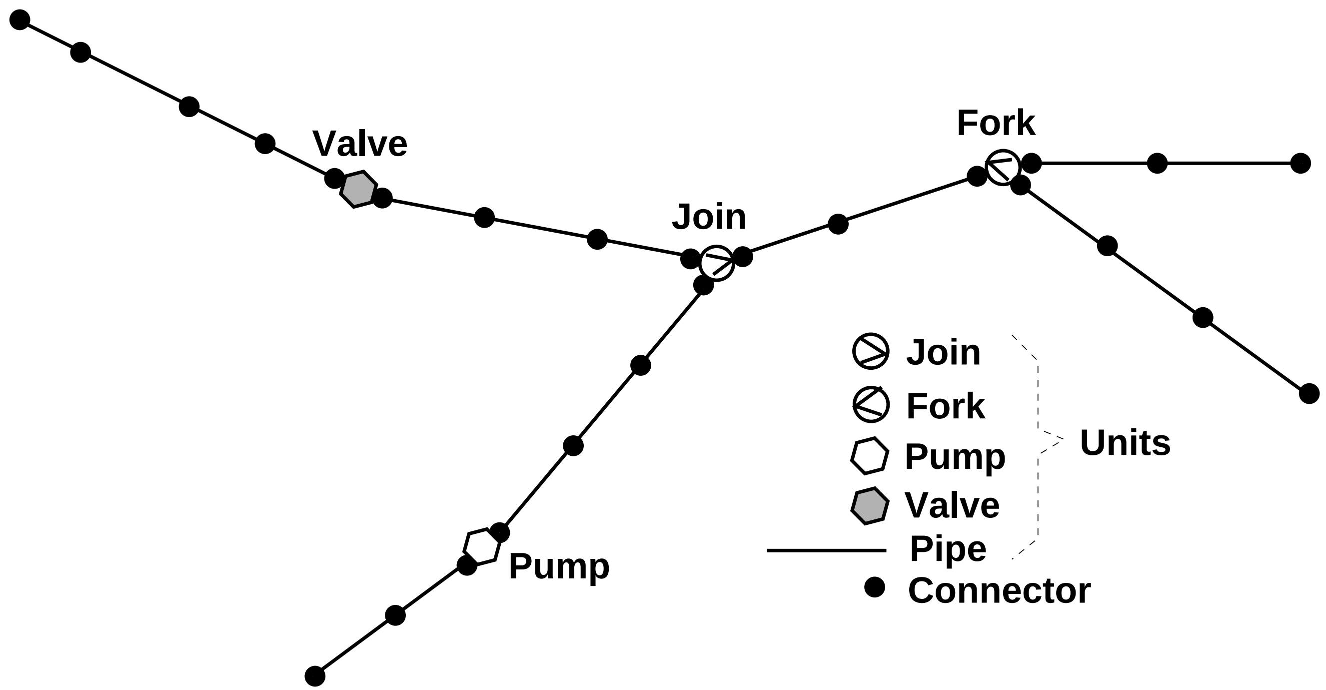


Figure 9: An oil pipeline system with 23 units (19 pipes) and 26 connectors

86. From an oil pipeline system, cf. Fig. 9 on previous slide, one can observe units and connectors.
87. Units are either pipe, or (flow, not extraction) pump, or valve, or join or fork units.
88. Units and connectors have unique identifiers.
89. From a connector one can observe the ordered pair of the identity of two (actual or pseudo) from-, respectively to-units that the connector connects.

type

86 OPLS, U, K

value

86 obs_Us: OPLS $\rightarrow$ U-**set**, obs_Ks: OPLS $\rightarrow$ K-**set**

87 is_PiU, is_PuU, is_VaU, is_JoU, is_FoU: U $\rightarrow$ **Bool** [mutually exclusive

type

88 UI, KI

value

88 obs_UI: U $\rightarrow$ UI, obs_KI: K $\rightarrow$ KI

axiom [uniqueness of identifiers]

88 $\forall$ opls:OPLS,u,u′:U,k,k′:K $\cdot$

$\{$u,u′$\}\subseteq$obs_Us(opls)$\wedge\{$k,k′$\}\subseteq$obs_Ks(opls)$\wedge$u$\neq$u′$\wedge$k$\neq$k′$\Rightarrow$

obs_UI(u)$\neq$obs_UI(u′)$\wedge$obs_KI(u)$\neq$obs_KI(u′)

value

89 obs_UIp: K $\rightarrow$ (UI|$\{$nil$\}$) $\times$ (UI|$\{$nil$\}$)

90. From a unit one can observe the identity of the connectors that provide input to, respectively that provide output from that unit — the two sets of identities are disjoint.
91. From a pipe, pump and valve units we can observe one input and one output connector identifier. From join units we can observe one output and two or more input connector identifiers, and from a fork unit the "reverse": one input and two or more output connector identifiers.
92. Given an oil pipeline system and a connector of that system, the observable ordered pair of actual identities of from- and to-units indeed do identify distinct units of that oil pipeline system.
93. No two connectors connect the same pair of units.

value

90 obs_iKls, obs_oKls: $U \rightarrow$ Kl-**set**

axiom

90 $\forall$ u:U $\cdot$ obs_iKls(u) $\cap$ obs_oKls(u) = {}

91 $\forall$ u:U $\cdot$

is_PiU(u)$\vee$is_VaU(u)$\vee$is_PuU(u) $\Rightarrow$ **card** obs_iKls(u)=1=**card** obs_oKls(u) $\wedge$

is_JoU(u) $\Rightarrow$ **card** obs_iKls(u)$\geq$2 $\wedge$ **card** obs_oKls(u)=1 $\wedge$

is_FoU(u) $\Rightarrow$ **card** obs_oKls(u)$\geq$2 $\wedge$ **card** obs_iKls(u)=1

92 $\forall$ opls:OPLS,k:K: k $\in$ obs_Ks(opls) $\Rightarrow$

let (fui,tui) = obs_UIp(k) **in**

fui$\neq$nil $\Rightarrow$ exist!u:U $\cdot$

u $\in$ obs_Us(opls)$\wedge$fui=obs_UI(u)$\wedge$obs_KI(k)$\in$ obs_oKls(u) $\wedge$

tui$\neq$nil $\Rightarrow$ exist!u:U $\cdot$

u $\in$ obs_Us(opls)$\wedge$tui=obs_UI(u)$\wedge$obs_KI(k)$\in$ obs_iKls(u) **end**

93 $\forall$ ols:OPLS,k,k$'$:K $\cdot$ {k,k$'$}$\subseteq$obs_Ks(opls)$\wedge$k$\neq$k$'\Rightarrow$

let ((fui,tui),(fui$'$,tui$'$)) = (obs_UIp(k),obs_UIp(k$'$)) **in**

nil$\neq$fui$\wedge$fui=fui$'\Rightarrow$tui$\neq$tui$'\wedge$ nil$\neq$tui$\wedge$tui=tui$'\Rightarrow$fui$\neq$fui$'$ **end**

94. An oil pipeline system thus has a set of input units, a set of output units and a set of routes from input to output units.

95. It follows from the above definitions that two two sets are non-empty.

value

94 $\mathrm{iUs,oUs}$: $\mathrm{OPLS} \to \mathrm{U}$-**set**

94 $\mathrm{iUs(opls)} \equiv$

$\quad \{\mathrm{u|u{:}U{\cdot}u} \in \mathrm{obs\_Us(opls)} \wedge$

$\quad\quad$ **let** $\mathrm{ikis} = \mathrm{obs\_iKIs(u)}$ **in**

$\quad\quad \sim\exists\ \mathrm{u'{:}U{\cdot}u'isin\ obs\_Us(opls)} \wedge \mathrm{ikis} \cap \mathrm{obs\_oKIs(u')} \neq \{\}$ **end**$\}$

94 $\mathrm{oUs(opls)} \equiv$

$\quad \{\mathrm{u|u{:}U{\cdot}u} \in \mathrm{obs\_Us(opls)} \wedge$

$\quad\quad$ **let** $\mathrm{okis} = \mathrm{obs\_oKIs(u)}$ **in**

$\quad\quad \sim\exists\ \mathrm{u'{:}U{\cdot}u'isin\ obs\_Us(opls)} \wedge \mathrm{okis} \cap \mathrm{obs\_iKIs(u')} \neq \{\}$ **end**$\}$

lemma:

95 $\forall\ \mathrm{opls{:}OPLS} \cdot \mathrm{iUs(opls)} \neq \{\} \wedge \mathrm{oUs(opls)} \neq \{\}$

96. We introduce the concept of a route being a special sequence of units.
97. **Basis Clause:** A unit, u, provides a route, $< u >$, of the oil pipeline system.
98. **Inductive Clause:** If r and r' are routes of the oil pipeline system
    (a) and the last unit, u of r, has an output connector identifier
    (b) which is an output connector identifier of the first unit, u' of r',

    then their concatenation is a route of the oil pipeline system.
99. **Extremal Clause:** Only such sequences of units are routes if that follows from a finite set of applications of clauses 97 and 98.

```
type
96   R′ = U*
96   R = {| r:R′ • wfR(r) |}
value
96   wfR: R′ → Bool
96   wfR(r) ≡
      case r of
97      ⟨u⟩ → true,
98      r′⌢r″ → wfR(r′) ∧ wfR(r″)
98(a)-98(b)  ∧ obs_oKIs(len r′)∩ obs_iKIS(hd r″)≠{}
      end
96   routes: U-set → R-set
96   routes(us) ≡
97     let urs = {⟨u⟩|u:U•u ∈ us} in
97     let rs = urs ∪
98          {r′⌢r″|r′,r″:R•{r′,r″}⊆rs ∧
98(a)-98(b)   obs_oKIs(len r′) = obs_iKIs(hd r″)}
99     rs end end
```

100. An oil pipeline system is well-formed, if — in addition to the earlier mentioned constraints —
    (a) there is a route from any input unit to some output unit,
    (b) there is a route leading to any output unit from some input unit and
    (c) the system of units and connectors “hang together”, that is, there is not a partition of these such that the sum of their routes equals the routes of the whole.

axiom

100 $\forall$ opls:OPLS $\cdot$

100(a) $\quad\forall$ iu:U $\cdot$ iu $\in$ obs_iUs(opls) $\Rightarrow$

100(a) $\quad\quad\exists$ ou:U $\cdot$ ou $\in$ obs_oUs(opls) $\wedge$

100(a) $\quad\quad\quad\exists$ r:R $\cdot$ r $\in$ routes(opls) $\wedge$

100(a) $\quad\quad\quad\quad$ **hd** r = iu $\wedge$ r(**len** r) = ou $\wedge$

100(b) $\quad\forall$ ou:U $\cdot$ ou $\in$ obs_oUs(opls) $\Rightarrow$

100(b) $\quad\quad\exists$ iu:U $\cdot$ iu $\in$ obs_iUs(opls) $\wedge$

100(b) $\quad\quad\quad\exists$ r:R $\cdot$ r $\in$ routes(opls) $\wedge$

100(b) $\quad\quad\quad\quad$ **hd** r = iu $\wedge$ r(**len** r) = ou $\wedge$

100(c) $\quad\sim\exists$ us,us$'$:U-**set** $\cdot$ us$\subset$obs_Us(opls) $\wedge$ us$'\subset$obs_Us(opls)

100(c) $\quad\quad\wedge$ us $\cap$ us$'$ = {} $\wedge$ us $\cup$ us$'$ = obs_Us(opls)

100(c) $\quad\quad\quad\Rightarrow$ routes(us)$\cup$ rotes(us$'$)

Dynamics of Pipelines

101. There is oil, $o : O$, and there is oil flow, $f : F$. We do not bother how oil volume is measured, but all oil is measured with the same measuring unit. Oil flow is measured by that measuring unit per some time units (for example, barrels per second).
102. One can observe the oil contained in oil pipeline units.
103. One can observe the oil flowing into and out of connectors of oil pipeline units.
104. Units leak oil.
105. The sum of the oil flowing into a unit minus its leak equals the sum of the oil flowing out of the unit.

type
101 O, F
value
102 obs_O: U $\rightarrow$ O
103 obs_ioFs: U $\rightarrow$ (KI $\xrightarrow[m]{}$F)$\times$(KI $\xrightarrow[m]{}$F)
104 obs_Leak: U $\rightarrow$ F
axiom
103 $\forall$ u:U $\cdot$
  let (ikis,okis) = (obs_iKIs(u),obs_iKIs(u)), (iflow,oflow) = obs_ioFs(u) **in**
  dom iflow = ikis $\wedge$ **dom** oflow = okis **end**
105 $\forall$ u:U $\cdot$ in_F(u) $-$ obs_Leak(u) = out_F(u)
value
 in_F,out_F: KI $\xrightarrow[m]{}$F $\rightarrow$ F
 in_F(fm),out_F(fm) $\equiv$ **case** fm **of** [ ]$\rightarrow$f$_0$,[ki$\mapsto$f]$\cup$ fm$'$$\rightarrow$f$\oplus$in_F(fm$'$) **end**

 $\oplus$: F $\times$ F $\rightarrow$ F
 f$_0$:F

f_0 is our way of designating the 'zero' flow, and $\oplus$ is our way of adding two flows.

106. Valve units can be in either of two states: `closed` or `open`.
107. Valves, when `closed`, also leak – in addition to “the usual” leak of units.
108. Pump units can be in either of two states: `pumping` or `not_pumping`.
109. If a valve unit is `closed` then the flows into and out from the unit are characterised by two leak flows.
110. If a pump unit is `not_pumping` then the flows into and out from the unit are characterised to be the same minus the leak of the pump unit.
111. If a pump unit is `pumping` then the flows into and out from the unit a characterised to still be the same minus the leak of the pump unit.

value

106 is_open: $U \rightarrow$ **Bool**

107 obs_Valve_Leak: $U \rightarrow F$

108 is_pumping: $U \rightarrow$ **Bool**

axiom

109 $\forall$ u:U $\cdot$ is_VaU(u) $\wedge$ $\sim$is_open(u) $\Rightarrow$

in_F(u) = obs_Leak(u) $\wedge$ out_F(u) = obs_Valve_Leak(u)

110-111 $\forall$ u:U $\cdot$ is_PuU(u) $\Rightarrow$ in_F(u) − obs_Leak(u) = out_F(u)

112. One can speak of the total leak of an oil pipeline system.
113. And one can speak of the total flow of oil into and the total flow of oil out from an oil pipeline system.
114. And, consequently one can conjecture a 'law' of oil pipeline systems: "what flows in is either lost to leaks or flows out".

value

112 total_Leak: U-**set** $\rightarrow$ F

112 total_Leak(us) $\equiv$ **case** us **of** {}$\rightarrow$f$_0$,{u}$\cup$ us′$\rightarrow$obs_Leak(u) $\cup$ total_Leak(us′) **end**

113 total_in_F, total_out_F: OPLS $\rightarrow$ F

113 total_in_F(opls) $\equiv$ tot_in_F(obs_iUs(opls))

113 total_out_F(opls) $\equiv$ tot_out_F(obs_oUs(opls))

113 tot_io_F: U-**set** $\rightarrow$ F

113 tot_in_F(us) $\equiv$ **case** us **of** {}$\rightarrow$f$_0$,{u}$\cup$ us′$\rightarrow$in_F(u) $\cup$ tot_in_F(us′) **end**

113 tot_out_F(us) $\equiv$ **case** us **of** {}$\rightarrow$f$_0$,{u}$\cup$ us′$\rightarrow$out_F(u) $\cup$ tot_out_F(us′) **end**

lemma:

114 $\forall$ opls:OPLS $\cdot$ total_in_F(opls) $-$ total_Leak(obs_Us(opls) $=$ total_in_F(opls)

This ends Example 55 ■

7.3.1. Principles

7.3.2. Discussion

-
-
-

End of Lecture 7

Domain Engineering: Opening and Intrinsics

Lecture 8

Domain Engineering: Support Techns., Mgt. & Org. and Rules & Regs.

7.4. Domain Support Technologies

Definition 45 – Domain Support Technology:

- *By domain support technology*
- *we mean a human or man-made technological device*
- *for the support of entities and behaviours, operations and events of the domain —*
- *with such a support thus enabling the existence of such phenomena and concepts in the domain.*

■

Example 56 – **Railway Switch Support Technology:** In "ye olde" days a railway switch (point machine [British], turn-out [US English], aguielette [French], sporskifte [Danish], weiche [German])

- was operated by a human, a railroad staff member;
- later, when quality of steel wires and pullers improved, the switch position could be controlled from the station cabin house;
- further on, in time, such mechanical gear was replaced by electro-mechanical gears,
- and, most recently, the monitoring and control of groups of switched could be (interlock) done with electronics interfacing to the electro-mechanics.

■

Usually, as hinted at in Example 56, several technologies may co-exist.

Example 57 – Air Traffic (II):

- By air traffic we mean
  - the time and position continuous movement of aircraft in and out of airports and in airspace,
  - that is,
    - for every time point there is a set of aircraft
    - in airspace
    - each with their (not necessarily) distinct positions
  - where an aircraft position is some triple of
    - latitude (ϕ),
    - longitude (λ) and
    - (true, indicated, height, pressure, or density) altitude (above sea level, above the terrain over which aircraft flying, etc.).

type

Time, Aircraft, Position

cAirTraffic = Time → (Aircraft $\overrightarrow{m}$ Position)

- How do we know the position of aircraft at any one time ?
- That is, can we record the continuous movement ?
  - In the above model time is assumed to be a linear, dense point set.
  - But can we record, measure, that ?
- The answer is: no we cannot !

- We, on the ground, can observe with our eyes, with binoculars, and with the aid of some radar (support) technology.
- The aircraft pilots can record altitude with a pressure altimeter (an aneroid barometer), and `LORAN` or a `Global Navigation Satellite System` (together with an aircraft chronometer) for determination of latitude and longitude.
- In any case, whether human or physical instrument-aided observation, one cannot record continuously.
- Instead any human or instrument awareness of movement is time and position discretised.

type

Time, Aircraft, Position

$\mathrm{cAirTraffic} = \mathrm{Time} \overrightarrow{_{m}} (\mathrm{Aircraft} \overrightarrow{_{m}} \mathrm{Position})$

- The difference between continuous and discretised air traffic,
- that is, between dAirTraffic and cAirTraffic,
- is the discretisation of Time.
- The way we get from cAirTraffic to dAirTraffic is by applying some SupportTechnology:

value

SupportTechnology: cAirTraffic $\rightarrow$ dAirTraffic

illustration:

axiom

$\forall$ cmvnt:cAirTraffic $\cdot$

$\exists$ dmvnt:dAirTraffic $\cdot$ dmvnt=SupportTechnology(cmvnt)

This ends Example 57 ■

Example 58 – Street Intersection Signalling:

- In this example of a support technology
  - we shall illustrate an abstraction
  - of the kind of semaphore signalling
  - one encounters at road intersections, that is, hubs.
- The example is indeed an abstraction:
  - we do not model the actual "machinery"
    - of road sensors,
    - hub-side monitoring & control boxes, and
    - the actuators of the green/yellow/red semaphore lamps.
  - But, eventually, one has to,
  - all of it,
  - as part of domain modelling.
- To model signalling we need to model hub and link states.

- We claim that the concept of hub and link states is an intrinsics facet of transport nets.
- We now introduce the notions of
  - hub and link states and state spaces and
  - hub and link state changing operations.
- A hub (link) state is the set of all traversals that the hub (link) allows.
  - A hub traversal is a triple of identifiers:
    * of the link from where the hub traversal starts,
    * of the hub being traversed, and
    * of the link to where the hub traversal ends.
  - A link traversal is a triple of identifiers:
    * of the hub from where the link traversal starts,
    * of the link being traversed, and
    * of the hub to where the link traversal ends.
  - A hub (link) state space is the set of all states that the hub (link) may be in.
  - A hub (link) state changing operation can be designated by
    * the hub and a possibly new hub state (the link and a possibly new link state).

type
  $\mathsf{L\Sigma' = L\_Trav\text{-}\mathbf{set}}$
  $\mathsf{L\_Trav = (HI \times LI \times HI)}$
  $\mathsf{L\Sigma = \{|\ lnk\sigma{:}L\Sigma' \bullet syn\_wf\_L\Sigma\{lnk\sigma\}\ |\}}$
value
  $\mathsf{syn\_wf\_L\Sigma\colon\ L\Sigma' \rightarrow \mathbf{Bool}}$
  $\mathsf{syn\_wf\_L\Sigma(lnk\sigma) \equiv}$
    $\mathsf{\forall\ (hi',li,hi''),(hi''',li',hi''''){:}L\_Trav \bullet \Rightarrow}$
      $\mathsf{(\{(hi',li,hi''),(hi''',li',hi'''')\} \in lnk\sigma \Rightarrow li = li' \wedge}$
      $\mathsf{hi' {\neq} hi'' \wedge hi''' {\neq} hi'''' \wedge \{hi',hi''\} = \{hi''',hi''''\})}$
type
  $\mathsf{H\Sigma' = H\_Trav\text{-}\mathbf{set}}$
  $\mathsf{H\_Trav = (LI \times HI \times LI)}$
  $\mathsf{H\Sigma = \{|\ hub\sigma{:}H\Sigma' \bullet wf\_H\Sigma\{hub\sigma\}\ |\}}$
value
  $\mathsf{syn\_wf\_H\Sigma\colon\ H\Sigma' \rightarrow \mathbf{Bool}}$
  $\mathsf{syn\_wf\_H\Sigma(hub\sigma) \equiv}$
    $\mathsf{\forall\ (li',hi,li''),(li''',hi',li''''){:}H\_Trav \bullet}$
      $\mathsf{\{(li',hi,li''),(li''',hi',li'''')\} {\subseteq} hub\sigma \Rightarrow hi = hi'}$

- The above well-formedness only checks syntactic well-formedness,
  - that is well-formedness when only considering the traversal designator,
  - not when considering the "underlying" net.
- Semantic well-formedness takes into account
  - that link identifiers designate existing links and
  - that hub identifiers designate existing hub.

value

sem_wf_LΣ: LΣ → N → **Bool**
sem_wf_HΣ: HΣ → N → **Bool**
sem_wf_LΣ(lnkσ)(ls,hs) ≡ lnkσ≠{}⇒
 ∀ (hi,li,hi′): LΣ • (hi,li,hi′) ∈ lnkσ ⇒
  ∃ h,h′:H • {h,h′}⊆hs ∧ obs_HI(h)=hi ∧ obs_HI(h′)=hi′
  ∃ l:L • l ∈ ls ∧ obs_LI(l)=li
 pre syn_wf_LΣ(lnkσ)
sem_wf_HΣ(hubσ)(ls,hs) ≡ hubσ≠{}⇒
 ∀ (li,hi,li′): HΣ • (li,hi,li′) ∈ hubσ ⇒
  ∃ l,l′:L • {l,l′}⊆ls ∧ obs_LI(l)=li ∧ obs_LI(l′)=li′
  ∃ h:H • h ∈ hs ∧ obs_HI(l)=hi
 pre syn_wf_HΣ(hubσ)
xtr_LIs: HΣ → LI-**set**
xtr_LIs(hubσ) ≡ {li,li′|(li,hi,li′):H_Trav • (li,hi,li′) ∈ hubσ}
xtr_HI: HΣ → HI
xtr_HI(hubσ) ≡ **let** (li,hi,li′):H_Trav • (li,hi,li′) ∈ hubσ **in** hi **end**
pre: hubσ≠{}
xtr_LI: LΣ → LI
xtr_LIs(lnkσ) ≡ **let** (hi,li,hi′):L_Trav • (hi,li,hi′) ∈ hubσ **in** li **end**
pre: lnkσ≠{}

$\quad$ xtr_HIs: $\mathsf{L\Sigma} \rightarrow$ HI-**set**
$\quad$ xtr_HIs(lnkσ) **as** his
$\quad$ **pre**: lnk$\sigma \neq \{\}$
$\quad$ **post** his$=\{$hi,hi$'|$(hi,li,hi$'$):L_Trav $\cdot$ (hi,li,hi$'$) $\in$ hub$\sigma\} \wedge$ **card** his=2
type
$\quad$ HΩ = HΣ-**set**, LΩ = LΣ-**set**
value
$\quad$ obs_HΩ: H $\rightarrow$ HΩ, obs_LΩ: L $\rightarrow$ LΩ
axiom
$\quad$ $\forall$ h:H $\cdot$ obs_HΣ(h) $\in$ obs_HΩ(h) $\wedge$ $\forall$ l:L $\cdot$ obs_LΣ(l) $\in$ obs_LΩ(l)
value
$\quad$ chg_HΣ: H $\times$ HΣ $\rightarrow$ H, chg_LΣ: L $\times$ LΣ $\rightarrow$ L
$\quad$ chg_HΣ(h,hσ) **as** h$'$
$\quad\quad$ **pre** hσ $\in$ obs_HΩ(h) **post** obs_HΣ(h$'$)=hσ
$\quad$ chg_LΣ(l,lσ) **as** l$'$
$\quad\quad$ **pre** lσ $\in$ obs_LΩ(h) **post** obs_HΣ(l$'$)=lσ

- Well, so far we have indicated that there is an operation that can change hub and link states.
- But one may debate whether those operations shown are really examples of a support technology. (That is, one could equally well claim that they remain examples of intrinsic facets.)
- We may accept that and then ask the question:
  - How to effect the described state changing functions ?
  - In a simple street crossing a semaphore does not instantaneously change from red to green in one direction while changing from green to red in the cross direction.
  - Rather there is are intermediate sequences of, for example, not necessarily synchronised green/yellow/red and red/yellow/green states to help avoid vehicle crashes and to prepare vehicle drivers.
- Our "solution" is to modify the hub state notion.

```
type
  Colour == red | yellow | green
  X = LI×HI×LI×Colour [ crossings of a hub ]
  HΣ = X-set [ hub states ]
value
  obs_HΣ: H → HΣ, xtr_Xs: H → X-set
  xtr_Xs(h) ≡
    {(li,hi,li′,c)|li,li′:LI,hi:HI,c:Colour•{li,li′}⊆obs_LIs(h)∧hi=obs_HI(h)}
axiom
  ∀ n:N,h:H • h ∈ obs_Hs(n) ⇒ obs_HΣ(h)⊆xtr_Xs(h) ∧
    ∀ (li1,hi2,li3,c),(li4,hi5,li6,c′):X •
      {(li1,hi2,li3,c),(li4,hi5,li6,c′)}⊆obs_HΣ(h) ∧
      li1=li4 ∧ hi2=hi5 ∧ li3=li6 ⇒ c=c′
```

- We consider the colouring, or any such scheme, an aspect of a support technology facet.
- There remains, however, a description of how the technology that supports the intermediate sequences of colour changing hub states.
- We can think of each hub being provided with a mapping from pairs of "stable" (that is non-yellow coloured) hub states (hσ_i,hσ_f) to well-ordered sequences of intermediate "un-stable' (that is yellow coloured) hub states
  - paired with some time interval information
  - $\langle(h\sigma', t\delta'),\ (h\sigma'', t\delta''),\ \ldots,\ (h\sigma'^{\cdots\prime}, t\delta'^{\cdots\prime})\rangle$
  - and so that each of these intermediate states can be set,
  - according to the time interval information,[28]
  - before the final hub state (hσ_f) is set.

[28] Hub state $h\sigma''$ is set $t\delta'$ time unites after hub state $h\sigma'$ was set.

type
  TI [time interval]
  Signalling = $(\mathsf{H}\Sigma \times \mathsf{TI})^*$
  Sema = $(\mathsf{H}\Sigma \times \mathsf{H}\Sigma) \xrightarrow[m]{}$ Signalling
value
  obs_Sema: H $\rightarrow$ Sema,
  chg_HΣ: H $\times$ HΣ $\rightarrow$ H,
  chg_HΣ_Seq: H $\times$ HΣ $\rightarrow$ H
  chg_HΣ(h,hσ) **as** h$'$
    pre hσ $\in$ obs_HΩ(h) **post** obs_HΣ(h$'$)=hσ
  chg_HΣ_Seq(h,hσ) $\equiv$
    let sigseq = (obs_Sema(h))(obs_Σ(h),hσ) **in** sig_seq(h)(sigseq) **end**
  sig_seq: H $\rightarrow$ Signalling $\rightarrow$ H
  sig_seq(h)(sigseq) $\equiv$
    if sigseq=$\langle\rangle$ **then** h **else**
    let (hσ,tδ) = **hd** sigseq **in let** h$'$ = chg_HΣ(h,hσ);
    wait tδ;
    sig_seq(h$'$)(**tl** sigseq) **end end end**

This ends Example 58 ■

7.4.1. A Formal Characterisation of a Class of Support Technologies

- We have presented an abstraction of the physical phenomenon of a road intersection semaphore.
- That abstraction has to be further concretised.
  - The electronic, electro-mechanical or other
  - and the data communication
  - monitoring of incoming street traffic
  - and the semaphore control box control
  - of when to start and end semaphore switching,
  - etcetera, must all be detailed.

Schema 1 – A Support Technology Evaluation Scheme:

- Let the support technology be one for observing and recording the movement of cars along roads, trains along rail tracks, aircraft in airspace (along air-lanes), or “some such thing”.
- We can evaluate the quality of “some such” support technology by interpreting the following specification pattern:
- Let *is_close* be a predicate which holds if two positions are close to one-another.
- Proximity is a fuzzy notion, so let the *is_close* predicate be “tunable”, i.e., by set to “any degree” of closeness.

```
type
  Vehicle, Position
  continuous_Movement = Time → (Vehicle →m Position)
  discrete_Movement = Time →m (Vehicle →m Position)
value
  obs_and_record_Mvmt: continuous_Movement → discrete_Movement
  is_close: Position × Position → Bool
  quality_Support_Technology: is_close → Bool
  quality_Support_Technology(is_close) ≡
    ∀ cmvt:continuous_Movement •
      let dmvt = obs_and_record_Mvmt(cvmt) in
      dom dvmt ⊆ 𝒟𝒪ℳ𝒜ℐ𝒩 cvmt ∧
      ∀ t:Time • t ∈ dom dvmt
        ∀ v:Vehicle • v ∈ dom dvmt(t) ⇒ is_close(((cvmt)(t))(v),((dvmt)(t))(v)) end
```

- The above scheme can be interpreted as follows:
  - For any given sub-domain of movement, be it road traffic, train traffic, air traffic or other, there is a set of technologies that enable observation and recording of such traffic.
  - For a given such technology and a given such traffic, that is, a traffic along a specific route, the predicate *is_close* has to be "instantiated", i.e., "tuned".
  - Then, to test whether the technology delivers an acceptable observation and recording, that is, is of a necessary and sufficient quality, a laboratory experiment — usually quite a resource (equipment, cost and time) consuming affair — has to be carried out before accepting acquisition and installation of that technology for that route.

- The experiment ideally compares the actual traffic to that observed and recorded by the contemplated technology.
- But the actual traffic "does not exist in any recorded form".
- Hence a "highest possible" movement recording (reference) support technology must first be (experimentally) developed and made available.
- We then say that whatever that reference technology represents is the actual, but discretised movement.
- It is that reference movement which is now compared — using *is_close* — to the discretised movement recorded by the support technology being tested.

■

7.4.2. Discussion

- For more detailed modelling of specific support technologies, including more concrete models of movement sensors and recorders and of street intersection signals, one will undoubtedly need use other formalisms than the ones mainly used in this paper, for example:
  - MSCs and LSCs,
  - Petri nets,
  - SCs,
  - DC,
  - TLA+,
  - STeP,
  - etcetera

7.4.3. Principles

7.5. Domain Management and Organisation

- The term management usually conjures an image of an institution of
  - owners,
  - two or three layers of (hierarchical or matrix) stratified management,
  - workers, and of
  - clients.

- Management is about resources and resources come in many shapes and forms:
  - manifest equipment, buildings, land,
  - services and/or production goods,
  - financial assets (liquid cash, bonds, stocks, etc.),
  - staff (personnel),
  - customer allegiance, and
  - goodwill[29].

[29] Goodwill: the favor or advantage that a business has acquired especially through its brands and its good reputation

- Management decisions as to the monitoring and controlling of resources are often, for pragmatic reasons, classified as
  - strategic,
  - tactical and
  - operational

  monitor and control decisions and actions.
- The borderlines between
  - strategic and tactical, and between
  - tactical and operational

  monitor and control decisions and actions
- is set by pragmatic concerns, that is, are hard to characterise precisely.
- But we shall try anyway.

Definition 46 – **Strategy:** *By strategy we shall understand*

- *the science and art*
- *of formulating the goals of an enterprise*
- *and of employing the*
  - *political,*
  - *economic,*
  - *psychological, and*
  - *institutional*

  resources of that enterprise
- *to achieve those goals.*

■

Definition 47 – **Tactics:** *By tactics we shall understand*

- *the art or skill*
- *of employing available resources*
- *to accomplish strategic goals.*

■

- We introduce three kinds of entities to model an essence of strategic, tactical and operational management.
  - Let RES (for resources) designate an indexed set of resources;
  - let ENV (for environment) designate a binding of resource names to resource locations and some of their more static properties — such a schedules, and
  - let Σ (for state) designate the association of resource locations to the more dynamic properties (attributes) of resources,
  - then we might be able to delineate the three major kinds of actions:

type

A, B, C, RES, ENV, Σ

value

strategic_action: $A \to RES \to ENV \to \Sigma \to RES$

tactical_action: $B \to RES \to ENV \to \Sigma \to ENV$

operational_action: $C \to RES \to ENV \to \Sigma \to \Sigma$

- A, B and C
  - are "inputs" chosen by management
  - to reflect strategic or tactical decisions.
- Sometimes tactical actions also change the state:

type

tactical_action: $B \to RES \to ENV \to \Sigma \to \Sigma \times ENV$

- A strategic action, $\mathsf{strategic\_action(a)(res)}(\rho)(\sigma)$ **as** res', in principle does not change the environment and state but sets up a new set of resources, res', for which “future” business is transacted.
- A tactical action, $\mathsf{tactical\_action(a)(res)}(\rho)(\sigma)$ **as** ρ', changes the environment — typically the scheduling and allocation components of environments.
- Operational actions, $\mathsf{operational\_action(a)(res)}(\rho)(\sigma)$ **as** σ', changes the state.

The above strategy/tactics/operations “abstraction” is an idealised “story”.

Definition 48 – Resource Monitoring:

- *By the monitoring of resources we mean the regular keeping track of these resources:*
  - *their current value,*
  - *state-of-quality,*
  - *location,*
  - *usage, etc. —*

  including changes in these (i.e., trends).

■

Definition 49 – Resource Control:

- *By the controlling of resources we mean the*
  - *acquisition (usually as the result of converting one resources into another),*
  - *regular scheduling and allocation*
  - *and final disposal (sale, renewal or "letting go")*

of these resources.

■

Definition 50 – **Management:** *By management we mean*

- *the strategic,*
- *tactical and*
- *operational*
- *monitoring and*
- *controlling of resources*

. ■

Definition 51 – **Organisation:** *By organisation we mean*

- *the stratification*
- *(arranging into graded classes)*
- *of management and enterprise actions.*

■

Example 59 – Management and Organisation:

- We can claim that
  - the set of models of the description given in Example 42
  - includes that of enterprise management and organisation.
  - We refer to Fig. 10 on next to slide.

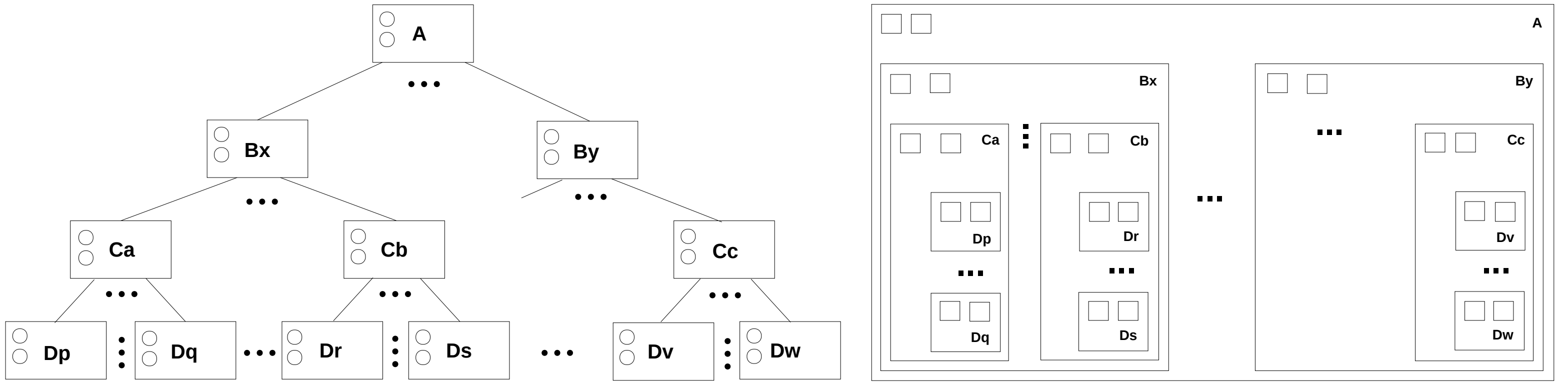


Figure 10: Conventional hierarchical organigram and its mereology diagram

- The small, quadratic round-corner boxes of Fig. 10
- can be thought of as designating staff or other (atomic) resources.

- We will now define a number of strategic/tactical operations on the organisation of an enterprise.
- For simplicity, bit without any loss of generality, we assume a notion of void parts, that is parts with no connections and, if assemblies, then with no sub-parts.
- The operations to be defined can be considered 'primitive' only in the sense that more realistic operations on non-void parts can be defined in terms of these primitive operations.
- Given this interpretation we can now postulate a number of management operations (over a given system s).

115. *Assign* a new, void resource p, to a given assembly (i.e., division or department) identified by i.
116. *Move* a given, void resource identified by i, from an assembly identified by f_i to another assembly identified by t_j.
117. *Delete* a given, void resource identified by i.

- We ignore, for the time, the issue of connectors.
- In order to model these operations we need first introduce some concepts:

118. Given a system, s, and a part, p, of that system,
119. the sequence $\langle\pi_1, \pi_2, ..., \pi_{n-1}, \pi\rangle$
120. is the sequence of part identifiers such that π_1 is that of the assembly that s is,
121. that is, is the 1st level part that embraces p, π_n
122. is the identifier of p, and $\imath_i$, for $1<i<n$,
123. is the i'th level part embracing p.

value

$\text{void\_P}: \text{P} \rightarrow \textbf{Bool}$

$\text{void\_P(p)} \equiv \text{obs\_KIs(p)}=\{\} \wedge \text{is\_A(p)} \Rightarrow \text{obs\_Ps(p)}=\{\}$

type

$\text{Path} = \text{AUI}^*$

value

$\text{gen\_Path}: \text{P} \rightarrow \text{A} \stackrel{\sim}{\rightarrow} \text{Path}$

$\text{gen\_Path(p)(a)} \equiv$

$\quad \langle \text{obs\_AUI(a)} \rangle \widehat{}$

$\quad \textbf{if } \text{p=a } \textbf{then } \langle\rangle$

$\quad \textbf{else let } \text{a}':\text{A} \cdot \text{a}' \in \text{obs\_Ps(a)} \wedge \text{p} \in \text{xtr\_Ps(a}') \textbf{ in } \text{gen\_Path(p)(a}') \textbf{ end end}$

$\quad \textbf{pre}: \text{p} \in \{\text{a}\} \cup \text{xtr\_Ps(a)}$

$\text{gen\_all\_Paths}: \text{A} \rightarrow \text{Path-}\textbf{set}$

$\text{gen\_all\_Paths(a)} \equiv \{\text{gen\_Path(p)(a)} | \text{p:P} \cdot \text{p} \in \text{xtr\_Ps(a)}\}$

124. *Assigning* a new, void part, p, to a system, s, results in a new system, s'. p is in this new system. Let the path to p be $\pi\ell$. Let the set of all paths of s be $pths$. Then the set of all paths of s' is $pths \cup \{\pi\ell\}$. Thus it follows that the set, ps', of all parts of s', is p together with the set, ps, of all parts of $s : ps' = ps \cup \{p\}$.
125. *Moving* a given, void part, p, of a system, s, results in a new system s'. Let the path to p in s be $\pi\ell$, and let the path to p in s' be $\pi\ell'$. Then the set of paths of the two systems relate as follows: $pths \setminus \{\pi\ell\} = pths' \setminus \{\pi\ell'\}$ and $ps' = ps \cup \{p\}$.
126. *Deleting* a given, void part, p, from a system, s, results in a new system, s'. The new system has exactly one less path than the set of all paths of s. And we have: $pths \setminus \{\pi\ell\} = pths'$ and $ps' = ps \setminus \{p\}$.

semantic types
S, A, U, P=A|U
value
$\text{get\_P}: S \to (AI|UI) \xrightarrow{\sim} P$
$\text{get\_P}(s)(i) \equiv$ **let** $p{:}P \cdot p \in \text{xtr\_Ps}(s) \wedge \text{obs\_AUI}(p)=i$ **in** p **end**
pre $\exists\, p{:}P \cdot p \in \text{xtr\_Ps}(s) \wedge \text{obs\_AUI}(p)=i$

syntactic types
MgtOp = AP | MP | DP | MA | CA
AP == AsgP(pt:P,ai:AI)
MP == MovP(ai:AI,fai:AI,tai:AI)
DP == DelP(ai:AI)

value

int_MgtOp: MgtOp $\xrightarrow{\sim}$ S $\xrightarrow{\sim}$ S

int_MgtOp(AsgP(p,i))(s) **as** s′
pre void_P(p) $\wedge$ obs_AUI(p)$\notin$ xtr_AUIs(s)
post obs_Ps(s) $\cup$ {u} = obs_Ps(s′) $\wedge$ obs_Ks(s)=obs_Ks(s′) $\wedge$
gen_all_Paths(s)$\cup${gen_Path(p)(s′)}=gen_all_Paths(s′)

int_MgtOp(MovP(i,fi,ti))(s) **as** s′
pre void_P(get_P(s)(i)) $\wedge$ i$\neq$fi$\wedge$i$\neq$ti$\wedge$fi$\neq$ti$\wedge${i,fi,ti}$\subseteq$xtr_AUIs(s)
post obs_Ps(s) = obs_Ps(s′) $\wedge$ obs_Ks(s)=obs_Ks(s′) $\wedge$
gen_all_Paths(s)\{gen_Path(p)(s)} = gen_all_Paths(s′)\{gen_Path(p)(s′)}

int_MgtOp(DelP(i))(s) **as** s′
pre void_P(get_P(s)(i)) $\wedge$ i $\in$ xtr_AUIs(s)
post obs_Ps(s′) = obs_Ps(s)\{get_P(s)(i)} $\wedge$ obs_Ks(s)=obs_Ks(s′) $\wedge$
gen_all_Paths(s′)=gen_all_Paths(s)\{gen_Path(p)(s)}

- Similar connector operations can be postulated (narrated and formalised):

127. *Insert* a new (internal or external) connector, k, in a system s between parts i and j, or just emanating from (incident upon) part i;
128. *Move* a given connector's connections from parts $\{i, j\}$ to parts $\{i, k\}$, $\{\ell, k\}$ or $\{k, j\}$; and
129. *Delete* a given connector.

These operations would have to suitably update connected parts' connector identifier attributes.

- The hierarchical organigram of Fig. 10 on Slide 459 portrays one organisation form.
- So-called matrix-organisations, cf. Fig. 11 on the following slide are likewise modelled by the mereology concept introduced in Examples 42 and 44.

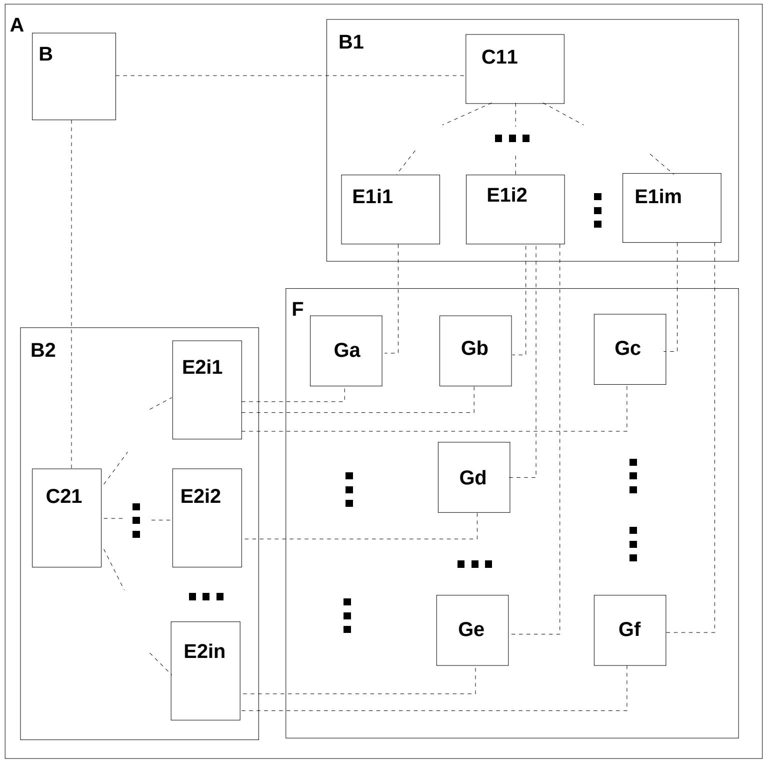


Figure 11: Conventional matrix organigram; mereology diagram is hinted at.

- We see, in Fig. 11, the use of connectors to underscore the two hierarchies: the strategic and tactical (*B1* and *B2*) and the matrix-sharing of production and service facilities (*F*, *Ga*, . . . , *Gf*).

This ends Example 59 ■

(7. **Domain Engineering** 7.5. **Domain Management and Organisation**)

7.5.1. Principles

7.5.2. Discussion

7.6. Domain Rules and Regulations

Definition 52 – Domain Rule: *By a domain rule we understand*

- *a text which prescribes how humans and/or technology are expected to behave, respectively function.*
- *A domain rule text thus denotes a predicate over states,*
  - *the state before,* σ_β, *and the state after,* σ_α,

  a human or a technology action.
- *If the predicate is satisfied, then the rule has been adhered to, i.e., the rule has not been "broken".*
- *The 'after' state,* σ_α,
- *following a rule that has been broken in some 'before' state*
- *will be referred to as a 'rule-braking state'.*

■

Definition 53 – **Domain Regulation:** *By a domain regulation we understand*

- *a text which prescribes [remedial] actions to be taken in case a domain rule has been "broken".*
- *A domain regulation text thus denotes an action, i.e., a state-to-state transformation,*
- *one that transforms a 'rule-braking state'* σ_α *into a (new 'after') state,* σ_{OK}*,*
- *in which the rule now appears to not have been broken.*

■

Example 60 – **Trains Entering and/or Leaving Stations:** For some train stations there is the rule that

- no two trains may enter and/or leave that station
- within any (sliding "window") n minute interval —
- where n typically is 2.
- If train engine men disregard this rule
  - they may be subject to disciplinary action —
  - as determined by some subsequent audit —
  - and the train may be otherwise diverted through actions from the train station cabin tower.

■

Example 61 – Rail Track Train Blocking:

- Usually rail tracks,
  - that is, longer sequences of linear rail units
  - connecting two train stations
- are composed of two or more blocks (also sequences of linear rail units).
- The train blocking rule for trains moving along such rail tracks (obviously in the same direction) is that
  - there must always be an empty block
  - between any two 'neighbouring' trains.
  - (We may consider the connecting stations to serve the rôle of such blocks.)
- Again, if the rule is broken by some train engine man,
  - then that person
  - may be subject to disciplinary action —
  - as determined by some subsequent audit — et cetera.

■

7.6.1. A Formal Characterisation of Rules and Regulations

Schema 2 – A Rules and Regulations Specification Pattern:

- Let Σ designate the state space of a domain;
- let Rule designate the syntax category of rules;
- let RULE designate the semantic type of rules, that is, the denotation of Rules: predicates over pairs of (before and after) states;
- let Stimulus designate the syntax category of stimuli that cause actions, hence state changes,
- that is, let STIMULUS finally designate the semantic type of Stimuli.
- valid_stimulus is now a predicate which "tests" whether a given stimulus and a given rule in a given state, σ, leads to a not-been-broken state.

type

Rule, Stimulus, Σ

RULE = $\Sigma \times \Sigma \rightarrow$ **Bool**

STIMULUS = $\Sigma \rightarrow \Sigma$

value

$\mathcal{M}_{rule}$: Rule $\rightarrow$ RULE

$\mathcal{M}_{stimulus}$: Stimulus $\rightarrow$ STIMULUS

$\mathcal{V}$alid_stimulus: Stimulus $\rightarrow$ Rule $\rightarrow \Sigma \rightarrow$ **Bool**

$\mathcal{V}$alid_stimulus(stimulus)(rule)(σ) $\equiv$

  (($\mathcal{M}_{rule}$)(rule))(σ,($\mathcal{M}_{stimulus}$(stimulus))(σ))

- Let Rule_and_Regulation designate the syntax category of pairs of rules and related regulations;
- let Regulation designate the semantic type of regulations, that is, the denotation of Regulations: state transformers from broken to OK states.

type
  Regulation
  Rule_and_Regulation = Rule × Regulation
  REGULATION = $\Sigma \rightarrow \Sigma$
value
  $\mathcal{M}_{regulation}$: Regulation $\rightarrow$ REGULATION
axiom
  $\forall$ (rule_syntax,regulation_syntax):Rule_and_Regulation, stimulus:Stimulus,
    $\sim\mathcal{V}$alid_stimulus(stimulus)(rule_syntax)(σ) $\Rightarrow$
      $\exists\ \sigma'$:Σ ·
        ($\mathcal{M}_{regulation}$(regulation_syntax))(σ)=σ'
        $\wedge$ ($\mathcal{M}_{rule}$(rule_syntax))(σ,σ')

■

7.6.2. Principles

- Rules and regulations are best treated by separately describinging
  - their pragmatics,
  - their semantics, and
  - their syntax

  — the latter two were hinted at in Sect. .

7.6.3. Discussion

- Many more examples could be given, and also formalised.
- We leave that to the next section, Sect. .

End of Lecture 8

Domain Engineering: Support Techns., Mgt. &Org. and Rules & Regs.

Lecture 9

Domain Engineering: Scripts

7.7. Domain Scripts, Licenses and Contracts

Definition 54 – Script: *By a domain script we shall understand*

- *a structured text*
- *which can be interpreted as a set of rules ("in disguise").*

■

Example 62 – Timetables:

- We shall view timetables as scripts.
- In on the present and next slides (482–499) we shall
  - first narrate and formalise the **syntax**, including the well-formedness of timetable scripts,
  - then we consider the **pragmatics** of timetable scripts,
    - including the bus routes prescribed by these journey descriptions and
    - timetables marked with the status of its currently active routes, and
  - finally we consider the **semantics** of timetable, that is, the traffic they denote.
- In Example. 65 on contracts for bus traffic, we shall assume the timetable scripts of this part of the lecture on scripts.

Figure 12: Some bus timetables: Italy, India and Norway

⊕ *The Syntax of Timetable Scripts* ⊕

130. Time is a concept covered earlier. Bus lines and bus rides have unique names (across any set of time tables). Hub and link identifiers, HI, LI, were treated from the very beginning.
131. A TimeTable associates to Bus Line Identifiers a set of Journies.
132. Journies are designated by a pair of a BusRoute and a set of BusRides.
133. A BusRoute is a triple of the Bus Stop of origin, a list of zero, one or more intermediate Bus Stops and a destination Bus Stop.
134. A set of BusRides associates, to each of a number of Bus Identifiers a Bus Schedule.
135. A Bus Schedule a triple of the initial departure Time, a list of zero, one or more intermediate bus stop Times and a destination arrival Time.
136. A Bus Stop (i.e., its position) is a Fraction of the distance along a link (identified by a Link Identifier) from an identified hub to an identified hub.
137. A Fraction is a **Real** properly between 0 and 1.
138. The Journies must be well_formed in the context of some net.

type

130. T, BLId, BId

131. $\mathrm{TT} = \mathrm{BLId} \underset{m}{\rightarrow} \mathrm{Journies}$

132. $\mathrm{Journies}' = \mathrm{BusRoute} \times \mathrm{BusRides}$

133. $\mathrm{BusRoute} = \mathrm{BusStop} \times \mathrm{BusStop}^* \times \mathrm{BusStop}$

134. $\mathrm{BusRides} = \mathrm{BId} \underset{m}{\rightarrow} \mathrm{BusSched}$

135. $\mathrm{BusSched} = \mathrm{T} \times \mathrm{T}^* \times \mathrm{T}$

136. BusStop == mkBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)

137. $\mathrm{Frac} = \{|\mathrm{r}{:}\mathbf{Real}\cdot 0<\mathrm{r}<1|\}$

138. $\mathrm{Journies} = \{|\mathrm{j}{:}\mathrm{Journies}'\cdot\exists\ \mathrm{n}{:}\mathrm{N}\cdot \mathrm{wf\_Journies(j)(n)}|\}$

- The free n in ∃ n:N · wf_Journies(j)(n) is the net given in the license.

⊕ *Well-formedness of Journies* ⊕

139. A set of journies is well-formed
140. if the bus stops are all different,
141. if a defined notion of a bus line is embedded in some line of the net, and
142. if all defined bus trips (see below) of a bus line are commensurable.

value

139. wf_Journies: Journies → N → **Bool**
139. wf_Journies((bs1,bsl,bsn),js)(hs,ls) ≡
140. diff_bus_stops(bs1,bsl,bsn) ∧
141. is_net_embedded_bus_line(⟨bs1⟩⌢bsl⌢⟨bsn⟩)(hs,ls) ∧
142. commensurable_bus_trips((bs1,bsl,bsn),js)(hs,ls)

143. The bus stops of a journey are all different
144. if the number of elements in the list of these equals the length of the list.

value

143. diff_bus_stops: BusStop $\times$ BusStop* $\times$ BusStop $\rightarrow$ **Bool**
143. diff_bus_stops(bs1,bsl,bsn) $\equiv$
144. **card elems** $\langle$bs1$\rangle$^bsl^$\langle$bsn$\rangle$ = **len** $\langle$bs1$\rangle$^bsl^$\langle$bsn$\rangle$

- We shall refer to the (concatenated) list ($\langle$bs1$\rangle$^bsl^$\langle$bsn$\rangle$ = **len** $\langle$bs1$\rangle$^bsl^$\langle$bsn$\rangle$) of all bus stops as the bus line.

145. To explain that a bus line is embedded in a line of the net
146. let us introduce the notion of all lines of the net, lns,
147. and the notion of projecting the bus line on link sector descriptors.
148. For a bus line to be embedded in a net then means that there exists a line, ln, in the net, such that a compressed version of the projected bus line is amongst the set of projections of that line on link sector descriptors.

value

145. is_net_embedded_bus_line: BusStop* → N → **Bool**
145. is_net_embedded_bus_line(bsl)(hs,ls)
146. **let** lns = lines(hs,ls),
147. cbln = compress(proj_on_links(bsl)(**elems** bsl)) **in**
148. ∃ ln:Line • ln ∈ lns ∧ cbln ∈ projs_on_links(ln) **end**

149. Projecting a list (*) of BusStop descriptors (mkBS(hi,li,f,hi′)) onto a list of Sector Descriptors ((hi,li,hi′))
150. we recursively unravel the list from the front:
151. if there is no front, that is, if the whole list is empty, then we get the empty list of sector descriptors,
152. else we obtain a first sector descriptor followed by those of the remaining bus stop descriptors.

value

149. proj_on_links: $\text{BusStop}^* \rightarrow \text{SectDescr}^*$
149. proj_on_links(bsl) $\equiv$
150. **case** bsl **of**
151. $\langle\rangle \rightarrow \langle\rangle$,
152. $\langle$mkBS(hi,li,f,hi′)$\rangle$^bsl′ $\rightarrow$ $\langle$(hi,li,hi′)$\rangle$^proj_on_links(bsl′)
152. **end**

153. By compression of an argument sector descriptor list we mean a result sector descriptor list with no duplicates.
154. The compress function, as a technicality, is expressed over a diminishing argument list and a diminishing argument set of sector descriptors.
155. We express the function recursively.
156. If the argument sector descriptor list an empty result sector descriptor list is yielded;
157. else
158. if the front argument sector descriptor has not yet been inserted in the result sector descriptor list it is inserted else an empty list is "inserted"
159. in front of the compression of the rest of the argument sector descriptor list.

```
153. compress: SectDescr* → SectDescr-set → SectDescr*
154. compress(sdl)(sds) ≡
155.   case sdl of
156.     ⟨⟩ → ⟨⟩,
157.     ⟨sd⟩⌢sdl′ →
158.       (if sd ∈ sds then ⟨sd⟩ else ⟨⟩ end)
159.       ⌢compress(sdl′)(sds\{sd}) end
```

- In the last recursion iteration (line 159.)
  - the continuation argument $\mathsf{sds} \backslash \{\mathsf{sd}\}$
  - can be shown to be empty: $\{\}$.

160. We recapitulate the definition of lines as sequences of sector descriptions.
161. Projections of a line generate a set of lists of sector descriptors.
162. Each list in such a set is some arbitrary, but ordered selection of sector descriptions.

type

160. $\mathrm{Line}' = (\mathrm{HI}\times\mathrm{LI}\times\mathrm{HI})^*$ **axiom** ... **type** $\mathrm{Line} = \ldots$

value

161. $\mathrm{projs\_on\_links}\colon \mathrm{Line} \rightarrow \mathrm{Line}'$**-set**

161. $\mathrm{projs\_on\_links}(\mathrm{ln}) \equiv$

162. $\{\langle \mathrm{isl}(i) | i{:}\langle 1..\mathbf{len}\ \mathrm{isl}\rangle\rangle | \mathrm{isx}{:}\mathbf{Nat\text{-}set}\cdot \mathrm{isx}\subseteq \mathbf{inds}\ \mathrm{ln} \wedge \mathrm{isl}=\mathrm{sort}(\mathrm{isx})\}$

163. sorting a set of natural numbers into an ordered list, isl, of these is expressed by a post-condition relation between the argument, isx, and the result, isl.
164. The result list of (arbitrary) indices must contain all the members of the argument set;
165. and "earlier" elements of the list must precede, in value, those of "later" elements of the list.

value

163. sort: $\textbf{Nat-set} \rightarrow \textbf{Nat}^*$

163. sort(isx) **as** isl

164. **post card** isx = lsn isl $\wedge$ isx = **elems** isl $\wedge$

165. $\forall$ i:**Nat** $\cdot$ $\{$i,i+1$\}\subseteq$**inds** isl $\Rightarrow$ isl(i)<isl(i+1)

166. The bus trips of a bus schedule are commensurable with the list of bus stop descriptions if the following holds:
167. All the intermediate bus stop times must equal in number that of the bus stop list.
168. We then express, by case distinction, the reality (i.e., existence) and timeliness of the bus stop descriptors and their corresponding time descriptors – and as follows.
169. If the list of intermediate bus stops is empty, then there is only the bus stops of origin and destination, and they must be exist and must fit time-wise.

170. If the list of intermediate bus stops is just a singleton list, then the bus stop of origin and the singleton intermediate bus stop must exist and must fit time-wise. And likewise for the bus stop of destination and the the singleton intermediate bus stop.
171. If the list is more than a singleton list, then the first bus stop of this list must exist and must fit time-wise with the bus stop of origin.
172. As for Item 171 but now with respect to last, resp. destination bus stop.
173. And, finally, for each pair of adjacent bus stops in the list of intermediate bus stops
174. they must exist and fit time-wise.

```
value
166. commensurable_bus_trips: Journies → N → Bool
166. commensurable_bus_trips((bs1,bsl,bsn),js)(hs,ls)
167.   ∀ (t1,til,tn):BusSched·(t1,til,tn)∈ rng js∧len til=len bsl∧
168.     case len til of
169.       0 → real_and_fit((t1,t2),(bs1,bs2))(hs,ls),
170.       1 → real_and_fit((t1,til(1)),(bs1,bsl(1)))(hs,ls)∧fit((til(1),t2),(bsl(1
171.       _ → real_and_fit((t1,til(1)),(bs1,bsl(1)))(hs,ls)∧
172.           real_and_fit((til(len til),t2),(bsl(len bsl),bsn))(hs,ls)∧
173.           ∀ i:Nat·{i,i+1}⊆inds til ⇒
174.             real_and_fit((til(i),til(i+1)),(bsl(i),bsl(i+1)))(hs,ls) end
```

175. A pair of (adjacent) bus stops exists and a pair of times, that is the time interval between them, fit with the bus stops if the following conditions hold:
176. All the hub identifiers of bus stops must be those of net hubs (i.e., exists, are real).
177. There exists links, l, l', for the identified bus stop links, li, li',
178. such that these links connect the identified bus stop hubs.
179. Finally the time interval between the adjacent bus stops must approximate fit the distance between the bus stops
180. The distance between two bus stops is a loose concept as there may be many routes, short or long, between them.
181. So we leave it as an exercise to the student to change/augment the description, in order to be able to ascertain a plausible measure of distance.
182. The approximate fit between a time interval and a distance must build on some notion of average bus velocity, etc., etc.
183. So we leave also this as an exercise to the student to complete.

175. real_and_fit: (T×T)×(BusStop×BusStop) → N → **Bool**
175. real_and_fit((t,t′),(mkBS(hi,li,f,hi′),mkBS(hi″,li′,f′,hi‴)))(hs,ls) ≡
176. {hi,hi′,hi″,hi‴}⊆his(hs)∧
177. ∃ l,l′:L•{l,l′}⊆ls∧(obs_LI(l)=li∧obs(l′)=li′)∧
178. obs_HIs(l)={hi,hi′}∧obs_HIs(l′)={hi″,hi‴}∧
179. afit(t′−t)(distance(mkBS(hi,li,f,hi′),mkBS(hi″,li′,f′,hi‴))(hs,ls))

180. distance: BusStop × BusStop → N → Distance
181. distance(bs1,bs2)(n) ≡ ... [left as an exercise !] ...

182. afit: TI → Distance → **Bool**
183. [time interval fits distance between bus stops]

This ends Example 62 ■

Definition 55 – **Licenses:** *By a domain license we shall understand*

- *a right or permission granted in accordance with law*
- *by a competent authority*
  - *to engage in some business or occupation,*
  - *to do some act,*
  - *or to engage in some transaction*
- *which*
  - *but for such license*
- *would be unlawful Merriam Webster On-line.*

■

Definition 56 – **Contract:** *By a domain contract we shall understand*

- *very much the same thing as a license:*
- *a binding agreement between two or more persons or parties —*
- *one which is legally enforceable.*

■

- The concepts of licenses and licensing express relations between
  - *actors* (licensors (the authority) and licensees),
  - *simple entities* (artistic works, hospital patients, public administration and citizen documents) and
  - *operations* (on simple entities), and as performed by actors.
- By issuing a license to a licensee, a licensor wishes to express and enforce certain permissions and obligations:
  - which operations
  - on which entities
  - the licensee is allowed (is licensed, is permitted) to perform.
- As such a license denotes a possibly infinite set of allowable behaviours.

- We shall consider four kinds of entities:
  - (i) digital recordings of artistic and intellectual nature:
    - music, movies, readings ("audio books"), and the like,
  - (ii) patients in a hospital:
    - as represented also by their patient medical records,
  - (iii) documents related to public government:
    - citizen petitions, law drafts, laws, administrative forms, letters between state and local government adminsitrators and between these and citizens, court verdicts, etc., and
  - (iv) bus timetables,
    - as part of contracts for a company to provide bus servises.

- The *permissions* and *obligations* issues are:
  - (i) for the owner (agent) of some intellectual property to be paid (i.e., an *obligation*) by users when they perform *permitted* operations (rendering, copying, editing, sub-licensing) on their works;
  - (ii) for the patient to be professionally treated — by medical staff who are basically *obliged* to try to cure the patient;
  - (iii) for public administrators and citizens to enjoy good governance: transparency in law making (national parliaments and local prefectures and city councils), in law enforcement (i.e., the daily administration of laws), and law interpretation (the judiciary) — by agents who are basically *obliged* to produce certain documents while being *permitted* to consult (i.e., read, perhaps copy) other documents;
  - (iv) for citizens to enjoy timely and reliable bus services and the local government to secure adequate price-performance standards.

Example 63 – A Health Care License Language:

- Citizens
  - go to hospitals
  - in order to be treated for some calamity (disease or other),
  - and by doing so these citizens become patients.
- At hospitals patients, in a sense, issue a request to be treated with the aim of full or partial restitution.
- This request is directed at medical staff, that is,
  - the patient authorises medical staff to perform a set of actions upon the patient.
  - One could claim, as we shall, that the patient issues a license.

⊕ *Patients and Patient Medical Records* ⊕

- So patients and their attendant patient medical records (PMRs) are the main entities, the "works" of this domain.
- We shall treat them synonymously: PMRs as surrogates for patients.
- Typical actions on patients — and hence on PMRs — involve
  - admitting patients,
  - interviewing patients,
  - analysing patients,
  - diagnosing patients,
  - planning treatment for patients,
  - actually treating patients, and,
  - under normal circumstance, to finally release patients.

⊕ *Medical Staff* ⊕

- Medical staff may request ('refer' to)
  - other medical staff to perform some of these actions.
  - One can conceive of describing action sequences (and 'referrals') in the form of hospitalisation (not treatment) plans.
  - We shall call such scripts for licenses.

⊕ *Professional Health Care* ⊕

- The issue is now,
  - given that we record these licenses,
  - their being issued and being honoured,
  - whether the handling of patients at hospitals
    - ∗ follow,
    - ∗ or does not follow

    properly issued licenses.

⊕ *A Notion of License Execution State* ⊕

- In the context of the Artistic License Language licensees could basically perform licensed actions in any sequence and as often as they so desired.
  - There were, of course, some obvious constraints.
    - ∗ Operations on local works could not be done before these had been created — say by copying.
    - ∗ Editing could only be done on local works and hence required a prior action of, for example, copying a licensed work.
- In the context of hospital health care most of the actions can only be performed if the patient has reached a suitable state in the hospitalisation.
- We refer to Fig. 13 on the next slide for an idealised hospitalisation plan.

(7. **Domain Engineering** 7.7. **Domain Scripts, Licenses and Contracts**)

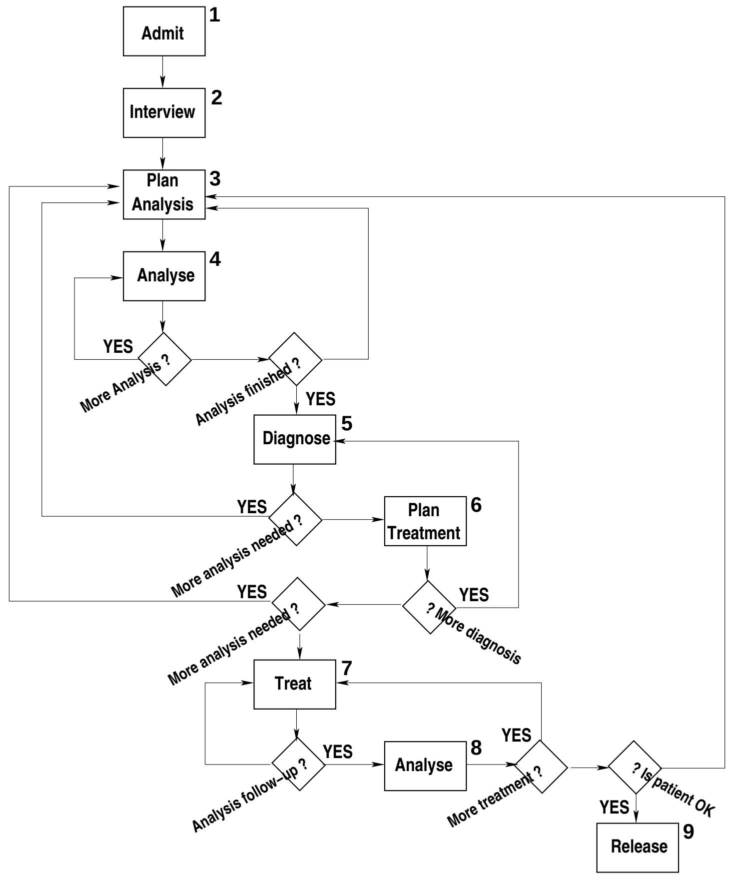


Figure 13: An example hospitalisation plan. States: {1,2,3,4,5,6,7,8,9}

- We therefore suggest
  - to join to the licensed commands
  - an indicator which prescribe the (set of) state(s) of the hospitalisation plan in which the command action may be performed.
- Two or more medical staff may now be licensed
  - to perform different (or even same !) actions
  - in same or different states.
  - If licensed to perform same action(s) in same state(s) —
  - well that may be "bad license programming" if and only if it is bad medical practice !
- One cannot design a language and prevent it being misused!

⊕ *The License Language* ⊕

- The syntax has two parts.
  - One for licenses being issued by licensors.
  - And one for the actions that licensees may wish to perform.

type

0. Ln, Mn, Pn
1. License = Ln × Lic
2. Lic == mkLic(staff1:Mn,mandate:ML,pat:Pn)
3. ML == mkML(staff2:Mn,to_perform_acts:CoL-**set**)
4 CoL = Cmd | ML | Alt
5. Cmd == mkCmd(σs:Σ-**set**,stmt:Stmt)
6 Alt == mkAlt(cmds:Cmd-**set**)
7. Stmt = **admit** | **interview** | **plan-analysis** | **do-analysis** | **diagnose** | **plan-treatment** | **treat** | **transfer** | **release**

- The above syntax is correct RSL.
- But it is decorated!
- The subtypes {|**boldface keyword**|} are inserted for readability.
- (0.) Licenses, medical staff and patients have names.
- (1.) Licenses further consist of license bodies (Lic).
- (2.) A license body names the licensee (Mn), the patient (Pn), and,
- (3.) through the "mandated" licence part (ML), it names the licensor (Mn) and which set of commands (C) or (o) implicit licenses (L, for CoL) the licensor is mandated to issue.
- (4.) An explicit command or licensing (CoL) is either a command (Cmd), or a sub-license (ML) or an alternative.

- (5.) A command (Cmd) is a state-labelled statement.
- (3.) A sub-license just states the command set that the sub-license licenses.
  - As for the Artistic License Language the licensee
  - chooses an appropriate subset of commands.
  - The context "inherits" the name of the patient.
  - But the sub-licensee is explicitly mandated in the license!
- (6.) An alternative is also just a set of commands.
  - The meaning is that
    - either the licensee choose to perform the designated actions
    - or, as for ML, but now freely choosing the sub-licensee,
    - the licensee (now new licensor) chooses to confer actions to other staff.

- (7.) A statement is either
  - an admit,
  - an interview,
  - a plan analysis,
  - an analysis,
  - a diagnose,
  - a plan treatment,
  - a treatment,
  - a transfer, or
  - a release

  directive
- Information given in the patient medical report
  - for the designated state
  - inform medical staff as to the details
  - of analysis, what to base a diagnosis on, of treatment, etc.

8. Action = Ln × Act
9. Act = Stmt | SubLic
10. SubLic = mkSubLic(sublicensee:Ln,license:ML)

- (8.) Each action actually attempted by a medical staff refers to the license, and hence the patient name.
- (9.) Actions are either of
  - an admit,
  - an interview,
  - a plan analysis,
  - an analysis,
  - a diagnose,
  - a plan treatment,
  - a treatment,
  - a transfer, or
  - a release

  actions.

- Each individual action is only allowed in a state σ
  - if the action directive appears in the named license
  - and the patient (medical record) designates state σ.
- (10.) Or an action can be a sub-licensing action.
  - Either the sub-licensing action that the licensee is attempting is explicitly mandated by the license (4. ML),
  - or is an alternative one thus implicitly mandated (6.).
  - The full sub-license, as defined in (1.–3.) is compiled from contextual information.

This ends Example 63 ■

Example 64 – A Public Administration License Language:

⊕ *The Three Branches of Government* ⊕

- By public government we shall,
  - following Charles de Secondat, baron de Montesquieu (1689–1755),
  - understand a composition of three powers:
    * the law-making (legislative),
    * the law-enforcing and
    * the law-interpreting

    parts of public government.
- Typically
  - national parliament and local (province and city) councils are part of law-making government,
  - law-enforcing government is called the executive (the administration),
  - and law-interpreting government is called the judiciary [system] (including lawyers etc.).

⊕ *Documents* ⊕

- A crucial means of expressing public administration is through *documents.*
- We shall therefore provide a brief domain analysis of a concept of documents.
- (This document domain description also applies
  - to patient medical records and,
  - by some "light" interpretation, also to artistic works —

  insofar as they also are documents.)

- Documents are
  - *created*,
  - *edited* and
  - *read*;
- and documents can be
  - *copied*,
  - *distributed*,
  - the subject of *calculations* (interpretations) and be
  - *shared* and
  - *shredded*

.

⊕ *Document Attributes* ⊕

- With documents one can associate, as attributes of documents, the *actors* who
  - created,
  - edited,
  - read,
  - copied,
  - distributed
    - (to whom distributed),
  - shared,
  - performed calculations and
  - shredded

  documents.
- With these operations on documents,
- and hence as attributes of documents one can, again conceptually,
- associate the
  - *location* and
  - *time*

  of these operations.

⊕ *Actor Attributes and Licenses* ⊕

- With actors (whether agents of public government or citizens)
  - one can associate the *authority* (i.e., the *rights*)
  - these actors have with respect to performing actions on documents.
- We now intend to express these *authorisations as licenses*.

⊕ *Document Tracing* ⊕

- An issue of public government is
  - whether citizens and agents of public government act in accordance with the laws —
  - with actions and laws reflected in documents
  - such that the action documents enables a trace from the actions to the laws "governing" these actions.

- We shall therefore assume that every document can be traced
  - back to its law-origin
  - as well as to all the documents any one document-creation or -editing was based on.

⊕ A Document License Language ⊕

- The syntax has two parts.
  - One for licenses being issued by licensors.
  - And one for the actions that licensees may wish to perform.

type

0. Ln, An, Cfn
1. L == Grant | Extend | Restrict | Withdraw
2. Grant == mkG(license:Ln,licensor:An,granted_ops:Op-**set**,licensee:An)
3. Extend == mkE(licensor:An,licensee:An,license:Ln,with_ops:Op-**set**)
4. Restrict == mkR(licensor:An,licensee:An,license:Ln,to_ops:Op-**set**)
5. Withdraw == mkW(licensor:An,licensee:An,license:Ln)
6. Op == Crea|Edit|Read|Copy|Licn|Shar|Rvok|Rlea|Rtur|Calc|Shrd

type

7. Dn, DCn, UDI
8. Crea == mkCr(dn:Dn,doc_class:DCn,based_on:UDI-**set**)
9. Edit == mkEd(doc:UDI,based_on:UDI-**set**)
10. Read == mkRd(doc:UDI)
11. Copy == mkCp(doc:UDI)

12a. Licn == mkLi(kind:LiTy)

12b. LiTy == grant | extend | restrict | withdraw

13. Shar == mkSh(doc:UDI,with:An-**set**)
14. Rvok == mkRv(doc:UDI,from:An-**set**)
15. Rlea == mkRl(dn:Dn)
16. Rtur == mkRt(dn:Dn)
17. Calc == mkCa(fcts:CFn-**set**,docs:UDI-**set**)
18. Shrd == mkSh(doc:UDI)

- (0.) The are names of licenses (Ln), actors (An), documents (UDI), document classes (DCn) and calculation functions (Cfn).
- (1.) There are four kinds of licenses: granting, extending, restricting and withdrawing.
- (2.) Actors (licensors) grant licenses to other actors (licensees).
  - An actor is constrained to always grant distinctly named licenses.
  - No two actors grant identically named licenses.
  - A set of operations on (named) documents are granted.

- (3.–5.) Actors who have issued named licenses may extend, restrict or withdraw the license rights (wrt. operations, or fully).
- (6.) There are nine kinds of operation authorisations. Some of the next explications also explain parts of some of the corresponding actions (see (16.–24.).
- (7.) There are names of documents (Dn), names of classes of documents (DCn), and there are unique document identifiers (UDI).

- (8.) **Creation** results in an initially void document which is
  - not necessarily uniquely named (dn:Dn) (but that name is uniquely associated with the unique document identifier created when the document is created)
  - typed by a document class name (dcn:DCn) and possibly
  - based on one or more identified documents (over which the licensee (at least) has reading rights).
  - We can presently omit consideration of the document class concept.
  - "based on" means that the initially void document contains references to those (zero, one or more) documents.
  - The "based on" documents are moved from licensor to licensee.

- (9.) **Editing** a document
  - may be based on "inspiration" from, that is, with reference to a number of other documents (over which the licensee (at least) has reading rights).
  - What this "be based on" means is simply that the edited document contains those references. (They can therefore be traced.)
  - The "based on" documents are moved from licensor to licensee
    - ∗ if not already so moved as the result of the specification of other authorised actions.

- (10.) **Reading** a document
  - only changes its “having been read” status.
  - The read document, if not the result of a copy, is moved from licensor to licensee — if not already so moved as the result of the specification of other authorised actions.

- (11.) **Copying** a document
  - increases the document population by exactly one document.
  - All previously existing documents remain unchanged except that the document which served as a master for the copy has been so marked.
  - The copied document is like the master document except that the copied document is marked to be a copy.
  - The master document, if not the result of a create or copy, is moved from licensor to licensee
    * if not already so moved as the result of the specification of other authorised actions.

- (12a.) A licensee can **sub-license** (sL) certain operations to be performed by other actors.
- (12b.) The granting, extending, restricting or withdrawing permissions,
  - cannot name a license (the user has to do that),
  - do not need to refer to the licensor (the licensee issuing the sub-license),
  - and leaves it open to the licensor to freely choose a licensee.
  - The licensor (the licensee issuing the sub-license) must choose a unique license name.

- (13.) A document can be **shared**
  - between two or more actors.
  - One of these is the licensee, the others are implicitly given read authorisations.
  - (One could think of extending, instead the licensing actions with a **shared** attribute.)
  - The shared document, if not the result of a create and edit or copy, is moved from licensor to licensee — if not already so moved as the result of the specification of other authorised actions.
  - Sharing a document does not move nor copy it.

- (14.) Sharing documents can be **revoked**. That is, the reading rights are removed.
- (15.) The **release** operation:
  - if a licensor has authorised a licensee to create a document
  - (and that document, when created got the unique document identifier udi:UDI)
  - then that licensee can **release** the created, and possibly edited document (by that identification)
  - to the licensor, say, for comments.
  - The licensor thus obtains the master copy.

- (16.) The **return** operation:
  - if a licensor has authorised a licensee to create a document
  - (and that document, when created got the unique document identifier udi:UDI)
  - then that licensee can **return** the created, and possibly edited document (by that identification)
  - to the licensor — "for good"!
  - The licensee relinquishes all control over that document.

- (17.) Two or more documents can be subjected to any one of a set of permitted **calculation** functions.
  - These documents, if not the result of a creates and edits or copies, are moved from licensor to licensee —
  - if not already so moved as the result of the specification of other authorised actions.
  - Observe that there can be many calculation permissions, over overlapping documents and functions.
- (18.) A document can be **shredded**.
  - It seems pointless to shred a document if that was the only right granted wrt. document.

17. Action = Ln × Clause
18. Clause = Cre | Edt | Rea | Cop | Lic | Sha | Rvk | Rel | Ret | Cal | Shr
19. Cre == mkCre(dcn:DCn,based_on_docs:UID-**set**)
20. Edt == mkEdt(uid:UID,based_on_docs:UID-**set**)
21. Rea == mkRea(uid:UID)
22. Cop == mkCop(uid:UID)
23. Lic == mkLic(license:L)
24. Sha == mkSha(uid:UID,with:An-**set**)
25. Rvk == mkRvk(uid:UID,from:An-**set**)
25. Rev == mkRev(uid:UID,from:An-**set**)
26. Rel == mkRel(dn:Dn,uid:UID)
27. Ret == mkRet(dn:Dn,uid:UID)
28. Cal == mkCal(fct:Cfn,over_docs:UID-**set**)
29. Shr == mkShr(uid:UID)

- A clause elaborates to a state change and usually some value.
- The value yielded by elaboration of the above
  - create, copy, and calculation

  clauses
- are **unique document identifiers**.
- These are chosen by the "system".

- (17.) Actions are **tagged** by the name of the license
  - with respect to which their authorisation and document names has to be checked.
  - No action can be performed by a licensee
  - unless it is so authorised by the named license,
  - both as concerns the operation (create, edit, read, copy, license, share, revoke, calculate and shred)
  - and the documents actually named in the action.
  - They must have been mentioned in the license,
  - or, created or copies of downloaded (and possibly edited) documents or copies of these — in which cases operations are inherited.

- (19.) A licensee may **create** documents if so licensed —
  - and obtains all operation authorisations to this document.
- (20.) A licensee may **edit** "downloaded" (edited and/or copied) or created documents.
- (21.) A licensee may **read** "downloaded" (edited and/or copied) or created and edited documents.
- (22.) A licensee may (conditionally) **copy** "downloaded" (edited and/or copied) or created and edited documents.
  - The licensee decides which name to give the new document, i.e., the copy.
  - All rights of the master are inherited to the copy.

- (23.) A licensee may **issue licenses**
  - of the kind permitted.
  - The licensee decides whether to do so or not.
  - The licensee decides
    - to whom,
    - over which, if any, documents,
    - and for which operations.
  - The licensee looks after a proper ordering of licensing commands:
    - first grant,
    - then sequences of zero, one or more either extensions or restrictions,
    - and finally, perhaps, a withdrawal.

- (24.) A "downloaded" (possibly edited or copied) document may (conditionally) be **shared** with one or more other actors.
  - Sharing, in a digital world, for example,
  - means that any edits done after the opening of the sharing session,
  - can be read by all so-granted other actors.

- (25.) Sharing may (conditionally) be **revoked**, partially or fully, that is, wrt. original “sharers”.
- (26.) A document may be **released**.
  - It means that the licensor who originally requested
  - a document (named dn:Dn) to be created
  - now is being able to see the results —
  - and is expected to comment on this document
  - and eventually to re-license the licensee to further work.

- (27.) A document may be **returned**.
  - It means that the licensor who originally requested
  - a document (named dn:Dn) to be created
  - is now given back the full control over this document.
  - The licensee will no longer operate on it.

- (28.) A license may (conditionally) apply any of a licensed set of **calculation functions**
  - to "downloaded" (edited, copied, etc.) documents,
  - or can (unconditionally) apply any of a licensed set of calculation functions
  - to created (etc.) documents.
  - The result of a calculation is a document.
  - The licensee obtains all operation authorisations to this document (— as for created documents).
- (29.) A license may (conditionally) **shred** a "downloaded" (etc.) document.

This ends Example 64 ■

Example 65 – A Bus Services Contract Language:

- In a number of steps
  - (‘A Synopsis’,
  - ‘A Pragmatics and Semantics Analysis’, and
  - ‘Contracted Operations, An Overview’)
- we arrive at a sound basis from which to formulate the narrative.
  - We shall, however, forego such a detailed narrative.
  - Instead we leave that detailed narrative to the student.
  - (The detailed narrative can be “derived” from the formalisation.)

⊕ *A Synopsis* ⊕

- Contracts obligate transport companies to deliver bus traffic according to a timetable.
- The timetable is part of the contract.
- A contractor may sub-contract (other) transport companies to deliver bus traffic according to timetables that are sub-parts of their own timetable.
- Contractors are either public transport authorities or contracted transport companies.
- Contracted transport companies may cancel a subset of bus rides provided the total amount of cancellations per 24 hours for each bus line does not exceed a contracted upper limit.
- The cancellation rights are spelled out in the contract.
- A sub-contractor cannot increase a contracted upper limit for cancellations above what the sub-contractor was told (in its contract) by its contractor.
- Etcetera.

⊕ *A Pragmatics and Semantics Analysis* ⊕

- The "works" of the bus transport contracts are two:
  - the timetables and, implicitly,
  - the designated (and obligated) bus traffic.
- A bus timetable appears to define one or more bus lines,
  - with each bus line giving rise to one or more bus rides.
- Nothing is (otherwise) said about regularity of bus rides.
- It appears that bus ride cancellations must be reported back to the contractor.
  - And we assume that cancellations by a sub-contractor is further reported back also to the sub-contractor's contractor.
  - Hence eventually that the public transport authority is notified.

- Nothing is said, in the contracts, such as we shall model them,
  - about passenger fees for bus rides
  - nor of percentages of profits (i.e., royalties) to be paid back from a sub-contractor to the contractor.
- So we shall not bother, in this example, about transport costs nor transport subsidies.
- The opposite of cancellations appears to be 'insertion' of extra bus rides,
  - that is, bus rides not listed in the time table,
  - but, perhaps, mandated by special events
  - We assume that such insertions must also be reported back to the contractor.

- We assume concepts of acceptable and unacceptable bus ride delays.
  - Details of delay acceptability may be given in contracts,
    * but we ignore further descriptions of delay acceptability.
    * but assume that unacceptable bus ride delays are also to be (iteratively) reported back to contractors.
- We finally assume that sub-contractors cannot (otherwise) change timetables.
  - (A timetable change can only occur after, or at, the expiration of a license.)
- Thus we find that contracts have definite period of validity.
  - (Expired contracts may be replaced by new contracts, possibly with new timetables.)

⊕ *Contracted Operations, An Overview* ⊕

- So these are the operations that are allowed by a contractor according to a contract:
  - (i) *start:* to perform, i.e., to start, a bus ride (obligated);
  - (ii) *cancel:* to cancel a bus ride (allowed, with restrictions);
  - (iii) *insert:* to insert a bus ride; and
  - (iv) *subcontract:* to sub-contract part or all of a contract.

⊕ *Syntax* ⊕

- We treat separately,
  - the syntax of contracts (for a schematised example see Slide 552) and
  - the syntax of the actions implied by contracts (for schematised examples see Slide 556).

Contracts

- An example contract can be ‘schematised’:

  cid: **contractor** cor **contracts sub-contractor** cee
  to perform operations
  `{"start","cancel","insert","subcontract"}`
  with respect to timetable tt.

- We assume a context (a global state)
  - in which all contract actions (including contracting) takes place
  - and in which the implicit net is defined.

184. contracts, contractors and sub-contractors have unique identifiers CId, CNm, CNm.

185. A **contract** has a unique **identification**, names the **contractor** and the **sub-contractor** (and we assume the contractor and sub-contractor names to be distinct). A **contract** also specifies a contract **body**.

186. A **contract body** stipulates a timetable and the set of **op**erations that are mandated or allowed by the **contractor**.

187. An **Op**eration is either a `"start"` (i.e., start a bus ride), a bus ride `"cancel"`lation, a bus ride `"insert"`, or a `"subcontract"`ing operation.

type

184. CId, CNm

185. Contract = CId × CNm × CNm × Body

186. Body = Op-**set** × TT

187. Op == "`start`" | "`cancel`" | "`insert`" | "`subcontract`"

An abstract example contract:

$$(\mathrm{cid},\mathrm{cnm}_i,\mathrm{cnm}_j,(\{\text{"}\mathtt{start}\text{"},\text{"}\mathtt{cancel}\text{"},\text{"}\mathtt{insert}\text{"},\text{"}\mathtt{sublicense}\text{"}\},\mathrm{tt}))$$

Actions

- Concrete example actions can be schematised:

(a) cid: **conduct bus ride** (blid,bid) **to start at time** t
(b) cid: **cancel bus ride** (blid,bid) **at time** t
(c) cid: **insert bus ride like** (blid,bid) **at time** t

- The schematised license (Slide 552) shown earlier is almost like an action; here is the action form:

(d) cid: **sub-contractor** cnm′ **is granted a contract** cid′
to perform operations {"conduct","cancel","insert",sublicense"}
with respect to timetable tt′.

7.7.0.1. Actions

- All actions are being performed by a sub-contractor in a context which defines
  - that sub-contractor **cnm**,
  - the relevant net, say **n**,
  - the base contract, referred here to by **cid** (from which this is a sublicense), and
  - a timetable **tt** of which **tt′** is a subset.
- contract name **cnm′** is new and is to be unique.
- The subcontracting action can (thus) be simply transformed into a contract as shown on Slide 552.

type

$\mathrm{Action} = \mathrm{CNm} \times \mathrm{CId} \times (\mathrm{SubCon} \mid \mathrm{SmpAct}) \times \mathrm{Time}$

$\mathrm{SmpAct} = \mathrm{Start} \mid \mathrm{Cancel} \mid \mathrm{Insert}$

$\mathrm{Conduct} == \mathrm{mkSta}(\mathrm{s\_blid}{:}\mathrm{BLId},\mathrm{s\_bid}{:}\mathrm{BId})$

$\mathrm{Cancel} == \mathrm{mkCan}(\mathrm{s\_blid}{:}\mathrm{BLId},\mathrm{s\_bid}{:}\mathrm{BId})$

$\mathrm{Insert} = \mathrm{mkIns}(\mathrm{s\_blid}{:}\mathrm{BLId},\mathrm{s\_bid}{:}\mathrm{BId})$

$\mathrm{SubCon} == \mathrm{mkCon}(\mathrm{s\_cid}{:}\mathrm{CId},\mathrm{s\_cnm}{:}\mathrm{CNm},\mathrm{s\_body}{:}(\mathrm{s\_ops}{:}\mathrm{Op}\text{-}\mathbf{set},\mathrm{s\_tt}{:}\mathrm{TT}))$

examples:

(a) $(\mathrm{cnm},\mathrm{cid},\mathrm{mkSta}(\mathrm{blid},\mathrm{id}),\mathrm{t})$

(b) $(\mathrm{cnm},\mathrm{cid},\mathrm{mkCan}(\mathrm{blid},\mathrm{id}),\mathrm{t})$

(c) $(\mathrm{cnm},\mathrm{cid},\mathrm{mkIns}(\mathrm{blid},\mathrm{id}),\mathrm{t})$

(d) $(\mathrm{cnm},\mathrm{cid},\mathrm{mkCon}(\mathrm{cid}',(\{''\texttt{conduct}'',''\texttt{cancel}'',''\texttt{insert}'',''\texttt{sublicense}''\},\mathrm{tt}'),\mathrm{t}))$

where: $\mathrm{cid}' = \mathrm{generate\_CId}(\mathrm{cid},\mathrm{cnm},\mathrm{t})$ See Item/Line 190 on Slide 562

- We observe that
  - the essential information given in the start, cancel and insert action prescriptions is the same;
  - and that the RSL record-constructors (mkSta, mkCan, mkIns) make them distinct.

Uniqueness and Traceability of Contract Identifications

188. There is a “root” contract name, rcid.
189. There is a “root” contractor name, rcnm.

value
188 rcid:CId
189 rcnm:CNm

- All other contract names are derived from the root name.
- Any contractor can at most generate one contract name per time unit.
- Any, but the root, sub-contractor obtains contracts from other sub-contractors, i.e., the contractor. Eventually all sub-contractors, hence contract identifications can be referred back to the root contractor.

190. Such a contract name generator is a function which given a contract identifier, a sub-contractor name and the time at which the new contract identifier is generated, yields the unique new contract identifier.

191. From any but the root contract identifier one can observe the contract identifier, the sub-contractor name and the time that "went into" its creation.

value

190 gen_CId: CId × CNm × Time → CId
191 obs_CId: CId $\stackrel{\sim}{\rightarrow}$ CIdL [**pre** obs_CId(cid):cid≠rcid]
191 obs_CNm: CId $\stackrel{\sim}{\rightarrow}$ CNm [**pre** obs_CNm(cid):cid≠rcid]
191 obs_Time: CId $\stackrel{\sim}{\rightarrow}$ Time [**pre** obs_Time(cid):cid≠rcid]

192. All contract names are unique.

axiom

192 $\forall$ cid,cid′:CId•cid≠cid′⇒

192 obs_CId(cid)≠obs_CId(cid′) ∨ obs_CNm(cid)≠obs_CNm(cid′)

192 ∨ obs_LicNm(cid)=obs_CId(cid′)∧obs_CNm(cid)=obs_CNm(cid′)

192 ⇒ obs_Time(cid)≠obs_Time(cid′)

193. Thus a contract name defines a trace of license name, sub-contractor name and time triple, "all the way back" to "creation".

```
type
  CIdCNmTTrace = TraceTriple*
  TraceTriple == mkTrTr(CId,CNm,s_t:Time)
value
193 contract_trace: CId → LCIdCNmTTrace
193 contract_trace(cid) ≡
193   case cid of
193     rcid → ⟨⟩,
193     _ → contract_trace(obs_LicNm(cid))^⟨obs_TraceTriple(cid)⟩
193   end

193 obs_TraceTriple: CId → TraceTriple
193 obs_TraceTriple(cid) ≡
193   mkTrTr(obs_CId(cid),obs_CNm(cid),obs_Time(cid))
```

- The trace is generated in the chronological order: most recent contract name generation times last.
- Well, there is a theorem to be proven once we have outlined the full formal model of this contract language:
- namely that time entries in contract name traces increase with increasing indices.

theorem

$$\forall\ \text{licn:LicNm} \cdot$$
$$\quad \forall\ \text{trace:LicNmLeeNmTimeTrace} \cdot \text{trace} \in \text{license\_trace(licn)} \Rightarrow$$
$$\quad\quad \forall\ \text{i:}\mathbf{Nat} \cdot \{\text{i,i+1}\} \subseteq \mathbf{inds}\ \text{trace} \Rightarrow \text{s\_t(trace(i))} < \text{s\_t(trace(i+1))}$$

⊕ *Execution State* ⊕

Local and Global States

- Each sub-contractor has an own local state and has access to a global state.
- All sub-contractors access the same global state.
- The global state is the bus traffic on the net.
- There is, in addition, a notion of running-state. It is a meta-state notion.
  - The running state "is made up" from the fact that
  - there are n sub-contractors, each communicating, as contractors,
  - over channels with other sub-contractors.
- The global state is distinct from sub-contractor to sub-contractor – no sharing of local states between sub-contractors.
- We now examine, in some detail, what the states consist of.

Global State

- The net is part of the global state (and of bus traffics).
- We consider just the bus traffic.

194. Bus traffic is a modelled as a discrete function from densely positioned time points to a pair of the (possibly dynamically changing) net and the position of busses. Bus positions map bus numbers to the physical entity of busses and their position.
195. A bus is positioned either
196. at a hub (coming from some link heading for some link), or
197. on a link, some fraction of the distance from a hub towards a hub, or
198. at a bus stop, some fraction of the distance from a hub towards a hub.

type

136. BusStop == mkBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)

194. BusTraffic = T $\overrightarrow{m}$ (N $\times$ (BusNo $\overrightarrow{m}$ (Bus $\times$ BPos)))
195. BPos = atHub | onLnk | atBS
196. atHub == mkAtHub(s_fl:LI,s_hi:HI,s_tl:LI)
197. onLnk == mkOnLnk(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
198. atBSt == mkAtBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)

    Frac = {|f:**Real**•0<f<1|}

- We shall consider BusTraffic (with its Net) to reflect the global state.

Local Sub-contractor Contract States: Semantic Types

- A sub-contractor state contains, as a state component, the zero, one or more contracts
  - that the sub-contractor has received and
  - that the sub-contractor has sublicensed.

type

$\text{Body} = \text{Op-}\textbf{set} \times \text{TT}$
$\text{Lic}\Sigma = \text{RcvLic}\Sigma \times \text{SubLic}\Sigma \times \text{LorBus}\Sigma$
$\text{RcvLic}\Sigma = \text{LorNm} \xrightarrow[m]{} (\text{LicNm} \xrightarrow[m]{} (\text{Body} \times \text{TT}))$
$\text{SubLic}\Sigma = \text{LeeNm} \xrightarrow[m]{} (\text{LicNm} \xrightarrow[m]{} \text{Body})$
$\text{LorBus}\Sigma$ ... [see "`Local sub-contractor Bus States: Semantic Types`" next] ...

- (Recall that LorNm and LeeNm are the same.)

Local Sub-contractor Bus States: Semantic Types

- The sub-contractor state further contains a bus status state component which records
  - which buses are free, FreeBusΣ, that is, available for dispatch, and where "garaged",
  - which are in active use, ActvBusΣ, and on which bus ride, and a bus history for that bus ride,
  - and histories of all past bus rides, BusHistΣ.
  - A trace of a bus ride is a list of zero, one or more pairs of times and bus stops.
  - A bus history, BusHistory, associates a bus trace to a quadruple of bus line identifiers, bus ride identifiers, contract names and sub-contractor name.

type

BusNo

$\text{Bus}\Sigma = \text{FreeBuses}\Sigma \times \text{ActvBuses}\Sigma \times \text{BusHists}\Sigma$

$\text{FreeBuses}\Sigma = \text{BusStop} \xrightarrow[m]{} \text{BusNo-}\mathbf{set}$

$\text{ActvBuses}\Sigma = \text{BusNo} \xrightarrow[m]{} \text{BusInfo}$

$\text{BusInfo} = \text{BLId}\times\text{BId}\times\text{LicNm}\times\text{LeeNm}\times\text{BusTrace}$

$\text{BusHists}\Sigma = \text{Bno} \xrightarrow[m]{} \text{BusInfo}^*$

$\text{BusTrace} = (\text{Time}\times\text{BusStop})^*$

$\text{LorBus}\Sigma = \text{LeeNm} \xrightarrow[m]{} (\text{LicNm} \xrightarrow[m]{} ((\text{BLId}\times\text{BId}) \xrightarrow[m]{} (\text{BNo}\times\text{BusTrace})))$

- A bus is identified by its unique number (i.e., registration) plate (BusNo).
- The two components are modified whenever a bus is commissioned into action or returned from duty, that is, twice per bus ride.

Local Sub-contractor Bus States: Update Functions

value

$\textbf{update\_Bus}\Sigma$: $\mathrm{Bno}\times(\mathrm{T}\times\mathrm{BusStop}) \rightarrow \mathrm{ActBus}\Sigma \rightarrow \mathrm{ActBus}\Sigma$

$\textbf{update\_Bus}\Sigma(\mathrm{bno},(\mathrm{t},\mathrm{bs}))(\mathrm{act}\sigma) \equiv$

$\quad\textbf{let}\ (\mathrm{blid},\mathrm{bid},\mathrm{licn},\mathrm{leen},\mathrm{trace}) = \mathrm{act}\sigma(\mathrm{bno})\ \textbf{in}$

$\quad\mathrm{act}\sigma\dagger[\,\mathrm{bno}\mapsto(\mathrm{licn},\mathrm{leen},\mathrm{blid},\mathrm{bid},\mathrm{trace}^\frown\langle(\mathrm{t},\mathrm{bs})\rangle)\,]\ \textbf{end}$

$\quad\textbf{pre}\ \mathrm{bno} \in \textbf{dom}\ \mathrm{act}\sigma$

$\textbf{update\_Free}\Sigma\textbf{\_Act}\Sigma$:

$\quad\mathrm{BNo}\times\mathrm{BusStop}\rightarrow\mathrm{Bus}\Sigma\rightarrow\mathrm{Bus}\Sigma$

$\textbf{update\_Free}\Sigma\textbf{\_Act}\Sigma(\mathrm{bno},\mathrm{bs})(\mathrm{free}\sigma,\mathrm{actv}\sigma) \equiv$

$\quad\textbf{let}\ (\_,\_,\_,\_,\mathrm{trace}) = \mathrm{act}\sigma(\mathrm{b})\ \textbf{in}$

$\quad\textbf{let}\ \mathrm{free}\sigma' = \mathrm{free}\sigma\dagger[\,\mathrm{bs} \mapsto (\mathrm{free}\sigma(\mathrm{bs}))\cup\{\mathrm{b}\}\,]\ \textbf{in}$

$\quad(\mathrm{free}\sigma',\mathrm{act}\sigma\backslash\{\mathrm{b}\})\ \textbf{end end}$

$\quad\textbf{pre}\ \mathrm{bno} \notin \mathrm{free}\sigma(\mathrm{bs}) \wedge \mathrm{bno} \in \textbf{dom}\ \mathrm{act}\sigma$

```
update_LorBusΣ:
   LorNm×LicNm×lee:LeeNm×(BLId×BId)×(BNo×Trace)
     →LorBusΣ→out {l_to_l[leen,lorn]|lorn:LorNm·lorn ∈ leenms\{leen}}
update_LorBusΣ(lorn,licn,leen,(blid,bid),(bno,tr))(lbσ) ≡
  l_to_l[leenm,lornm]!Licensor_BusHistΣMsg(bno,blid,bid,libn,leen,tr) ;
  lbσ†[leen↦(lbσ(leen))†[licn↦((lbσ(leen))(licn))†[(blid,bid)↦(bno,trace)]
  pre leen ∈ dom lbσ ∧ licn ∈ dom (lbσ(leen))

update_ActΣ_FreeΣ:
  LeeNm×LicNm×BusStop×(BLId×BId)→BusΣ→BusΣ×BNo
update_ActΣ_FreeΣ(leen,licn,bs,(blid,bid))(freeσ,actvσ) ≡
  let bno:Bno · bno ∈ freeσ(bs) in
  ((freeσ\{bno},actvσ ∪ [bno↦(blid,bid,licnm,leenm,⟨⟩)]),bno) end
  pre bs ∈ dom freeσ ∧ bno ∈ freeσ(bs) ∧ bno ∉ dom actvσ ∧ [bs exists .
```

Constant State Values

- There are a number of constant values, of various types, which characterise the "business of contract holders". We define some of these now.

199. For simplicity we assume a constant **net** — constant, that is, only with respect to the set of identifiers links and hubs. These links and hubs obviously change state over time.
200. We also assume a constant set, **leens**, of sub-contractors. In reality sub-contractors, that is, transport companies, come and go, are established and go out of business. But assuming constancy does not materially invalidate our model. Its emphasis is on contracts and their implied actions — and these are unchanged wrt. constancy or variability of contract holders.

201. There is an initial bus traffic, tr.
202. There is an initial time, t_0, which is equal to or larger than the start of the bus traffic tr.
203. To maintain the bus traffic “spelled out”, in total, by timetable tt one needs a number of buses.
204. The various bus companies (that is, sub-contractors) each have a number of buses. Each bus, independent of ownership, has a unique (car number plate) bus number (BusNo).
   These buses have distinct bus (number [registration] plate) numbers.
205. We leave it to the student to define a function which ascertain the minimum number of buses needed to implement traffic tr.

value

199. net : N,
200. leens : LeeNm-**set**,
201. tr : BusTraffic, **axiom** wf_Traffic(tr)(net)
202. t_0 : T · $t_0 \geq$ **min dom** tr,
203. min_no_of_buses : **Nat** · necessary_no_of_buses(itt),
204. busnos : BusNo-**set** · **card** busnos ≥ min_no_of_buses
205. necessary_no_of_buses: TT → **Nat**

206. To "bootstrap" the whole contract system we need a distinguished contractor, named init_leen, whose only license originates with a "ghost" contractor, named root_leen (o, for outside [the system]).
207. The initial, i.e., the distinguished, contract has a name, root_licn.
208. The initial contract can only perform the `"sublicense"` operation.
209. The initial contract has a timetable, tt.
210. The initial contract can thus be made up from the above.

value

206. root_leen,init_ln : LeeNm • root_leen $\notin$ leens $\wedge$ initi_leen $\in$ leens,
207. root_licn : LicNm
208. iops : Op-**set** = {″`sublicense`″},
209. itt : TT,
210. init_lic:License = (root_licn,root_leen,(iops,itt),init_leen)

Initial Sub-contractor Contract States

type

$$\mathrm{InitLic\Sigma s} = \mathrm{LeeNm} \ \overrightarrow{m}\ \mathrm{Lic\Sigma}$$

value

$$\mathrm{il}\sigma\mathrm{:Lic\Sigma}=([\ \mathrm{init\_leen} \mapsto [\ \mathrm{root\_leen} \mapsto [\ \mathrm{iln} \mapsto \mathrm{init\_lic}\ ]\ ]\ ]$$
$$\cup\ [\ \mathrm{leen} \mapsto [\,]\ |\ \mathrm{leen:LeeNm} \cdot \mathrm{leen} \in \mathrm{leenms} \backslash \{\mathrm{init\_leen}\}\ ],[\,],[\,])$$

Initial Sub-contractor Bus States

211. Initially each sub-contractor possesses a number of buses.
212. No two sub-contractors share buses.
213. We assume an initial assignment of buses to bus stops of the free buses state component and for respective contracts.
214. We do not prescribe a "satisfiable and practical" such initial assignment (ibσs).
215. But we can constrain ibσs.
216. The sub-contractor names of initial assignments must match those of initial bus assignments, allbuses.
217. Active bus states must be empty.
218. No two free bus states must share buses.
219. All bus histories are void.

type

211. $\mathrm{AllBuses}' = \mathrm{LeeNm} \underset{m}{\rightarrow} \mathrm{BusNo}\text{-}\mathbf{set}$

212. $\mathrm{AllBuses} = \{|\mathrm{ab}:\mathrm{AllBuses}' \cdot \forall\ \{\mathrm{bs},\mathrm{bs}'\} \subseteq \mathbf{rng}\ \mathrm{ab} \wedge \mathrm{bns} \neq \mathrm{bns}' \Rightarrow \mathrm{bns} \cap \mathrm{bns}' = \{\}|\}$

213. $\mathrm{InitBus}\Sigma\mathrm{s} = \mathrm{LeeNm} \underset{m}{\rightarrow} \mathrm{Bus}\Sigma$

value

212. $\mathrm{allbuses}:\mathrm{Allbuses} \cdot \mathbf{dom}\ \mathrm{allbuses} = \mathrm{leenms} \cup \{\mathrm{root\_leen}\} \wedge \cup\ \mathbf{rng}\ \mathrm{allbuses} = \mathrm{busnos}$

213. $\mathrm{ib}\sigma\mathrm{s}:\mathrm{InitBus}\Sigma\mathrm{s}$
214. $\mathrm{wf\_InitBus}\Sigma\mathrm{s}: \mathrm{InitBus}\Sigma\mathrm{s} \rightarrow \mathbf{Bool}$
215. $\mathrm{wf\_InitBus}\Sigma\mathrm{s}(\mathrm{i}\sigma\mathrm{s}) \equiv$
216. $\quad \mathbf{dom}\ \mathrm{i}\sigma\mathrm{s} = \mathrm{leenms} \wedge$
217. $\quad \forall\ (\_,\mathrm{ab}\sigma,\_):\mathrm{Bus}\Sigma \cdot (\_,\mathrm{ab}\sigma,\_) \in \mathbf{rng}\ \mathrm{i}\sigma\mathrm{s} \Rightarrow \mathrm{ab}\sigma = [\,] \wedge$
218. $\quad \forall\ (\mathrm{fbi}\sigma,\mathrm{abi}\sigma),(\mathrm{fbj}\sigma,\mathrm{abj}\sigma):\mathrm{Bus}\Sigma \cdot$
218. $\qquad \{(\mathrm{fbi}\sigma,\mathrm{abi}\sigma),(\mathrm{fbj}\sigma,\mathrm{abj}\sigma)\} \subseteq \mathbf{rng}\ \mathrm{i}\sigma\mathrm{s}$
218. $\qquad\quad \Rightarrow (\mathrm{fbi}\sigma,\mathrm{acti}\sigma) \neq (\mathrm{fbj}\sigma,\mathrm{actj}\sigma)$
218. $\qquad\qquad \Rightarrow \mathbf{rng}\ \mathrm{fbi}\sigma \cap \mathbf{rng}\ \mathrm{fbj}\sigma = \{\}$
219. $\qquad\qquad\quad \wedge\ \mathrm{acti}\sigma = [\,] = \mathrm{actj}\sigma$

⊕ *Communication Channels* ⊕

- The running state is a meta notion. It reflects the channels over which
  - contracts are issued;
  - messages about committed, cancelled and inserted bus rides are communicated, and
  - fund transfers take place.

Sub-Contractor↔Sub-Contractor Channels

- Consider each sub-contractor (same as contractor) to be modelled as a behaviour.
- Each sub-contractor (licensor) behaviour has a unique name, the LeeNm.
- Each sub-contractor can potentially communicate with every other sub-contractor.
- We model each such communication potential by a channel.
- For n sub-contractors there are thus $n \times (n-1)$ channels.

channel { l_to_l[fi,ti] | fi:LeeNm,ti:LeeNm • {fi,ti}⊆leens ∧ fi≠ti } LLMSG
type LLMSG = ...

Sub-Contractor↔Bus Channels

- Each sub-contractor has a set of buses. That set may vary.
- So we allow for any sub-contractor to potentially communicate with any bus.
- In reality only the buses allocated and scheduled by a sub-contractor can be “reached” by that sub-contractor.

channel { l_to_b[l,b] | l:LeeNm,b:BNo · l ∈ leens ∧ b ∈ busnos } LBMSG
type LBMSG = ...

Sub-Contractor↔Time Channels

- Whenever a sub-contractor wishes to perform a contract operation
- that sub-contractor needs know the time.
- There is just one, the global time, modelled as one behaviour: time_clock.

channel $\{$ l_to_t[l] $|$ l:LeeNm $\cdot$ l $\in$ leens $\}$ LTMSG
type LTMSG = ...

Bus↔Traffic Channels

- Each bus is able, at any (known) time to ascertain where in the traffic it is.
- We model bus behaviours as processes, one for each bus.
- And we model global bus traffic as a single, separate behaviour.

channel { b_to_tr[b] | b:BusNo · b ∈ busnos } LTrMSG
type BTrMSG == reqBusAndPos(s_bno:BNo,s_t:Time) | (Bus×BusPos)

Buses↔Time Channel

- Each bus needs to know what time it is.

channel { b_to_t[b] | b:BNo · b ∈ busnos } BTMSG
type BTMSG ...

⊕ *Run-time Environment* ⊕

:

- So we shall be modelling the transport contract domain as follows:
  - As for behaviours we have this to say.
    * There will be n sub-contractors. One sub-contractor will be initialised to one given license.
    * Each sub-contractor is modelled, in RSL, as a CSP-like process.
    * With each sub-contractor, l_i, there will be a number, b_i, of buses. That number may vary from sub-contractor to sub-contractor.
    * There will be b_i channels of communication between a sub-contractor and that sub-contractor's buses, for each sub-contractor.
    * There is one global process, the traffic. There is one channel of communication between a sub-contractor and the traffic. Thus there are n such channels.

– As for operations, including behaviour interactions we assume the following.
  * All operations of all processes are to be thought of as instantaneous, that is, taking nil time !
  * Most such operations are the result of channel communications
    · either just one-way notifications,
    · or inquiry requests.
  * Both the former (the one-way notifications) and the latter (inquiry requests) must not be indefinitely barred from receipt, otherwise holding up the notifier.
  * The latter (inquiry requests) should lead to rather immediate responses, thus must not lead to dead-locks.

⊕ *The System Behaviour* ⊕

- The system behaviour starts by establishing a number of
  - licenseholder
  - and
  - bus_ride

  behaviours and the single
  - time_clock
  - and
  - bus_traffic

  behaviours

$$
\begin{array}{l}
\textbf{value} \\
\quad \textsf{system}: \ \textbf{Unit} \to \textbf{Unit} \\
\quad \textsf{system}() \equiv \\
\quad\quad \textbf{licenseholder}(\mathrm{init\_leen})(\mathrm{il}\sigma(\mathrm{init\_leen}),\mathrm{ib}\sigma(\mathrm{init\_leen})) \\
\quad\quad \parallel (\parallel \{\textbf{licenseholder}(\mathrm{leen})(\mathrm{il}\sigma(\mathrm{leen}),\mathrm{ib}\sigma(\mathrm{leen})) \\
\quad\quad\quad\quad \mid \mathrm{leen{:}LeeNm{\cdot}leen} \in \mathrm{leens}\backslash\{\mathrm{init\_leen}\}\}) \\
\quad\quad \parallel (\parallel \{\textbf{bus\_ride}(\mathrm{b,leen})(\mathrm{root\_lorn},\text{"}\mathtt{nil}\text{"}) \\
\quad\quad\quad\quad \mid \mathrm{leen{:}LeeNm,b{:}BusNo} \ {\cdot}\mathrm{leen} \in \textbf{dom} \ \mathrm{allbuses} \wedge \mathrm{b} \in \mathrm{allbuses(leen)}\}) \\
\quad\quad \parallel \textbf{time\_clock}(\mathrm{t}_0) \parallel \textbf{bus\_traffic}(\mathrm{tr})
\end{array}
$$

- The initial licenseholder behaviour states are individually initialised
  - with basically empty license states and
  - by means of the global state entity bus states.
- The initial bus behaviours need no initial state.
- Only a designated licenseholder behaviour is initialised
  - to a single, received license.

⊕ *Semantic Elaboration Functions* ⊕

The Licenseholder Behaviour

220. The licenseholder behaviour is a sequential, but internally non-deterministic behaviour.

221. It internally non-deterministically ($\sqcap$) alternates between

   (a) performing the licensed operations (on the net and with buses),

   (b) receiving information about the whereabouts of these buses, and informing contractors of its (and its subsub-contractors') handling of the contracts (i.e., the bus traffic), and

   (c) negotiating new, or renewing old contracts.

220. **licenseholder**: LeeNm → (LicΣ×BusΣ) → **Unit**
221. **licenseholder**(leen)(licσ,busσ) ≡
221.   **licenseholder**(leen)((**lic_ops**$\sqcap$**bus_mon**$\sqcap$**neg_licenses**)(leen)(licσ,busσ))

The Bus Behaviour

222. Buses ply the network following a timed bus route description.

     A timed bus route description is a list of timed bus stop visits.

223. A timed bus stop visit is a pair: a time and a bus stop.

224. Given a bus route and a bus schedule one can construct a timed bus route description.

     (a) The first result element is the first bus stop and origin departure time.

     (b) Intermediate result elements are pairs of respective intermediate schedule elements and intermediate bus route elements.

     (c) The last result element is the last bus stop and final destination arrival time.

225. Bus behaviours start with a “nil” bus route description.

type

222. $\mathrm{TBR} = \mathrm{TBSV}^*$

223. $\mathrm{TBSV} = \mathrm{Time} \times \mathrm{BusStop}$

value

224. $\mathrm{conTBR}\colon \mathrm{BusRoute} \times \mathrm{BusSched} \rightarrow \mathrm{TBR}$

224. $\mathrm{conTBR}((\mathrm{dt},\mathrm{til},\mathrm{at}),(\mathrm{bs1},\mathrm{bsl},\mathrm{bsn})) \equiv$

224(a)) $\langle(\mathrm{dt},\mathrm{bs1})\rangle$

224(b)) $\widehat{\ } \langle(\mathrm{til}[\,i\,],\mathrm{bsl}[\,i\,])|i{:}\mathbf{Nat}\cdot i{:}\langle 1..\mathbf{len}\ \mathrm{til}\rangle\rangle$

224(c)) $\widehat{\ } \langle(\mathrm{at},\mathrm{bsn})\rangle$

pre: **len** til = **len** bsl

type

225. BRD == "`nil`" | TBR

226. The bus behaviour is here abstracted to only communicate with some contract holder, time and traffic,
227. The bus repeatedly observes the time, t, and its position, po, in the traffic.
228. There are now four case distinctions to be made.
229. If the bus is idle (and a a bus stop) then it waits for a next route, brd′ on which to engage.
230. If the bus is at the destination of its journey then it so informs its owner (i.e., the sub-contractor) and resumes being idle.
231. If the bus is 'en route', at a bus stop, then it so informs its owner and continues the journey.
232. In all other cases the bus continues its journey

```
value
226. bus_ride: leen:LeeNm × bno:Bno → (LicNm × BRD) →
226.    in,out l_to_b[ leen,bno ], in,out b_to_tr[ bno ], in b_to_t[ bno ] Unit
226. bus_ride(leen,bno)(licn,brd) ≡
227.   let t = b_to_t[ bno ]? in
227.   let (bus,pos) = (b_to_tr[ bno ]!reqBusAndPos(bno,t) ; b_to_tr[ bno ]?) in
228.   case (brd,pos) of
229.     ("nil",mkAtBS(_,_,_,_)) →
229.          let (licn,brd′) = (l_to_b[ leen,bno ]!reqBusRid(pos);l_to_b[ leen,bno ]?) in
229.          bus_ride(leen,bno)(licn,brd′) end
230.     (⟨(at,pos)⟩,mkAtBS(_,_,_,_)) →
230s          l_to_b[ l,b ]!BusΣMsg(t,pos);
230          l_to_b[ l,b ]!BusHistΣMsg(licn,bno);
230          l_to_b[ l,b ]!FreeΣ_ActΣMsg(licn,bno) ;
230          bus_ride(leen,bno)(ilicn,"nil"),
231.     (⟨(t,pos),(t′,bs′)⟩⌢brd′,mkAtBS(_,_,_,_)) →
231s          l_to_b[ l,b ]!BusΣMsg(t,pos) ;
231          bus_ride(licn,bno)(⟨(t′,bs′)⟩⌢brd′),
232.     _ → bus_ride(leen,bno)(licn,brd) end end end
```

- In formula line 227 of **bus_ride** we obtained the **bus**.
- But we did not use "that" bus !
- We we may wish to record, somehow, number of passengers alighting and boarding at bus stops, bus fees paid, one way or another, etc.
- The **bus**, which is a time-dependent entity, gives us that information.
- Thus we can revise formula lines 230s and 231s:

Simple: 230s l_to_b[l,b]!**BusΣMsg**(pos);
Revised: 230r l_to_b[l,b]!**BusΣMsg**(pos,**bus_info**(**bus**));

Simple: 231s l_to_b[l,b]!**BusΣMsg**(pos);
Revised: 231r l_to_b[l,b]!**BusΣMsg**(pos,**bus_info**(**bus**));

type
  Bus_Info = Passengers × Passengers × Cash × ...
value
  bus_info: Bus → Bus_Info
  bus_info(**bus**) ≡ (obs_alighted(**bus**),obs_boarded(**bus**),obs_till(**bus**),...)

The Global Time Behaviour

233. The time_clock is a never ending behaviour — started at some time t_0.
234. The time can be inquired at any moment by any of the licenseholder behaviours and by any of the bus behaviours.
235. At any moment the time_clock behaviour may not be inquired.
236. After a skip of the clock or an inquiry the time_clock behaviour continues, non-deterministically either maintaining the time or advancing the clock!

value

233. **time_clock**: T $\rightarrow$
233. **in**,**out** {l_to_t[leen] | leen:LeeNm $\cdot$ leen $\in$ leenms}
233. **in**,**out** {b_to_t[bno] | bno:BusNo $\cdot$ bno $\in$ busnos} **Unit**
233. **time_clock**:(t) $\equiv$
235. (**skip** $\sqcap$
234. ($\sqcap${l_to_t[leen]? ; l_to_t[leen]!t | leen:LeeNm$\cdot$leen $\in$ leens})
234. $\sqcap$ ($\sqcap${b_to_t[bno]? ; b_to_t[bno]!t | bno:BusNo$\cdot$bno $\in$ busnos})) ;
236. (**time_clock**:(t) $\sqcap$ **time_clock**(t+δ_t))

The Bus Traffic Behaviour

237. There is a single bus_traffic behaviour. It is, “mysteriously”, given a constant argument, “the” traffic, tr.
238. At any moment it is ready to inform of the position, bps(b), of a bus, b, assumed to be in the traffic at time t.
239. The request for a bus position comes from some bus.
240. The bus positions are part of the traffic at time t.
241. The bus_traffic behaviour, after informing of a bus position reverts to “itself”.

```
value
237. bus_traffic: TR → in,out {b_to_tr[ bno ]|bno:BusNo•bno ∈ busnos}
237. bus_traffic(tr) ≡
239.   ⌷ { let reqBusAndPos(bno,time) = b_to_tr[ b ]? in assert b=bno
238.        if time ∉ dom tr then chaos else
240.        let (_,bps) = tr(t) in
238.        if bno ∉ dom tr(t) then chaos else
238.        b_to_tr[ bno ]!bps(bno) end end end end | b:BusNo•b ∈ busnos}
241.   bus_traffic(tr)
```

License Operations

242. The lic_ops function models the contract holder choosing between and performing licensed operations.
243. To perform any licensed operation the sub-contractor needs to know the time and
244. must choose amongst the four kinds of operations that are licensed.
    - The choice function, which we do not define, makes a basically non-deterministic choice among licensed alternatives.
    - The choice yields the contract number of a received contract and,
    - based on its set of licensed operations,
    - it yields either a simple action or a sub-contracting action.
245. Thus there is a case distinction amongst four alternatives.
246. This case distinction is expressed in the four lines identified by: 246.
247. All the auxiliary functions, besides the action arguments, require the same state arguments.

value

242. **lic_ops**: LeeNm → (LicΣ×BusΣ) → (LicΣ×BusΣ)
242. **lic_ops**(leen)(licσ,busσ) ≡
243. **let** t = (time_channel(leen)!req_Time;time_channel(leen)?) **in**
244. **let** (licn,act) = choice(licσ)(busσ)(t) **in**
245. (**case** act **of**
246. mkCon(blid,bid) → **cndct**(licn,leenm,t,act),
246. mkCan(blid,bid) → **cancl**(licn,leenm,t,act),
246. mkIns(blid,bid) → **insrt**(licn,leenm,t,act),
246. mkLic(leenm′,bo) → **sublic**(licn,leenm,t,act) **end**)(licσ,busσ) **end end**

cndct,cancl,insert: SmpAct→(LicΣ×BusΣ)→(LicΣ×BusΣ)
sublic: SubLic→(LicΣ×BusΣ)→(LicΣ×BusΣ)

Bus Monitoring

- Like for the **bus_ride** behaviour we decompose the **bus_mon**itoring behaviour into two behaviours.
  - The **local_bus_mon**itoring behaviour monitors the buses that are commissioned by the sub-contractor.
  - The **licensor_bus_mon**itoring behaviour monitors the buses that are commissioned by sub-contractors sub-contractd by the contractor.

value

bus_mon: l:LeeNm $\rightarrow$ (Lic$\Sigma \times$BusΣ)
    $\rightarrow$ **in** {l_to_b[l,b]|b:BNo•b $\in$ allbuses(l)} (Lic$\Sigma \times$BusΣ)
bus_mon(l)(licσ,busσ) $\equiv$
  local_bus_mon(l)(licσ,busσ) $\sqcap$ **licensor_bus_mon**(l)(licσ,busσ)

248. The **local_bus_mon**itoring function models all the interaction between a contract holder and its despatched buses.
249. We show only the communications from buses to contract holders.
250.
251.
252.
253.
254.
255.
256.
257.
258.

```
248. local_bus_mon: leen:LeeNm → (LicΣ×BusΣ)
249.        → in {l_to_b[ leen,b ]|b:BNo•b ∈ allbuses(l)} (LicΣ×BusΣ)
248. local_bus_mon(leen)(licσ:(rlσ,slσ,lbσ),busσ:(fbσ,abσ)) ≡
250.    let (bno,msg) = ⌷{(b,l_to_b[ l,b ]?)|b:BNo•b ∈ allbuses(leen)} in
254.    let (blid,bid,licn,lorn,trace) = abσ(bno) in
251.    case msg of
252.      BusΣMsg(t,bs) →
256.        let abσ′ = update_BusΣ(bno)(licn,leen,blid,bid)(t,bs)(abσ) in
256.        (licσ,(fbσ,abσ′,histσ)) end,
258.      BusHistΣMsg(licn,bno) →
258.        let lbσ′ =
258.          update_LorBusΣ(obs_LorNm(licn),licn,leen,(blid,bid),(b,trace))(lbσ)  in
258.        l_to_l[ leen,obs_LorNm(licn) ]!Licensor_BusHistΣMsg(licn,leen,bno,blid,bid,tr
258.        ((rlσ,slσ,lbσ′),busσ) end
257.      FreeΣ_ActΣMsg(licn,bno) →
258.        let (fbσ′,abσ′) = update_FreeΣ_ActΣ(bno,bs)(fbσ,abσ) in
258.        (licσ,(fbσ′,abσ′)) end
258.    end end end
```

259.

260.

261.

262.

263.

```
259. licensor_bus_mon: lorn:LorNm → (LicΣ×BusΣ)
259.       → in {l_to_l[ lorn,leen ]|leen:LeeNm•leen ∈ leenms\{lorn}} (LicΣ×BusΣ)
259. licensor_bus_mon(lorn)(licσ,busσ) ≡
259.    let (rlσ,slσ,lbhσ) = licσ in
259.    let (leen,Licensor_BusHistΣMsg(licn,leen″,bno,blid,bid,tr))
                  = ⌷{(leen′,l_to_l[ lorn,leen′ ]?)|leen′:LeeNm•leen′ ∈ leenms\{lorn}} in
259.    let lbhσ′ =
259.       update_BusHistΣ(obs_LorNm(licn),licn,leen″,(blid,bid),(bno,trace))(lbhσ) in
259.    l_to_l[ leenm,obs_LorNm(licnm) ]!Licensor_BusHistΣMsg(b,blid,bid,lin,lee,tr);
259.    ((rlσ,slσ,lbhσ′),busσ)
259.    end end end
```

License Negotiation

264.

265.

266.

267.

268.

269.

270.

271.

272.

273.

274.

275.

264.

265.

266.

267.

268.

269.

270.

271.

272.

273.

274.

The Conduct Bus Ride Action

276. The conduct bus ride action prescribed by $(\mathsf{ln}, \mathsf{mkCon}(\mathsf{bli}, \mathsf{bi}, \mathsf{t}'))$ takes place in a context and shall have the following effect:
   (a) The action is performed by contractor li and at time t. This is known from the context.
   (b) First it is checked that the timetable in the contract named ln does indeed provide a journey, j, indexed by bli and (then) bi, and that that journey starts (approximately) at time t' which is the same as or later than t.
   (c) Being so the action results in the contractor, whose name is "embedded" in ln, receiving notification of the bus ride commitment.

(d) Then a bus, selected from a pool of available buses at the bust stop of origin of journey j, is given j as its journey script, whereupon that bus, as a behaviour separate from that of sub-contractor li, commences its ride.

(e) The bus is to report back to sub-contractor li the times at which it stops at en route bus stops as well as the number (and kind) of passengers alighting and boarding the bus at these stops.

(f) Finally the bus reaches its destination, as prescribed in j, and this is reported back to sub-contractor li.

(g) Finally sub-contractor li, upon receiving this 'end-of-journey' notification, records the bus as no longer in actions but available at the destination bus stop.

276.
276(a))
276(b))
276(c))
276(d))
276(e))
276(f))
276(g))

The Cancel Bus Ride Action

277. The cancel bus ride action prescribed by $(\mathsf{ln},\mathsf{mkCan}(\mathsf{bli},\mathsf{bi},\mathsf{t}'))$ takes place in a context and shall have the following effect:
    (a) The action is performed by contractor li and at time t. This is known from the context.
    (b) First a check like that prescribed in Item 276(b)) is performed.
    (c) If the check is OK, then the action results in the contractor, whose name is “embedded” in ln, receiving notification of the bus ride cancellation.

    That’s all !

277.

277(a))

277(b))

277(c))

The Insert Bus Ride Action

278. The insert bus ride action prescribed by (ln,mkIns(bli,bi,t′) takes place in a context and shall have the following effect:

(a) The action is performed by contractor li and at time t. This is known from the context.

(b) First a check like that prescribed in Item 276(b)) is performed.

(c) If the check is OK, then the action results in the contractor, whose name is "embedded" in ln, receiving notification of the new bus ride commitment.

(d) The rest of the effect is like that prescribed in Items 276(d))–276(g)).

278.
278(a))
278(b))
278(c))
278(d))

The Contracting Action

279. The subcontracting action prescribed by (ln,mkLic(li′,(pe′,ops′,tt′))) takes place in a context and shall have the following effect:

(a) The action is performed by contractor li and at time t. This is known from the context.

(b) First it is checked that timetable tt is a subset of the timetable contained in, and that the operations ops are a subset of those granted by, the contract named ln.

(c) Being so the action gives rise to a contract of the form (ln′,li,(pe′,ops′,tt′),li′). ln′ is a unique new contract name computed on the basis of ln, li, and t. li′ is a sub-contractor name chosen by contractor li. tt′ is a timetable chosen by contractor li. ops′ is a set of operations likewise chosen by contractor li.

(d) This contract is communicated by contractor li to sub-contractor li′.

(e) The receipt of that contract is recorded in the license state.

(f) The fact that the contractor has sublicensed part (or all) of its obligation to conduct bus rides is recorded in the modified component of its received contracts.

279.
279(a))
279(b))
279(c))
279(d))
279(e))
279(f))

⊕ *Discussion* ⊕

-
-
-
-

This ends Example 65 ■

(7. Domain Engineering 7.7. Domain Scripts, Licenses and Contracts)

7.7.1. Principles

7.7.2. Discussion

End of Lecture 9

Domain Engineering: Scripts

Lecture 10

Domain Engineering: Human Behaviour and Closing Stages

7.8. Domain Human Behaviour

Definition 57 – Human Behaviour: *By* **human behaviour** *we mean*

- *any of a quality spectrum of carrying out assigned work:*
  - *from* **careful, diligent** *and* **accurate**,

  via

  - **sloppy** *dispatch, and*
  - **delinquent** *work,*

  to

  - *outright* **criminal** *pursuit.*

■

Example 66 – **A Casually Described Bank Script:** Our formulation amounts to just a (casual) rough sketch. It is followed by a series of three larger examples (Examples 67–69). 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:
  * the interest on the balance of the loan since the most recent repayment,
  * the handling fee, normally considered fixed,
  * 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.
– The interest is subtracted from the mortgage holder's demand/deposit account and added to the bank's interest (income) account.
– The handling fee is subtracted from the mortgage holder's demand/deposit account and added to the bank's fee (income) account.
– 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.

This ends Example 66 ■

Example 67 – **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
  $AY' = \textbf{Real}, AY = \{| ay:AY' \cdot 0 < ay \leq r |\}$
  $MI' = \textbf{Real}, MI = \{| mi:MI' \cdot 0 < mi \leq r' |\}$
  $Bank' = A\_Register \times Accounts \times M\_Register \times Loans$
  $Bank = \{| \beta:Bank' \cdot wf\_Bank(\beta) |\}$
  $A\_Register = C \xrightarrow[m]{} A\text{-}\textbf{set}$
  $Accounts = A \xrightarrow[m]{} Balance$
  $M\_Register = C \xrightarrow[m]{} M\text{-}\textbf{set}$
  $Loans = M \xrightarrow[m]{} (Loan \times Date)$
  $Loan, Balance = P$
  $P = \textbf{Nat}$

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) ≡ $\cup$ **rng** ρ = **dom** α $\wedge$ $\cup$ **rng** μ = **dom** ℓ

axiom

ay<mi [$\wedge$ ...]

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) $\equiv$
  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

This ends Example 67 ■

Example 68 – Bank Staff or Programmer Behaviour:

- Let us assume a bank clerk, "in ye olde" days, when calculating, say mortgage repayments (cf. Example 67).
  - 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 67).
  - 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.

This ends Example 68 ■

Example 69 – A Human Behaviour Mortgage Calculation:

$$
\begin{array}{l}
\mathsf{int\_Cmd(mkPM(c,a,m,p,d))}(\rho,\alpha,\mu,\ell) \equiv \\
\quad \textbf{let } \mathsf{(b,d')} = \ell(\mathsf{m}) \textbf{ in} \\
\quad \textbf{if } \mathsf{q}(\alpha(\mathsf{a}),\mathsf{p}) \ /\!*\ \alpha(\mathsf{a})\leq\mathsf{p}\vee\alpha(\mathsf{a})=\mathsf{p}\vee\alpha(\mathsf{a})\leq\mathsf{p}\vee\ldots\ */ \\
\qquad \textbf{then} \\
\qquad\quad \textbf{let } \mathsf{i} = \mathsf{f}_1(\mathsf{interest(mi,b,period(d,d'))}), \\
\qquad\qquad \ell' = \ell \dagger [\,\mathsf{m}\mapsto\mathsf{f}_2(\ell(\mathsf{m})-(\mathsf{p}-\mathsf{i}))\,] \\
\qquad\qquad \alpha' = \alpha \dagger [\,\mathsf{a}\mapsto\mathsf{f}_3(\alpha(\mathsf{a})-\mathsf{p}),a_i\mapsto\mathsf{f}_4(\alpha(a_i)+\mathsf{i}), \\
\qquad\qquad\qquad\quad a_{\text{“staff”}}\mapsto\mathsf{f}_{\text{“staff”}}(\alpha(a_{\text{“staff”}})+\mathsf{i})\,] \textbf{ in} \\
\qquad\quad ((\rho,\alpha',\mu,\ell'),\mathsf{ok}) \textbf{ end} \\
\qquad \textbf{else} \\
\qquad\quad ((\rho,\alpha',\mu,\ell),\mathsf{nok}) \\
\quad \textbf{end end} \\
\quad \textbf{pre } \mathsf{c} \in \textbf{dom } \mu \wedge \mathsf{m} \in \mu(\mathsf{c})
\end{array}
$$

$$
\begin{array}{l}
\mathsf{q}\colon \mathsf{P} \times \mathsf{P} \xrightarrow{\sim} \textbf{Bool} \\
\mathsf{f}_1,\mathsf{f}_2,\mathsf{f}_3,\mathsf{f}_4,\mathsf{f}_{\text{“staff”}}\colon \mathsf{P} \xrightarrow{\sim} \mathsf{P}\ \ [\text{typically: } \mathsf{f}_{\text{“staff”}} = \lambda \mathsf{p}.\mathsf{p}\,]
\end{array}
$$

The predicate q and the functions f_1, f_2, f_3, f_4 and $f_{\text{“staff”}}$ of Example 67 are deliberately left undefined. They are being defined by the “staffer” when performing (incl., programming) the mortgage calculation routine.

The point of Example 67 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 67 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. This ends Example 69 ■

Example 70 – Transport Net Building:

- We show the example in two stages:
  - First we show a description of a diligent operation;
  - then of a less careful operation.

Sub-example 1 *(of Example 70 –)* A Diligent Operation:

- The int_Insert operation of Example 10 Slide 67
  - was expressed without stating necessary pre-conditions:

11 [30] The insert operation takes an Insert command and a net and yields either a new net or **chaos** for the case where the insertion command "is at odds" with, that is, is not semantically well-formed with respect to the net.

[30] See Page 69 for Item 11 et cetera.

12 We characterise the "is not at odds", i.e., is semantically well-formed, that is: pre_int_Insert(op)(hs,ls), as follows: it is a propositional function which applies to Insert actions, op, and nets, (hs.ls), and yields a truth value if the below relation between the command arguments and the net is satisfied.

Let (hs,ls) be a value of type N.

13 If the command is of the form 2oldH(hi′,l,hi′) then

- ⋆1 hi′ must be the identifier of a hub in hs,
- ⋆2 l must not be in ls and its identifier must (also) not be observable in ls, and
- ⋆3 hi″ must be the identifier of a(nother) hub in hs.

14 If the command is of the form 1oldH1newH(hi,l,h) then

- ⋆1 hi must be the identifier of a hub in hs,
- ⋆2 l must not be in ls and its identifier must (also) not be observable in ls, and
- ⋆3 h must not be in hs and its identifier must (also) not be observable in hs.

15 If the command is of the form 2newH(h′,l,h″) then

- ⋆1 h′ — left to the reader as an exercise (see formalisation !),
- ⋆2 l — left to the reader as an exercise (see formalisation !), and
- ⋆3 h″ — left to the reader as an exercise (see formalisation !).

```
value
 12′  pre_int_Insert: Ins → N → Bool
 12″  pre_int_Insert(Ins(op))(hs,ls) ≡
 ⋆2     s_l(op)∉ ls ∧ obs_LI(s_l(op)) ∉ iols(ls) ∧
      case op of
 13       2oldH(hi′,l,hi″) → {hi′,hi″}⊆iohs(hs),
 14       1oldH1newH(hi,l,h) → hi ∈ iohs(hs)∧h∉ hs∧obs_HI(h)∉ iohs(hs),
 15       2newH(h′,l,h″) → {h′,h″}∩ hs={}∧{obs_HI(h′),obs_HI(h″)}∩ iohs(hs
      end
```

- These must be **carefully** expressed and adhered to
- in order for staff to be said to carry out the link insertion operation **accurately**.

This ends Sub-example 70.1 ■

Sub-example 2 *(of Example 70 –)* **A Sloppy via Delinquent to Criminal Operation:**

- We replace systematic checks ($\wedge$) with partial checks ($\vee$), etcetera,
- and obtain various degrees of **sloppy** to **delinquent**, or even **criminal** behaviour.

value

12′ pre_int_Insert: Ins → N → **Bool**

12″ pre_int_Insert(Ins(op))(hs,ls) ≡

⋆2 s_l(op)∉ ls ∧ obs_LI(s_l(op)) ∉ iols(ls) ∧

case op **of**

13 2oldH(hi′,l,hi″) → hi′ ∈ iohs(hs)∨hi″isin iohs(hs),

14 1oldH1newH(hi,l,h) → hi ∈ iohs(hs)∨h∉ hs∨obs_HI(h)∉ iohs(hs),

15 2newH(h′,l,h″) → {h′,h″}∩ hs={}∨{obs_HI(h′),obs_HI(h″)}∩ iohs(hs)={}

end

This ends Sub-example 70.2 ■

This ends Example 70 ■

7.8.1. A Meta Characteristic 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.

Schema 3 – A Human Behaviour Specification Pattern:

type

[1] α: Action $= \Sigma \xrightarrow{\sim} \Sigma$-**infset**

value

[2] hum_int: Rule $\rightarrow \Sigma \rightarrow$ RUL-**infset**

[3] action: Stimulus $\rightarrow \Sigma \rightarrow \Sigma$

[4] hum_beha: Stimulus $\times$ Rules $\rightarrow$ Action $\rightarrow \Sigma \xrightarrow{\sim} \Sigma$-**infset**

[5] hum_beha(sy_sti,sy_rul)(α)(σ) **as** σset

[6] **post**

[7] σset $= \alpha(\sigma) \wedge$ action(sy_sti)(θ) $\in \theta$set

[8] $\wedge \forall \sigma'$:Σ·$\sigma' \in \sigma$set $\Rightarrow$

[9] $\exists$ se_rul:RUL·se_rul $\in$ hum_int(sy_rul)(σ)$\Rightarrow$se_rul(σ,σ')

■

- The above is, necessarily, sketchy:
  - [1] There is a possibly infinite variety of ways of interpreting some rules.
  - [2] 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.

7.8.2. Principles

7.8.3. Discussion

End of Lecture 10

Domain Engineering: Human Behaviour and Closing Stages

Lecture 11

Domain Engineering: Opening and Closing Stages

7.9. Opening and Closing Stages

- We cover in this lecture the following aspects of domain engineering:
  - opening stages ;
  - closing stages ; and
  - domain engineering documentation — .
- Sections

7.9.1. Opening Stages

- For completeness, we shall briefly list the opening stages of domain engineering.

1. domain **stake-holder identification** (and subsequent **liaison**);
2. rough sketching of **business processes**;
3. domain **acquisition**
   - literature study,
   - Internet study,
   - on-site interviews,
   - questionnaire preparation,
   - questionnaire fill-in, and
   - questionnaire handling —

   resulting in a great number of **domain description units**;
4. domain **analysis** (based on domain description units) and **concept formation**, and
5. domain "**terminologisation**".

7.9.1.1. Stakeholder Identification and Liaison

7.9.1.2. Domain Acquisition

7.9.1.3. Domain Analysis

-
-
-
-

7.9.1.4. Terminoligisation

-
-
-
-

7.9.2. Closing Stages

- For completeness, we shall, as in Sect. on Slide 649, briefly list the closing stages of domain engineering.
- They are:
  1. **domain verification, model checking and testing** – the assurance of properties of the formalisation of the domain model ;
  2. **domain validation** – the assurance of the veracity of the informal, i.e., the narrative domain description ; and
  3. **domain theory formation** .
- Other than this brief mentioning we shall not cover these, from an engineering view-point rather important stages.

7.9.2.1. Verification, Model Checking and Testing

7.9.2.2. Domain Validation

-
-
-
-

7.9.2.3. Domain Theory

7.9.3. Domain Engineering Documentation

•

•

•

•

7.9.4. Conclusion

•

•

•

End of Lecture 11

Domain Engineering: Opening and Closing Stages

Lecture 12

Reqs. Eng.: Opening, Acquisition & BPR

8. Requirements Engineering

8.1. Characterisations

Definition 58 – **IEEE Definition of 'Requirements':**

- *By a requirements we understand (cf. IEEE Standard 610.12):*
  - *"A condition or capability*
  - *needed by a user*
  - *to solve a problem*
  - *or achieve an objective".*

■

Principle 4 – **Requirements Engineering [1]:** *Prescribe only those requirements that can be objectively shown to hold for the designed software.* ■

Principle 5 – **Requirements Engineering [2]:** *When prescribing requirements,*

- *formulate, at the same time, tests (theorems, properties for model checking)*
- *whose actualisation should show adherence to the requirements*

■

Definition 59 – Requirements:

- *By requirements we shall understand a document which prescribes desired properties of a machine:*
  - *(i) what entities the machine shall "maintain", and*
  - *what the machine shall (must; not should) offer of*
    * *(ii) functions and of*
    * *(iii) behaviours*
  - *(iv) while also expressing which events the machine shall "handle".*

■

- A requirements prescription ideally specifies
- externally observable *properties* of
  - simple entities,
  - functions,
  - events and
  - behaviours
- of **the machine**
- such as the requirements stake-holders wish them to be.

- Above we used the term ‘ideally’.
  - Even in good practice the requirements engineer
  - may, here and there in the requirements prescription,
  - resort to prescribe the requirements more by *how* it effects the *what*
  - rather than only (i.e., ‘ideally’) prescribe the requirements by *what* the machine is to offer.

- The **machine** is what is required, that is,
  - the **hardware** and
  - **software**
- that is to be designed and
- which are to satisfy the requirements.
- It is a highlight of this document that
  - requirements engineering has a scientific foundation
  - and that that scientific foundation is the domain theory,
  - that is the properties of the domain as modelled by a domain description.

- Conventional requirements engineering,
  - as covered in a great number of software engineering textbooks,
  - does not have (such) a scientific foundation.
- This foundation allows us to pursue requirements engineering in quite a new manner.
- The key idea of the kind of requirements engineering that we shall present is
  - that a major part of the requirements can be systematically "derived"
  - from a description of the domain in which the requirements 'reside'.

8.2. The Core Stages of Requirements Engineering

- The core stages of requirements engineering are therefore those of 'deriving' the following **requirements facets**:
  - business process re-engineering,
  - domain requirements,
  - interface requirements and
  - machine requirements.

8.3. On Opening and Closing Requirements Engineering Stages

-
-
-
-
- We shall treat the stages and steps of RE opening and closing stages later.

8.4. Business Process Re-Engineering

Definition 60 – Business Process Re-Engineering: *By business process re-engineering we understand*

- *the reformulation of previously adopted business process descriptions,*
- *together with additional business process engineering work.*

■

- Business process re-engineering (BPR) is about *change*,
- and hence BPR is also about *change management*.
- The concept of workflow is one of these "hyped" as well as "hijacked" terms:
  - They sound good, and they make you "feel" good.
  - But they are often applied to widely different subjects, albeit having some phenomena in common.
- By workflow we shall, very loosely, understand the physical movement of people, materials, information and "centre ('locus') of control" in some organisation (be it a factory, a hospital or other).

8.4.1. Michael Hammer's Ideas on BPR

1. *Understand a method of re-engineering before you do it for serious.*
2. *One can only re-engineer processes.*
3. *Understanding the process is an essential first step in re-engineering.*
4. *If you proceed to re-engineer without the proper leadership, you are making a fatal mistake.*
5. *Re-Engineering requires radical, breakthrough ideas about process design.*

6. *Before implementing a process in the real world create a laboratory version in order to test whether your ideas work.*
7. *You must re-engineer quickly.*
8. *You cannot re-engineer a process in isolation. Everything must be on the table.*
9. *Re-Engineering needs its own style of implementation: fast, improvisational, and iterative.*
10. *Any successful re-engineering effort must take into account the personal needs of the individuals it will affect.*

8.4.2. What Are BPR Requirements?

- Two "paths" lead to business process re-engineering:
  - A client wishes to improve enterprise operations by deploying new computing systems (i.e., new software).
    - In the course of formulating requirements for this new computing system
    - a need arises to also re-engineer the human operations within and without the enterprise.
  - An enterprise wishes to improve operations by redesigning the way staff operates within the enterprise and the way in which customers and staff operate across the enterprise-to-environment interface.
    - In the course of formulating re-engineering directives
    - a need arises to also deploy new software, for which requirements therefore have to be enunciated.

- One way or the other, business process re-engineering is an integral component in deploying new computing systems.

8.4.3. Overview of BPR Operations

- We suggest six domain-to-business process re-engineering operations.
  - They are based on the facets that were prominent in the process of constructing a domain description.

1. introduction of some new and removal of some old **intrinsics;**
2. introduction of some new and removal of some old **support technologies;**
3. introduction of some new and removal of some old **management and organisation substructures;**
4. introduction of some new and removal of some old **rules and regulations;**
5. related **scripting;** and
6. introduction of some new and removal of some old work practices (relating to **human behaviours**);

8.4.4. BPR and the Requirements Document

8.4.4.1. Requirements for New Business Processes

- The reader must be duly "warned":
  - The BPR requirements are not for a computing system,
  - but for the people who "surround" that (future) system.
  - The BPR requirements state, unequivocally,
  - how those people are to act,
  - i.e., to use that system properly.
- Any implications, by the BPR requirements,
- as to concepts and facilities of the new computing system
- must be prescribed (also) in the domain and interface requirements.

8.4.4.2. Place in Narrative Document

- We shall thus, in the later part of this lecture, treat a number of BPR facets.
- Each of whatever you decide to focus on,
- in any one requirements development,
- must be prescribed.
- And the prescription must be put into the overall requirements prescription document.

- As the BPR requirements "rebuilds" the business process description part of the domain description[31],
- and as the BPR requirements are not directly requirements for the machine,
- we find that they (the BPR requirements texts) can be simply put in a separate section.

[31]— Even if that business process description part of the domain description is "empty" or nearly so!

- There are basically two ways of "rebuilding"
  - the domain description's business process's description part (D_{BP})
  - into the requirements prescription part's BPR requirements (R_{BPR}).
  - Either
    * you keep all of D as a base part in R_{BPR},
    * and then you follow that part (i.e., R_{BPR})
    * with statements, R'_{BPR}, that express the new business process's "differences"
    * with respect to the "old" (D_{BP}).
    * Call the result R_{BPR}.

– Or
  * you simply rewrite (in a sense, the whole of) D_{BP} directly into R_{BPR},
  * copying all of D_{BP},
  * and editing wherever necessary.

8.4.4.3. Place in Formalisation Document

- The above statements as how to express the "merging" of BPR requirements into the overall requirements document apply to the narrative as well as to the formalised prescriptions.

Principle 6 – Documentation:

- *We may assume that there is a formal domain description, $\mathcal{D}_{BP}$, (of business processes) from which we develop the formal prescription of the BPR requirements.*

- *We may then decide to*
  - *either develop entirely new descriptions of the new business processes, i.e., actually prescriptions for the business re-engineered processes,* $\mathcal{R}_{BPR}$;
  - *or develop, from* $\mathcal{D}_{BP}$, *using a suitable schema calculus, such as the one in* `RSL`, *the requirements prescription* $\mathcal{R}_{BPR}$, *by suitable parameterisation, extension, hiding, etc., of the domain description* $\mathcal{D}_{BP}$.

■

8.4.5. Intrinsics Review and Replacement

Definition 61 – Intrinsics Review and Replacement: *By intrinsics review and replacement we understand an evaluation*

- *as to whether current intrinsics stays or goes, and*
- *as to whether newer intrinsics need to be introduced.*

■

Example 71 – **Intrinsics Replacement:** A railway net owner changes its business from owning, operating and maintaining railway nets (lines, stations and signals) to operating trains. Hence the more detailed state changing notions of rail units need no longer be part of that new company's intrinsics while the notions of trains and passengers need be introduced as relevant intrinsics. ■

Replacement of intrinsics usually point to dramatic changes of the business and are usually not done in connection with subsequent and related software requirements development.

8.4.6. Support Technology Review and Replacement

Definition 62 – **Support Technology Review and Replacement:** *By support technology review and replacement we understand an evaluation*

- *as to whether current support technology as used in the enterprise is adequate, and*
- *as to whether other (newer) support technology can better perform the desired services.*

■

Example 72 – Support Technology Review and Replacement:

- Currently the main information flow of an enterprise is taken care of by printed paper, copying machines and physical distribution. All such documents, whether originals (masters), copies, or annotated versions of originals or copies, are subject to confidentiality.
- As part of a computerised system for handling the future information flow, it is specified, by some domain requirements, that document confidentiality is to be taken care of by encryption, public and private keys, and digital signatures.
- However, it is realised that there can be a need for taking physical, not just electronic, copies of documents.

- The following business process re-engineering proposal is therefore considered:
  - Specially made printing paper and printing and copying machines are to be procured, and so are printers and copiers whose use requires the insertion of special signature cards which, when used, check that the person printing or copying is the person identified on the card, and that that person may print the desired document.
  - All copiers will refuse to copy such copied documents — hence the special paper.
  - Such paper copies can thus be read at, but not carried outside the premises (of the printers and copiers).
  - And such printers and copiers can register who printed, respectively who tried to copy, which documents.
  - Thus people are now responsible for the security (whereabouts) of possible paper copies (not the required computing system).

- The above, somewhat construed example, shows the "division of labour" between the contemplated (required, desired) computing system (the "machine") and the "business re-engineered" persons authorised to print and possess confidential documents.
- It is implied in the above that the re-engineered handling of documents would not be feasible without proper computing support.
- Thus there is a "spill-off" from the business re-engineered world to the world of computing systems requirements.

■

8.4.7. Management and Organisation Re-Engineering

Definition 63 – Management and Organisation Re-Engineering: *By management and organisation re-engineering we understand an evaluation*

- *as to whether current management principles and organisation structures as used in the enterprise are adequate, and*
- *as to whether other management principles and organisation structures can better monitor and control the enterprise.*

■

Example 73 – Management and Organisation Re-Engineering:

- A rather complete computerisation of the procurement practices of a company is being contemplated.
- Previously procurement was manifested in the following physically separate as well as designwise differently formatted paper documents: *requisition form, order form, purchase order, delivery inspection form, rejection and return form*, and *payment form*.
- The supplier had corresponding forms: *order acceptance and quotation form, delivery form, return acceptance form, invoice form, return verification form*, and *payment acceptance form*.
- The current concern is only the procurement forms, not the supplier forms.

- The proposed domain requirements are mandating
  - that all procurer forms disappear in their paper version,
  - that basically only one, the procurement document, represents all phases of procurement,
  - and that order, rejection and return notification slips, and payment authorisation notes,
  - be effected by electronically communicated and duly digitally signed messages that represent appropriate subparts of the one, now electronic procurement document.

- The business process re-engineering part may now
  - "short-circuit" previous staff's review and
  - acceptance/rejection of former forms,
- in favour of fewer staff interventions.
- The new business procedures, in this case, subsequently find their way into proper domain requirements: those that support, that is monitor and control all stages of the re-engineered procurement process.

■

8.4.8. Rules and Regulations Re-Engineering

Definition 64 – **Rules and Regulation Re-Engineering:** *By rules and regulations re-engineering we understand an evaluation*

- *as to whether current rules and regulations as used in the enterprise are adequate, and*
- *as to whether other rules and regulations can better guide and regulate the enterprise.*

■

- Here it should be remembered that rules and regulations principally stipulate business engineering processes.
- That is, they are — i.e., were — usually not computerised.

Example 74 – Rules and Regulations Re-Engineering:

- Our example continues that of Example 60 on Slide 473.
- Assume now, due to re-engineered support technologies, that interlock signalling can be made magnitudes safer than before, without interlocking.
- Thence it makes sense to re-engineer the rule of Example 60
  - from: *In any three-minute interval at most one train may either arrive to or depart from a railway station*
  - into: *In any 20-second interval at most two trains may either arrive to or depart from a railway station.*
- This re-engineered rule is subsequently made into a domain requirements, namely that the software system for interlocking is bound by that rule.

■

8.4.9. Script Re-Engineering

- On one hand, there is the engineering of the contents of rules and regulations,
- and, on another hand, there are
  - the people (management, staff) who script these rules and regulations,
  - and the way in which these rules and regulations are communicated to managers and staff concerned.

Definition 65 – Script Re-Engineering: *By script re-engineering we understand evaluation*

- *as to whether the way in which rules and regulations are scripted and made known (i.e., posted) to stakeholders in and of the enterprise is adequate, and*
- *as to whether other ways of scripting and posting are more suitable for the enterprise.*

■

Example 75 – Health-care Script Re-Engineering:

- We refer to Example 63 (Pages 505–517).
- Let us assume that the situation before this business re-engineering process starts, in relation to hospital health-care, was that
  - there was no physically visible notion of a health-care license language,
  - but the the requirements now calls for such a language to be introduced
  - with as much computer & communication support as is reasonable
  - and that the hospital(s) in question are to become “paper-less”.
- Now we can foresee a number of business process re-engineering based on the concept that such a health-care license language has been designed.

- For every action performed by a medical staff, whether an admittance, annamnese, planning analysis, carrying out our analysis, diagnostics, treatment planning, treatment (in all forms), et cetera, action (cf. Fig. 13 on Slide 510),
- there now has to be prescribed, by and for the hospital health-care staff a BPR prescription, which outlines what the staff members must do
  - ∗ in preparation of the action,
  - ∗ the action (probably nothing new here),
  - ∗ and in concluding the action

  so that the medical staff performs the necessary "chores" assumed by the health-care license language software.

8.4.10. Human Behaviour Re-Engineering

Definition 66 – **Human Behaviour Re-Engineering:** *By human behaviour re-engineering we understand an evaluation*

- *as to whether current human behaviour as experienced in the enterprise is acceptable, and*
- *as to whether partially changed human behaviours are more suitable for the enterprise.*

■

Example 76 – Human Behaviour Re-Engineering:

- A company has experienced certain lax attitudes among members of a certain category of staff.
- The progress of certain work procedures therefore is re-engineered,
- implying that members of another category of staff are henceforth expected to follow up on the progress of “that” work.
- In a subsequent domain requirements stage the above re-engineering
- leads to a number of requirements for computerised monitoring of the two groups of staff.

■

8.4.11. A Specific Example of BPR

Example 77 – A Toll-road System (I):

- Example 10 (Pages 62–77) outline a generic model of a domain of roads (links) and their intersections (hubs).
- We shall base some of the requirements examples of Sect. on an instantiation of that domain model (Example 10) to a specific toll-road system.
- In this example we shall rough sketch that toll-road system.
- First we refer to Fig. 14 on following slide.

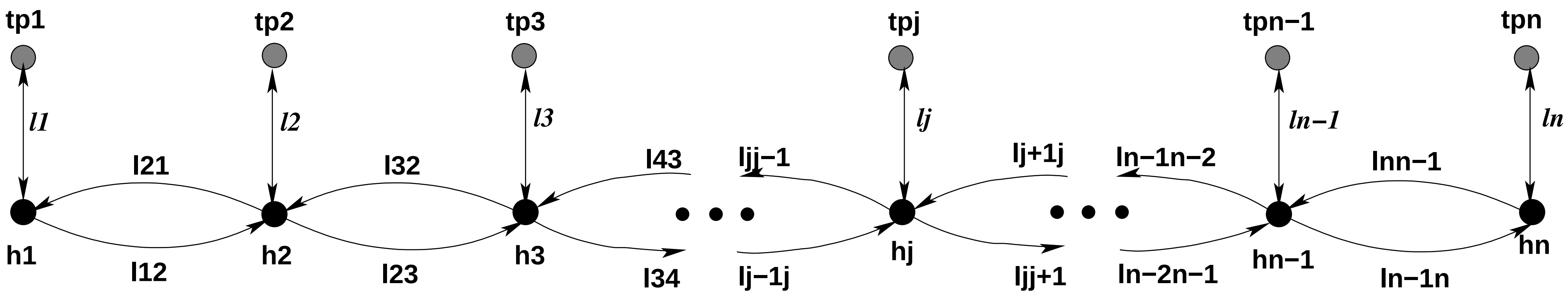


Figure 14: A simple, linear toll road net:

tp_i: *t*boll *p*laza i,

ti_1, ti_n: *t*erminal (or *t*oll plaza) *i*ntersection k,

h_k: intermediate intersection (hub) k, $1<k<n$

ℓ_i: toll plaza *l*ink i,

l_{xy}: toll-way *l*ink from i_x to i_y, y=x+1 or y=x-1 and $1 \leq x < n$.

- We first explain the kind of toll-roads semi-generically hinted at by Fig. 14 on the previous slide.
  - The core of the (semi-generic) toll-road is the "linear" stretch of pairs of one-way links $(\ell_{j_{j+1}}, \ell_{{j+1}_j})$ between adjacent hubs h_j and h_{j+1}. This is the actual toll-road.
  - In order to enter and leave the toll-road there are entries and exits. These are in the form of toll plazas tp_i.
  - Simple two-way links, ℓ_j, connect toll plaza tp_j (via toll plaza intersection t_{ij} to toll-road intersection h_j.

- We must here state that toll plazas are equipped with toll booths
  - for cars entering the toll-road system and
  - for cars leaving the toll-road system.
- The basic business process of this toll-road system includes
  - (i) the maintenance of all roads, intersections and toll plaza toll booths;
  - (ii) the travel, through the toll-road system of a toll-paying car; and
  - (iii) the monitoring and control of toll-paying car traffic within the toll-road system.

- We shall only summarise the travel (i) business process:
  - (1) A car enters the toll-road system at toll plaza tp_i.
  - (2) A toll booth gate prevents the car from fully entering the system.
  - (3) The car is issued a toll slip by an entry toll-booth at toll plaza tp_i.
  - (4) The toll-booth gate allows the car from entering the system.
  - (5) The car travels along toll plaza link ℓ_j to toll-road hub h_j. [Let, in the following h_j now be h_i.]

– (6) At toll-road hub h_i the car decides whether to continue driving along the toll-road proper or to leave the toll-road system along toll plaza link ℓ_j to toll plaza tp_i. In the latter case the business process description continues with Item (8).
– (7) The car travels, along toll-road link either $\ell_{i_{i+1}}$ or $\ell_{i_{i-1}}$, between toll-road hub h_i and hub h_{i+1} respectively hub h_{i+-}. [Let, in the following this target hub now be h_i.] The next business process step is now described in Item (6).
– (8) At toll plaza tp_i the car enters an exit toll-booth.
– (9) A toll-booth gate prevents the car from leaving the toll-road system.

– (10) The previously issued toll slip is now presented at the toll-booth.
– (11) The toll-booth calculates the fare from entry plaza to exit plaza[32].
– (13) The car (driver) somehow[33]
– (14) The toll-booth gate allows the car to leave the toll-road system.
– (15) The car leaves the toll-road system.

This ends Example 77 ■

[32]We shall not detail this calculation. Its proper calculation may involve that the system has traced the car's passage through all hubs.
[33]We shall not detail that "somehow": whether it is by cash payment, via credit card, or by means of a toll-road system credit mechanism "built-into" the car.

8.4.12. Discussion: Business Process Re-Engineering

8.4.12.1. Who Should Do the Business Process Re-Engineering?

- It is not in our power, as software engineers,
- to make the kind of business process re-engineering decisions implied above.
- Rather it is, perhaps, more the prerogative of appropriately educated, trained and skilled (i.e., gifted) other kinds of engineers or business people
- to make the kinds of decisions implied above.
- Once the BP re-engineering has been made, it then behooves the client stakeholders to further decide whether the BP re-engineering shall imply some requirements, or not.

- Once that last decision has been made in the affirmative, we, as software engineers, can then apply our abstraction and modelling skills, and,
- while collaborating with the former kinds of professionals,
- make the appropriate prescriptions for the BPR requirements.
- These will typically be in the form of domain requirements, which are covered extensively in later lectures.

8.4.12.2. General

- Business process re-engineering is based on the premise
- that corporations must change their way of operating,
- and, hence, must "reinvent" themselves.
- Some corporations (enterprises, businesses, etc.) are
  - "vertically" structured
    - along functions, products or geographical regions.
  - Others are "horizontally" structured
    - along coherent business processes.
  - In either case adjustments may need to be made as the business (i.e., products, sales, markets, etc.) changes.

8.5. Requirements Acquisition

- Requirements 'reside' in the domain.
- That means:
  - one can not possibly utter a reasonably comprehensive set of requirements
  - without stating the domain "to which they apply".
- Therefore
  - we first describe the domain before
  - we next prescribe the requirements.
- And therefore
  - we shall "base our requirements acquisition"
  - on a supposedly existing domain description.

- To *'base our requirements acquisition ... etc.'* shall mean
  - that we carefully go through the domain description
  - (found most appropriate for the requirements at hand)
  - with the requirements stake-holders
  - asking them a number of questions.
- Which these questions are will be dealt with soon.
- For domain acquisition there were, in principle, no prior domain description documents, really, to refer to.
  - Hence an elaborate set of procedures had to be followed
  - in order to solicit and elicit domain acquisition units.
  - Before such elicitation could be done in any systematic fashion the domain engineer had to study the domain, by whatever informal means available.
  - Now there is the domain description.

- From a purely linguistic point of view we can think of decomposing requirements acquisition relative to the domain description along three axes:
  - the first axis of **domain requirements** — being those which can be expressed sôlely using terms from the domain;
  - the second axis of **machine requirements** — being those which can be expressed sôlely using terms from the machine; and
  - the third axis of **interface requirements** — being those which can be expressed using terms from both the domain and the machine.
- The next three sections,
  - Sects. –,
  - shall therefore be structured into two parts:
    * the respective requirements acquisition part
    * and the corresponding requirements modelling part.

End of Lecture 12

Reqs. Eng.: Opening, Acquisition & BPR

Lecture 13

Domain Requirements Engineering

8.6. Domain Requirements

8.6.1. A Small Domain Example

- In exemplifying several of the very many kinds of domain and machine requirements we need a small domain description.
- This small domain description will be that of a **timetable**, cf. Example 78.
- The sequence of machine requirements examples based on Example 78 will, furthermore, be expressed using the **scheme** and **class** modularisation constructs of `RSL`.

Example 78 – A Domain Example: an 'Airline Timetable System':

- We choose a very simple domain:
- that of a traffic timetable, say flight timetable.
- In the domain you could, in "ye olde days", hold such a timetable in your hand, you could browse it, you could look up a special flight, you could tear pages out of it, etc.
- There was no end as to what you could do to such a timetable.
- So we will just postulate a sort, *TT*, of timetables.

- Airline customers, *clients*, in general, only wish to inquire a timetable (so we will here omit treatment of more or less "malicious" or destructive acts).
- But you could still count the number of digits "7" in the timetable, and other such ridiculous things.
- So we postulate a broadest variety of inquiry functions, *qu:QU*, that apply to timetables, *tt:TT*, and yield values, *val:VAL*.

- Specifically designated airline *staff* may, however, in addition to what a *client* can do, update the timetable.
- But, recalling human behaviours, all we can ascertain for sure is that update functions, *up:UP*, apply to timetables and yield two things: another, replacement timetable, *tt:TT*, and a result, *res:RES*, such as: *"your update succeeded"*, or *"your update did not succeed"*, etc.
- In essence this is all we can say for sure about the domain of timetable creations and uses.

- We can view the domain of the
  - timetable,
  - clients and
  - staff
- as a behaviour
  - which nondeterministically alternates ($\lceil\rceil$) between
  - the *client* querying the timetable *client_0(tt)*,
  - and the *staff* updating the same *staff_0(tt)*.

```
scheme TI_TBL_0 =
  class
    type
      TT, VAL, RES
      QU = TT → VAL
      UP = TT → TT × RES
    value
      client_0: TT → VAL, client_0(tt) ≡ let q:QU in q(tt) end
      staff_0: TT → TT × RES, staff_0(tt) ≡ let u:UP in u(tt) end

      tim_tbl_0: TT → Unit
      tim_tbl_0(tt) ≡
          (let v = client_0(tt) in tim_tbl_0(tt) end)
        ⌈⌉ (let (tt′,r) = staff_0(tt) in tim_tbl_0(tt′) end)
  end
```

This ends Example 78 ■

8.6.2. Acquisition

- Common to the acquisition and modelling of domain requirements are the following sub-stages:
  - projection,
  - instantiation,
  - determination,
  - extension and
  - fitting.

  sub-stages.

- With each and every stake-holder group the domain engineer(s) go through the domain description and asks the following questions:
  - Which of the simple entities, functions, events and behaviour (parts) of the domain do you wish to be represented somehow in, i.e., **projected** onto the machine ?

- Which of the simple entities, functions, events and behaviour (parts) of the domain do you wish to be less generic, more **instantiated** in the machine ?
- Which of the simple entities, functions, events and behaviour (parts) of the domain do you wish to appear more **deterministic** in the machine ?
- Are there simple entities, functions, events and behaviours that could be in the domain but are not there because their "existence" is not feasible — if so, with computing and communication are they now feasible and should the domain thus be **extended** ?
- Given that there may be several, parallel ongoing requirements development for related parts of the domain, should they be **fitted** ?

- For each of these five sub-stages of domain requirements the acquisition consists in asking these questions and marking the domain description cum emerging domain requirements document with the answers:
  - circling-in the domain description parts that are to be part of the domain requirements (i.e., projection)
  - marking those parts with possible directives as to instantiation and determination;
  - making adequate notes on possible extensions
  - and fittings.
- Once this domain requirements acquisition has taken place for all groups of stake-holders the requirements engineers can proceed to interface requirements acquisition.

8.6.3. Projection

Definition 67 – Projection: *By domain projection we understand an operation*

- *that applies to a domain description*
- *and yields a domain requirements prescription.*
- *The latter represents a projection of the former*
- *in which only those parts of the domain are present*
- *that shall be of interest in the ongoing requirements development*

■

Example 79 – Projection: A Road Maintenance System:

- The requirements are for a road maintenance system.
  - That is, maintenance of link and hub (road segment and road intersection) surfaces,
  - the monitoring of their quality
  - and road repair.
- Instead of listing all the phenomena and concepts of the domain that are "projected away", we list those few that remain:
  - hubs,
  - links,
  - hub identifiers and
  - link identifiers;
  - nets,
  - observer functions, and
  - axioms.

[34] Formula numbers refer to narrative text items as from Page 62 etc.

type

1: H, L,[34]

2: $N = H\text{-}\mathbf{set} \times L\text{-}\mathbf{set}$

axiom

2: $\forall$ (hs,ls):N $\cdot$ **card** hs$\geq$2 $\wedge$ **card** ks$\geq$1

type

3: HI, LI

value

4a: obs_HI: H $\rightarrow$ HI, obs_LI: L $\rightarrow$ LI

axiom

4b: $\forall$ h,h$'$:H, l,l$'$:L $\cdot$ h$\neq$h$'$

$\Rightarrow$ obs_HI(h)$\neq$obs_HI(h$'$) $\wedge$ l$\neq$l$'$$\Rightarrow$obs_LI(l)$\neq$obs_LI(l$'$)

value

$$
\begin{array}{l}
\text{5a:}\ \ \mathsf{obs\_HIs}:\ \mathsf{L} \rightarrow \mathsf{HI}\text{-}\mathbf{set} \\
\text{6a:}\ \ \mathsf{obs\_LIs}:\ \mathsf{H} \rightarrow \mathsf{LI}\text{-}\mathbf{set} \\
\text{5b:}\ \ \forall\ \mathsf{l:L} \cdot \mathbf{card}\ \mathsf{obs\_HIs(l)}=2\ \wedge \\
\text{6b:}\ \ \forall\ \mathsf{h:H} \cdot \mathbf{card}\ \mathsf{obs\_LIs(h)}\geq 1\ \wedge \\
\text{5(a):}\ \ \forall\ \mathsf{(hs,ls):N} \cdot \forall\ \mathsf{h:H} \cdot \mathsf{h} \in \mathsf{hs} \Rightarrow \forall\ \mathsf{li:LI} \cdot \mathsf{li} \in \mathsf{obs\_LIs(h)} \Rightarrow \\
\qquad \exists\ \mathsf{l':L} \cdot \mathsf{l'} \in \mathsf{ls} \wedge \mathsf{li}=\mathsf{obs\_LI(l')} \wedge \mathsf{obs\_HI(h)} \in \mathsf{obs\_HIs(l')} \wedge \\
\text{6(a):}\qquad \forall\ \mathsf{l:L} \cdot \mathsf{l} \in \mathsf{ls} \Rightarrow \\
\qquad \exists\ \mathsf{h',h'':H} \cdot \{\mathsf{h',h''}\}\subseteq \mathsf{hs} \wedge \mathsf{obs\_HIs(l)}=\{\mathsf{obs\_HI(h')},\mathsf{obs\_HI(h'')}\} \\
\text{7:}\qquad \forall\ \mathsf{h:H} \cdot \mathsf{h} \in \mathsf{hs} \Rightarrow \mathsf{obs\_LIs(h)} \subseteq \mathsf{iols(ls)} \\
\text{8:}\qquad \forall\ \mathsf{l:L} \cdot \mathsf{l} \in \mathsf{ls} \Rightarrow \mathsf{obs\_HIs(h)} \subseteq \mathsf{iohs(hs)} \\
\quad \mathsf{iohs}:\ \mathsf{H}\text{-}\mathbf{set} \rightarrow \mathsf{HI}\text{-}\mathbf{set},\ \mathsf{iols}:\ \mathsf{L}\text{-}\mathbf{set} \rightarrow \mathsf{LI}\text{-}\mathbf{set} \\
\quad \mathsf{iohs(hs)} \equiv \{\mathsf{obs\_HI(h)}|\mathsf{h:H}\cdot\mathsf{h} \in \mathsf{hs}\} \\
\quad \mathsf{iols(ls)} \equiv \{\mathsf{obs\_LI(l)}|\mathsf{l:L}\cdot\mathsf{l} \in \mathsf{ls}\}
\end{array}
$$

This ends Example 79 ■

Example 80 – Projection: A Toll-road System:

- For the 'Toll-road System', as outlined in Example 77, in addition to what was projected for the 'Road Maintenance System' of Example 79, the following entities and most related functions are projected:
  - hubs, links,
  - hub and link identifiers;
  - nets, that is,
  - hub state and hub state spaces and
  - link states and link state spaces,
  - corresponding observer functions,
  - corresponding axioms and
  - syntactic and
  - semantic wellformedness predicates.

type

$L\Sigma' = L\_Trav\text{-}\mathbf{set}$

$L\_Trav = (HI \times LI \times HI)$

$L\Sigma = \{|\ lnk\sigma{:}L\Sigma' \cdot syn\_wf\_L\Sigma\{lnk\sigma\}\ |\}$

$H\Sigma' = H\_Trav\text{-}\mathbf{set}$

$H\_Trav = (LI \times HI \times LI)$

$H\Sigma = \{|\ hub\sigma{:}H\Sigma' \cdot wf\_H\Sigma\{hub\sigma\}\ |\}$

$H\Omega = H\Sigma\text{-}\mathbf{set},\ L\Omega = L\Sigma\text{-}\mathbf{set}$

value

$obs\_H\Sigma{:}\ H \rightarrow H\Sigma,\ obs\_L\Sigma{:}\ L \rightarrow L\Sigma$

$obs\_H\Omega{:}\ H \rightarrow H\Omega,\ obs\_L\Omega{:}\ L \rightarrow L\Omega$

axiom

$\forall\ h{:}H \cdot obs\_H\Sigma(h) \in obs\_H\Omega(h) \wedge \forall\ l{:}L \cdot obs\_L\Sigma(l) \in obs\_L\Omega(l)$

This ends Example 80 ■

8.6.4. Instantiation

Definition 68 – Instantiation: *By domain instantiation we understand an operation*

- *that applies to a (projected and possibly determined) domain description, i.e., a requirements prescription,*
- *and yields a domain requirements prescription,*
- *where the latter has been made more specific, usually by constraining a domain description*

■

Example 81 – **Instantiation: A Road Maintenance System:** We continue Example 79.

280. The road net consist of a sequence of one or more road segments.
281. A road segment can be characterised by a pair of hubs and a pair of links connected to these hubs.
282. Neighbouring road segments share a hub.
283. All hubs are otherwise distinct.
284. All links are distinct.
285. The two links of a road segment connects to the hubs of the road segment.
286. We can show that road nets are specific instances of concretisations of the former, thus more abstract road nets.

type

280 RN = RS*,

281 RS = H × (L × L) × H

axiom

∀ rn:RN ·

282 ∀ i:**Nat** · {i,i+1}⊆**inds** rn ⇒ **let** (_,_,h)=rn(i),(h′,_,_)=rn(i+1) **in** h=h′ **end** ∧

283 **len** rn + 1 = **card**{h,h′|h,h′:H·(h,_,h′)∈ **elems** rn} ∧

284 2∗(**len** rn) = **card**{l,l′|l,l′:L·(_,(l,l′),_)∈ **elems** rn} ∧

285 ∀ (h,(l,l′),h′):RS · (h,(l,l′),h′) ∈ **elems** rn ⇒

obs_Σ(l)={(obs_Hl(h),obs_Hl(h′))} ∧ obs_Σ(l′)={(obs_Hl(h′),obs_Hl(h))}

value

286 abs_N: RN → N

abs_N(rsl) ≡

({h,h′|(h,_,h′):RS · (h,_,h′) ∈ **elems** rsl},{l,l′|(_,(,l,l′),_):RS · (_.(l,l′),_) ∈ **elems** rsl})

This ends Example 81 ■

Example 82 – **Instantiation: A Toll-road System:** We continue Example 80.

- The 1st version domain requirements prescription, Example 80, is now updated with respect to the properties of the toll-road net:
  - We refer to Fig. 14 on Slide 701 and the preliminary description given in Example 77.
  - There are three kinds of hubs:
    - ∗ tollgate hubs and
    - ∗ intersection hubs:
      - · terminal intersection hubs and
      - · proper, intermediate intersection hubs.
  - Tollgate hubs have one connecting two way link.
    - ∗ linking the tollgate hub to its associated intersection hub.

- Terminal intersection hubs have three connecting links:
  - (i) one, a two-way link, to a tollgate hub,
  - (ii) one one-way link emanating to a next up (or down) intersection hub, and
  - (iii) one one-way link incident upon this hub from a next up (or down) intersection hub.
- Proper intersection hubs have five connecting links:
  - one, a two way link, to a tollgate hub,
  - two one way links emanating to next up and down intersection hubs, and
  - two one way links incident upon this hub from next up and down intersection hub.
- etc.

- As a result we obtain a 2nd version domain requirements prescription.

type

$\quad \mathsf{TN} = ((\mathsf{H} \times \mathsf{L}) \times (\mathsf{H} \times \mathsf{L} \times \mathsf{L}))^* \times \mathsf{H} \times (\mathsf{L} \times \mathsf{H})$

value

$\quad \mathsf{abs\_N}: \mathsf{TN} \rightarrow \mathsf{N}$

$\quad \mathsf{abs\_N(tn)} \equiv \mathsf{(tn\_hubs(tn),tn\_hubs(tn))}$

$\quad$ **pre** $\mathsf{wf\_TN(tn)}$

$\quad \mathsf{tn\_hubs}: \mathsf{TN} \rightarrow \mathsf{H}$-**set**,

$\quad \mathsf{tn\_hubs(hll,h,(\_,hn))} \equiv$

$\qquad \{\mathsf{h,hn}\} \cup \{\mathsf{thj,hj} \mid ((\mathsf{thj,tlj}),(\mathsf{hj,lj,lj'})):((\mathsf{H}\times\mathsf{L})\times(\mathsf{H}\times\mathsf{L}\times\mathsf{L}))\cdot((\mathsf{thj,tlj}),(\mathsf{hj,lj,lj'}))\in \textbf{elems}\ \mathsf{hlll}\}$

$\quad \mathsf{tn\_links}: \mathsf{TN} \rightarrow \mathsf{L}$-**set**

$\quad \mathsf{tn\_links(hll,\_,(ln,\_))} \equiv$

$\qquad \{\mathsf{ln}\} \cup \{\mathsf{tlj,lj,lj'} \mid ((\mathsf{thj,tlj}),(\mathsf{hj,lj,lj'})):((\mathsf{H}\times\mathsf{L})\times(\mathsf{H}\times\mathsf{L}\times\mathsf{L}))\cdot((\mathsf{thj,tlj}),(\mathsf{hj,lj,lj'}))\in \textbf{elems}\ \mathsf{hlll}\}$

theorem $\forall\ \mathsf{tn:TN} \cdot \mathsf{wf\_TN(tn)} \Rightarrow \mathsf{wf\_N(abs\_N(tn))}$

```
type
  LnkM == plaza | way
value
  wf_TN: TN → Bool
  wf_TN(tn:(hll,h,(ln,hn))) ≡
     wf_Toll_Lnk(h,ln,hn)(plaza) ∧ wf_Toll_Ways(hll,h) ∧
     wf_State_Spaces(tn) [ to be defined under Determination ]
  wf_Toll_Ways: ((H×L)×(H×L×L))* × H → Bool
  wf_Toll_Ways(hll,h) ≡
    ∀ j:Nat • {j,j+1}⊆inds hll ⇒
      let ((thj,tlj),(hj,ljj′,lj′j)) = hll(j),(_,(hj′,_,_)) = hll(j+1) in
      wf_Toll_Lnk(thj,tlj,hj)(plaza) ∧
      wf_Toll_Lnk(hj,ljj′,hj′)(way) ∧ wf_Toll_Lnk(hj′,lj′j,hj)(way) end ∧
      let ((thk,tlk),(hk,lk,lk′)) = hll(len hll) in
      wf_Toll_Lnk(thk,tlk,hk)(plaza) ∧
      wf_Toll_Lnk(hk,lk,hk′)(way) ∧ wf_Toll_Lnk(hk′,lk′,hk)(way)  end
```

wf_Toll_Lnk: $(H \times L \times H) \rightarrow$ LnkM $\rightarrow$ **Bool**
wf_Toll_Lnk(h,l,h′)(m) $\equiv$
  obs_Ps(l)={(obs_HI(h),obs_LI(l),obs_HI(h′)),(obs_HI(h′),obs_LI(l),obs_HI(h))} $\wedge$
  obs_Σ(l) = **case** m **of**
      plaza $\rightarrow$ obs_Ps(l),
      way $\rightarrow$ {(obs_HI(h),obs_LI(l),obs_HI(h′))} **end**

This ends Example 82 ■

8.6.5. Determination

Definition 69 – Determination: *By domain determination we understand an operation*

- *that applies to a (projected) domain description, i.e., a requirements prescription,*
- *and yields a domain requirements prescription,*
- *where the latter has made deterministic, or specific, some function results or some behaviours of the former*

■

Example 83 – Timetable System Determination:

- We make airline timetables more specific, more deterministic.
  - There are given notions
    - of departure and arrival times, and
    - of airports, and
    - of airline flight numbers.

```
scheme TI_TBL_2 =
  extend TI_TBL_1 with
    class
      type
        T, An, Fn
    end
```

- A timetable consists of a number of air flight journey entries.
  - Each entry has a flight number,
  - and a list of two or more airport visits.
    * an airport visit consists of three parts: An airport name, and a pair of (gate) arrival and departure times.

scheme TI_TBL_3 =
 extend TI_TBL_2 **with**
  class
   type
    $\mathsf{JR}' = (\mathsf{T} \times \mathsf{An} \times \mathsf{T})^*$
    $\mathsf{JR} = \{|\ \mathsf{jr}:\mathsf{JR}' \cdot \textbf{len}\ \mathsf{jr} \geq 2 \wedge \ldots\ |\}$
    $\mathsf{TT} = \mathsf{Fn} \xrightarrow[m]{} \mathsf{JR}$
  end

- We illustrate just one, simple form of airline timetable queries.
- A simple airline timetable query
  - either just browses all of an airline timetable,
  - or inquires of the journey of a specific flight.

- The simple
  - browse query thus need not provide specific argument data,
  - whereas the flight journey query needs to provide a flight number.
- A simple
  - update query inserts a new pairing of a flight number and a journey to the timetable,
  - whereas a delete query need just provide the number of the flight to be deleted.

- The result of a query is a value:
  - the specific journey inquired,
  - or the entire timetable browsed.
- The result of an update is a possible timetable change
  - and either an “OK” response if the update could be made,
  - or a “Not OK” response if the update could not be made:
    * Either the flight number of the journey to be inserted was already present in the timetable,
    * or the flight number of the journey to be deleted was not present in the timetable.

```
scheme TI_TBL_3Q =
  extend TI_TBL_3 with
    class
      type
        Query == mk_brow() | mk_jour(fn:Fn)
        Update == mk_inst(fn:Fn,jr:JR) | mk_delt(fn:Fn)
        VAL = TT
        RES == ok | not_ok
    end
```

Then we define the semantics of the query commands:

scheme TI_TBL_3U =
  extend TI_TBL_3 **with**
    class
      value
        $\mathcal{M}_q$: Query → QU
        $\mathcal{M}_q$(qu) ≡
          case qu **of**
            mk_brow() → λtt:TT·tt,
            mk_jour(fn)
              → λtt:TT · **if** fn ∈ **dom** tt
                                then [fn↦tt(fn)] **else** [ ] **end**
    end   **end**

```
scheme TI_TBL_3U =
  extend TI_TBL_3 with
    class
        𝓜_u: Update → UP
        𝓜_u(up) ≡
          case qu of
            mk_inst(fn,jr) → λtt:TT ·
              if fn ∈ dom tt
                then (tt,not_ok) else (tt ∪ [ fn↦jr ],ok) end,
            mk_delt(fn) → λtt:TT ·
              if fn ∈ dom tt
                then (tt \ {fn},ok) else (tt,not_ok) end
    end   end
```

- Before we had:

value
$\text{tim\_tbl\_0: TT} \rightarrow \textbf{Unit}$
$\text{tim\_tbl\_0(tt)} \equiv$
$\quad (\textbf{let}\ \text{v} = \text{client\_0(tt)}\ \textbf{in}\ \text{tim\_tbl\_0(tt)}\ \textbf{end})$
$\ \ \lceil\rceil\ (\textbf{let}\ (\text{tt}',\text{r}) = \text{staff\_0(tt)}\ \textbf{in}\ \text{tim\_tbl\_0(tt}')\ \textbf{end})$

- Now we get:

value
$\text{system: TT} \rightarrow \textbf{Unit}$
$\text{system()} \equiv$
$\quad (\textbf{let}\ \text{q:Query}\ \textbf{in}\ \textbf{let}\ \text{v} = \mathcal{M}_q\text{(q)(tt)}\ \textbf{in}\ \text{system(tt)}\ \textbf{end}\ \textbf{end})$
$\ \ \lceil\rceil\ (\textbf{let}\ \text{u:Update}\ \textbf{in}\ \textbf{let}\ (\text{r},\text{tt}') = \mathcal{M}_u\text{(q)(tt)}\ \textbf{in}\ \text{system(tt}')\ \textbf{end}\ \textbf{end})$

Or, for use in Example 99:

system(tt) $\equiv$ client(tt) $\sqcap$ staff(tt)

client: TT $\rightarrow$ **Unit**
client(tt) $\equiv$
  let q:Query **in let** v = $\mathcal{M}_q$(q)(tt) **in** system(tt) **end end**

staff: TT $\rightarrow$ **Unit**
staff(tt) $\equiv$
  let u:Update **in let** (r,tt′) = $\mathcal{M}_u$(q)(tt) **in** system(tt′) **end end**

This ends Example 83 ■

Example 84 – Determination: A Road Maintenance System:

We continue Example 81.

- We shall, in this example, claim that the following items constitute issues of more determinate nature of the 'Road Management System' under development.
  - fixing the states of links and hubs;
  - endowing links and hubs with such attributes as
    * road surface material (concrete, asphalt, etc.),
    * state of road surface wear-and-tear,
    * hub and link areas, say in m^2,
    * time units needed for and cost of ordinary cleaning of m^2s of hub and link surface;
    * time units needed for and cost of ordinary repairs of m^2s of hub and link surface;
    * etcetera.

287. The two links of a road segment are open for traffic in one direction and in opposite directions only.
288. Hubs are always in the same state, namely one that allows traffic from incoming links to continue onto all outgoing links.
289. Hubs and Links have a number of attributes that allow for the monitoring and planning of hub and link surface conditions, i.e., whether in ordinary or urgent need of cleaning and/or repair.

```
axiom
  ∀ rn:RN •
287 ∀ (h,(l,l′),h′):RS • (h,(l,l′),h′) ∈ elems rn ⇒
      obs_LΣl(l) = {(obs_HI(h),obs_LI(l),obs_HI(h′))} ∧
      obs_LΣl(l′) = {(obs_HI(h′),obs_LI(l′),obs_HI(h))} ∧
288 ∀ i:Nat • {i,i+1}⊆inds rn •
      let ((h,(l,l′),h′),(h′,(l″,l‴),h″)) = (rn(i),rn(i+1)) in
      case i of
        1 →        obs_HΣ(h) = {(obs_LI(l),obs_HI(h),obs_LI(l′))},
        len rn → obs_HΣ(h′) = {(obs_LI(l′),obs_HI(h′),obs_LI(l))},
        _ →        obs_HΣ(h′)
                 = {(obs_LI(l),obs_HI(h′),obs_LI(l′)),(obs_LI(l),obs_HI(h′),obs_LI(l′))}
      end end
type
289 Surface, WearTear, Area, OrdTime, OrdCost, RepTime, RepCost, ...
value
289 obs_Surface: (H|L)→Surface, obs_WearTear: (H|L)→WearTear, ...
```


This ends Example 84 ■

Example 85 – **Determination: A Toll-road System:** We continue Example 82.

- We single out only two 'determinations':
  - *The link state spaces.*
    - There is only one link state: the set of all paths through the link,
    - thus any link state space is the singleton set of its only link state.
  - *The hub state spaces* are the singleton sets of the "current" hub states which allow these crossings:
    - (i) from terminal link back to terminal link,
    - (ii) from terminal link to emanating tollway link,
    - (iii) from incident tollway link to terminal link, and
    - (iv) from incident tollway link to emanating tollway link.
- Special provision must be made for expressing the entering from the outside and leaving toll plazas to the outside.

```
wf_State_Spaces: TN → Bool
wf_State_Spaces(hll,hn,(thn,tln)) ≡
  let ((th1,tl1),(h1,l12,l21)) = hll(1),
     ((thk,ljk),(hk,lkn,lnk)) = hll(len hll) in
  wf_Plaza(th1,tl1,h1) ∧ wf_Plaza(thn,tln,hn) ∧
  wf_End(h1,tl1,l12,l21,h2) ∧ wf_End(hk,tln,lkn,lnk,hn) ∧
  ∀ j:Nat • {j,j+1,j+2}⊆inds hll ⇒
    let (,(hj,ljj,lj′j)) = hll(j),((thj′,tlj′),(hj′,ljj′,lj′j′)) = hll(j+1) in
    wf_Plaza(thj′,tlj′,hj′) ∧ wf_Interm(ljj,lj′j,hj′,tlj′,ljj′,lj′j′) end end

wf_Plaza(th,tl,h) ≡
  obs_HΣ(th) = [ crossings at toll plazas ]
    {(external,obs_HI(th),obs_LI(tl)),
      (obs_LI(tl),obs_HI(th),external),
      (obs_LI(tl),obs_HI(th),obs_LI(tl))} ∧
     obs_HΩ(th)={obs_HΣ(th)} ∧
     obs_LΩ(tl) = {obs_LΣ(tl)}
```

$$
\begin{array}{l}
\mathsf{wf\_End(h,tl,l,l')} \equiv \\
\quad \mathsf{obs\_H\Sigma(h)} = [\text{crossings at 3}-\text{link end hubs}] \\
\quad\quad \mathsf{\{(obs\_LI(tl),obs\_HI(h),obs\_LI(tl)),(obs\_LI(tl),obs\_HI(h),obs\_LI(l)),} \\
\quad\quad\ \mathsf{(obs\_LI(l'),obs\_HI(h),obs\_LI(tl)),(obs\_LI(l'),obs\_HI(h),obs\_LI(l))\}} \wedge \\
\quad \mathsf{obs\_H\Omega(h)} = \{\mathsf{obs\_H\Sigma(h)}\} \wedge \\
\quad \mathsf{obs\_L\Omega(l)} = \{\mathsf{obs\_L\Sigma(l)}\} \wedge \mathsf{obs\_L\Omega(l')} = \{\mathsf{obs\_L\Sigma(l')}\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{wf\_Interm(ul\_1,dl\_1,h,tl,ul,dl)} \equiv \\
\quad \mathsf{obs\_H\Sigma(h)} = \{[\text{crossings at properly intermediate, 5}-\text{link hubs}] \\
\quad\quad \mathsf{(obs\_LI(tl),obs\_HI(h),obs\_LI(tl)),(obs\_LI(tl),obs\_HI(h),obs\_LI(dl\_1)),} \\
\quad\quad \mathsf{(obs\_LI(tl),obs\_HI(h),obs\_LI(ul)),(obs\_LI(ul\_1),obs\_HI(h),obs\_LI(tl)),} \\
\quad\quad \mathsf{(obs\_LI(ul\_1),obs\_HI(h),obs\_LI(ul)),(obs\_LI(ul\_1),obs\_HI(h),obs\_LI(dl\_1)),} \\
\quad\quad \mathsf{(obs\_LI(dl),obs\_HI(h),obs\_LI(tl)),(obs\_LI(dl),obs\_HI(h),obs\_LI(dl\_1)),} \\
\quad\quad \mathsf{(obs\_LI(dl),obs\_HI(h),obs\_LI(ul))\}} \wedge \\
\quad\quad \mathsf{obs\_H\Omega(h)} = \{\mathsf{obs\_H\Sigma(h)}\} \wedge \mathsf{obs\_L\Omega(tl)} = \{\mathsf{obs\_L\Sigma(tl)}\} \wedge \\
\quad\quad \mathsf{obs\_L\Omega(ul)} = \{\mathsf{obs\_L\Sigma(ul)}\} \wedge \mathsf{obs\_L\Omega(dl)} = \{\mathsf{obs\_L\Sigma(dl)}\}
\end{array}
$$

This ends Example 85 ■

8.6.6. Extension

Definition 70 – Extension: *By domain extension we understand an operation*

- *that applies to a (projected and possibly determined and instantiated) domain description, i.e., a (domain) requirements prescription,*
- *and yields a (domain) requirements prescription.*
- *The latter prescribes that a software system is to support, partially or fully, an operation that is not only feasible but also computable in reasonable time*

■

Example 86 – Timetable System Extension:

- We assume a projected and instantiated timetable (see Sect. 83 on Slide 737).
- A query of a timetable may, syntactically, specify an airport of origin, a_o, an airport of destination, a_d, and a maximum number, n, of intermediate stops.
- The query semantically designates the set of all those trips of one up to n direct air journeys between a_o and a_d, i.e., trips where the passenger may change flights (up to $n-1$ times) at intermediate airports.

```
scheme TI_TBL_3C =
  extend TI_TBL_3 with
    class
      type
        Query′ == Query | mk_conn(fa:An,ta:An,n:Nat)
        VAL′ = VAL | CNS
        CNS = (JR*)-set
      value
        𝓜q(mk_conn(fa,ta,n)) as
        pre ...
        post ...
    end
```

This ends Example 86 ∎

Example 87 – **Extension: A Toll-road System:** We continue Examples 77 and 85.

- In the rough sketch of the toll-road business processes (Example 77)
- references were made to a concept of a toll-booth.
- The domain extension is that of the controlled access of vehicles to and departure from the toll road net:
  - the entry to (and departure from) tollgates from (respectively to) an `"an external"` net — which we do not describe;
  - the new entities of tollgates with all their machinery;

- the user/machine functions:
  - upon entry:
    - driver pressing entry button,
    - tollgate delivering ticket;
  - upon exit:
    - driver presenting ticket,
    - tollgate requesting payment,
    - driver providing payment, etc.

- One added (extended) domain requirements:
  - as vehicles are allowed to cruise the entire net
  - payment is a function of the totality of links traversed, possibly multiple times.

– This requires, in our case,
  * that tickets be made such as to be sensed somewhat remotely,
  * and that intersections be equipped with sensors which can record
  * and transmit information about vehicle intersection crossings.
    · (When exiting the tollgate machine can then access the exiting vehicles sequence of intersection crossings — based on which a payment fee calculation can be done.)
    · All this to be described in detail — including all the thinks that can go wrong (in the domain) and how drivers and tollgates are expected to react.

• We suggest only some signatures:

type
  Mach, Ticket, Cash, Payment, Map_TN
value
  obs_Cash: Mach $\rightarrow$ Cash, obs_Tickets: M $\rightarrow$ Ticket-**set**
  obs_Entry, obs_Exit: Ticket $\rightarrow$ HI, obs_Ticket: V $\rightarrow$ (Ticket|nil)
  calculate_Payment: (HI×HI) $\rightarrow$ Map_TN $\rightarrow$ Payment
  press_Entry: M $\rightarrow$ M × Ticket [gate up]
  press_Exit: M × Ticket $\rightarrow$ M × Payment
  payment: M × Payment $\rightarrow$ M × Cash [gate up]

This ends Example 87 ■

8.6.7. Fitting

Definition 71 – Fitting: *By domain requirements fitting we understand an operation*

- *that applies to two or more, say m, projected and possibly determined, instantiated and extended domain descriptions, i.e., to two or more, say m, original domain requirements prescriptions,*
- *and yields $m + n$ (resulting, revised original plus new, shared) domain requirements prescriptions.*
- *The m revised original domain requirements prescriptions resulting from the fitting prescribe most of the original (m) domain requirements.*
- *The n (new, shared) domain requirements prescriptions resulting from the fitting prescribe requirements that are shared between two or more of the m revised original domain requirements*

■

Example 88 – **Fitting: Road Maintenance and Toll-road Systems:** We end the series of examples that illustrate requirements for a road maintenance respectively a toll-road system (Examples 77–87).

- We postulate two domain requirements:
  - We have outlined a domain requirements development for software support for road maintenance;
  - and we have outlined a domain requirements development for software support for a toll-road system.

- We can therefore postulate that there are two domain requirements developments, both based on the transport domain:
  - one, $d_{r_{\text{road-maint.}}}$, for a toll road computing system monitoring and controlling vehicle flow in and out of toll plazas, and
  - another, $d_{r_{\text{toll-road}}}$, for a toll link and intersection (i.e., hub) building and maintenance system monitoring and controlling link and hub quality and for development.

- The fitting procedure now identifies the shared of awareness of the net by both $dr_{\text{road-maint.}}$ and $dr_{\text{toll-road}}$ of nets (N), hubs (H) and links (L).
  - We conclude from this that we can single out a common requirements for software that manages net, hubs and links.
  - Such software requirements basically amounts to requirements for a database system.
  - A suitable such system, say a relational database management system, DBrel, may already be available with the customer.

– In any case, where there before were two requirements ($d_{r_{\text{road-maint.}}}$, $d_{r_{\text{toll-road}}}$) there are now four:
  * $d'_{r_{\text{road-maint.}}}$, a modification of $d_{r_{\text{road-maint.}}}$ which omits the description parts pertaining to the net;
  * $d'_{r_{\text{toll-road}}}$, a modification of $d_{r_{\text{toll-road}}}$ which likewise omits the description parts pertaining to the net;
  * $d_{r_{\text{net}}}$, which contains what was basically omitted in $d'_{r_{\text{road-maint.}}}$ and $d'_{r_{\text{toll-road}}}$; and
  * $d_{r_{\text{db:i/f}}}$ (for database interface) which prescribes a mapping between type names of $d_{r_{\text{net}}}$ and relation and attribute names of DB_{rel}.

• Much more can and should be said, but this suffices as an example.

This ends Example 88 ■

8.7. A Caveat: Domain Descriptions versus Requirements Prescriptions

8.7.1. Domain Phenomena

- When in the domain we describe simple entities by:

  type L, H, N

  then we mean that

  – L, H and N denote types of real, actually in the domain occurring phenomena
  – l:L, h:H and n:N
  – (as here, from Example 10, links, hubs and nets).

8.7.2. Requirements Concepts

- When, however, in the requirements, we describe simple entities by the same identifiers
  - then we mean that
  - L, H and N denote types of representation of domain phenomena l:L, h:H and n:N,
  - not the "the real thing", but "only" representations thereof.

8.7.3. A Possible Source of Confusion

- We have decided not to make a syntactic distinction between these two kinds of (simple entity, operation, event and behaviour) names.
- The context, that is, the fact that such names occur in a section on requirements, is enough, we think, to make the distinction clear.
- When there can be doubt, as we shall see in the next section, on Interface Requirements (Sect.), then we shall "spell out" the difference, viz., L (domain) versus LINK (requirements).

8.7.4. Relations of Requirements Concepts to Domain Phenomena

- We did not bother to warn the readerearlier,
  - about the possible source of confusion
  - that lies in mistaking a requirements concepts for "the real thing": its domain phenomena "counterpart".
- But we find that it is high time now,
  - before we enter the section on 'Interface Requirements' (just below),
  - to highlight that the simple entities, operations, events and behaviours referred to in requirements
  - are concept
  - whereas those of domains
  - are phenomena.

- The reason (for now emphasising the difference) is simple:
  - interface requirements are about the relations between
  - phenomena of the domain and
  - concepts of the software (being required).
- When,
  - in the domain, we name a (simple entity, operation, event or behaviour) phenomena $\mathcal{D}$, and when
  - in the requirements, we name a corresponding (simple entity, operation, event or behaviour) $\mathcal{R}$;
- then, by corresponding,
  - we mean that there is an unprovable relation

$$\mathcal{R} \models \mathcal{D}.$$

- We cannot possibly formally claim that

$$\mathcal{R} = \mathcal{D} \quad \text{or even} \quad \mathcal{R} \simeq \mathcal{D},$$

  - The $\mathcal{R}$ is a mathematical model of some requirements concept
  - whereas $\mathcal{D}$ is thought of as “the real thing”.
- Let us understand the above, seemingly contradictory statement:
  - The $\mathcal{D}$ is expressed mathematically, so it must be conceptual, as is $\mathcal{R}$.
  - Therefore they ought be comparable.
  - If we take this view that both are mathematical models, then all is OK and we can compare them.
  - If, however, we take the view that the names (of what is assumed, or claimed, to be domain phenomena in $\mathcal{D}$) denote “the real things, out there in the actual world”, then we cannot compare them.

- How do we, in the following, reconcile these two views ?
- We do so as follows:
  - On one hand we write $\mathcal{R} \models \mathcal{D}$ to mean that the requirements abstractly models the domain,
  - while, when we write
    - $\mathcal{R}$ abstractly refines $\mathcal{D}$

    or

    - abs_D($\mathcal{R}$eq) = $\mathcal{D}$om

    we mean that

    - the mathematical model of the requirements
    - is a refinement of the mathematical model of the domain —
    - in which latter phenomena names are considered names of mathematical concepts.

8.7.5. Sort versus Type Definitions

- As a principle we prefer to use sorts and observer functions:

Example 89 – Domain Types and Observer Functions:

type
  N, L, H, LI, HI, Location, Length
value
  obs_Ls: N → L-**set**, obs_Hs: N → H-**set**
  obs_LI: L → LI, obs_HI: H → HI
  obs_LIs: H → LI-**set**, obs_HIs: L → HI-**set**
  obs_Location: L → Location, obs_Length: L → **Real**

■

- rather then type definitions:

Example 90 – Requirements Types and Decompositions:

type
  LI, HI, Location, Length
  N = L-**set** × H-**set**
  L = LI × (HI×HI) × Location × Length × ...
  H = HI × LI-**set** × Length × ...
value
  (ls,hs):N, (li,(fhi,thi),loc,len,...):L, (hi,lis,len,...):H

■

- when defining simple domain entities.
- The type definitions are then typically introduced in requirements prescriptions.
- As shown in the last formula line of Example 90, the observer functions of domain descriptions can then be simply effected by decompositions.

8.7.5.1. Discussion

End of Lecture 13

Domain Requirements Engineering

Lecture 14

Interface Requirements Engineering

8.8. Interface Requirements

8.8.1. Acquisition

- Interface requirements acquisition evolves around a notion of
- shared phenomena and concepts of the domain. These are listed now.
- **Shared Simple Entities:** The shared simple entities are those simple entities that 'occur' in the domain but must also be represented by the machine.
- **Shared Operations:** The shared operations are those operations of the domain that can only be partially 'executed' by the machine.
- **Shared Events:** The shared events are those events of the domain that must be brought to the attention of the machine.
- **Shared Behaviours:** The shared behaviours are those behaviours of the domain that can only be partially 'processed' by the machine.

- Again the requirements engineers "walk" through the domain description together with each group of requirements stake-holders
  - marking up all the shared phenomena and concepts,
  - and decides their basic principles resolution,
  - which are duly noted in the evolving interface requirements document.

8.8.2. Shared Simple Entity Requirements

Definition 72 – Shared Simple Entity:

- *By a shared simple entity we understand*
  - *a simple entity that 'occurs' in the domain*
  - *but must also be represented by the machine.*

■

Example 91 – Shared Simple Entities: Railway Units:

- We may think of a train traffic monitoring and control system being interface requirements developed.
- The following phenomena are then identified as among those being shared:
  - *rail units,*
  - *signals,*
  - *road level crossing gates,*
  - *train sensors* (optical sensor sensing passing trains) and
  - *trains.*

This ends Example 91 ■

Example 92 – **Shared Simple Entities: Toll-road Units:**

- We may think of a toll-road traffic monitoring and control system being interface requirements developed.
- The following phenomena are identified as among those being shared:
  - *links,*
  - *hubs,*
  - *cars,* (optical sensors sensing passing cars)
  - *toll-both gates,*
  - *toll-booth externally arriving car sensor,*
  - *toll-booth internally arriving car sensor,*
  - *toll-booth request slip sensor* and
  - *toll-booth accept slip sensor.*

This ends Example 92 ■

Example 93 – Shared Simple Entities: Transport Net Data Representation:

- We deliberately formulated
  - Examples 91: "Shared Simple Entities: Railway Units"
  - and 92: "Shared Simple Entities: Toll-road Units"
- so as to conjure the image of two very similar set of requirements.
- These are now made into one set. In this example we focus on the machine representation of simple entities.
- We now continue these examples as well as Example 84.

290. The shared simple entities are the links and the hubs. In the domain we referred to these by the sort names *L* and *H*, in the machine they will be represented by the types *LINK* and *HUB*

291. Now we must make sure that we can abstract *LINK*s and *HUB*s "back" into *L* and *H*.

292. A number of properties that could be observed of links and hubs in the domain must be represented, somehow, in the machine. (Again we refer to Example 84.) Some properties are:
    - link and hub *Location*,
    - link *Length*,
    - road (link and hub) *Surface* material (concrete, macadamised, dirt road, etc.),
    - road (link and hub) *WearTear* (surface quality),
    - *Date* last surveyed (i.e., monitored)
    - *Date* last maintained (i.e., controlled) with respect to surface quality,
    - next scheduled *Date* of survey, etc.

293. Let us call the pair of sets of representations of *LINK*s and *HUB*s for *NET*. We omit, in this example, the modelling of net attributes.
294. We postulate an abstraction function, abs_N, which from a concretely represented *net:NET* abstracts the abstract *n*(et) in *N*(et).
295. Tentatively we might impose the following representation theorem (a relation) between concrete and abstract nets: the links [hubs] (in *L* [in *H*]) that can be abstracted from any concrete net *net* must be those observable in the abstracted net.

```
type
290  Length, Surface, WearTear, Date, L_Location, H_Location
292  LINK = LI×(HI×HI)×L_Location×Length×Surface×WearTear×(Date×Date×Date)×
292  HUB = HI×H_Location×Surface×WearTear×(Date×Date×Date)× ...
293  NET = LINK-set × HUB-set
value
291  abs_L: LINK → L, abs_H: HUB → L
294  abs_N: NET → N
theorems:
294  ∀ (links,hubs):NET, ∃ n:N •
294    (links,hubs) ⊨ n ∧
294    abs_N(links,hubs) abstractly refines n ∧
295    let ls = {abs_L(link)|link:LINK • link ∈ links}
295        hs = {abs_H(hub)|hub:HUB • hub ∈ hubs} in
295    obs_Ls(abs_N(net))⊨ls ∧ obs_Hs(abs_N(net))⊨hs end
```

- We shall discuss the representation of concrete hubs in Example 94.

This ends Example 93 ■

- In Example 93 ("Shared Simple Entities: Transport Net Data Representation")
- we kept an abstract representation of links and hubs.
- At some time in the software development process we are forced to decide on a concrete type representation of links and hubs so that we can implement those types through the use of a practical programming language.
- Here we shall choose, as already hinted at in Example 88, to represent links and hubs as tuples in relations of a relational database management system.

Example 94 – Representation of Transport Net Hubs:

- We continue Example 93 ("Shared Simple Entities: Transport Net Data Representation").
- With the hints given in the text paragraph just preceding this example
- we are now ready to suggest a concrete type for *LINK*s and *HUB*s,
  - namely as tuples or respective relations,
  - but with the twist that we do not endow a concrete hub representation with the set of link identifiers that, in the domain, can be observed from that hub
  - since, as we shall shortly show, that information can be calculated from the set of links having the same hub identifiers as that of the hub.

- You may now object,
  - if you did not already wonder way back in Example 10,
  - as to why we did not already include this in the domain model of the net.
  - The answer is: Yes, we could have done that, but we prefer to have modelled links and hubs, as we did it in Example 10, since we think that that is a most abstract, "no tricks" model.
  - The "tricks" we refer to is represented below by the *xtr_LIs* function.

```
type
  L_Location, H_Location
  LINK = LI×(HI×HI)×L_Location×Length×Surface×WearTear×(Date×Dat
  HUB = HI×H_Location×Surface×WearTear×(Date×Date×Date)× ...
  RDB = LINK-set × HUB-set
value
  xtr_LIs: HUB → RDB → LI-set
  xtr_LIs(hi,hl,s,wt,(ld,md,nd),...)(ls,hs) ≡
    {li|li:LI•
      ∃ link:(li′,(hi′,hi″),ll,lgt,s,wt,(d′,d″,d‴),...):LINK•
      link ∈ ls ∧ (hi=hi′ ∨ hi=hi″)}
```

This ends Example 94 ■

- The requirements example that now follows
- to some extent deviate from the ideal of expressing requirements:
  - rather than expressing properties, the *what*,
  - these requirements express an *abstract design*, the *how*.
- One might very well claim that the example that now follows
  - really should be moved to a section on Software Design
  - since it can be said to be such an *abstract design*.
- Be that as it may,
  - we have chosen to place the next example here,
  - under shared entity data initialisation,
  - as it illustrates that concept rather well.
- The point is that
  - the more we include considerations of the machine the
  - the more operational, that is, the less *what* the more *how*
  - the interface requirements becomes.

Example 95 – Shared Simple Entities: Transport Net Data Initialisation:

- We continue Example 93 ("Shared Simple Entities: Transport Net Data Representation").
- We now focus on the initialisation of simple entity data.

296. Input of representations of simple transport net entities, that is, representations of links and hubs, is by means of a software package, call it `NetDataInput`.
297. `NetDataInput` assumes a rather old-fashioned constellation of a graphic user interface (GUI), `NetDataGUI`, in-data and a conventional relational database `NetDataRDB`.
298. The `NetDataRDB` is here thought of as just consisting of two relations `ls:LINKS` and `hs:HUBS`.

299. Each relation consists of a set of `LINK`, respectively `HUB` “tupleisations” of links and hubs — which, to repeat, are representations of links and hubs .

300. When `NetDataInput` is invoked, the `NetDataInput` GUI shall open in a window with a click-able, simple *either/or* choice icon: `Road Net` or `Rail Net`.

301. Clicking one of these shall result in replacing the *either/or* window being replaced by a window for the input of net units for the selected choice of net.

In the following we shall treat only the `Road Net` variant of this interface requirements.

302. The `Road Net` (GUI) window has the following alternative click-able choice icons: `link` and `hub`.

303. Clicking the `Road Net` `link` icon shall result in replacing the `Road Net` window being replaced by a window for the input of representations of road links. Similarly for clicking the `Road Net` `link` icon.

In the following we shall treat only the `Road Net` `link` icon variant of this interface requirements.

304. The `Road Net link` window has the following named fields:

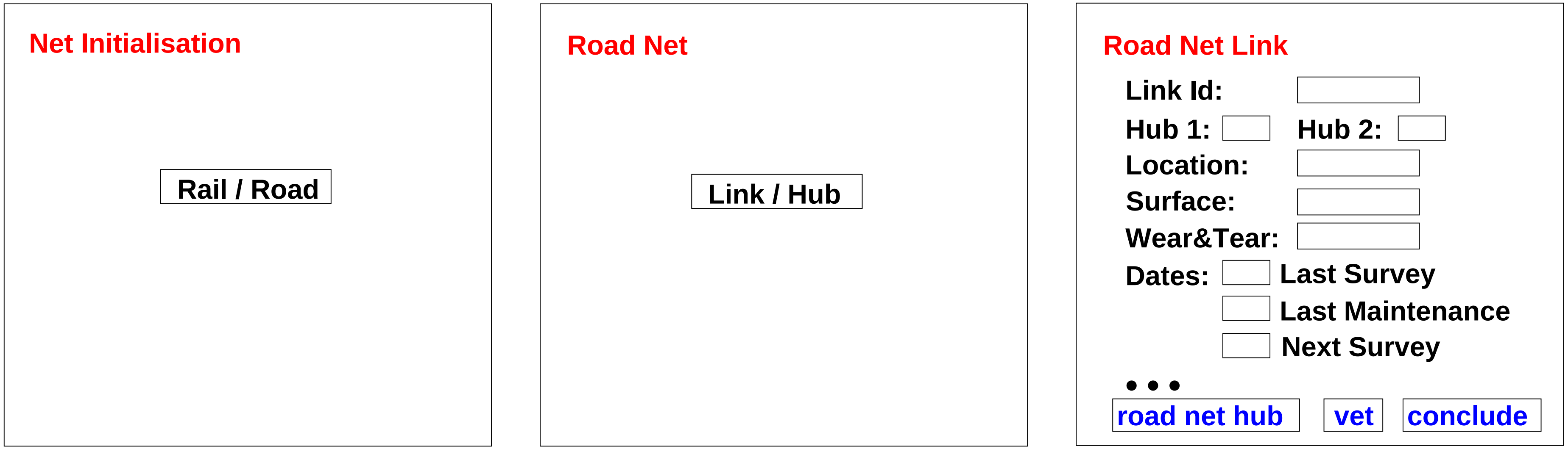
- `link identifier` FILL IN
- `hub identifier 1` FILL IN
- `hub identifier 2` FILL IN
- `link location` FILL IN
- `link surface` FILL IN
- `link wear & tear` FILL IN
- a triple of link dates:
  - `last survey` FILL IN
  - `last maintenance` FILL IN
  - `next survey` FILL IN
- et cetera.
- `submit`

- Each field, except for the `submit` icon, consists of a name part and an input, fill in, part.
- (In the formalisation below the type names "cover" both parts.)
- The system shall assign unique link identifiers, i.e., "fill-in" the link identifier automatically.

305. Clicking the `submit` icon shall result in the following checks:

- The composite in-data, keyed into the input parts of the `Road Net link` window fields are vetted:
  - Is the `link identifier` already defined ?
  - Do the `location` co-ordinates conflict with earlier input ?
  - Are the dates in an appropriate chronological order ?
  - Et cetera.
- If checks are OK, then the following actions are performed:
  - The *NetDataRDB* `link` relation is updated to reflect the new link tuple.
  - The `Road Net` system then reverts to the `Road Net link` window, allowing, however, the input staff to select the alternative
  - (i) `Road Net hub` window, or
  - (ii) to request a partial or full vetting of the state of the *NetDataRDB* `link` and `hub` relations,
  - (iii) or to conclude the input on net data.

(8. **Requirements Engineering** 8.8. **Interface Requirements** 8.8.2. **Shared Simple Entity Requirements**)



■ ■ ■

Figure 15: Three snapshots of `NetDataInput`

type

296 NDI
298 RDB = Links × Hubs
299 Links = LINK-**set**
299 Hubs = HUB-**set**
300 GUI == EitherOrW | RoadNetW | RailNetW
300 EitherOrW == roadnet | railnet
302 RoadNetW == link | hub
304 LINK = LI×(HI×HI)×L_Location×Surface×WearTear×(Date×Date×Date)× ...
304 HUB = HI×H_Location×Surface×WearTear×(Date×Date×Date)× ...
Response == ok | (not_ok × Error_Msg)
Error_Msg

value

297 obs_GUI: NDI → GUI, obs_RDB: NDI → RDB
301 select_RoadOrRail: GUI → GUI
303 select_Link_or_Hub: GUI → GUI
305 submit_Link_input: GUI → GUI × Response

- We leave it to the reader to complete the interface requirements for shared simple entity initialisation.

This ends Example 95 ■

8.8.3. Shared Operation Requirements

Definition 73 – **Shared Operation:** *By a shared operation we understand an operation of the domain that can only be partially 'executed' by the machine — with the remaining operation parts being "executed" by a human or some "gadget" of the domain "outside" of our concern.* ■

- We start by giving a consolidated domain description of a fragment of a financial services industry,
  - in other words: Example 96
  - is not a requirements prescription.
- But it will be the basis for a shared operations interface requirements prescription.

Example 96 – Shared Operations: Personal Financial Transactions:

- With the advent of the Internet,
  - i.e., computing and communications,
  - and with the merging of in-numerous functionalities of the financial service sector,
  - we are witnessing the ability of some clients of the financial service industry to handle most of their transactions "themselves".
- In the first part, Items 306–314 of this example, we rough-sketch the state and the signatures of some of the client operations of a financial service industry.
- In the second part, Items 319–331, we rough sketch client states and behaviours.

306. The financial service industry includes one or mare banking and securities instrument trading services.
    (a) Banks are uniquely identified (*BId*).
    (b) Banks offer accounts:
        i. a client may have one or more demand/deposit accounts and
        ii. one or more mortgage accounts, identified by account numbers;

        accounts, in this simplified example, holds a balance of (deposited or mortgaged) money.
    (c) Two or more clients may share accounts and bank registers correlate client names (*C*) to account and (*A*) mortage (*M*) account numbers.
    (d) We do not describe bank identifiers, client names, demand/deposit account numbers, mortgage account numbers,

type

306 Banks, SecTrad

306(a) Banks = BId $\overrightarrow{m}$ Bank

306(b) Bank = Registers × Accounts × Mortgages × ...

306((b))i Accounts = A $\overrightarrow{m}$ Account

306((b))ii Mortgages = M $\overrightarrow{m}$ Mortgage

306(c) Registers = C $\overrightarrow{m}$ (A|M)-**set**

306(d) C, A, BId, Account, Mortgage, Account, Mortgage, OrdNr

307. Bank clients
    (a) *open account* and
    (b) *close* (demand/deposit and mortgage) *accounts*,
    (c) *deposit* money into accounts,
    (d) *transfer* money to (possibly other client) accounts, and
    (e) *withdraw* (cash) money (say, through an ATM).
308. Client supplied arguments to and
309. responses from these banking operations are also not further described.

value

307(a) openacct: $\text{Arg}^* \rightarrow \text{Banks} \rightarrow \text{Banks} \times \text{Response}$

307(b) closeacct: $\text{Arg}^* \rightarrow \text{Banks} \rightarrow \text{Banks} \times \text{Response}$

307(c) deposit: $\text{Arg}^* \rightarrow \text{Banks} \rightarrow \text{Banks} \times \text{Response}$

307(d) transfer: $\text{Arg}^* \rightarrow \text{Banks} \rightarrow \text{Banks} \times \text{Response}$

307(e) withdraw: $\text{Arg}^* \rightarrow \text{Banks} \rightarrow \text{Banks} \times \text{Response}$

type

308 Arg = BId | C | A | M | OrdNr | Amount | Cash | Date | Time

309 Response = (A | M | Cash | ... | Date | Time)-**set**

310. A securities exchange keeps track of buy and sell offers, of suspended such offerings and of transacted trading.
311. Basic concepts of trading, apart from buying, selling, suspension and concluded trading, are
    (a) securities instrument identifications (*SId*);
    (b) quantities offered for selling or buying, or traded (*Quant*);
    (c) the order numbers of placed offers (*OrdNr*),
    (d) prices (*Price*),
    (e) dates (*Date*) and
    (f) times (*Time*).

type

310 SecTrad = BuyOfrs × SellOfrs × Suspension × Tradings

311 SId, Quant, OrdNr, Price, Date, Time, ...

312. Securities trading allows clients
    (a) to place buy and
    (b) sell offers,

    giving their client and bank identification, their bank demand/deposit account number (from which to withdraw [i.e., demand], resp. into which to deposit) buying or selling prices), the securities instrument [e.g., stock] identifier, quantity to be bought or sold, the high, respectively the low price acceptable, and the last date of the offer.
313. Clients may inquire as to the trading status of their offer.
314. We can therefore think of the following kinds of client transaction "codes" (*Cmd*): *omkt* (observe the market), *open* (some kind of bank or securities trading account: demand/deposit, mortgage, trading, etc.), *deposit, withdraw, transfer, close, buy offer, sell offer, inquire,* et cetera.

value

312(a) int_buyofr: Arg* → SecTrad → SecTrad × Response
312(b) int_sellofr: Arg* → SecTrad → SecTrad × Response
313 trading: SecTrad → SecTrad × Response

type

314 Cmd == obsmkt|analmkt|openacct|deposit|withdraw|transfer|closeac

We can, finally, suggest crucial components of the securities exchange state:

315. *BuyOfr*s map client names, *C*, into (client) bank identifiers, *BId*, and client account numbers, *A*, which then map into *OrdNr*s, which (then again) map into a quadruple of securities instrument identifications, *SId*,*Quant*ity of instrument to be bought, the preferred lowest *Price* and the *Date* of placement or order.

316. *SellOfr*s have same components as buy offers — but now the *Price* designate a highest price.

317. *Suspension*s just list order number and date and time of suspension.

318. *Tradings* list pertinent information.

type

315 $\quad \mathrm{BuyOfr} = \mathrm{OrdNr} \ \overrightarrow{m} \ (\mathrm{C} \times \mathrm{BId} \times \mathrm{A}) \ \overrightarrow{m} \ (\mathrm{SId} \times \mathrm{Quant} \times \mathrm{Price} \times \mathrm{Date})$

316 $\quad \mathrm{SellOfr} = \mathrm{OrdNr} \ \overrightarrow{m} \ (\mathrm{C} \times \mathrm{BId} \times \mathrm{A}) \ \overrightarrow{m} \ (\mathrm{SId} \times \mathrm{Quant} \times \mathrm{Price} \times \mathrm{Date})$

317 $\quad \mathrm{Suspension} = \mathrm{OrdNr} \ \overrightarrow{m} \ (\mathrm{C} \times \mathrm{Date} \times \mathrm{Time})$

318 $\quad \mathrm{Tradings} = \mathrm{OrdNr} \ \overrightarrow{m} \ \mathrm{SId} \times \mathrm{Quant} \times \mathrm{Price} \times (\mathrm{C}\times\mathrm{Bid}\times\mathrm{A}\times(\mathrm{Date}\times\mathrm{Time}))$

- We now rough-sketch a concept of 'personal finance management' operations.
  - It is in this part, not the first, that this example reveals that it is an example of an operation that is shared between the domain and the machine.
  - We maintain, however, that the example is still that of a domain description.
  - The operation is that of a client managing own, personal finances.

– This 'personal finance management' operation is a composite operation.

  * It is a sequence of "one-step" operations, each operation being a banking or a securities trading. (For simplicity, but without any loss of generality, we limit the example to just these two sets of operations.)
  * Each such operation results in a date- and time-stamped response (Item 309 on Slide 802).
  * Each response is studied by the client.
  * The client may then decide to proceed with further 'one-step' operations or end this sequence "at this time" — allowing, of course, the client to resume 'personal finance management' at a later 'personal finance management session'.

309. Each of the banking and securities instrument operations result in a response.

319. This response becomes part of the client's 'finance management' state, $\Pi\Phi\Sigma$.

320. We refer to the global financial service industry state as Ω.

    Besides the banks and securities trading, the global financial service industry state (ω:Ω) is thought of as including all those aspects of the clients of this industry which affects and/or reflects the financial situation.

321. A 'personal finance management (*pfm*) *session*' is now a conditional iteration (formula Line 325 below) of personal finance management operations.

322. A client can always *obs*erve the dated and timed *responses* received as a result of past *p*ersonal *f*inance *m*anagement operations.

323. An iteration of 'personal finance management' starts with the client analysing (*ω_anal_mkt*) the market based on *past responses*;

324. followed by an analysis (*cli_anal_mkt*) of the personal financial situation based on the market *response*.

325. If the analysis advises some 'personal finance management'

326. then the client inquires, *what_to_do*, past responses and as to which transaction, *cmd*, and with which arguments, *argl*, such a transaction should be performed.

    This operation, *what_to_do*, is not computable. It is an operation performed basically by the client.

327. The client then performs (*Int_Cmd*) this (i.e., the *cmd*) transaction. The transaction usually transforms the finance industry state (ω) into a next state (ω') and always yields a *date*- and
328. Once the transaction has been concluded the client reverts to the 'personal finance management (pfm) session' with an updated "past responses" (merge_pfm) and in the new global state, ω'.
329. Else, that is, if the analysis "advises" no transactions, the 'personal finance management' state, $\pi\phi\sigma$ and the global financial state, ω, is left unchanged and the (i.e., this) session ends.
330. To perform a transaction depends on which kind of transaction, cmd, has been advised.
331. We leave the interpretation of Int_Cmd to the reader.

π The lines, below, marked π designate actions that are performed by the client or jointly between the client and the financial system (designated by an ω or ω' argument).

type
319–320 $\Pi\Phi\Sigma$, Ω
322 Responses = (Date$\times$Time) $\xrightarrow[m]{}$ Response
314 Cmd == obsmkt|analmkt|openacct|deposit|withdraw|transfer|closeacct|...|buyofr|sellofr|trading|...
value
322 obs_Responses: $\Pi\Phi\Sigma \to$ Responses
321 pfm_session: $\Pi\Phi\Sigma \to \Omega \to \Omega \times \Pi\Phi\Sigma$
321 pfm_session($\pi\phi\sigma$)(ω) $\equiv$
322π **let** past_responses = obs_Responses($\pi\phi\sigma$) **in**
323π **let** ω_response = ω_anal_mkt(past_responses)(ω) **in**
324π **let** $\pi\phi\sigma$_response = cli_anal_mkt(response)($\pi\phi\sigma$) **in**
325π **if** advice_pfm_action($\pi\phi\sigma$_response)
326π **then let** (cmd,argl) = what_to_do($\pi\phi\sigma$_response)(ω) **in**
327 **let** (response,ω') = Int_Cmd(cmd,argl)(ω) **in**
328π pfm_session(merge_pfm(response,date,time)($\pi\phi\sigma$))(ω') **end end end**
329 **else** (ω,$\pi\phi\sigma$) **end end end**
323 ω_anal_mkt: Responses $\to \Omega \to$ Response
324π cli_anal_mkt: Response $\to \Pi\Phi\Sigma \to$ Response
325 advice_pfm_action: Response $\to$ **Bool**
326π what_to_do: Responses $\to \Omega \to$ Cmd $\times$ Arg*
328π merge_pfm: Responses $\to \Pi\Phi\Sigma \to \Omega \to \Omega \times \Pi\Phi\Sigma$

330. To perform a transaction depends on which kind of transaction, cmd, has been advised.

332. A case distinction is made between the very many kinds of transactions (listed in Item 314).

333. The *obs_mkt* and *anal_pfm* operations do not change the state of the financial industry.

    We think of these operations as not being computable functions. Rather we think of them as a more-or-less “informed” study, by the client of the market of financial instruments including the status of those enterprises whose stocks are traded.

334. The *arg*ument *l*ist of the *open* transaction indicates which kind of account is to be established (demand/deposit, mortgage, etc.).

335. The *arg*ument *l*ist of the *buy offer* transaction indicates which kind of securities (stocks, oil, metals, or other commodities) is sought, in which quantity, at which price level, up till which date, et cetera.

```
value
330   Int_Cmd: (Cmd × Arg*) → Ω → Ω × Response
330   Int_Cmd(cmd,argl)(ω) ≡
332     case cmd of
333π        obsmkt → (ω,eval_obs_mkt(argl)(ω)),
333π        analmkt → (ω,ω_anal_mkt(argl)(ω)),
        ...
334π        openacct → int_open_acct(argl)(ω),
        ...
335π        buyofr → int_buyofr(argl)(ω),
        ...
332     end

333π  eval_obs_mkt: Arg* → Ω → Response
333π  ω_anal_mkt: Arg* → Ω → Response
334   int_open_acct: Arg* → Ω → Ω × Response
335   int_buyofr: Arg* → Ω → Ω × Response
```


This ends Example 96 ■

- The reader may well ask:
  - What in Example 96
  - illustrates the shared operations interface requirements?
  - We have already indicated part of the answer to this question by the π annotations.
- Why is this a reasonable question?
  - It is a reasonable question because we have not made that abundantly clear.
  - That is, we have not discussed the placement of π annotations in much detail.
  - Example 96 could really be construed as a domain description based in the intrinsics, support technology, management and organisation and human behaviour regime.

8.8.4. Shared Event Requirements

Definition 74 – **Shared Event:** *By a shared event we understand an event of the domain that must be brought to the attention of the machine.* ∎

- We defer the exemplification of shared events till our treatment of ‘shared behaviour’.
- We therefore progress right on to exemplify ‘shared behaviours’.
- The ‘step of development’,
  - from the specification of Example 96 (*Shared Operations*)
  - to the specification of Example 97 (*Shared Behaviours*)
- is not a formal refinement:
  - but it can be made into such a formally verifiable refinement.
  - So we pose that as a relevant MSc Thesis topic.

8.8.5. Shared Behaviour Requirements

Definition 75 – Shared Behaviour:

- *By a shared behaviours we understand a behaviour of the domain*
  - *that can only be partially 'processed' by the machine —*
  - *with the remaining behaviour being provided by*
  - *humans or some "gadgets" of the domain, "outside" of our concern.*

■

Example 97 – Shared Behaviours: Personal Financial Transactions:

336. There is an index set, *CI*, of clients.
337. For each client there is an "own"
338. 'personal finance management' state $\pi\phi\sigma_{ci} : \Pi\Phi\Sigma$ (cf. 319 on Slide 810).
339. The finance industry "grand state" $\omega : \Omega$ is as before (cf. Item and formula line 320 on Slide 810).
340. The system consists of
    (a) an indexed set of *client* behaviours
    (b) and one finance industry "grand state" behaviour *omega*.
341. We model communications between clients and the financial industry to occur over client-industry channels.
342. We model communications over these channels as being of type *M*. *M* will be "revealed" as we go on.

type

336 CI

337 $\Pi\Phi\Sigma$s = CI $\overrightarrow{m}$ $\Pi\Phi\Sigma$

338 $\Pi\Phi\Sigma$

339 Ω = Banks $\times$ SecTrad $\times$ ...

value

336 cis:CI-**set**

337 $\pi\phi\sigma$s:$\Pi\Phi\Sigma$s

339 ω:Ω

340 system: **Unit** $\rightarrow$ **Unit**

340 system() $\equiv$

340(a) $\|$ {client(ci)($\pi\phi\sigma$s(ci))|ci:CI•ci $\in$ cis}

340(b) $\|$ omega(ω)

channel

341 {c_ω_ch[ci]|ci:CI•ci $\in$ cis} M

type

342 M

- Let us first consider the issue of events.
- First the events arise in "the market",
- here symbolised with the global state $\omega : \Omega$.

343. The *omega*(ω) behaviour and the *client(ci)*($\pi\phi\sigma$*s(ci)*) behaviours, for all *clients*, are cyclic — expressed through 'tail recursion' over possibly updated states.

344. To model events we let the *omega*(ω) behaviour alternate between either
   (a) inquiring its state as to unusual situations in, the status *status* of, "the market", and,
   (b) if so, inform an arbitrary subsets of *clients* of such "events" and
   (c) continuing in an unchanged global financial system state

   or

345. servicing *client* requests — from any client.

type

Event

value

omega: $\Omega \rightarrow$ **in**,**out** {c_ω_ch[ci]|ci:CI·ci $\in$ cis} **Unit**

omega(ω) $\equiv$

344(b) (**let** scis:CI-**set** · scis$\subseteq$cis **in**

344(a) **let** event = ωstatus(ωsnapshot(ω)) **in**

344(b) **if** nok_status(event) **then** {c_ω_ch[ci]!event|ci:CI·ci $\in$ scis} **end**

344(c) omega(ω) **end end**)

344 □

345 □ {**let** req = c_ω_ch[ci] ? **in** ... see Items350–357(b) ... **end** | ci:CI ·

344(a) ωsnapshot: $\Omega \rightarrow \Omega$

344(a) ωstatus: $\Omega \rightarrow$ Event

344(b) nok_status: Event $\rightarrow$ **Bool**

- Then we consider how clients respond
- to becoming aware of unusual events in “the market”.

346. Clients alternate between handling events:
347. first deciding (345–347) to “listen to the market”,
    (a) then updating an own personal finance state ($\pi\phi\sigma$),
    (b) and then coming a client behaviour in that new personal finance state,

    and
348. handling ordinary, that is, personal finance management (*pfm*) or
349. just idling.

value

346 client: ci:CI $\rightarrow \Pi\Phi\Sigma \rightarrow$ **in**,**out** c_ω_ch[ci] **Unit**

346 client(ci)($\pi\phi\sigma$) $\equiv$

347 (**let** event = c_ω_ch[ci]? **in**

347(a) **let** $\pi\phi\sigma'$ = update_$\pi\phi\sigma$($\pi\phi\sigma$)(event) **in**

347(b) client(ci)($\pi\phi\sigma'$) **end end**)

346 $\bigcap$

348 client(ci)(pfm_session(ci)($\pi\phi\sigma$))

346 $\bigcap$

349 client(ci)($\pi\phi\sigma$)

- Let us now turn to the treatment of usual financial transactions.
- The main functions, in Example 96, are
  - *pfm_session* (Items 321–329 and formulas on Slides 813–813) and
  - *Int_Cmd* (Items 332–335 and formulas on Slides 815–815)
- We now analyse these two functions.
- We refer, in the following to the formula lines on Slides 813–813 and Slides 815–815.
- The analysis is with respect to
  - what actions, π, are expected to occur in the *client* behaviour and
  - what actions are expected from the financial industry (i.e., to occur in the *omega* behaviour).

- In *pfm_session* (Slides 813–813), formula lines
  - 322π: *obs_Responses*$(\pi\phi\sigma)$,
  - 324π: *cli_anal_mkt(response)*$(\pi\phi\sigma)$,
  - 325π: **if** *ok_or_nok* = *ok* and
  - 328π: *merge_pfm(response,date,time)*,

  are expected from the *client* behaviour, all others from the industry, i.e., the *omega* behaviour.
- In *Int_Cmd* (Slides 815–815), formula lines
  - 333π: *eval_obs_mkt(argl)*(ω),
  - 333π: ω*_anal_mkt(argl)*(ω),
  - 334π: *int_open_acct(argl)*(ω),
  - ...,
  - 335π: *int_buyofr(argl)*(ω),
  - ...,

  are expected from the *client* behaviour.

- Of the functions itemised above, the

350. (323π) *ω_anal_mkt(past_responses)(ω)*,
351. (333π) *eval_obs_mkt(ω)*,
352. (334π) *int_open_acct(argl)(ω)*,
353. . . . ,
354. (335π) *int_buyofr(argl)(ω)*,
355. . . . ,

  functions — as invoked by the *client*, in the *pfm_session* and the *Int_Cmd* behaviours —

- require access to the global financial service industry state ω.
- The idea is now, for the *client*,
  - in its *Int_Cmd* (see Formula lines 330.0 onwards [Slide 833])
  - to communicate these functions, as function named argument lists,
  - cf. Formula lines 350–355 below.

type

FCT == Anal_mkt|Obs_mkt|...|Open_aact|...|Buy_Ofr|...

350 Anal_mkt = mkAnalMkt(argl:Arg*)

351 Obs_mkt = mkObsMkt(argl:Arg*)

352 Open_acct = mkOpenAcct(argl:Arg*)

353 ...

354 Buy_Ofr = mkBuyOfr(argl:Arg*)

355 ...

channel

341 {c_ω_ch[ci]|ci:CI•ci $\in$ cis}: Event | FCT | Response

356. The *omega* behaviour thus alternates
    (a) between accepting and responding to either of the many forms of functions, *FCT*
    (b) or generating event notifications.
357. If the *omega* behaviour of its own will, that is, internally non-deterministically chooses to accept a client initiate request it externally non-deterministically chooses which client request to serve.
    (a) The *omega* behaviour deciphers the request;
    (b) applies the communicated function to (possibly communicated arguments) and the ω"grand state"; and communicates the result back to the chosen client.

```
value
   omega: Ω → in,out {c_ω_ch[ ci ]|ci:CI•ci ∈ cis}  Unit
   omega(ω) ≡
356(b)     (let scis:CI-set • scis⊆cis in
356(b)      let event = ωstatus(ωsnapshot(ω)) in
356(b)      if nok_status(event) then {c_ω_ch[ ci ]!event|ci:CI•ci ∈ scis} end;
356(b)      omega(ω) end end)
356    ⌷
357    ⌷ {let req = c_ω_ch[ ci ] ? in
357(a)         case req of
356(a)           mkObsMkt(argl) → c_ω_ch[ ci ] ! eval_obs_mkt(argl)(ω) ; omega(ω),
356(a)           mkAnalMkt(argl) → c_ω_ch[ ci ] ! ω_anal_mkt(argl)(ω) ; omega(ω),
356(a)           mkOpenAcct(argl) →
357(a)             let (ω′,res) = int_open_acct(argl)(ω) in c_ω_ch[ ci ] ! res ; omega(ω′) end,
356(a)           ...
356(a)           mkBuyOfr(argl) →
357(a)             let (ω′,res) = int_buyofr(argl)(ω) in c_ω_ch[ ci ] ! res ; omega(ω′) end,
356(a)           ...
357(a)         end end | ci:CI • ci ∈ cis}
```

Some auxiliary functions:

value

333π eval_obs_mkt: Arg$^* \rightarrow \Omega \rightarrow$ Response
333π ω_anal_mkt: Arg$^* \rightarrow \Omega \rightarrow$ Response
334 int_open_acct: Arg$^* \rightarrow \Omega \rightarrow \Omega \times$ Response
335 int_buyofr: Arg$^* \rightarrow \Omega \rightarrow \Omega \times$ Response

- There is only minor changes, marked $\surd$, to pfm_session:
  - (321) an additional argument, ci:CI and
  - (327) an additional argument, (ci) to Int_Cmd(ci)(cmd,argl).

```
value
321√  pfm_session: ci:CI → ΠΦΣ → in,out ωch  ΠΦΣ
321√  pfm_session(ci)(πφσ) ≡
322     let past_responses = obs_Responses(πφσ) in
323     let response = ωch[ ci ]!mkObsMkt(past_responses); ωch[ ci ] ? in
324     let response = analyse_pfm(past_responses)(πφσ) in
325     if advice_pfm_action(response)
326       then (let (cmd,argl) = what_pfm_to_do(response) in
327√            let response = Int_Cmd(ci)(cmd,argl) in
328√            pfm_session(ci)(merge_pfm(response)(πφσ)) end end) end
329       else πφσ end end end

325  advice_pfm_action: Response → Bool
324  analyse_pfm: Responses → ({|ok|}|Event) → {|ok|nok|}
```

- Similarly the changes to Int_Cmd are obvious and
- marked, as before, with $.i, i = 0, 1, 2, 3$:

value

```
330.0  Int_Cmd: ci:CI × (Cmd × Arg*) → in,out ωch  Response
330    Int_Cmd(ci)(cmd,argl) ≡
330       case cmd of
330.2        obsmkt → ωch[ ci ]!mkObsMkt(argl) ; ωch[ ci ]?
330.2        analpfm → ωch[ ci ]!mkAnalMkt(argl) ; ωch[ ci ]?
330          ...
330.2        openacct → ωch[ ci ]!mkOpenAcct(argl) ; ωch[ ci ]?
330          ...
330.3        buyofr → ωch[ ci ]!mkBuyOfr(argl) ; ωch[ ci ]?
330          ...
330       end
```

This ends Example 97 ■

8.8.5.1. Discussion

-

- TO BE WRITTEN

-

-

End of Lecture 14

Interface Requirements Engineering

Lecture 15

Reqs. Eng.: Machine Reqs. & Closing Stages

8.9. Machine Requirements

Definition 76 – **Machine Requirements:** *By machine requirements we understand*

- *those requirements that can be expressed*
- *sôlely in terms of (or with prime reference to) machine concepts*

■

8.9.1. An Enumeration of Machine Requirements Issues

- There are many separable machine requirements.
- To find one's way around all these separable machine requirements we shall start by enumerating the very many that we shall overview.

8.9.2. Performance Requirements

Definition 77 – Performance Requirements: *By performance requirements we understand machine requirements that prescribe*

- *storage consumption,*
- *(execution, access, etc.) time consumption,*
- *as well as consumption of any other machine resource:*
  - *number of CPU units (incl. their quantitative characteristics such as cost, etc.),*
  - *number of printers, displays, etc., terminals (incl. their quantitative characteristics),*
  - *number of "other", ancillary software packages (incl. their quantitative characteristics),*
  - *of data communication bandwidth,*
  - *etcetera*

■

- Pragmatically speaking, performance requirements translate into financial resources spent, or to be spent.

Example 98 – **Timetable System Performance:** We continue Example 86 on Slide 755.

- The machine shall serve 1000 users and 1 staff member.
- Average response time shall be at most 1.5 seconds,
- when the system is fully utilised.

This ends Example 98 ■

8.9.2.1. General

- Till now we may have expressed certain (functions and) behaviours as generic (functions and) behaviours.
- From now on we may have to "split" a specified behaviour
  - into an indexed family of behaviours,
  - all "near identical" save for the unique index.
- And we may have to separate out, as a special behaviour, (those of) shared entities.

Example 99 – **Timetable System Users and Staff:** We continue Example 83 on Slide 737 and Example 98 on Slide 840.

- In Example 83 the sharing of the timetable between users and staff was expressed parametrically.

system(tt) $\equiv$ client(tt) $\sqcap$ staff(tt)

client: TT $\rightarrow$ **Unit**
client(tt) $\equiv$ **let** q:Query **in let** v = $\mathcal{M}_q$(q)(tt) **in** system(tt) **end end**

staff: TT $\rightarrow$ **Unit**
staff(tt) $\equiv$
 let u:Update **in let** (r,tt′) = $\mathcal{M}_u$(u)(tt) **in** system(tt′) **end end**

- We now factor the timetable entity out as a separate behaviour,
- accessible, via indexed communications, i.e., channels,
- by a family of client behaviours and the staff behaviour.

type
  CIdx /∗ Index set of, say 1000 terminals ∗/
channel
  { ct[i]:QU,tc[i]:VAL | i:CIdx }
  st:UP,ts:RES
value
  system: TT → **Unit**
  system(tt) ≡ time_table(tt) ∥ (∥ {client(i)|i:CIdx}) ∥ staff()

client: i:CIdx $\rightarrow$ **out** ct[i] **in** tc[i] **Unit**
client(i) $\equiv$ **let** qc:Query **in** ct[i]!$\mathcal{M}_q$(qc) **end** tc[i]?;client(i)

staff: **Unit** $\rightarrow$ **out** st **in** ts **Unit**
staff() $\equiv$ **let** uc:Update **in** st!$\mathcal{M}_u$(uc) **end** **let** res = ts? **in** staff() **end**

time_table: TT $\rightarrow$ **in** {ct[i]|i:CIdx},st **out** {tc[i]|i:CIdx},ts **Unit**
time_table(tt) $\equiv$
  $\square$ {**let** qf = ct[i]? **in** tc[i]!qf(tt) **end** | i:CIdx}
  $\square$ **let** uf = st? **in** **let** (tt′,r)=uf(tt) **in** ts!r; time_table(tt′) **end** **end**

- Please observe the "shift"
  - from using $\sqcap$ in *system* earlier in this example
  - to $[]$ just above.
- The former expresses nondeterministic internal choice.
- The latter expresses nondeterministic external choice.
- The change can be justified as follows:
  - The former, the nondeterministic internal choice, was "between" two expressions which express no external possibility of influencing the choice.
  - The latter, the nondeterministic external choice, is "between" two expressions where both express the possibility of an external input, i.e., a choice.
- The latter is thus acceptable as an implementation of the former.

This ends Example 99 ■

- The next example, Example 100, continues the performance requirements expressed just above.
- Those two requirements could have been put in one phrase, i.e., as one prescription unit.
- But we prefer to separate them, as they pertain to different kinds (types, categories) of resources: terminal + data communication equipment facilities versus time and space.

Example 100 – **Storage and Speed for n-Transfer Travel Inquiries:** We continue Example 86 on Slide 755.

- When performing the n-*Transfer Travel Inquiry* (rough sketch) prescribed above,
  - the first — of an expected many — result shall be communicated back to the inquirer in less than 5 seconds after the inquiry has been submitted,
  - and, at no time during the calculation of the "next" results must the storage buffer needed to calculate these exceed around 100,000 bytes.

■

8.9.2.2. Storage Requirements

8.9.2.3. Machine Cycle Requirements

8.9.2.4. Other Resource Consumption

8.9.3. Dependability Requirements

- To properly define the concept of *dependability* we need first introduce and define the concepts of
  - *failure*,
  - *error*, and
  - *fault*.

Definition 78 – Failure:

- *A machine failure occurs*
- *when the delivered service*
- *deviates from fulfilling the machine function,*
- *the latter being what the machine is aimed at*

■

Definition 79 – Error:

- *An error*
- *is that part of a machine state*
- *which is liable to lead to subsequent failure.*
- *An error affecting the service*
- *is an indication that a failure occurs or has occurred*

Definition 80 – **Fault:**

- *The adjudged (i.e., the ‘so-judged’) or hypothesised cause of an error*
- *is a fault*

- The term hazard is here taken to mean the same as the term fault.
- One should read the phrase: “adjudged or hypothesised cause” carefully:
- In order to avoid an unending trace backward as to the cause,
- we stop at *the cause which is intended to be prevented or tolerated.*

Definition 81 – **Machine Service:** *The service delivered by a machine*

- *is its behaviour*
- *as it is perceptible by its user(s),*
- *where a user is a human, another machine or a(nother) system*
- *which interacts with it*

■

Definition 82 – **Dependability:** *Dependability is defined*

- *as the property of a machine*
- *such that reliance can justifiably be placed on the service it delivers*

■

- *Impairments* to dependability are the unavoidably expectable circumstances causing or resulting from "undependability": faults, errors and failures.
- *Means* for dependability are the techniques enabling one
  - to provide the ability to deliver a service on which reliance can be placed,
  - and to reach confidence in this ability.
- *Attributes* of dependability enable
  - the properties which are expected from the system to be expressed,
  - and allow the machine quality resulting from the impairments and the means opposing them to be assessed.

8.9.3.1. Dependability Tree

- Having already discussed the "threats" aspect,
- we shall therefore discuss the "means" aspect of the *dependability tree*.
- Attributes:
  - Accessibility
  - Availability
  - Integrity
  - Reliability
  - Safety
  - Security

8.9.3.2. Dependability Tree

- Means:
  - Procurement
    - Fault prevention
    - Fault tolerance
  - Validation
    - Fault removal
    - Fault forecasting
- Threats:
  - Faults
  - Errors
  - Failures

8.9.3.3. Dependability Concepts

- Despite all the principles, techniques and tools aimed at *fault prevention*,
- *faults* are created.
- Hence the need for *fault removal.*
- *Fault removal* is itself imperfect.
- Hence the need for *fault forecasting.*
- Our increasing dependence on computing systems in the end brings in the need for *fault tolerance.*

8.9.3.4. Dependability Concepts

Definition 83 – Dependability Attribute: *By a dependability attribute we shall mean either one of the following:*

- *accessibility,*
- *availability,*
- *integrity,*
- *reliability,*
- *robustness,*
- *safety and*
- *security.*

8.9.3.5. Dependability Concepts

That is, a machine is dependable if it satisfies some degree of "mixture" of being accessible, available, having integrity, and being reliable, safe and secure ■

- The crucial term above is "satisfies".
- The issue is: To what "degree"?
- As we shall see — in a later later lecture — to cope properly
  - with dependability requirements and
  - their resolution

  requires that we deploy

  - mathematical formulation techniques,
  - including analysis and simulation,

  from statistics (stochastics, etc.).

8.9.3.6. Accesability Requirements

- Usually a desired, i.e., the required, computing system, i.e., the machine, will be used by many users — over "near-identical" time intervals.
- Their being granted access to computing time is usually specified, at an abstract level, as being determined by some internal nondeterministic choice, that is: essentially by *"tossing a coin"!*
- If such internal nondeterminism was carried over, into an implementation, some *"coin tossers"* might never get access to the machine.

8.9.3.7. Accesability Requirements

Definition 84 – **Accessibility:** *A system being accessible — in the context of a machine being dependable —*

- *means that some form of "fairness"*
- *is achieved in guaranteeing users "equal" access*
- *to machine resources, notably computing time (and what derives from that)*

■

Example 101 – Timetable Accessibility:

- The timetable (system) shall be “inquirable” by any number of users,
- and shall be update-able by a few, so authorised, airline staff.
- At any time it is expected that up towards a thousand users are directing queries at the timetable (system).
- And at regular times, say at midnights between Saturdays and Sundays, airline staff are making updates to the timetable (system).
- No matter how many users are “on line” with the timetable (system), each user shall be given the appearance that that user has exclusive access to the timetable (system).

■

8.9.3.8. Availability Requirements

- Usually a desired, i.e., the required, computing system, i.e., the machine, will be used by many users — over "near-identical" time intervals.
- Once a user has been granted access to machine resources, usually computing time, that user's computation may effectively make the machine unavailable to other users —
- by "going on and on and on"!

Definition 85 – Availability: *By availability — in the context of a machine being dependable — we mean*

- *its readiness for usage.*
- *That is, that some form of "guaranteed percentage of computing time" per time interval (or percentage of some other computing resource consumption)*
- *is achieved — hence some form of "time slicing" is to be effected*

■

Example 102 – Timetable Availability: We continue Examples 83 on Slide 737, 86 on Slide 755 and 101 on Slide 864:

- No matter which query composition any number of (up to a thousand) users are directing at the timetable (system),
- each such user shall be given a reasonable amount of compute time per maximum of three seconds,
- so as to give the psychological appearance that each user — in principle — “possesses” the timetable (system).
- If the timetable system can predict that this will not be possible, then the system shall so advise all (relevant) users.

■

8.9.3.9. Integrity Requirements

Definition 86 – Integrity: *A system has integrity — in the context of a machine being dependable — if*

- *it is and remains unimpaired,*
- *i.e., has no faults, errors and failures,*
- *and remains so, without these,*
- *even in the situations where the environment of the machine has faults, errors and failures*

■

- Integrity seems to be a highest form of dependability,
- i.e., a machine having integrity is 100% dependable!
- The machine is sound and is incorruptible.

8.9.3.10. Reliability Requirements

Definition 87 – **Reliability:** *A system being reliable — in the context of a machine being dependable — means*

- *some measure of continuous correct service,*
- *that is, measure of time to failure*

■

Example 103 – **Timetable Reliability:**

- Mean time between failures shall be at least 30 days,
- and downtime due to failure (i.e., an availability requirements) shall, for 90% of such cases, be less than 2 hours.

■

8.9.3.11. Safety Requirements

Definition 88 – Safety: *By safety — in the context of a machine being dependable — we mean*

- *some measure of continuous delivery of service of*
  - *either correct service, or incorrect service after benign failure,*
- *that is: Measure of time to catastrophic failure*

■

Example 104 – Timetable Safety:

- Mean time between failures
- whose resulting downtime is more than 4 hours
- shall be at least 120 days.

This ends Example 104 ■

8.9.3.12. Security Requirements

- Security requires a notion of *authorised user*,
- with authorised users being fine-grained authorised to access only a well-defined subset of system resources (data, functions, etc.).
- An *un-authorised user* (for a resource) is anyone who is not authorised access to that resource.

8.9.3.13. Security Requirements

Definition 89 – Security: *A system being secure — in the context of a machine being dependable —*

- *means that an un-authorised user, after believing that he or she has had access to a requested system resource:*
  - *cannot find out what the system resource is doing,*
  - *cannot find out how the system resource is working*
  - *and does not know that he/she does not know!*
- *That is, prevention of un-authorised access to computing and/or handling of information (i.e., data)*

■

- The characterisation of security is rather abstract.
- As such it is really no good as an a priori design guide.
- That is, the characterisation gives no hints as how to implement a secure system.
- But, once a system is implemented, and claimed secure, the characterisation is useful as a guide on how to test for security!

Example 105 – Timetable Security: We continue Examples 83 on Slide 737, 86 on Slide 755, 101 on Slide 864, and 102 on Slide 867.

- Timetable users can be any airline client logging in as a user, and such (logged-in) users may inquire the timetable.
- The timetable machine shall be secure against timetable updates from any user.
- Airline staff shall be authorised to both update and inquire, in a same session.

■

Example 106 – Hospital Information System Security:

- General access to (including copying rights of) patient's medical journals is granted only to designated hospital staff.
- In certain forms of emergency situations any hospital staff may , get access to a hospital patient's medical journal.
- Such incidents shall be duly and properly recorded and reported.

This ends Example 106 ■

8.9.3.14. Robustness Requirements

Definition 90 – Robustness: *A system is robust — in the context of dependability —*

- *if it retains its attributes*
  - *after failure, and*
  - *after maintenance*

■

- Thus a robust system is "stable"
  - across failures
  - and "across" possibly intervening "repairs"
  - and "across" other forms of maintenance.

8.9.4. Maintenance Requirements

Definition 91 – Maintenance Requirements: *By maintenance requirements we understand a combination of requirements with respect to:*

- *adaptive maintenance,*
- *corrective maintenance,*
- *perfective maintenance,*
- *preventive maintenance and*
- *extensional maintenance*

■

- Maintenance of building, mechanical, electro-technical and electronic artifacts — i.e., of artifacts based on the natural sciences — is based both on documents and on the presence of the physical artifacts.
- Maintenance of software is based just on software, that is, on all the documents (including tests) entailed by software.

8.9.4.1. Adaptive Maintenance Requirements

Definition 92 – Adaptive Maintenance: *By adaptive maintenance we understand such maintenance*

- *that changes a part of that software so as to also, or instead, fit to*
  - *some other software, or*
  - *some other hardware equipment*

  (i.e., other software or hardware which provides new, respectively replacement, functions)

■

Example 107 – Timetable System Adaptability:

- The timetable system is expected to be implemented in terms of a number of components that implement respective domain and interface requirements, as well as some (other) machine requirements.
- The overall timetable system shall have these components connected, i.e., interfaced with one another — where they need to be interfaced — in such a way that any component can later be replaced by another component ostensibly delivering the same service, i.e., functionalities and behaviour.

This ends Example 107 ■

8.9.4.2. Corrective Maintenance Requirements

Definition 93 – Corrective Maintenance: *By corrective maintenance we understand such maintenance which*

- *corrects a software error*

■

Example 108 – Timetable System Correct-ability:

- Corrective maintenance shall be done remotely:
  - from a developer site,
  - via secure Internet connections.

■

8.9.4.3. Perfective Maintenance Requirements

Definition 94 – Perfective Maintenance: *By perfective maintenance we understand such maintenance which*

- *helps improve (i.e., lower) the need for*
- *hardware (storage, time, equipment),*
- *as well as software*

■

Example 109 – Timetable System Perfectability:

- The system shall be designed in such a way as to
  - clearly be able to monitor the use of
  - "scratch" (i.e., buffer) storage and compute time

  for any instance of any query command.

■

8.9.4.4. Preventive Maintenance Requirements

Definition 95 – Preventive Maintenance: *By preventive maintenance we understand such maintenance which*

- *helps detect, i.e., forestall, future occurrence*
- *of software or hardware errors*

■

8.9.4.5. Extensional Maintenance Requirements

Definition 96 – Extensional Maintenance: *By extensional maintenance we understand such maintenance which adds new functionalities to the software, i.e., which implements additional requirements* ∎

Example 110 – Timetable System Extendability:

- Assume a release of a timetable software system to implement a requirements that, for example, expresses
  - that shortest routes
  - but not that fastest routes be found
  - in response to a travel query.
- If a subsequent release of that software
  - is now expected to also calculate fastest routes
  - in response to a travel query,
- then we say that the implementation of that last requirements
- constitutes extensional maintenance.

■

- Whenever a maintenance job has been concluded, the software system is to undergo an extensive acceptance test:
- a predetermined, large set of (typically thousands of) test programs has to be successfully executed.

8.9.5. Platform Requirements

Definition 97 – Platform: *By a [computing] platform is here understood a combination of hardware and systems software — so equipped as to be able to execute the software being requirements prescribed — and 'more'* ∎

Definition 98 – Platform Requirements: *By platform requirements we mean a combination of the following:*

- *development platform requirements,*
- *execution platform requirements,*
- *maintenance platform requirements and*
- *demonstration platform requirements*

∎

Example 111 – **Space Satellite Software Platforms:** Elsewhere prescribed software for some space satellite function is to satisfy the following platform requirements:

- shall be developed on a Sun workstation under `Sun UNIX`,
- shall execute on the military `MI1750` hardware computer running its proprietary `MI1750 Operating System`,
- shall be maintained at the NASA Houston, TX installation of `MI1750 Emulating Sun Sparc Stations`, and
- shall be demonstrated on ordinary Sun workstations under `Sun UNIX`.

■

8.9.5.1. Development Platform Requirements

Definition 99 – Development Platform Requirements: *By development platform requirements we shall understand such machine requirements which*

- *detail the specific software and hardware*
- *for the platform on which the software*
- *is to be developed*

■

8.9.5.2. Execution Platform Requirements

Definition 100 – **Execution Platform Requirements:** *By execution platform requirements we shall understand such machine requirements which*

- *detail the specific (other) software and hardware*
- *for the platform on which the software*
- *is to be executed*

■

8.9.5.3. Maintenance Platform Requirements

Definition 101 – Maintenance Platform Requirements: *By maintenance platform requirements we shall understand such machine requirements which*

- *detail the specific (other) software and hardware*
- *for the platform on which the software*
- *is to be maintained*

■

8.9.5.4. Demonstration Platform Requirements

Definition 102 – Demonstration Platform Requirements: *By demonstration platform requirements we shall understand such machine requirements which*

- *detail the specific (other) software and hardware*
- *for the platform on which the software*
- *is to be demonstrated to the customer — say for acceptance tests, or for management demos, or for user training*

■

8.9.5.5. Discussion

- Example 111 is rather superficial.
- And we do not give examples for each of the specific four platforms.
- More realistic examples would go into rather extensive details,
- listing hardware and software product names, versions, releases, etc.

8.9.6. Documentation Requirements

Definition 103 – Documentation Requirements: *By documentation requirements we mean requirements of any of the software documents that together make up software:*

- *not only code that may be the basis for executions by a computer,*
- *but also its full development documentation:*
  - *the stages and steps of application domain description,*
  - *the stages and steps of requirements prescription, and*
  - *the stages and steps of software design prior to code,*

with all of the above including all validation and verification (incl., test) documents.

- *In addition, as part of our wider concept of software, we also include*
- *a comprehensive collection of supporting documents:*
  - *training manuals,*
  - *installation manuals,*
  - *user manuals,*
  - *maintenance manuals, and*
  - *development and maintenance logbooks*

■

- We do not attempt, in our characterisation, to detail what such documentation requirements could be.
- Such requirements could cover a spectrum
  - from the simple presence, as a delivery, of specific ones,
  - to detailed directions as to their contents, informal or formal.

End of Lecture 15

Reqs. Eng.: Machine Reqs. & Closing Stages

Lecture 16

Reqs. Eng.: Opening and Closing Stages

8.10. Opening and Closing Stages

8.10.1. Opening Stages

8.10.1.1. Stakeholder Identification and Liaison

8.10.1.2. Requirements Acquisition

8.10.1.3. Requirements Analysis

8.10.1.4. Terminoligisation

-
-
-
-

8.10.2. Closing Stages

- For completeness, we shall, as in Sects. on Slide 654 and on Slide 899, briefly list the closing stages of requirements engineering.
- They are:
  1. **requirements verification, model checking and testing** – the assurance of properties of the formalisation of the requirements model;
  2. **requirements validation** – the validation of the veracity of the informal, i.e., the narrative requirements prescription;
  3. **requirements feasibility and satisfiability**; and
  4. **requirements theory formation**.
-
-
-

8.10.2.1. Verification, Model Checking and Testing

8.10.2.2. Requirements Validation

8.10.2.3. Requirements Satisfiability & Feasibility

8.10.2.4. Requirements Theory

-
-
-
-

8.10.3. Requirements Engineering Documentation

8.10.4. Conclusion

End of Lecture 16

Reqs. Eng.: Opening and Closing Stages

Lecture 17

Domain Demos

9. Domain Demos

(9. **Domain Demos**)

9.1. What Is a Demo ?

9.2. Demo Entities, Operations, Events and Behaviours

9.3. Demos and Simulation

(9. Domain Demos 9.3. Demos and Simulation)

9.3.1. Timing

9.4. Discussion

End of Lecture 17

Domain Demos

Lecture 18

Domain-based Innovation

10. Domain Engineering as a Basis for Innovation

(10. **Domain Engineering as a Basis for Innovation**)

10.1. What Is Innovation ?

- Innovation takes place when scientific results "meets" the market-place !

10.2. Where Is Innovation Hidden in the Domain Models ?

10.3. How to Extract Innovations from Domain Models ?

10.4. Discussion

End of Lecture 18

Domain-based Innovation

Lecture 19

Conclusion

11. Conclusion

-
-
-

11.1. What Have We Achieved ?

11.2. What Have We Omitted ?

-
-
-

11.3. What Have We Not Been Able to Cover ?

11.4. What Is Next ?

11.5. How Do You Now Proceed ?

12. Acknowledgements

- This book has been written after I retired from almost 32 years as professor at The Technical University of Denmark, as of April 1, 2007.
- I have therefore been basically deprived of the daily, invigorating interaction with dear colleagues at DTU Informatics.
- Instead I was invited, or, more-or-less, invited myself, to lecture at universities in Nancy[35], Graz[36] and Saarbrücken[37].
- The present book underwent a number of iterations while presenting its material at intensive 30 lectures plus 15 afternoons of project tutoring.

[35] Oct.–Dec., 2007, University of Henri Poincare (UHP) and INRIA, France
[36] Oct.–Dec., 2008, Technical University of Graz (TUG), Austria
[37] March 2009, University of Saarland, Germany

(12. **Acknowledgements**)

- I am therefore profoundly grateful (alphabetically listed) to
  - Bernhard Aichernig (TUG),
  - Hermann Maurer (TUG),
  - Dominique Méry (UHP/INRIA),
  - Wolfgang Paul (UdS),
  - Thomas in der Rieden (UdS) and
  - Franz Wotawa (TUG)
- for hosting me and for giving me the opportunity to refine the course material and be challenged by bright young people from a dozen European and Asian countries.
- Sir Tony Hoare provided ...

End of Lecture 19

Conclusion and Acknowledgements