DRAFT

Dines Bjørner

A Financial Services Industry

Domain Descriptions

Banking, Stock Brokerage, Credit Cards, Insurance, Portfolio Management, *&c*

January 21, 2008, 10:00

Draft Notes — Rough Sketches

Fredssvej 11, DK-2840 Holte, Denmark
bjorner@gmail.com, http://www.imm.dtu/~db

Version History

1. 19-Jan-2008: A first version was e-mailed to Mr. Dennis Yap, NOA Advance Technology Solution Co. Ltd., Suite 1012, Kolon Science Valley II, Guro-dong, Guro-Gu, Seoul 150-050, Korea; Tel.: +82-2-850-3760, Fax: +82-2-850-3764
2. 20-Jan-2008: A second version with minor changes, was likewise sent to Mr. Dennis Yap (in three versions). Changes were:
   (a) Draft status was added to every page.
   (b) This front matter page, VI, was populated.
   (c) Additional Preface text, lower half of Page VII was inserted.
   (d) Initial project start phase material moved from Sect. 1.10 to a new section, Sect. 1.14.1.
   (e) Sect. 1.10 text is new.
   (f) Additional material on entities, operations, events, and behaviours, and on UML was added to Volume II's front pages (Pages 11–12).
   (g) Additions made to texts in Sects. 4.1.2 on page 23 to 4.1.3 on page 27.
   (h) Provisions made to print two versions of this document:
   - One which does not show any formula (Mr. Yap was sent such a Chap. 4 and was also sent a complete file),
   - and one which shows extensively annotated formulas — explaining the RAISE specification language (RSL) constructs being used in the particular formula lines (Mr. Yap was sent such a Chap. 4 and was also sent a complete file).
3. 21-Jan-2008
   - New preface framed text inserted, on top of Page VII, before first itemised three bullets.
   - Extensions made to texts in Sect. 4.1.2 on page 23: the command syntaxes and semantics.
   - This version is with formulas. These and their annotations can all be simply removed.

Part I

Preface

- This document is "vastly" incomplete.
- It may not illustrate the kind of finance handling issues that the reader may be looking for.
- But trust me, Dines Bjørner,
  - ⋆ I do have enough enough experience in also this field,
  - ⋆ to know that whatever can be meaningfully described,
  - ⋆ where a meaningful description is something which designates,
  - ⋆ which points to some phenomenon or otherwise well-known banking, insurance, brogerage, etc., concept,
  - ⋆ that that can also be precisely narrated (and, if need be, formalised).
- But only about a few days of work by a single person has so far been extended on this report.

- The current status of this document is rough sketch incomplete draft.
- It is being developed during the period 18 January 2008 to (probably) early February 2008.
- It is intended as a document that might help a Korean software house in its project/contract discussions with a Korean client.

The main objective of releasing this document, at this early and vastly incomplete stage, is to help NOA, a Korean Software House, showing a Korean client what a domain description entails.
The document does not show how domain descriptions can be used further. We refer to [1, to appear] for a 30 page introduction, and to the book [2] for a comprehensive treatment. We refer to Appendix D (starting Page 201).

In [3, to appear] I give a concise overview of domain engineering; in [1, to appear] one of domain and requirements engineering as they relate; and in [4, to appear] I relate domain engineering, requirements engineering and software design to software management. In [5] I present a number of domain engineering research challenges. In [6, to appear] — which also covers research challenges of domain engineering — I additionally present a rather large example of the container line industry domain.

Contents

Volume I

Informal Documents

This volume only contains one chapter: Information.

1

Information

This chapter is rather "thin". More detail should be given to each item.

1.1 Project Title

The project title is:

FiSeIn: A Domain Description of Segments of the Financial Service Industry

1.2 Name, Place and Date

This document is edited (written) by

- **Name:** Dines Bjørner,
- **Place:**
  - Fredsvej 11,
  - DK-2840 Holte,
  - Denmark;
  - phone: +45-45 42 21 41,
  - e-mail:bjorner@gmail.com,
  - URL: http://www.imm.edu.dk/~db

- **Date:**
  - Writing started 18 January 2008.
  - Present version was compiled January 21, 2008: 10:00

1.3 Partners

Partners may be:

1. DB: Dines Bjørner (as prospective consultant);
2. NOA Advance Technology Solution Co. Ltd. "NOA"
   - Address:
     - ⋆ Suite 1012,
     - ⋆ Kolon Science Valley II,
     - ⋆ Guro-dong, Guro-Gu,
     - ⋆ Seoul 150-050,
     - ⋆ Korea;
     - ⋆ Tel.: +82-2-850-3760
     - ⋆ Fax: +82-2-850-3764
   - Persons:
     (a) Mr. Moon, CEO,
     (b) Mr. Dennis Yap;

   and
3. Company XXX.

Do we need more on NOA ?

Do we need more on Company XXX ?

1.4 Current Situation

This section is tentative.

At the moment, January 2008, NOA is negotiating with Company XXX a contract that involves fitting an existing contract around

> a documentation framework and a document which is well organized and applicable to multiple product groups and trading activities in the capital market business. The document is meant to provide detail guideline for its staff (front, middle and back office) and to set up standard procedure for processing each financial instrument. The scope include product definition, market data structure, pricing, dealing, sales, trade work flow, risk analysis, back office processing, compliance and reporting in a bank. The document will be used for generating a requirements document where our software system will be used to automatic the process.

Further to the above:

> NOA expects to be introducing Dines Bjørner's expertise into the NOA and Company XXX consortium. This consortium needs a full picture of the values of creating these documents.

Moreover:

> We are selling the license of a implemented trading and management system to a financial institute. There are some gap between their requirement and our solution where we will be extending our system to meet their requirement.

Summarising:

- The plan is to create a full domain document where our client's employee can use to understand their own business process.
- The requirements document that follow will mainly focus on the gap between our package and their requirement.

1.5 Needs and Ideas

These subsections are tentative.

1.5.1 The Need

There is a need

> to understand how a domain description, requirement document and prescription document can be used, to advantage, in the current NOA/XXX project.

1.5.2 The Idea

The idea

> is to use the TripTych concept of
> - informative,
> - descriptional and
> - analytic

> documentation as from (and including)
> - domain engineering,
> - via requirements engineering to
> - software implementation.

1.6 Concepts and Facilities

These subsections are tentative.

1.6.1 Concepts

The concepts of Company XXX are those of the

- multiple product groups and trading activities in the capital market business.

Further

- The document is meant to provide detail guideline for its staff (front, middle and back office) and to set up standard procedure for processing each financial instrument.

1.6.2 Facilities

- TO BE WRITTEN

1.7 Scope and Span

These subsections are tentative.

1.7.1 Scope

The scope of Company XXX's business is that of the financial service industry:

- banking,
- stock brokerage,
- credit cards,
- insurance, and
- portfolio management.

Banks may offer any subset of the implied services of these five areas.

1.7.2 Span

The span include

- product definition,
- market data structure,
- pricing,
- dealing,
- sales,
- trade work flow,
- risk analysis,
- back office processing,
- compliance and

- reporting

in a bank.

DB needs understand the above 10 bullet (•) itemisation

The present document attempts to sketch domain descriptions of the scope.

1.8 Assumptions and Dependencies

These subsections are tentative.

1.8.1 Assumptions

The assumptions — on which this project (between the partners: Dines Bjørner (DB), the consultant, NOA and Company XXX) is based — are

- that the business domain of Company XXX can be openly determined;
- that this business domain, at the moment, is stable[1];
- that there is the will amongst the partners to provide information to all partners about the business domain of Company XXX; and
- that all partners will feed back, i.e., report, to all other partners in timely fashion.

1.8.2 Dependencies

The corresponding dependencies, within the span of the project, are:

- that
-
-

1.9 Implicit/Derivative Goals

These subsections are tentative.

1.9.1 Implicit Goals

- Creating the domain description leads to improved understanding of the financial service industry.
- Creating and circulating the resulting domain description leads to improved awareness of of the business processes of the financial service industry.

[1] The business domain of Company XXX must be stable, but it can be expected that its requirements remain in a flux, that is, "change" during the domain description part of this project.

1.9.2 Derivative Goals

- The domain description of the financial service industry can be used (and this is not IT work)
  - ⋆ to perform business process re-engineering, and
  - ⋆ to train future and re-train existing staff in the financial service industry .

1.10 Synopsis

This section is tentative.

1.11 Standards Compliance

This section is tentative.

Insofar as it is reasonable[2] the project work shall otherwise adhere to the following standards:

1. IEEE-1058-1998, Standard for Software Project Management Plans
2. IEEE-730-2002, Standard for Software Quality Assurance Plans
3. IEEE-830-1998, Recommended Practice for Software Requirements Specifications
4. IEEE-1012-1998, Standard for Software Verification and Validation
5. IEEE-1016-1998, Recommended Practice for Software Design Descriptions
6. IEEE-1028-1997, Standard for Software Reviews
7. ISO/IEC 12207 Information Technology-Software Life Cycle Processes, with amendments
8. ISO/IEC 15288 Systems Engineering — Systems Life Cycle Processes
9. IEEE/EIA 12207.0 1996 Industry Implementation of International Standard ISO/IEC: 12207:1995 Software life cycle processes.
10. ISO 9001-2000, Quality Management Systems - Requirements.
11. ISO/IEC 15288, Systems Engineering - Systems Life Cycle Processes.
12. Standard ISO/IEC 17799:2005 Information technology — Security techniques — Code of practice for information security management.
13. ISO/IEC 90003:2004, Software Engineering. Guidelines for the application of ISO 9001:2000 to computer software.

1.12 Contracts

This section will remain “empty”.

[2]

1.13 The Teams

This section will remain "empty".

1.13.1 Management

1.13.2 Developers

1.13.3 Client Staff

1.13.4 Consultants

This section will remain "empty".

1.14 Plans

1.14.1 An Upstart Phase

The FiSeIn project has the following components:

1. Establishing the NOA/Company XXX FiSeIn contract.
2. Establishing a consultancy contract with DB.
3. Establishing the NOA project FiSeIn team.
4. Establishing the Company XXX project FiSeIn team.
5. Possible initial visit of Dines Bjørner (DB) to NOA.
   (a) Lectures
      i. Domain Engineering,
      ii. Documentation,
      iii. TripTych Software Engineering (SE) in general,
      iv. TripTych SE Management.
   (b) Initial Project Work:
      i. DB to assist NOA (and Company XXX ?) in establishing initial document library,
      ii. DB to assist NOA (and Company XXX ?) in rough sketching the kind of documents otherwise shown in this report.
   (c) Main body of work, phase I
      - either NOA & and Company XXX to work 203 weeks "alone",
      - or DB to work with NOA & and Company XXX on initial volumes and parts of the FiSeIn documentation
   (d) DB to return to Denmark for 2-3 weeks
   (e) Repeated phases, like Phase I, see Item 5(c).
6. Etcetera.

It may be that DB may be needed for the bulk of the 2nd and 3rd Quarter of 2008.

1.14.2 Project Graph

1.14.3 Budget

1.14.4 Funding

1.14.5 Accounts

1.15 Management

This section will remain "empty".

1.15.1 Assessment

1.15.2 Improvement

Plans

Actions

Volume II

Domain Descriptions

This volume contains one chapter and five parts.

The chapter, Business Processes, is common to four[3] of the five parts. The parts cover main aspects of the domain descriptions of

1. banking,
2. stock brokerage,
3. credit cards,
4. insurance and
5. portfolio management.

These aspects, in the present edition of this book, includes

- stakeholders, acquisition process and terminology
- and the facets of
  - ⋆ intrinsics,
  - ⋆ support facilities,
  - ⋆ management and organisation,
  - ⋆ rules and regulations.
  - ⋆ scripts and
  - ⋆ human behaviour.

Each aspect is (to be) treated according to the following description ontology:

- entities (types and values),
- operations (function and actions),
- events, and
- behaviours.

[3] Chapter 2, Business Processes, currently misses the 'Insurance' sub-domain.

Relation to UML

By describing and operation signatures we cover, in a more rigid and engineeringly sound manner what is sometimes covered by software engineers using UML's Class Diagram construct.

By describing operation definitions we go well beyond what UML covers.

By covering events and behaviours we cover, again in a more rigid and engineeringly sound manner what is sometimes covered by software engineers using UML's use case, state machine, etc. concepts.

2

Business Processes

This chapter need be updated: Insurance is currently not included.

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

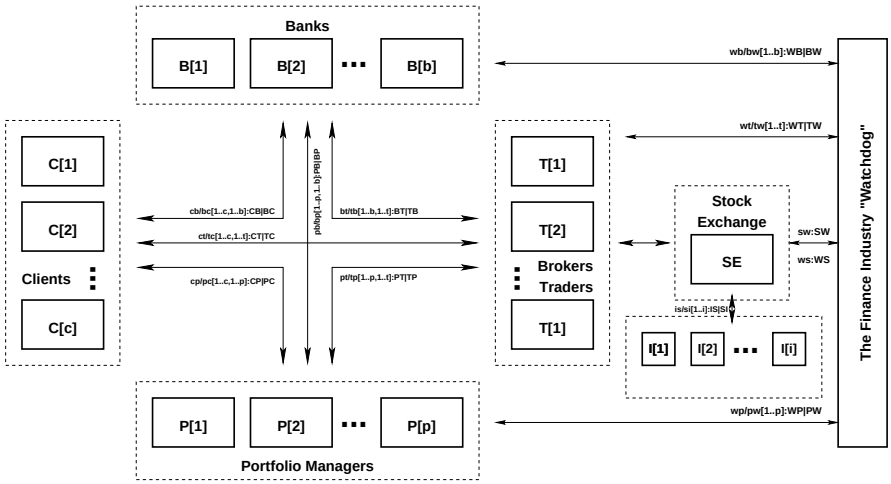

Fig. 2.1. A financial behavioural system abstraction

The above figure need be redrawn to include 'Insurance'

(i) Clients engage in a number of business processes: (i.1) they open, deposit into, withdraw from, obtain statements about, transfer sums between and close demand/deposit, mortgage and other accounts; (i.2) they request brokers to buy or sell, or to withdraw buy/sell orders for securities instruments (bonds, stocks, futures, etc.); and (i.3) they arrange with portfolio managers to look after their bank and securities instrument assets, and occasionally they reinstruct portfolio managers in those respects.

(ii) 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.

(iii) 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.

(iv) Portfolio managers engage with clients, banks, and brokers and traders in exchanges related to client portfolios.

(v) 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.

(vi) 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.

(vii) The financial industry watchdog engages with banks, portfolio managers, brokers and traders and with the stock exchanges.

The above list need be extended: 1/2 to 1/1 page more, at most

Part I

Banking

3

Banking: Preamble

3.1 Stakeholders

3.2 The Acquisition Process

3.2.1 Studies

3.2.2 Interviews

3.2.3 Questionnaires

3.2.4 Indexed Description Units

3.3 Terminology

4

Banking: Intrinsics

We describe the intrinsics of two main-street set of banking concepts: Demand/Deposit banking and Mortgage (or: Savings & Loan) banking.

4.1 Demand/Deposit Banking

In 'Demand/Deposit Banking' banks offer clients that they can open demand/deposit accounts, deposit monies into such accounts, withdraw monies from such accounts, obtain chronologically listed statements of deposit, withdraw, and statement transactions, and close accounts. Banks can offer credit limits on a per customer and account basis, and banks usually offer yields on [positive] deposits and (always) ascribe interests on [negative] deposits within the credit limits. Banks therefore perform such operations on accounts as calculation of yields and interests and the update of accounts with these amounts. Banks may notify clients of account liabilities.

> We now describe banking systematically based on the above synopsis. In this more systematic description we introduce further banking concepts than those mentioned above. The description is first in narrative form, that is, informal, in English, using only terms from the banking profession or such terms which are generally understood.
> The casual reader may wish to skip the formulas in a first, or in any reading !

4.1.1 The Banking State

Narrative: Accounts

1. The bank maintains a set of all client demand/deposit accounts.
2. Each account has a distinct account name.
3. Each account has an account balance

4. To each account is associated two rates: yield and interest.
5. One can, to each account, associate the transactions performed on this account as a chronologically ordered list of chronologically ordered list of statements[1].
6. From an account one can thus observe its name, balance, yield and interest rates, and its list of list of statements.
7. A statement is either an open account, a deposit, a withdrawal, a close account, a calculate interest or yield, or an inform customer statement.
   (a) An open account statement records the client name, date and time of invocation of command, amount of initial deposit, the yield and interest rate agreed with the bank, and the new account name (e.g. a numeral) resulting from a successful transaction.
   (b) A deposit statement records the client name, account number, date and time of invocation of command, and amount of deposit.
   (c) A withdrawal statement records the client name, account number, date and time of invocation of command, and amount of withdrawal.
   (d) A close account statement records the client name, account number, date and time of invocation of command, and rest amount of account balance.
   (e) A calculate interest or yield statement records the client name, account number, date and time of invocation of command, and the amount of yield or interest of which one of the two is definitely zero.
   (f) An inform statement records the client name, account number, date and time of invocation of command, and the list of statements issued to the client.

Formalisation: Accounts

type
1. DDA = Anm $\xrightarrow[m]{}$ Acct
2. Anm
3. AcctBal
4. Yield, Interest
5. Stmt

value
6. obs_Anm: Acct → Anm
6. obs_AcctBal: Acct → AcctBal
6. obs_Yield: Acct → Yield
6. obs_Interest: Acct → Interest

[1] We shall operate with to strongly related concepts: statements and commands. A transaction records the invocation (i.e., execution) of a command, see Sects. 4.1.2–4.1.3 and Sects. 4.2.2–4.2.3. The statement records the time and date of invocation, the type and input arguments of the command as well as the results of the invocation.

6. obs_StatList: Acct → (Stmt*)*

type

7. Stmt = OpenS | DepoS | WithS | ClosS | CalcS | InfoS
7(a). OpenS == mkOpS(cn:CliNm,sh:Shrd,i:Info,r1:Res1,re2:Res2)
7(b). DepoS == mkDeS(an:Anm,sh:Shrd,de:Mon,r:Res1)
7(c). WithS == mkWiS(an:Anm,sh:Shrd,de:Mon,r:Res1)
7(d). ClosS == mkClS(an:Anm,sh:Shrd,de:Mon,r:Res1)
7(e). CalcS == mkCaS(an:Anm,sh:Shrd,ab:InYi,r:Res1)
7(f). InfoS == mkInS(an:Anm,sh:Shrd,sl:Stmt*,r:Res1)
α Res1 == ok | nok
β Res2 == nil | mkRes2(y:Yld,i:Intrst,an:Anm) [nil if Res1 is nok]
γ Date, Time, Info [information] Yld [yield], Intrst [interest]
δ Shrd = Date × Time [shared info]
ϵ InYi == mkIntrst(m:Mon) | mkYld(m:Mon)

α Every command execution results in either a successful termination, i.e., ok, or the command is rejected, i.e. nok.

β An ok open demand/deposit account command execution further results in the client being informed of the yield and interest rates, and the new account number.

γ When opening an account some information was given (on the basis of which the open command was conditionally executed).

δ Every transaction records the execution, on a certain date and at a certain time, of a corresponding command.

RSL Annotations: We use identifiers A, B, C, ... to stand for the type names occuring in the referenced formulas.

1. "**type** A = B $\overrightarrow{m}$ C" defines A to be a concrete type of maps, i.e., finite definition set, i.e., discrete functions from values of type B into values of type C.

2.– 5. "**type** A, B, ..." defines A, B, ... to be abstract types, i.e., an algebraic sorts, of, in line 2.– 5. further undescribed values.

6. "**value** obs_A: B → A" defines obs_A to be a function which when applied to a value of type B yields a value of type A.

7. "**type** A = B | ... | C" defines A to be a type of either type B or . . . typeC values.

7(a).–7(f). "**type** A == mkA(sx:X,...,sy:Y)" defines A to be a Cartesian type of values, each one can be thought of as a 'tree' with root label mkA and with sub-tree values (possibly terminal leaves) of type X, . . . , Y; the x, . . . , y designate selector functions which when applied to a value of type A yield respective sub-tree.

$\beta, \gamma, \delta, \epsilon$ The bracketed [text] is commentary and has no semantics meaning.

δ "**type** A = B × C × ... × D" defines type A values to be Cartesian (groupings) values of B, C, ..., D values, that is, (b,c,...,d) values.

ϵ "**type** A == mkB(...) | mkC(...) | ... | mkD(...)" defines A to be a types of disjoint types of values, either a mkB(...) value, or a mkC(...) value ... or a mkD(...) value. These value types are disjoint sôlely because their "tree root" labels, mkB, mkC, ..., mkD, are distinct.$\surd$ [2]

Narrative: Bank Registers

In order to be able to administrate the interface between demand/deposit accounts and clients, and to be able for two or more clients to share an account the bank maintains the following registers:

8. There is a client-to-accounts-register: The client-to-accounts-register records for each client which zero, one or more demand/deposit accounts, by name, that client has rights to.
9. And there is an accounts-to-client-register:
   (a) The accounts-to-client-register records for each account which
   The accounts-to-client-register contents can be computed from the client-to-accounts-register.
10. Thus the two registers must be commensurate: If a client can access an account then that must be recorded accordingly in both registers. Thus there can be no discrepancies.

Formalisation

type
  8. CtAR
  8. CtAR = Cnm $\overrightarrow{m}$ Anm-**set**
  9. AtCR
  9(a). AtCR = Anm $\overrightarrow{m}$ Cnm-**set**

value
  10. wf_CtAR_AtCR: CtAR × AtCR → **Bool**

Narrative: A Preliminary Bank State

11. A bank state includes the demand/deposit accounts, the client-to-accounts register and the accounts-to-clients register.
12. The bank state must be well-formed:
   (a) If the demand/deposit accounts associate an account number (say a) with some balance, then that account number must be linked to one or more client names in the accounts-to-clients register,
   (b) and for each such name for that account number in the accounts-to-clients register there must be an association from that client name to that account number in the client-to-accounts register.

[2] $\surd$ designates end of RSL annotations.

Formalisation: A Preliminary Bank State

type
11. Prel_Bank_State′ = DDA × CtAR × AtCR × ...
12. Prel_Bank_State = {| bank:Prel_Bank_State′ • wf_Prel_Bank_State(bank) |}

value
12. wf_Prel_Bank_State: Prel_Bank_State′ → **Bool**
12. wf_Prel_Bank_State(dda,ctar,atcr,...) ≡
12(a). **dom** dda = **dom** atcr ∧
∀ a:Anm•a ∈ **dom** dda ⇒ ∀ c:Cnm•c ∈ atcr(a) ⇒ c ∈ dda(a) ∧
12(b). **dom** ctar = ∪ **rng** atcr

RSL Annotations:

11. The primed (′) A′ in **type** D′ = ... is intended to "signal" that the type D′ is "overspecified".
12. The sub-type definition **type** D = {| d:D′ • wfD(d) |} expresses that D consists just of those values a of type D′ which are well-formed according to wfD(d).

12(a). **dom** map expresses the definition set, the 'domain', of map map
The universally quantified predicate expression ∀ x:X • P(a) expresses that "for all x of type X it is the case (•) that the predicate P holds of x.
x ∈ xset ⇒ Q(x) expresses that x is in the set xset implies that the predicate Q holds of x.

12(b). The expression ∪ **rng** map expresses the distributed union of the range elements of map.

4.1.2 Client Transactions

There are, to illustration, six client comands: open, deposit into, withdraw from, statement of, close and share account. We shall treat these in separate sub-sections each consisting of four parts: two narratives and two, respective formalisations of the syntax, respectively the semantics of these commands.

Narrative: Syntax of the Open Account Command

Clients interact with the bank by means of commands.

13. A client command is either an open account, a deposit, a withdraw, a request statements or a close account or a share account command.
    (a) An open command includes a client name and the initial deposit of (zero, one or more units of) money.

Formalisation: Syntax of the Open Account Command

type
13. Cmd = OpeC | DepC | WitC | ReqC | CloC | ShaC
13(a). OpeC = mkOpC(cn:Cnm,i:Info)

Narrative: Semantics of the Open Account Command

14. An open a demand/deposit account command means that the client, cn:Cnm, is assigned a new, i.e., a fresh, hitherto unused demand/deposit account number after the bank has negotiated with the client a credit limit, a yield rate on a positive account balance, an interest rate on a negative account balance, and aspects not further described here — all based on prevailing bank policy and client information.

Formalisation: Semantics of the Open Account Command

value
14. int_OpeC: OpeC → Prel_Bank_Bank_State
→ Prel_Bank_Bank_State × (nok|(ok×Anm))
int_OpeC(mkOpC(cn,info))(β) ≡
0. **let** (dda,ctas,atcs) = obs_Prel_Bank_State(β) **in**
1. **if** is_client_OK(cn,info)(dda,ctas,atcs)
2. **then**
3. **let** a : Anm • a ∉ **dom** dda,
4. (y,i) = negotiate_Yld_Int(cn,info)(dda,ctas,atcs),
5. **let** s = mkInfoS(a,(date(β),time(β)),⟨⟩,ok) **in**
6. dda′ = dda ∪ [a ↦ makeAcct(y,i,0,⟨s⟩)],
7. atcs′ = atcs ∪ [a ↦ {cn}],
8. as = **if** cn ∈ **dom** ctas **then** ctas(cn) **else** {} **end in**
9. **let** ctas′ = ctas † [cn ↦ as ∪ a] **in**
10. (merge_Prel_Bank_states((dda′,ctas′,atcs′),β),(ok,a)) **end end end**
11. **else** (merge_Prel_Bank_States((dda,ctas,atcs),β),nok)
end end

obs_Prel_Bank_State: Prel_Bank_State → DDA×CtAR×AtCR
is_client_OK: (Cnm×Info) → (DDA×CtAR×AtCR) → **Bool**
negotiate_Yld_Int: (Cnm×Info) → (DDA×CtAR×AtCR) → (Yld×Intrst)
merge_Prel_Bank_states: (DDA×CtAR×AtCR)×Prel_Bank_State → Prel_Bank_State

RSL Annotations: First we explain some of the clauses and operator/-operand expressions:

0.,4. The clause **let** (a,b,...,c) = function(args), ... binds a, b, ... and c to the respective components of the Cartesian result of the function application function(args).

3. The clause **let** x:X • x ∉ xset binds the name x to a value of type X such that (•) x is not in the set xset.

5.–9. The clause **let** a = expression, ... binds the name a to the result of the evaluation of expression.

6.,7. The expression map ∪ [a ↦ b] evaluates to a map which is like map except that it now has a further association (↦) mapping a into b.

6.,7. The expression [$a_1 \mapsto b_1$, $a_2 \mapsto b_2$, ..., $a_n \mapsto b_n$,] designates a map which associates a_i with b_i (for i in 1, 2, ..., n).

6. The expression {a_1, a_2, ..., a_n } designates a set of the listed element values; {a} designates a singleton set (of just one element); {} designates the empty set.

7. The expression ⟨a_1, a_2, ..., a_n ⟩ designates a list of the listed element values; ⟨a⟩ designates a singleton list (of just one element); ⟨⟩ designates the empty list.

Then we "gently" read the function that gices semantics to an Open Account command:

1. If the client name and information is OK with the bank
2. then
3. an account number that has not (never ?) been used before is allocated, and
4. a yield and an interest rate is negotiated for the client, and
5. as a temporary "measure", the open account statement is compiled, and
6. the demand/deposit accounts are updated to record for the new account its yield and interest, that the initial balance is 0, and the statement list is the singleton of the open account statement, and
7. the accounts to clients register is updated to reflect that the new account, at this moment, has just one holder, and
8. as a temporary "measure" the set of account numbers for that client is calculated, and
9. the client to accounts register is suitably updated to reflect that this client now has the new account, and
10. a new banking state results from this transaction as does the fact that the transaction went well,
11. else the banking state is unchanged and a not ok client result is returned.

Narrative: Syntax of the Deposit Command

13. A client command is ... a deposit ... command.

13(b). A deposit command includes an account number and the amount of monies to be deposited.

Formalisation: Syntax of the Deposit Command

type
13. Cmd = OpeC | DepC | WitC | ReqC | CloC | ShaC
13(b). DepC = mkDeC(an:Anm,mo:Money)

Narrative: Semantics of the Deposit Command

15.

Formalisation: Semantics of the Deposit Command

Narrative: Syntax of the Withdraw Command

13. A client command is ... a withdraw ... command.
13(c). A withdraw command includes an account number and the amount of monies to be withdrawn.

Formalisation: Syntax of the Withdraw Command

type
13. Cmd = OpeC | DepC | WitC | ReqC | CloC | ShaC
13(c). WitC = mkWiC(an:Anm,am:Amount)

Narrative: Semantics of the Withdraw Command

16.

Formalisation: Semantics of the Withdraw Command

Narrative: Syntax of the Request Statement Command

13. A client command is ... a withdraw ... command.
13(d). A request statements command includes an account number.

Formalisation: Syntax of the Request Statement Command

type
13. Cmd = OpeC | DepC | WitC | ReqC | CloC | ShaC
13(d). ReqC = mkReC(an:Anm)

Narrative: Semantics of the Request Statement Command

17.

Formalisation: Semantics of the Request Statement Command

Narrative: Syntax of the Close Accoount Command

13. A client command is ... a withdraw ... command.
13(e). A close command includes an account number.

Formalisation: Syntax of the Close Account Command

type
13. Cmd = OpeC | DepC | WitC | ReqC | CloC | ShaC
13(e). CloC = mkClC(an:Anm)

Narrative: Semantics of the Close Accoount Command

18.

Formalisation: Semantics of the Close Accoount Command

Narrative: Syntax of the Share Account Command

13. A client command is ... a share ... command.
13(f). A share command mentions an account and a client (who is "invited" to share the account).

Formalisation: Syntax of the Share Account Command

type
13. Cmd = OpeC | DepC | WitC | ReqC | CloC | ShaC
13(f). ShaC = mkSha(an:Anm,c:Cnm)

Narrative: Semantics of the Share Account Command

19.

Formalisation: Semantics of the Share Account Command

4.1.3 Bank Transactions

The bank, in order to be able to manage its (trusted and own) resources need to be able to perform, at is own discretion, some transactions.

Narrative: Syntax

20. A bank command is either a
    (a)
    (b)
    (c)
21.
22.
23.
24.

Formalisation

20.
20(a).
20(b).
20(c).
21.
22.
23.
24.

4.2 Mortgage Banking

4.2.1 The Banking State

Narrative

25.
26.
27.
28.
29.
30.
31.

Formalisation

25.
26.
27.
28.
29.
30.
31.

4.2.2 Customer Transactions

Narrative: Syntax

32.
33.
34.
35.
36.
37.
38.
39.

Formalisation: Syntax

32.
33.
34.
35.
36.
37.
38.
39.

4.2.3 Bank Transactions

Narrative: Syntax

40.
41.
42.
43.
44.
45.
46.
47.

Formalisation: Syntax

40.
41.
42.
43.
44.
45.

46.
47.

5

Banking: Support Facilities

We describe the support facilties that enable the intrinsics, the management & organisation, the rules & regulations, the scripts and that support diligent or cause slopy, neglient, or outright crimilal bahaviour of the main-street set of banking concepts of Demand/Deposit banking and Mortgage (or: Savings & Loan) banking.

6

Banking: Management and Organisation

7

Banking: Rules and Regulations

We describe the rules and regulations that pertain to client use of banking and the banks obligations to clients and national, regional and international banking.

8

Banking: Scripts

We describe the scripts that pertain to the use of demand/deposit and savings & loan (mortgage) accounts.

9

Banking: Human Behaviour

10

Banking: Conclusion

Part II

Stock Brokerage

11

Stock Brokerage: Preamble

12

Stock Brokerage: Intrinsics

13

Stock Brokerage: Support Facilities

14

Stock Brokerage: Management and Organisation

15

Stock Brokerage: Rules and Regulations

16

Stock Brokerage: Scripts

17

Stock Brokerage: Human Behaviour

18

Stock Brokerage: Conclusion

Part III

Credit Cards

19

Credit Cards: Preamble

19.1 Stakeholders

19.2 The Acquisition Process

19.2.1 Studies

19.2.2 Interviews

19.2.3 Questionnaires

19.2.4 Indexed Description Units

19.3 Terminology

20

Credit Cards: Intrinsics

21

Credit Cards: Support Facilities

22

Credit Cards: Management and Organisation

23

Credit Cards: Rules and Regulations

24

Credit Cards: Scripts

25

Credit Cards: Human Behaviour

26

Credit Cards: Conclusion

Part IV

Insurance

27

Insurance: Preamble

27.1 Stakeholders

27.2 The Acquisition Process

27.2.1 Studies

27.2.2 Interviews

27.2.3 Questionnaires

27.2.4 Indexed Description Units

27.3 Terminology

28

Insurance: Intrinsics

29

Insurance: Support Facilities

30

Insurance: Management and Organisation

31

Insurance: Rules and Regulations

32

Insurance: Scripts

33

Insurance: Human Behaviour

34

Insurance: Conclusion

Part V

Portfolio Management

35

Portfolio Management: Preamble

36

Portfolio Management: Intrinsics

37

Portfolio Management: Support Facilities

38

Portfolio Management: Management and Organisation

39

Portfolio Management: Rules and Regulations

40

Portfolio Management: Scripts

41

Portfolio Management: Human Behaviour

42

Portfolio Management: Conclusion

Volume III

Analyses

This volume contains only one chapter: Analyses.

43

Analysis

Volume IV

Closing

This volume contains one chapter:

- Review, Discussion and Conclusion.

And then it contains the appendices:

- Examples of Rough Sketch Descriptions of Financial Services,
- Methodology,
- Indexes, and
- Bibliographical Notes

44

Review, Discussion and Conclusion

Part I

Appendices

A

Examples of Rough Sketch Descriptions of Financial Services

A.1 Financial Service Industry Business Processes

Example A.1 *Financial Service Industry Business Processes:* The main business process behaviours of a financial service system are the following: (i) clients, (ii) banks, (iii) securities instrument brokers and traders, (iv) portfolio managers, (v) (the, or a, or several) stock exchange(s), (vi) stock incorporated enterprises and (vii) the financial service industry "watchdog". We rough-sketch the behaviour of a number of business processes of the financial service industry.

(i) Clients engage in a number of business processes: (i.1) they open, deposit into, withdraw from, obtain statements about, transfer sums between and close demand/deposit, mortgage and other accounts; (i.2) they request brokers to buy or sell, or to withdraw buy/sell orders for securities instruments (bonds, stocks, futures, etc.); and (i.3) they arrange with portfolio managers to look after their bank and securities instrument assets, and occasionally they reinstruct portfolio managers in those respects.

(ii) 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.

(iii) 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.

(iv) Portfolio managers engage with clients, banks, and brokers and traders in exchanges related to client portfolios.

(v) 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.

(vi) 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.

(vii) The financial industry watchdog engages with banks, portfolio managers, brokers and traders and with the stock exchanges. ∎

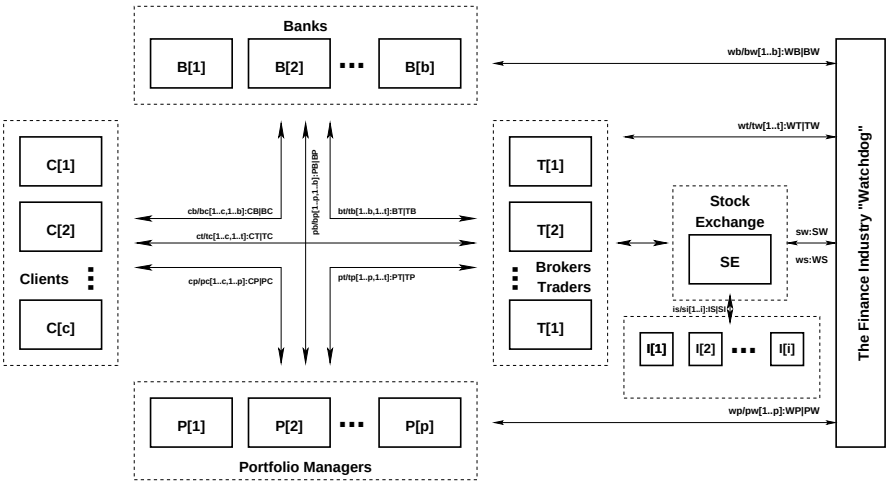

Fig. A.1. A financial behavioural system abstraction

Example A.2 *Atomic Component — A Bank Account:* When we informally speak of the phenomena that can be observed in connection with a bank account, we may first bring up such things as: (i) The balance (or cash, a noun), the credit limit (noun), the interest rate (noun), the yield (noun); and (ii) the opening (verb) of, the deposit (verb) into, the withdrawal (verb) from and the closing (verb) of an account. Then we may identify (iii) the events that trigger the opening, deposit, withdrawal and closing actions. We may thus consider a bank account — with this structure of (i) values, (ii) actions (predicates, functions, operations), and (iii) ability to respond to external events (to open, to deposit, etc.) — to be a component, i.e., a process. ∎

Example A.3 *Composite Component — A Bank:* Likewise, continuing the above example, we can speak of a bank as consisting of any number of bank accounts, i.e., as a composite component of proper constituent bank account components. Other proper constituent components are: the customers (who own the accounts), the bank tellers (whether humans or machines) who services the accounts as instructed by customers, etc. ∎

In the above we have stressed the "internals" of the atomic components. When considering the composite components we may wish to emphasise the interaction between components.

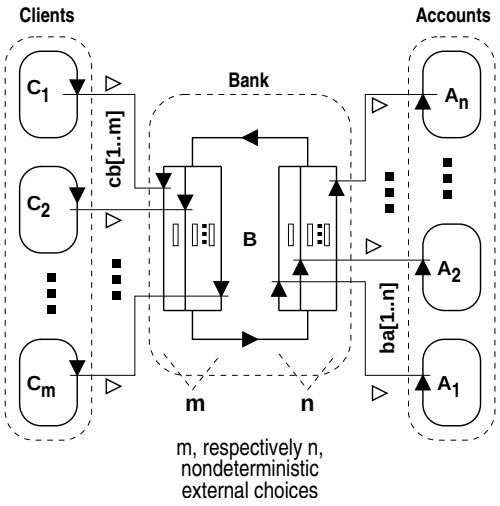

Fig. A.2. A fifth schematic "rendezvous" class

Example A.4 *One-Way Composite Component Interaction:* We illustrate a simple one-way client-to-account deposit. A customer may instruct a bank teller to deposit monies handed over from the customer to the bank teller into an appropriate account, and we see an interaction between three "atomic" components: the client(s), the bank teller(s) and the account(s).
Figure A.2 shows a set of distinct client processes. A client may have one or more accounts and clients may share accounts. For each distinct account there is an account process. The bank (i.e., the bank teller) is a process. It is at any one time willing to input a cash-to-account (a,d) request from any client (c). There are as many channels into (out from) the bank process as there are distinct clients (resp. accounts).

Using formal notation we can expand on the informal picture of Fig. A.2.

type
  Cash, Cash, Cidx, Aidx
channel
  { cb[c]:(Aidx×Cash) | c:Cidx }
  { ba[a]:Cash | a:Aidx }
value
  S5: **Unit** → **Unit**
  S5() ≡ Clients() ‖ B() ‖ Accounts()

  Clients: **Unit** → **out** { cb[c] | c:Cidx } **Unit**
  Clients() ≡ ‖ { C(c) | c:Cidx }

  C: c:Cidx → **out** cp[c] **Unit**

C(c) ≡ **let** (a,d):(Aidx×Cash) = ... **in** cb[c] ! (a,d) **end** ; C(c)

type
A_Bals = Aindex $\xrightarrow[m]{}$ Cash
value
abals: A_Bals

Accounts: **Unit** → **in** { ba[a] | a:AIndex } **Unit**
Accounts() ≡ ‖ { A(a,abals(a)) | a:AIndex }

A: a:Aindex × Balance → **in** ba[a] **Unit**
A(a,d) ≡ **let** d′ = ba[a] ? **in** A(a,d+d′) **end**

B: **Unit** → **in** { cb[c] | c:Cidx } **out** { ba[a] | a:Aidx } **Unit**
B() ≡ [] {**let** (a,d) = cb[c] ? **in** ba[a] ! d **end** | c:Cidx} ; B()

We comment on the deposit example. With respect to the use of notation above, there are Cindex client-to-bank channels, and Aindex bank-to-account channels. The banking system (S5) consists of a number of concurrent processes: Cindex clients, Aindex accounts and one bank. From each client process there is one output channel, and into each account process there is one input channel. Each client and each account process cycles around depositing, respectively cashing monies. The bank process is nondeterministically willing ([]) to engage in a rendezvous with any client process, and passes any such input onto the appropriate account.

Generally speaking, we illustrated a banking system of many clients and many accounts. We only modelled the deposit behaviour from the client via the bank teller to the account. We did not model any reverse behaviour, for example, informing the client as to the new balance of the account. So the two bundles of channels were both one-way channels. We shall later show an example with two-way channels. ∎

Example A.5 *Multiple, Diverse Component Interaction:* We illustrate composite component interaction. At regular intervals, as instructed by some service scripts associated with several distinct kinds of accounts, transfers of monies may take place between these. For example, a regular repayment of a loan may involve the following components, operations and interactions: An appropriate repayment amount, p, is communicated from client k to the bank's script servicing component se (3).[1]Based on the loan debt and its interest rate (d,ir) (4), and this repayment (p), a distribution of annuity (a), fee (f) and interest (i) is calculated.[2]The loan repayment sum total, p, is subtracted from the balance, b, of the demand/deposit account, dd_a, of the client (5). A loan service fee, f, is added to the (loan service) fee account, f_a, of the bank (7). The interest on the balance of the loan since the last repayment is added to the interest account, i_a, of the bank (8), and the difference, a, (the effective

repayment), between the repayment, p, and the sum of the fee and the interest is subtracted from the principal, p, of the mortgage account, m_a, of the client (6).

In process modelling the above we are stressing the communications. As we shall see, the above can be formally modelled as below.

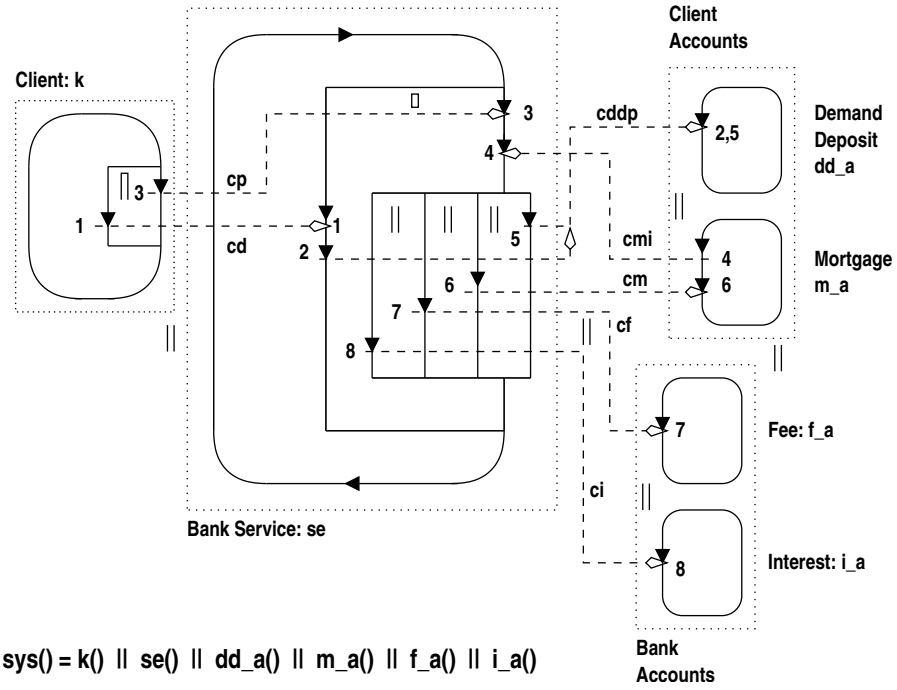

Fig. A.3. A loan repayment scenario

type
Monies,Deposit,Loan,
Interest_Income,Fee_Income = **Int**,
Interest = Rat
channel
cp,cd,cddp,cm,cf,ci:Monies, cmi:Interest
value
sys: **Unit** → **Unit**,
sys() ≡ se() ‖ k() ‖ dd_a(b) ‖ m_a(p) ‖ f_a(f) ‖ i_a(i)

[1] For references (3–8) we refer to Fig. A.3.
[2] See line four of the body of the definition of the se process below.

```
k: Unit → out cp,cd  Unit
k() ≡
      (let p:Nat • /∗ p is some repayment, 1 ∗/ in cp ! p end
   ⌈⌉
      let d:Nat • /∗ d is some deposit, 2 ∗/ in cd ! d  end)
   ; k()

se: Unit → in cd,cp,cmi out cddp,cm,cf,ci  Unit
se() ≡
      ((let d = cd ? in cddp ! d end) /∗ 1,2 ∗/
   ⌈⌉
      (let (p,(ir,ℓ)) = (cp ?,cmi ?) in /∗ 3,4 ∗/
      let (a,f,iv) = o(p,ℓ,ir) in
      (cddp ! (−p) ‖ cm ! a ‖ cf ! f ‖ ci ! iv) end end)) /∗ 5,6,7,8 ∗/
   ; se()


dd_a: Deposit → in cddp  Unit
dd_a(b) ≡ dd_a(b + cddp ?) /∗ 2,5 ∗/

m_a: Interest × Loan → out cmi in cm  Unit
m_a(ir,ℓ) ≡ cmi ! (ir,ℓ) ; m_a(ir,ℓ− cm ?) /∗ 4;6 ∗/

f_a: Fee_Income → in cf  Unit
f_a(f) ≡ f_a(f + cf ?) /∗ 7 ∗/

i_a: Interest Income → in ci  Unit
i_a(i) ≡ i_a(i + ci ?) /∗ 8 ∗/
```

The formulas above express:

- The composite component, a bank, consists of:
  - ⋆ a customer, k, connected to the bank (service), se, via channels cd, cp
  - ⋆ that customer's demand/deposit account, dd_a, connected to the bank (service) via channels cdb, cddp
  - ⋆ that customer's mortgage account, m_a, connected to the bank (service) via channel cm
  - ⋆ a bank fees income account, f_a, connected to the bank (service) via channel cf
  - ⋆ a bank interest income account, i_a, connected to the bank (service) via channel ci
- The customer demand/deposit account is willing, at any time, to nondeterministically engage in communication with the service: either accepting (?) a deposit or loan repayment (2 or 5), or delivering (!) information about the loan balance and interest rate (4).

- We model this "externally inflicted" behaviour by (what is called) the *external nondeterministic choice*, $[]^3$, operation.
- The service component, in a nondeterministic external choice, [], either accepts a customer deposit (cd?) or a mortgage payment (cp?).
- The deposit is communicated (cddp!d) to the demand/deposit account component.
- The fee, interest and annuity payments are communicated in parallel (‖) to each of the respective accounts: bank fees income (cf!f), bank interest income (ci!i) and client mortgage (cm!a) account components.
- The customer is unpredictable, may issue either a deposit or a repayment interaction with the bank.
- We model this "self-inflicted" behaviour by (what is called) the *internal nondeterministic choice*, $\sqcap^4$, operation.

■

Characterisation. By a *nondeterministic external choice* we mean a nondeterministic decision which is effected, not by actions prescribed by the text in which the [] operator occurs, but by actions in other processes. That is, speaking operationally, the process honouring the [] operation does so by "listening" to the environment. ■

Characterisation. By *nondeterministic internal choice* we mean a nondeterministic decision that is implied by the text in which the ⊓ operator occurs. Speaking operationally, the decision is taken locally by the process itself, not as the result of any event in its surroundings. ■

A.1.1 Some Modelling Comments — An Aside

Examples A.4 and A.5 illustrated one-way communication, from clients via the bank to accounts. Example A.4 illustrated bank "multiplexing" between several (m) clients and several (n) accounts. Example A.5 illustrated a bank with just one client and one pair of client demand/deposit and mortgage accounts. Needless to say, a more realistic banking system would combine the above. Also, we have here chosen to model each account as a process. It is reasonable to model each client as a separate process, in that the collection of all clients can be seen as a set of independently and concurrently operating components. To model the large set of all accounts as a similarly large set of seemingly independent and concurrent processes can perhaps be considered a "trick": It makes, we believe, the banking system operation more transparent.

[3] See the definition of what is meant by nondeterministic external choice right after this example.

[4] See the definition of what is meant by nondeterministic internal choice right after this example.

In the next — and final — example of this introductory section we augment the first example with an account balance response being sent back from the account via the bank to the client.

A.1.2 Examples Continued

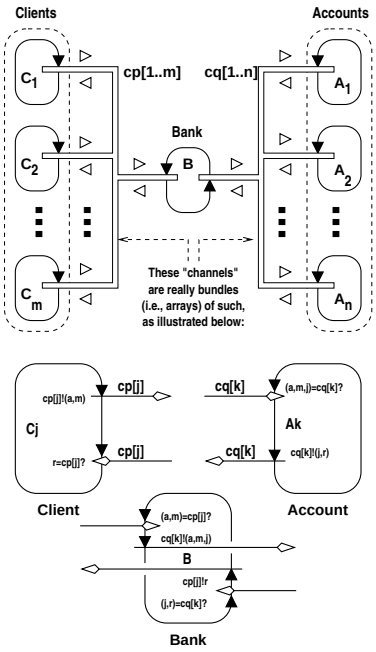

Fig. A.4. Two-way component interaction

Example A.6 *Two-Way Component Interaction:* The present example "contains" that of the one-way component interaction of Example A.4. Each of the client, bank and account process definitions are to be augmented as shown in Fig. A.4 and in the formulas that follow (cf. Fig. A.2 and the formulas in Example A.4).

type
  Cash, Balance, CIndex, AIndex
  CtoB = AIndex × Cash,
  BtoC = Balance,
  BtoA = Cindex × Cash,
  AtoB = Cindex × Balance
channel
  cb[1..m] CtoB|BtoC, ba[1..n] BtoA|AtoB
value

```
S6: Unit → Unit
S6() ≡
    ‖ { C(c) | c:CIndex } ‖ B() ‖
    ‖ { A(a,b,r) | a:AIndex, b:Balance, r:Response • ... }

C: c:CIndex → out cp[ c ] Unit
C(c) ≡
    let (a,d):(AIndex×Cash) = ... in
    cb[ c ] ! (d,a) end let r = cb[ c ] ? in C(c) end

B: Unit → in,out {cb[ c ]|c:CIndex} in,out {ba[ a ]|a:AIndex} Unit
B() ≡ [] {let (d,a) = cb[ c ] ? in ba[ a ] ! (c,d) end | c:Cindex} []
       [] {let (c,b) = ba[ a ] ? in bc[ c ] ! b end | a:Aindex} ; B()

A: a:Aindex × Balance → in,out ba[ a ] Unit
A(a,b) ≡ let (c,m) = ba[ a ] ? in ba[ a ] ! (m+b) ; A(a,m+b) end
```

We explain the formulas above. Both the C and the A definitions specify pairs of communications: deposit output followed by a response input, respectively a deposit input followed by a balance response output. Since many client deposits may occur while account deposit registrations take place, client identity is passed on to the account, which "returns" this identity to the bank — thus removing a need for the bank to keep track of client-to-account associations. The bank is thus willing, at any moment, to engage in any deposit and in any response communication from clients, respectively accounts. This is expressed using the nondeterministic external choice combinator []. ∎

Example A.7 *A Bank System Context and State:*

The Context

We focus in this example on the demand/deposit aspects of an ordinary bank. The bank has clients k:K. Clients have one or more numbered accounts c:C. Accounts, a:A, may be shared between two or more clients. Each account is established and "governed" by an initial contract, ℓ:L ('L' for legal).The account contract specifies a number of parameters: the yield, by rate (i.e., percentage), y:Y, due the client on positive deposits; the interest, by rate (i.e., percentage), i:I, due the bank on negative deposits less than a normal credit limit, n:N; the period (frequency), f:F, between (of) interest and yield calculations; the number of days, d:D, between bank statements sent to the client; and personal client information, p:P (name, address, phone number, etc.).

The State

Above we focused on the "syntactic" notion of a client/account contract and what it prescribed. We now focus on the "semantic" notion of the client ac-

count. The client account a:A contains the following information: the balance, b:B (of monies in the account, whether debit or credit, i.e., whether positive or negative), a list of time-stamped transactions "against" the account: establishment, deposits, withdrawals, transfers, interest/yield calculation, whether the account is frozen (due to its exceeding the credit limit), or (again) freed (due to restoration of balance within credit limits), issue of statement, and closing of account. Each transaction records the transaction type, and if deposit, withdrawal or transfer and the amount involved, as well as possibly some other information.

A Model

We consider contract information a contextual part of the bank configuration, while the account part is considered a state part of the bank configuration. We may then model the bank as follows:

type
K, C, Y, I, N, D, P, B, T
[Bank: Configuration]
Bank = $\Gamma \times \Sigma$
[Γ: Context]
Γ = (K $\overrightarrow{m}$ C-**set**) × (C $\overrightarrow{m}$ L)
L == mkL(y:Y,i:I,n:N,f:F,d:D,p:P)
[Σ: State]
Σ = C $\overrightarrow{m}$ A
A = {free|frozen} × B × (T × Trans)*
Trans = Est|Dep|Wth|Xfr|**Int**|Yie|Frz|Fre|Stm|Sha|Clo
Dep == deposit(m:**Nat**)
Wth == withdraw(m:**Nat**)
Xfr == toxfer(to:C,m:**Nat**) | fmxfer(fm:C,m:**Nat**)
Sha == share(new:C,old:C)

Bank is here the configuration.[5] Γ is the context. Σ is the state. ∎

The banking system so far outlined is primarily a dynamic, programmable system: Most transactions, when obeyed, change the (account) state σ:Σ. A few (to wit: establish, share) change the context γ:Γ. Establishment occurs exactly once in the lifetime of an account. Initially contracts, from which the γ:Γ configuration component is built, are thought of as specifying only one client. Hence the share transaction, which "joins" new clients to an account, could as well be thought of as an action: one changing the state, rather than the context. We have arbitrarily chosen to model it as a context changing "action"! All this to show that the borderline between context and state is "soft": It is a matter of choice.

[5] But, the bank configuration could, in more realistic situations, include many other components not related directly to the client/account "business".

Notice that, although time enters into the banking model, we did not model time flow explicitly. Here, in the man-made system model, it is considered "outside" the model. We claim that the concepts of context and state enter, in complementary ways, into both physical systems and man-made systems. Before proceeding with more detailed analysis of the configuration (cum context ⊕ state) ideas, let us recall that these concepts are pragmatic.

48. *No money printing:* Financial transactions between financial institutions (transfers of monies between banks, or to or from insurance companies, stockbrokers, portfolio managers, etc.) do not themselves "generate monies": The sum total of monies within the system is unchanged — money is only "moved".
49. *Life is like a sewer, what you put into it is what you get out of it (II):* The only changes in the sum total of monies of a financial system (of banks, insurance companies, stockbrokers, funds managers, etc.) is when clients residing outside this system deposits or withdraws funds.
50. *Financial services:*
    The system of banks (including a national or federal, etc., bank), insurance companies, stockbrokers and traders, stock exchanges, portfolio managers, and the external clients of these "components" (bank account holders, insurance holders, buyers and sellers of securities instruments, etc.), as well as the externally observable events within as well as between these "system" components and between these and their clients, could form a domain. Some of these events trigger actions, such as: opening an account, depositing monies, withdrawing monies, transferring monies, buying or selling stocks, etc.

A.2 Bank Scripts

A.2.1 Bank Scripts: A Denotational, Ideal Description

Example A.8 *Bank Scripts, I:* Without much informal explanation, i.e., narrative, we define a small bank, small in the sense of offering but a few services. One can open and close demand/deposit accounts. One can obtain and close mortgage loans, i.e., obtain loans. One can deposit into and withdraw from demand/deposit accounts. And one can make payments on the loan. In this example we illustrate informal rough-sketch scripts while also formalising these scripts.

In the following we first give the formal specification, then a rough-sketch script. You may prefer to read the pairs, formal specification and rough-sketch script, in the reverse order.

Bank State

Formal Presentation: Bank State

type
C, A, M
AY′ = **Real**, AY = {| ay:AY′ • 0<ay≤10 |}
MI′ = **Real**, MI = {| mi:MI′ • 0<mi≤10 |}
Bank′ = A_Register × Accounts × M_Register × Loans
Bank = {| β:Bank′ • wf_Bank(β)|}
A_Register = C $\xrightarrow[m]{}$ A**-set**
Accounts = A $\xrightarrow[m]{}$ Balance
M_Register = C $\xrightarrow[m]{}$ M**-set**
Loans = M $\xrightarrow[m]{}$ (Loan × Date)
Loan,Balance = P
P = **Nat**

There are clients (c:C), account numbers (a:A), mortgage number (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.

State Well-formedness

Formal Presentation: State Well-formedness

value
ay:AY, mi:MI

wf_Bank: Bank → **Bool**
wf_Bank(ρ,α,μ,ℓ) ≡ ∪ **rng** ρ = **dom** α ∧ ∪ **rng** μ = **dom** ℓ
axiom
ai<mi

We assume a fixed yield, ai, on demand/deposit accounts, and a fixed interest, mi, on loans. A bank is well-formed if all accounts named in the accounts register are indeed accounts, and all loans named in the mortgage register are indeed mortgages. No accounts and no loans exist unless they are registered.

Client Transactions

Formal Presentation: Syntax of Client Transactions

```
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)
  CloM == mkCM(c:C,m:M,p:P)
  Reply = A | M | P | OkNok
  OkNok == ok | notok
```

The client can issue the following commands: Open Account, Close Account, Deposit monies (p:P), Withdraw monies (p:P), Obtain loans (of size p:P) and Pay installations on loans (by transferring monies from an account). Loans can be Closed when paid down.

Open Account Transaction

Formal Presentation: Semantics of Open Account Transaction

```
value
  int_Cmd: Cmd → Bank → Bank × Reply

  int_Cmd(mkOA(c))(ρ,α,μ,ℓ) ≡
    let a:A • a ∉ dom α in
    let as = if c ∈ dom ρ then ρ(c) else {} end ∪ {a} in
    let ρ′ = ρ † [ c↦as ],
        α′ =  α ∪ [ a↦0 ] in
    ((ρ′,α′,μ,ℓ),a) end end end
```

When opening an account the new account number is registered and the new account set to 0. The client obtains the account number.

Close Account Transaction

Formal Presentation: Semantics of Close Account Transaction

```
int_Cmd(mkCA(c,a))(ρ,α,μ,ℓ) ≡
    let ρ′ = ρ † [ c↦ρ(c)\{a} ],
        α′ = α \ {a} in
    ((ρ′,α′,μ,ℓ),α(a)) end
    pre c ∈ dom ρ ∧ a ∈ ρ(c)
```

When closing an account the account number is deregistered, the account is deleted, and its balance is paid to the client. It is checked that the client is a bona fide client and presents a bona fide account number. The well-formedness condition on banks secures that if an account number is registered then there is also an account of that number.

Deposit Transaction

Formal Presentation: Semantics of Deposit Transaction

```
int_Cmd(mkD(c,a,p))(ρ,α,μ,ℓ) ≡
    let α′ = α † [ a↦α(a)+p ] in
    ((ρ,α′,μ,ℓ),ok) end
    pre c ∈ dom ρ ∧ a ∈ ρ(c)
```

When depositing into an account that account is increased by the amount deposited. It is checked that the client is a bona fide client and presents a bona fide account number.

Withdraw Transaction

Withdrawing monies can only occur if the amount is not larger than that deposited in the named account. Otherwise the amount, p:P, is subtracted from the named account. It is checked that the client is a bona fide client and presents a bona fide account number.

Formal Presentation: Semantics of Withdraw Transaction

```
int_Cmd(mkW(c,a,p))(ρ,α,μ,ℓ) ≡
    if α(a)≥p
        then
            let α′ = α † [ a↦α(a)−p ] in
            ((ρ,α′,μ,ℓ),p) end
        else
            ((ρ,α,μ,ℓ),nok)
```

```
    end
    pre c ∈ dom ρ ∧ a ∈ dom α
```

Open Mortgage Account Transaction

Formal Presentation: Semantics of Open Mortgage Account Transaction

```
int_Cmd(mkOM(c,p))(ρ,α,μ,ℓ) ≡
    let m:M • m ∉ dom ℓin
    let ms = if c ∈ dom μ then μ(c) else {} end ∪ {m} in
    let mu′ = μ † [c↦ms],
        α′ = α † [a_ℓ↦α(a_ℓ)−p],
        ℓ′ = ℓ ∪ [m↦p] in
    ((ρ,α′,μ′,ℓ′),m) end end end
```

To obtain a loan, p:P, is to open a new mortgage account with that loan (p:P) as its initial balance. The mortgage number is registered and given to the client. The loan amount, p, is taken from a specially designated bank capital acount, a_ℓ. The bank well-formedness condition should be made to reflect the existence of this account.

Close Mortgage Account Transaction

Formal Presentation: Semantics of Close Mortgage Account Transaction

```
int_Cmd(mkCM(c,m))(ρ,α,μ,ℓ) ≡
    if ℓ(m) = 0
      then
        let μ′ = ρ † [c↦μ(c) \ {m}],
            ℓ′ = ℓ \ {m} in
        ((ρ,α,μ′,ℓ′),ok) end
      else
        ((ρ,α,μ,ℓ),nok)
    end
    pre c ∈ dom μ ∧ m ∈ μ(c)
```

One can only close a mortgage account if it has been paid down (to 0 balance). If so, the loan is deregistered, the account removed and the client given an OK. If not paid down the bank state does not change, but the client is given a NOT OK. It is checked that the client is a bona fide loan client and presents a bona fide mortgage account number.

Loan Payment Transaction

Formal Presentation: Semantics of Loan Payment Transaction

To pay off a loan is to pay the interest on the loan since the last time interest was paid. That is, interest, i, is calculated on the balance, b, of the loan for the period $d' - d$, at the rate of mi. (We omit defining the interest computation.) The payment, p, is taken from the client's demand/deposit account, a; i is paid into a bank (interest earning account) a_i and the loan is diminished with the difference $p-i$. It is checked that the client is a bona fide loan client and presents a bona fide mortgage account number. The bank well-formedness condition should be made to reflect the existence of account a_i.

```
int_Cmd(mkPM(c,a,m,p,d′))(ρ,α,μ,ℓ) ≡
    let (b,d) = ℓ(m) in
    if α(a)≥p
        then
            let i = interest(mi,b,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 ∧ m ∈ μ(c)
```

This ends the first stage of the development of a script language. ∎

A.2.2 Bank Scripts: A Customer Language

Example A.9 *Bank Scripts, II:* From each of the informal/formal bank script descriptions we systematically "derive" a script in a possible bank script language. The derivation, for example, for how we get from the formal descriptions of the individual transactions to the scripts in the "formal" bank script language is not formalised. In this example we simply propose possible scripts in the formal bank script language.

Open Account Transaction

Formal Presentation: Open Account Transaction

```
value
    int_Cmd(mkOA(c))(ρ,α,μ,ℓ) ≡
        let a:A • a ∉ dom α in
```

```
    let as = if c ∈ dom ρ then ρ(c) else {} end ∪ {a} in
    let ρ′ = ρ † [ c↦as ],
        α′ = α ∪ [ a↦0 ] in
    ((ρ′,α′,μ,ℓ),a) end end end
```

Derived Bank Script: Open Account Transaction

```
routine open_account(c in "client",a out "account") ≡
    do
      register c with new account a ;
      return account number a to client c
    end
```

Close Account Transaction

Formal Presentation: Close Account Transaction

```
int_Cmd(mkCA(c,a))(ρ,α,μ,ℓ) ≡
    let ρ′ = ρ † [ c↦ρ(c)\{a} ],
        α′ = α \ {a} in
    ((ρ′,α′,μ,ℓ),α(a)) end
    pre c ∈ dom ρ ∧ a ∈ ρ(c)
```

Derived Bank Script: Close Account Transaction

```
routine close_account(c in "client",a in "account" out "monies") ≡
    do
      check that account client c is registered ;
      check that account a is registered with client c ;
      if
        checks fail
          then
            return NOT OK to client c
          else
            do
              return account balance a to client c ;
              delete account a
            end
      fi
    end
```

Deposit Transaction

Formal Presentation: Deposit Transaction

```
int_Cmd(mkD(c,a,p))(ρ,α,μ,ℓ) ≡
  let α′ = α † [a↦α(a)+p] in
  ((ρ,α′,μ,ℓ),ok) end
  pre c ∈ dom ρ ∧ a ∈ ρ(c)
```

Derived Bank Script: Deposit Transaction

```
routine deposit(c in "client",a in "account",ma in "monies") ≡
    do
      check that account client c is registered ;
      check that account a is registered with client c ;
      if
        checks fail
          then
            return NOT OK to client c
          else
            do
              add ma to account a ;
              return OK to client c
            end
      fi
    end
```

Withdraw Transaction

Formal Presentation: Withdraw Transaction

```
int_Cmd(mkW(c,a,p))(ρ,α,μ,ℓ) ≡
  if α(a)≥p
    then
      let α′ = α † [a↦α(a)−p] in
      ((ρ,α′,μ,ℓ),p) end
    else
      ((ρ,α,μ,ℓ),nok)
  end
  pre c ∈ dom ρ ∧ a ∈ dom α
```

Derived Bank Script: Withdraw Transaction

```
routine withdraw(c in ″client″,a in ″account″,
                                    ma in ″amount″ out ″monies″) ≡
    do
      check that account client c is registered ;
      check that account a is registered with client c ;
      check that account a has ma or more balance;
      if
        checks fail
          then
            return NOT OK to client c
          else
            do
              subtract ma from account a ;
              return ma to client c
            end
      fi
    end
```

Obtain Loan Transaction

Formal Presentation: Obtain Loan Transaction

$$
\begin{array}{l}
\text{int\_Cmd}(\text{mkOM}(\text{c,p}))(\rho,\alpha,\mu,\ell) \equiv \\
\quad \textbf{let } \text{m:M} \bullet \text{m} \notin \textbf{dom } \ell \textbf{ in} \\
\quad \textbf{let } \text{ms} = \textbf{if } \text{c} \in \textbf{dom } \mu \textbf{ then } \mu(\text{c}) \textbf{ else } \{\} \textbf{ end} \cup \{\text{m}\} \textbf{ in} \\
\quad \textbf{let } \text{mu}' = \mu \dagger [\,\text{c} \mapsto \text{ms}\,], \\
\qquad \alpha' = \alpha \dagger [\,\text{a}_\ell \mapsto \alpha(\text{a}_\ell) - \text{p}\,], \\
\qquad \ell' = \ell \cup [\,\text{m} \mapsto \text{p}\,] \textbf{ in} \\
\quad ((\rho,\alpha',\mu',\ell'),\text{m}) \textbf{ end end end}
\end{array}
$$

Derived Bank Script: Obtain Loan Transaction

```
routine get_loan(c in ″client″,p in ″amount″,m out ″loan number″) ≡
    do
      register c with loan m amount p;
      subtract p from account bank's loan capital
      return loan number m to client c
    end
```

Close Loan Transaction

Formal Presentation: Close Loan Transaction

```
int_Cmd(mkCM(c,m))(ρ,α,μ,ℓ) ≡
    if ℓ(m) = 0
       then
          let μ′ = ρ † [c↦μ(c)\{m}],
              ℓ′ = ℓ \ {m} in
          ((ρ,α,μ′,ℓ′),ok) end
       else
          ((ρ,α,μ,ℓ),nok)
    end
    pre c ∈ dom μ ∧ m ∈ μ(c)
```

Derived Bank Script: Close Loan Transaction

```
routine close_loan(c in "client",m in "loan number") ≡
      do
         check that loan client c is registered ;
         check that loan m is registered with client c ;
         check that loan m has 0 balance;
         if
            checks fail
               then
                  return NOT OK to client c
               else
                  do
                     close loan m
                     return OK to client c
                  end
         fi
      end
```

Loan Payment Transaction

Formal Presentation: Loan Payment Transaction

```
int_Cmd(mkPM(c,a,m,p,d′))(ρ,α,μ,ℓ) ≡
    let (b,d) = ℓ(m) in
    if α(a)≥p
       then
          let i = interest(mi,b,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 ∧ m ∈ μ(c)
```

Derived Bank Script: Loan Payment Transaction

```
routine pay_loan(c in "client",m in "loan number",p in "amount") ≡
    do
        check that loan client c is registered ;
        check that loan m is registered with client c ;
        check that account a is registered with client c ;
        check that account a has p or more balance ;
        if
            checks fail
                then
                    return NOT OK to client c
                else
                    do
                        compute interest i for loan m on date d ;
                        subtract p−i from loan m ;
                        subtract p from account a ;
                        add i to account bank's interest
                        return OK to client c ;
                    end
        fi
    end
```

This ends the second stage of the development of a script language. ∎

A.2.3 Syntax of Bank Script Language

Example A.10 *Bank Scripts, III:* We now examine the proposed scripts. Our objective is to design a syntax for the language of bank scripts. First, we list the statements as they appear in Example A.9 on page 140, except for the first two statements.

Routine Headers

We first list all routine "headers":

open_account(c **in** ″client″,a **out** ″account″)
close_account(c **in** ″client″,a **in** ″account″ **out** ″monies″)
deposit(c **in** ″client″,a **in** ″account″,ma **in** ″monies″)
withdraw(c **in** ″client″,a **in** ″account″,ma **in** ″amount″ **out** ″monies″)
get_loan(c **in** ″client″,p **in** ″amount″,m **out** ″loan number″)
close_loan(c **in** ″client″,m **in** ″loan number″)
pay_loan(c **in** ″client″,m **in** ″loan number″,p **in** ″amount″)

We then schematise a routine "header":

routine name(v1 io ″t″,v2 io ″t2″,...,vn io ″tn″) ≡

where:

io = **in** | **out**

and:

ti is any text

Example Statements

do stmt_list ***end***
if test_expr ***then*** stmt ***else*** stmt ***fi***

register c ***with new account*** a
register c ***with loan*** m ***amount*** p

add p ***to account*** a
subtract p ***from account*** a
subtract p−i ***from loan*** m
add i ***to account bank's interest***
subtract p ***from account bank's loan capital***
add p ***to account bank's loan capital***
compute interest i ***for loan*** m ***on date*** d

delete account a
close loan m

return ret_expr ***to client*** c
check that check_expr

The interest variable i is a ***local*** variable. The date variable d is an "oracle" (see below), but will be treated as a ***local*** variable.

Example Expressions

test_expr:

checks fail

ret_expr:

account number a
account balance a
NOT OK
OK
p
loan number m

check_expr:

account client c ***is registered***
account a ***is registered with client*** c
account a ***has*** p ***or more balance***
loan client c ***is registered***
loan m ***is registered with client*** c
loan m ***has*** 0 ***balance***

Abstract Syntax for Syntactic Types

We analyse the above concrete schemas (i.e., examples). Our aim is to find a reasonably simple syntax that allows the generation of the scripts of Example A.9. After some experimentation we settle on the syntax shown next.

Formal Presentation: Bank Script Language Syntax

type
RN, V, C, A, M, P, I, D

Routine = Header × Clause

Header == mkH(rn:RN,vdm:(V $\overrightarrow{m}$ (IOL × **Text**)))
IOL == **in** | **out** | **local**

Clause = DoEnd | IfThEl | Return | RegA | RegL | Check
| Add | Sub | 2Sub | DelA | DelM | ComI | RetE |

DoEnd == mkDE(cl:Clause*)

```
    IfThEl == mkITE(tex:Test_Expr,cl:Clause,cl:Clause)

    Return == mkR(rex:Ret_Expr,c:V)
    RegA == mkRA(c:V,a:V)
    RegL == mkRL(c:V,m:V,p:V)
    Chk = mkC(cex:Chk_Expr)
    Add == mkA(p:V,t:(V|BA))
    Sub == mkS(p:V,t:(V|BA))
    2Sub == mk2S(p:V,i:V,t:(AN|MN|BA))
      AN == mkAN(a:V)
      MN == mkMN(m:V)
      BA == bank_i | bank_c
    DelA == mkDA(c:V,a:V)
    DelM == mkDM(c:V,m:V)
    Comp == mkCP(m:V,fn:Fn,argl:(V|D)*)

      Fn == interest | ...

  Test_Expr = mkTE()

  Chk_Expr == CisAReg(c:V) | AisReg(a:V,c:V) | AhasP(a:V,p:V)
            |  CisMReg(c:V) | MisReg(m:V,c:V) | Mhas0(m:V)

  RetE == mkAN(a:V)|mkAB(a:V)|ok|nok|mkP(p:V)|mkMN(m:V)
```

■

A.2.4 Semantics of Bank Script Language

Example A.11 *Bank Scripts, IV:*

Formal Presentation: Semantics of Bank Script Language

We now give semantics to the bank script language of Example A.10 on page 145.

Semantic Types Abstract Syntax

```
type
  V, C, A, M, P, I
type
  AY′ = Real, AY = {| ay:AY′ • 0<ay≤10 |}
  MI′ = Real, MI = {| mi:MI′ • 0<mi≤10 |}
  Bank′ = A_Register × Accounts × M_Register × Loans
  Bank = {| β:Bank′ • wf_Bank(β)|}
```

```
  A_Register = C -m-> A-set
  Accounts = A -m-> Balance
  M_Register = C -m-> M-set
  Loans = M -m-> (Loan × Date)
  Loan,Balance = P
  P = Nat
  Σ = (V -m-> (C|A|M|P|I)) ⋃ (Fn -m-> FCT)
  FCT = (...|Date)* → Bank → (P|...)
value
  a_ℓ,a_i:A
axiom
  ∀ (ρ,α,μ,ℓ):B {a_ℓ,a_i} ⊆ dom α
```

The only difference between the above semantics types and those of Example A.9 is the Σ state. The purpose of this auxiliary bank state component is to provide (i) a binding between the (always fixed) formal parameters of the script routines and the actual arguments given by the bank client or bank clerk when invoking any one of the routines, and (ii) a binding of a variety of "primitive", fixed, banking functions, FCT, named Fn, like computing the interest on loans, etc.

Semantic Functions

```
channel
  k:(C|A|M|P|Text), d:Date
```

There is, in this simplifying example, one channel, k, between the bank and the client. It transfers text messages from the bank to the client, and client names (c:C), client account numbers (a:A), client mortgage numbers (m:M), and amount requests and monies (p:P) from the client to the bank. There is also a "magic", a demonic channel, d, which connects the bank to a date "oracle".

```
value
  date: Date → out d  Unit
  date(da) ≡ (d!da ; date(da+Δ))
```

Each routine has a header and a clause. The purpose of the header is to initialise the auxiliary state component σ to appropriate bindings of formal routine parameters to actual, client-provided arguments. Once initialised, interpretation of the routine clause can take place.

```
  int_Routine: Routine → Bank → out k  Bank×Σ
  int_Routine(hdr,cla)(β) ≡
    let σ = initialise(hdr)([ ]) in
```

```
    Int_Clause(cla)(σ)(true)(β) end
```

For each formal parameter used in the body, i.e., in the clause, of the routine, there is a formal parameter definition in the header, and only for such. We have not expressed the syntactic well-formedness condition — but leave it as an exercise to the reader. And for each such formal parameter of the header a binding has now to be initially established. Some define input arguments, some define local variables and the rest define, i.e., name, output results. For each input argument the meaning of the header therefore specifies that an interaction is to take place, with the environment, as here designated by channel k, in order to obtain the actual value of that argument.

```
initialise: Header → Σ → out,in k  Σ
initialise(hdr)(σ) ≡
  if hdr = [ ]
    then σ
    else
      let v:V • v ∈ dom hdr in
      let (iol,txt) = hdr(v) in
      let σ′ =
        case iol of
          in → k!txt ; σ ∪ [ v ↦ k? ],
          _  → σ ∪ [ v ↦ undefined ]
        end in
      initialise(hdr\{v})(σ′)
  end end end end
```

In general, a clause is interpreted in a configuration consisting of three parts: (i) the local, auxiliary state, $\sigma : \Sigma$, which binds routine formal parameters to their values; (ii) the current 'check' state, tf:Check, which records the "sum total", i.e., the conjunction status of the check commands so far interpreted, i.e., initially tf = **true**; and (iii) the proper bank state, β:Bank, exactly as also defined and used in Example A.9. The result of interpreting a clause is a configuration: $(\Sigma \times \mathsf{Check} \times \mathsf{Bank})$.

```
type
  Check = Bool
value
  Int_Clause: Clause→Σ→Check→Bank→out k,in d (Σ×Check×Bank)
```

A **do ... end** clause is interpreted by interpreting each of the clauses within the clauses in the **do ... end** clause list, and in their order of appearance. The result of a check clause is "anded" (conjoined) to the current tf:Check status.

```
  Int_Clause(mkDE(cll))(σ)(tf)(β) ≡
```

```
    if cll = ⟨⟩
      then (σ,tf,β)
      else
        let (σ′,tf′,β′) = Int_Clause(hd cl)(σ)(tf)(β) in
        Int_Clause(mkDE(tl cll))(σ′)(tf∧tf′)(β′)
    end end
```

if ... then ... else fi clauses only test the current check status (and propagate this status).

```
  Int_Clause(mkITE(tex,ccl,acl))(σ)(tf)(β) ≡
    if tf
      then
        Int_Clause(ccl)(σ)(true)(β)
      else
        Int_Clause(acl)(σ)(false)(β)
    end
```

Interpretation of a **return** clause does not change the configuration "state". It only leads to an output, to the environment, via channel k, of a return value, and as otherwise directed by any of the six return expressions (rex).

```
  Int_Clause(mkRet(rex))(σ)(tf)(ρ,α,μ,ℓ) ≡
    k!(case rex of
         mkAN(a)
           → ″Your new account number:″ σ(a),
         mkAB(a)
           → ″Your account balance paid out:″ α(a),
         mkP(p)
           → ″Monies withdrawn:″ σ(p),
         mkMN(m)
           → ″Your loan number:″ σ(m),
         OK
           → ″Transaction was successful″,
         NOK
           → ″Transaction was not successful″
       end);
    (σ,true,(ρ,α,μ,ℓ))
```

Interpretation of a **register account** clause is as you would expect from Example A.9 — anything else would "destroy" the whole purpose of having a bank script. That purpose is, of course, to effect basically the same as the not yet "script-ised" semantics of Example A.9.

```
  Int_Clause(mkRA(c,a))(σ)(tf)(ρ,α,μ,ℓ) ≡
    let av:A • av ∉ dom α in
```

```
    let σ′ = σ † [ a ↦ av ],
        as = if c ∈ dom ρ then ρ(c) else {} end,
        ρ′ = ρ † [ c ↦ as ∪ {av} ],
        α′ = α ∪ [ av ↦ 0 ] in
    (σ′,tf,(ρ′,α′,μ,ℓ))
    end end
```

The same holds for the **register loan** clause (as for the **register account** clause).

```
Int_Clause(mkRL(c,m,p))(σ)(tf)(ρ,α,μ,ℓ) ≡
    let mv:M • mv ∉ dom ℓin
    let σ′ = σ † [ m ↦ mv ],
        ms = if c ∈ dom μ then μ(c) else {} end,
        μ′ = μ † [ c ↦ ms ∪ {mv} ],
        ℓ′ = ℓ ∪ [ mv ↦ p ] in
    (σ′,tf,(ρ,α,μ′,ℓ′))
    end end
```

It can be a bit hard to remember the "meaning" of the mnemonics, so we repeat them here in another form:

- CisAReg: Client named in c is registered:
  $\sigma(\mathsf{c}) \in \mathbf{dom}\ \rho$.
- AisReg: Client named in c has account named in a:
  $\sigma(\mathsf{c}) \in \mathbf{dom}\rho \wedge \sigma(\sigma(\mathsf{a})) \in \rho(\sigma(\mathsf{c}))$.
- AhasP: Account named in a has at least the balance given in p:
  $\alpha(\sigma(\mathsf{a})) \geq \sigma(\mathsf{p})$.
- CisMReg: Client named in c has a mortgage:
  $\sigma(\mathsf{c}) \in \mathbf{dom}\ \mu$.
- MisReg: Client named in c has mortgage named in m:
  $\sigma(\mathsf{c}) \in \mathbf{dom}\mu \wedge \sigma(\mathsf{m}) \in \mu(\sigma(\mathsf{c}))$.
- Mhas0: Mortgage named in m is paid up fully:
  $\ell(\sigma(\mathsf{m}))=0$.

Then it should be easier to "decipher" the logics:

```
Int_Clause(mkChk(cex))(σ)(tf)(ρ,α,μ,ℓ) ≡
    (σ,case cex of
          CisAReg(c) → σ(c) ∈ dom ρ,
          AisReg(a,c) → σ(c) ∈ domρ∧σ(σ(a)) ∈ρ(σ(c)),
          AhasP(a,p) → α(σ(a))≥σ(p),
          CisMReg(c) → σ(c) ∈ dom μ,
          MisReg(m,c) → σ(c) ∈ domμ∧σ(m) ∈μ(σ(c)),
          Mhas0(m) → ℓ(σ(m))=0
       end,(ρ,α,μ,ℓ))
```

There are a number of ways of adding amounts, designated in p, to accounts and mortgages:

- mkAN(a): to account named in a
- mkMN(m): to mortgage named in m
- bank_i: to the bank's own interest account
- bank_c: to the bank's own capital account

Int_Clause(mkA(p,t))(σ)(tf)(ρ,α,μ,ℓ) $\equiv$
  case t **of**
    mkAN(a) $\rightarrow$ (σ,**true**,(ρ,α†[a$\mapsto\alpha$(σ(a))+σ(p)],μ,ℓ))
    mkMN(m) $\rightarrow$ (σ,**true**,(ρ,α,μ,ℓ†[σ(m)$\mapsto\ell$(σ(m))+σ(p)]))
    bank_i $\rightarrow$ (σ,**true**,(ρ,α†[$a_i\mapsto\alpha(a_i)+\sigma$(p)],$\mu$,$\ell$))
    bank_c $\rightarrow$ (σ,**true**,(ρ,α†[$a_\ell\mapsto\alpha(a_\ell)+\sigma$(p)],$\mu$,$\ell$))
  end

The case, as above for adding, also holds for subtraction.

Int_Clause(mkS(p,t))(σ)(tf)(ρ,α,μ,ℓ) $\equiv$
  case t **of**
    mkAN(a) $\rightarrow$ (σ,**true**,(ρ,α†[σ(a)$\mapsto\alpha$(σ(a))$-\sigma$(p)],μ,ℓ))
    mkMN(m) $\rightarrow$ (σ,**true**,(ρ,α,μ,ℓ†[σ(m)$\mapsto\ell$(σ(m))$-\sigma$(p)]))
    bank_i $\rightarrow$ (σ,**true**,(ρ,α†[$a_i\mapsto\alpha(a_i)-\sigma$(p)],$\mu$,$\ell$))
    bank_c $\rightarrow$ (σ,**true**,(ρ,α†[$a_\ell\mapsto\alpha(a_\ell)-\sigma$(p)],$\mu$,$\ell$))
  end

And it holds as for subtraction, but subtracting two amounts, of values designated in p and i.

Int_Clause(mk2S(p,i,t))(σ)(tf)(ρ,α,μ,ℓ) $\equiv$
  let pi = σ(p)$-\sigma$(i) **in**
  case t **of**
    mkAN(a) $\rightarrow$ (σ,**true**,(ρ,α†[σ(a)$\mapsto\alpha$(σ(a))$-$pi],μ,ℓ))
    mkMN(m) $\rightarrow$ (σ,**true**,(ρ,α,μ,ℓ†[σ(m)$\mapsto\ell$(σ(m))$-$pi]))
    bank_i $\rightarrow$ (σ,**true**,(ρ,α†[$a_i\mapsto\alpha(a_i)-$pi],μ,ℓ))
    bank_c $\rightarrow$ (σ,**true**,(ρ,α†[$a_\ell\mapsto\alpha(a_\ell)-$pi],$\mu$,$\ell$))
  end end

To delete an account is to remove it from both the account register and the accounts.

Int_Clause(mkDA(c,a))(σ)(tf)(ρ,α,μ,ℓ) $\equiv$
  ($\sigma\backslash${a},**true**,(ρ†[σ(c)$\mapsto\alpha$(σ(c))$\backslash\{\sigma$(a)}],$\alpha\backslash\{\sigma$(a)},μ,))

Similarly, to delete a mortgage is to remove it from both the mortgage register and the mortgages.

```
Int_Clause(mkDM(c,m))(σ)(tf)(ρ,α,μ,ℓ) ≡
    (σ\{m},true,(ρ,α,μ†σ(c)[↦μ(σ(c))\{σ(m)}],ℓ\{β(m)}))
```

To compute a special function requires a place, i, to put, i.e., to store, the resulting, the yielded, value. It also requires the name, fn, of the function, and the actual argument list, aal, i.e., the list of values to be applied to the named function, fct. As an example we illustrate the "built-in" function of computing the interest on a loan, a mortgage.

```
Int_Clause(mkCP(i,fn,aal))(σ)(tf)(ρ,α,μ,ℓ) ≡
    let fct = σ(fn) in
    let val = case fn of
                  "interest" →
                      let ⟨m,d⟩ = aal in fct(⟨μ(σ(m)),d?⟩) end
                  ... → ...
              end in
    (σ†[σ(i)↦val],true,(ρ,α,μ,ℓ)) end end
```

This ends the last stage of the development of a script language. ∎

Example A.12 *Script Reengineering:* We refer to Examples A.8–A.11. They illustrated the description of a perceived bank script language. One that was used, for example, to explain to bank clients how demand/deposit and mortgage accounts, and hence loans, "worked".

With the given set of "schematised" and "user-friendly" script commands, such as they were identified in the referenced examples, only some banking transactions can be described. Some obvious ones cannot, for example, *merge two mortgage accounts, transfer money between accounts in two different banks, pay monthly and quarterly credit card bills, send and receive funds from stockbrokers*, etc.

A reengineering is therefore called for, one that is really first to be done in the basic business processes of a bank offering these services to its customers. We leave the rest as an exercise, cf. Exercise A.1. ∎

A.2.5 A Student Exercise

Exercise A.1 *Banking Script Language.* We refer to Example A.12 — and all of the examples referenced initially in Example A.12. Redefine, as suggested there, the banking script language to allow such transactions as: (i) *merge two mortgage accounts*, (ii) *transfer money between accounts in two different banks*, (iii) *pay monthly and quarterly credit card bills*, (iv) *send and receive funds from stockbrokers*, etc.

A.3 Financial Service Industry

- We model only two of the players in the financial services market:
  - ⋆ Banks and
  - ⋆ securities (typically stock and bond) exchanges.
- Also: We do not model their interaction, that is, transfers of securities between banks and stock exchanges.
- Such as was done in earlier examples.
- The models presented now lend themselves to such extensions rather easily.

A.3.1 Banking

Domain Analysis

We start out with a major analysis cum domain narrative!

Account Analysis:

We choose a simple, ordinary person oriented banking domain.

(This is in contrast to for example an import/export, or an investment, or a portfolio bank domain. And it is in contrast to the many other perspectives that one could model: securities and portfolio management, foreign currency trading, customer development, etc.)

On one hand there are the s, k:K, and on the other hand there is the . (We initially assume that the is perceived, by the s, as a single, "monolithic thing" — although it may have a geographically widely distributed net of branch offices.) Each person or other legal entity, who is a , may have several s.

Each has an identity, c:C, and is an otherwise complex quantity, a:A, whose properties will be unfolded slowly. A , k:K, may have more than one , but has at least one — otherwise there would be no need to talk about "a " (but perhaps about a prospective). (So the ing domain includes all the accounts and the .) Two or more may share .

Account Types:

have : Some are ; and some are ; yet other are (or) ; salary/earnings , etc. With each we associate a which is set up when the is first established.

Contract Rules & Regulations:

The establishes that determine several properties.

Example are *The account* (in question)(i) s y% , and (ii) has a of ℓ currency units. (iii) When the is between 0 and the (negative) , then the owed the is j%; (iv) s carry from the day after ; (v) on a is

otherwise calculated as follows: ...,[6] (vi) the client is sent a of s every d days (typically *every month, or every quarter*, or for every d transactions, or some such arrangement), ... *the lists, in chronological order, all as well as initiated s involving this and as from (ie. since) the last time a was issued.* (vii) s for handling certain (or any) s could be as follows: ment e, s, i, ing (overdraw) o_ℓ, t, etc. The , also called the s (of the), for any specific of , may differ from to , and may change over time.

The are set up when the is ed. Some may be changed by the , and some by the — giving to the . ing an , its s and an are examples of joint / or just s.

Transactions:

Depending on the a number of different kinds of s can be issued "against", ie. concerning (primarily) a specifically named, c:C, , a:A.

- s:
  can (i) monies into and (ii) monies from a (rather freely — and the may stipulate so); (iii) can money in a (and the contract may stipulate minimum monthly savings); (iv) clients can money from their (and the will undoubtedly state frequency and size limits on such s).
  (v) may obtain a large loan whereafter one regularly, as stipulated in the , (vi) repays the by ing — for example — three kinds of monies: (vi.1) on the (these are monies that go to a of the), (vi.2) on the (this is a quantity which is deducted from the s') and (vi.3) s (again monies that go to some [other]). (vii) And a may produce a () of a .
  A is a list of summaries of s. The listed s give the and of the s, its nature[7], the amounts involved (and, in cases according to which they were calculated), the resulting (current) , etc., etc. ! A also lists the "executed against" the but by the . See next.
- Bank Transactions:
  The bank regularly performs s "against" several accounts: (viii) calculation of s due the s (say on demand/deposit and), and (ix) calculation of s due the (say on n and on loan accounts). The may regularly inform as to the of their : (x) , (xi) s of s (s, , s), (xii) warnings on overdue payments, information on or s (say of salary) into (salary) accounts, etc. (xiii) Finally the may the rules & regulations of s, and (xiv) may transactions on (ie.) an .

Immediate & Deferred Transaction Handling:

When a is issued, say at time t, some of its implications are "immediately", some are red. Examples are: installation of , and s on a is expected to

[6] ...: here follows a detailed (pseudo-algorithmic) explanation on how is calculated.

[7] , , (al from a), ation of , and s on a (ment), between , including salary and other payment deposits as well as s on for example s of other , on credit cards, etc.

immediately lead to the on and s, while a , to be issued by the , namely for a to be issued, say, some period prior to a quarter later, to that (concerning amounts of next s), is deferred. Other s are also red in relation to this example. A red will be if the has not responded — as assumed — to a by providing a . That red will be ed if a proper takes place. The , if eventually , as its time "comes up", will lead to further s as well as of rates, etc. s concerning these s and s, etc., are also contained in the .

Thus we see, on one hand, that the is a serious and complex document. In effect its rule & regulation conditions define a number of named s that are applied when relevant s are handled (executed). These s, in the domain, are handled either manually, semi-automatically or (almost fully) automated. The staff (or, in cases, perhaps even s) who handle the manual parts of these s may and will make mistakes. And the semi or fully automated s may be incorrect !

Summary

We can summarise the analysis as follows:

- Transactions are initiated by:
  - ⋆ Clients:
    - · Establishment and closing of accounts
    - · demand (withdrawal) and deposits of monies
    - · borrowing and repayment of loans
    - · transfer of monies into or out of accounts
    - · request for (instantaneous or regular) statements
    - · *&c.*
  - ⋆ and the bank:
    - · Regular calculation of yield and interest
    - · regular payment of bills
    - · regular issue of statements
    - · reminder of loan repayments
    - · warning on overdue payments
    - · annual account reports
    - · change in (and advice about) account conditions
    - · *&c.*
- Transactions are handled by the bank:
  - ⋆ immediately: certain parts of f.ex. als, s, s, etc.
  - ⋆ overnight:[8] remaining parts of f.ex. above
  - ⋆ deferred: issue of s and preparation for s, of s, s, and s. etc.
  - ⋆ conditionally:[9] issue of s, etc.
- In the domain this handling may be by any combination of human and machine (incl. computer) labour.

[8] We will treat overnight transactions as deferred transactions.
[9] We will treat conditional transactions as deferred transactions.

- **Support technology** is here seen as the various means whereby transactions are processed and their effect recorded.
- Examples of **support technology** are: The paper forms, including (paper) books, used during transaction and kept as records; mechanical, electro-mechanical and electronic, hand-operated calculators; chops (used in authentication on paper forms); typewriters; computers (and hence data communication equipment).

Abstraction of Immediate and Deferred Transaction Processing

We proceed by first giving — again — a rather lengthy analysis, cum narrative, of transaction processing related concepts of a bank.

We have a situation where s are either "immediately" handled, or are red. For the domain we choose to model this seeming "distinction" by obliterating it ! Each is instead red and affixed the time interval when it should be . If a is issued at time t and if parts or all of it is to be handled "immediately" then it is red to the time interval (t, t). There is therefore, as part of the , a of *time interval marked transaction requests.* The (staff, computers, etc.) now is expected to repeatedly, ie. at any time t', inspect the . Any s that remain in the such that t' falls in the interval of requests are then to be handled "immediately". In the model we assume that the handling time is 0, but that requests that are eligible for "immediate" handling are chosen non-deterministically. This models the reality of a domain, but perhaps not a desirable one!

Account Temporality:

Time is a crucial concept in banking: s are calculated over time during which the changes and so do the rates — with no synchronisation between for example these two. Because of that temporality, we shall — in the domain model — "stack" all s (initialisations and updates) to the ual s () such that all such s are remembered and with a time-stamp of their occurrence.

Likewise most other account components will be time-stamped and past component values kept, likewise time-stamped.

Summary:

We shall subsequently repeat and expand on the above while making it more precise and while also providing an emerging formal specification of a domain model.

Before we do so we will, however, summarise the above:

- There are s, k:K, and s may have more than one , and s are identified, c:C.
- With each there is a . The lists the , including all the that shall govern the handling of any "against" the .

- are either client initiated such as , , , , s, etc., or are bank initiated such as interest s, s, s, issuance of requested regular s, etc.
- are expected handled within a certain time-interval — which may be "now" or later. For simplicity we treat all as red (till now or later!).
- So there are requests and processing. The latter corresponds to the actual, possibly piecemeal, handling of requests.
- And there are . This term — which is also a computing science and software engineering term — has here a purely banking connotation.
- And there are commands. The actual handling of a is decribed by means of a program in a hypothetical , `BaPL`. Programs in `BaPL` are commands, and commands may be composite and consist of other commands !
- So please keep the five concepts separate: Transaction requests, transaction processing, statements, routines and commands. Their relations are simple: Transaction requests lead to the eventual execution of one or more routines, each as described by means of commands. The excution of transaction request related routines constitute the transaction (ie. the transaction processing). One kind of transaction request may be that of "printing" a client account statement.

We have given a normative overview of the structure and the logic of some base operations of typical banks.

That is: We have mentioned a number of important bank state components and hinted at their inter-relation. But we have not detailed what actions actually occur when a transaction is "executed": what specific arithmetic is performed on account balances, what specific logic applies to conditional actions on account components, etc.

We shy away from this as it is normally not a normative property, but highly specialised: differs from bank to bank, from account to account, etc. These arithmetics and logics are properties of instantiated banks and accounts. With repect to the latter the arithmetic and logic transpire from the bank rules & regulations.

Modelling

The essence of the above analysis is the notion of deferred action. The consequence of this modelling decision is twofold: (i) First we are able to separate the possibly human (inter)action between clients and tellers, or between clients and 'automatic teller machines' (ATMs) from the actual "backroom" (action) processing; (ii) and then we are able to abstract this latter considerably wrt. for example the not so abstract model we shall later give of bank accounts.

There are client, k:K, account identifiers, c:C, accounts a:A, and transactions, tr:Trans. And there is the repository r:R. The repository contains for different time intervals (t,t′) [where t may be equal to t′] and for different client account identifiers zero, one or more "deferred" transactions (to be executed).

Each transaction is modelled as a pair: a transaction routine name, rn:Rn, and a list of arguments (values) to be processed by the routine.

We assume that (for example) client accounts, a:A, contain routine descriptions (scripts).

type
  K, C, A
  B = ({} $\overrightarrow{m}$ (K $\overrightarrow{m}$ C-**set**))
    $\bigcup$ ({} $\overrightarrow{m}$ (C $\overrightarrow{m}$ A))
    $\bigcup$ ({} $\overrightarrow{m}$ R)
    $\bigcup$ ({conditions} $\overrightarrow{m}$ (C $\overrightarrow{m}$ (Rn $\overrightarrow{m}$ Routine-**set**)))
  R = (T × T) $\overrightarrow{m}$ Jobs
  Jobs = C $\overrightarrow{m}$ Trans-**set**
  Trans == mk_Trans(rn:Rn,vl:VAL*)
  Routine = /* **BaPL** Program */

Client Transactions:

A client may issue a transaction, tr:Trans, w.r.t. to an account, c:C, and at time t:T. Honouring that request for a transaction the banking system defers the transaction by repositing it for execution in the (instantaneous) time interval (t,t). The client may already, for some reason or another, have a set of such reposited transactions.

Insert One Transaction:

value
  client: C × Trans → T → B → B
  client(c,trans)(t)(b) ≡ insert([(t,t) ↦ [c ↦ {trans}]])(b)

We can safely assume that no two identical:

  [(t,t) ↦ [c ↦ tsk]]

can be submitted to the bank since time passes for every one client or bank transaction.

Insertion of Arbitrary Number of Transactions:

You may wish to skip the next two function definitions. They show that one can indeed express the insertion and merge of deferred transactions into the bank repository.

value
  insert: R $\xrightarrow{\sim}$ B $\xrightarrow{\sim}$ B
  insert(r)(β) ≡

```
    if r = [ ]
      then beta
      else
        let r′ = β(), (t,t′):(T×T) • (t,t′) ∈ dom r in
        let r″ =
          if (t,t′) ∈ dom r′
            then
              let bjobs = r′(t,t′), cjobs = r(t,t′) in
              r′ † [ (t,t′) ↦ merge(bjobs,cjobs) ] end
            else
              r′ ∪ [ (t,t′) ↦ cjobs ] end
        insert(r \ {(t,t′)})(β † [↦ r′])
  end end end
```

Merge of Jobs: Client Transactions:

```
value
  merge: Jobs × Jobs ~→ Jobs
  merge(bjobs,cjobs) ≡
    if cjobs=[ ]
      then bjobs
      else
        let c:C • c ∈ dom cjobs in
        let jobs =
          if c ∈ dom bjobs
            then [ c ↦ cjobs(c) ∪ bjobs(c) ]
            else [ c ↦ cjobs(c) ] end in
        merge(bjobs † jobs,cjobs \ {c}) end end
    end
```

The Banking Cycle:

The bank at any time t:T investigates whether a transaction is ("defer") scheduled [ie. "deferred" for handling] at, or around, that time. If not, nothing happens — and the bank is expected to repeat this investigation at the next time click ! If there is a transaction, tr:Trans, then it is fetched from the repository together with the time interval (t',t'') for which it was scheduled and the identity, c:C, of the client account. (c may be the identity of an account of the bank itself!)

```
value
  bank: B → T ~→ B
  bank(β)(t) ≡
    if β() = [ ] then β else
```

```
    if is_ready_Task(β)(t)
      then
        let (((t′,t″),c,mk_Task(rn,al)),β′) = sel_rmv_Task(β)(t) in
        let rout:Routine • rout ∈ ((β′(conditions))(c))(rn) in
        let (β′,r) = E(c,rout)(al)(t,t′,t″)(β′) in
        bank(insert(r)(β″))(t) end end end
      else
        let t‴:T • t‴ = t + Δτ in bank(β)(t‴) end
    end end
```

E: C × Routine $\xrightarrow{\sim}$ VAL* $\xrightarrow{\sim}$ (T×T×T) $\xrightarrow{\sim}$ B $\xrightarrow{\sim}$ B × R

The expression $\Delta\tau$ yields a minimal time step value.

Auxiliary Repository Inspection Functions:

```
value
  is_ready_Task: B → T ~→ Bool
  is_ready_Task(β)(t) ≡
    ∃ (t′,t″):T×T • (t′,t″) ∈ dom β() ∧ t′ ≤ t ∧ t ≤ t″

  sel_rmv_Task: B → T ~→ (((T×T) × C × Task) × B)
  sel_rmv_Task(β)(t) ≡
    let r = β() in
    let (t′,t″):T×T •  (t′,t″) ∈ dom r ∧ t′ ≤ t ∧ t ≤ t″ in
    let jobs = r(t′,t″) in
    let c:C • c ∈ dom jobs in
    let tasks = jobs(c) in
    let task:Task • task ∈ tasks in
    let jobs′ = if tasks\{task} = {}
          then jobs\{c} else jobs † [ c ↦ tasks\{task} ] end in
    let r′ = if jobs′ = [ ]
          then r\{(t′,t″)} else r † [ (t′,t″) ↦ jobs′ ] end in
    (((t′,t″),c,task),β † [↦ r′])
    end end end end end end end end
```

- Performing the execution as prescribed by the transaction, tr:Trans, besides a changed bank — except for "new" deferred transactions — results in zero, one or more new deferred transactions, trs.
- These are inserted in the bank repository.
- And the bank is expected to "re-cycle": ie. to search for, ie. select new, pending transactions "at that time"!
- That is: the bank is expected to handle, ie. execute all its deferred transactions before advancing the clock!

Merging the Client and the Bank Cycles:

- On one hand clients keep coming and going: submitting transactions at irregular, unpredictable times.
- On the other hand the bank keeps inspecting its repository for "outstanding" tasks.
- These two "processes" intertwine.
- The client_step function extends the client function.
- The bank_step function "rewrites" the (former) bank function:

```
value
  cycle: B ~→ B
  cycle(β) ≡ let β′ = client_step(β) ⫿ bank_step(β) in cycle(β′) end

  client_step: B ~→ B
  client_step(β) ≡
    let (c,tr) = client_ch?, t = clock_ch? in client(c,tr)(t)(β) end

  bank_step: B ~→ B
  bank(β) ≡
    if β() = [ ]
      then β
      else
        let t = clock_ch? in
        if is_ready_Task(β)(t)
          then
            let (((t′,t″),c,mk_Task(rn,al)),β′) = sel_rmv_Task(β)(t) in
            let rout:Routine • rout ∈ ((β′(conditions))(c))(rn) in
            let (β′,r) = E(c,rout)(al)(t,t′,t″)(β′) in
            insert(r)(β″) end end end
          else β end
    end end
```

- The cycle function (internal choice) non–deterministically chooses between either a client step or a bank step.
- The client step **input**s a transaction at time t from some client.
- This is modelled by a channel communication.
- Both the client and the bank steps "gets to know what time it is" from the system clock.

A.4 Securities Trading

A.4.1 "What is a Securities Industry ?"

In line with our approach, we again ask a question — see the section title line just above! And we give a synopsis answer.

Synopsis

The securities industry consists of:

- the following components:
  - ⋆ one or more stock exchanges,
  - ⋆ one or more commodities exchanges,
  - ⋆ *&c.*
  - ⋆ one or more brokers,
  - ⋆ one or more traders,
  - ⋆ *&c.*
  - ⋆ and associated regulatory agencies,
- together with all their:
  - ⋆ stake-holders,
  - ⋆ states,
  - ⋆ events that may and do occur,
  - ⋆ actions (operations) that change or predicates that inspect these states,
  - ⋆ intra and inter behaviours and
  - ⋆ properties of the above!

A Stock Exchange "Grand" State

- Domain-wise we will just model a simple stock exchange — and from that model "derive" domain models of simple brokers and traders.
- Technically we model the "grand" state space as a sort, and name a few additional sorts whose values are observable in states.
- To help your intuition we "suggest" some concrete types for all sorts, but they are only suggestions.

type
S, O, T, Q, P, R
SE = (Buy × Sell) × ClRm
Buy, Sell = S $\xrightarrow[m]{}$ Ofrs
Ofrs = O $\xrightarrow[m]{}$ Ofr ~~!~~
Ofr = (T×T) $\xrightarrow[m]{}$ (Q × (lo:P×hi:P) × ...)
ClRm = O $\xrightarrow[m]{}$ Clrd | Rmvd
Clrd = S × P × T × Ofrs × Ofrs
Rmvd = S × T × O × Ofr
Market = T → SE

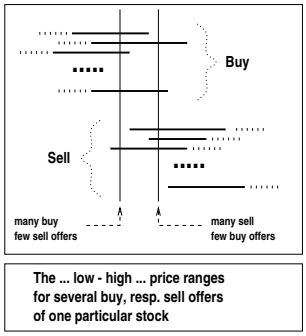

Fig. A.5. A "Snapshot" Stock Exchange View of Current Offers of a Single Stock

- The main (state) components of a stock exchange — reflecting, as it were, 'the market' — are the current state of stocks offered
  - ⋆ ie. placed)
  - ⋆ for buying Buy,
  - ⋆ respectively selling Sell,
  - ⋆ and a summary of those cleared (that is bought & sold)
  - ⋆ and those removed.
- The placement of an offer of a stock, s:S, results, r:R, in the offer being marked by a unique offer identification, o:O.
- The offer otherwise is associated with information about the time interval, (bt,et):T×T, during which the offer is valid — an offer that has not been cleared during that time interval is to be removed from buy or sell status, or it can be withdrawn by the placing broker — the quantity offered and the low to high price range of the offer. (There may be other information (...).)

Observers and State Structure

- Having defined abstract types (ie. sorts) we must now define a number of observers. Which one we define we find out, successively, as we later sketch signatures of functions as well as sketching their definition.
- As we do the latter we discover that it would "come in handy" if one had "such and such an observer"!
- Given the suggested concrete types for the correspondingly named abstract ones we can also postulate any larger number of observers — most of which it turns out we will (rather: up to this moment has) not had a need for!

value
obs_Buy: SE → Buy, obs_Sell: SE → Sell,
obs_ClRm: SE → ClRm
obs_Ss: (Buy|Sell) → S-**set**

obs_Ofrs: S × (Buy|Sell) $\xrightarrow{\sim}$ Ofrs
obs_Q: Ofr → Q
obs_Qs: Ofrs → Q
obs_lohi: Ofr → P×P
obs_TT: Ofr → T×T
obs_O: R → O
obs_OK: R → {ok|nok}

Main State Generator Signatures

The following three generators seems to be the major ones:

- place: expresses the placement of either a buy or a sell offer, by a broker for a quantity of stocks to be bought or sold at some price suggested by some guiding price interval (lo,hi), such that the offer is valid in some time (bt,et) interval.[10]

value
place: {buy|sell}×B×Q×S×(lo:P×hi:P)×(bt:T×et:T)×... → SE
$\xrightarrow{\sim}$ SE × R

- wthdrw: expresses the withdrawal of an offer o:O (by a broker who has the offer identification).
- next: expresses a state transition — afforded just by inspecting the state and effecting either of two kinds of state changes or none!

value
wthdrw: O × T → SE $\xrightarrow{\sim}$ SE × R
next: T × SE → SE

A Next State Function

- At any time, but time is a "hidden state" component,
- the stock exchange either clears (fclr) a batch of stocks —
- if some can be cleared (pclr) —
- or removes (frmv) elapsed (prmv) offers,
- or does nothing!

[10] We shall [probably] understand the buy (lo,hi) interval as indicating: buy as low as possible, do not buy at a pricer higher than hi, but you may buy when it is lo or as soon after it goes below lo. Similarly for sell (lo,hi): sell as high as possible, do not sell at a pricer lower than lo, but you may sell when it is hi or as soon after it goes above hi; the place action is expected to return a response which includes giving a unique offer identification o:O.

```
value
  next: T × SE → SE
  next(t,se) ≡
    if pclr(t,se)
      then fclr(t,se)
      else
    if prmv(t,se)
      then frmv(t,se)
      else se
    end end

  pclr: T × SE → Bool, fclr: T × SE → SE
  prm: T × SE → Bool, frm: T × SE → SE
```

Next State Auxiliary Predicates

- A batch (bs,ss) of (buy, sell) offered stocks of one specific kind(s) can be cleared if a price (p) can be arrived at,
- one that satisfies the low to high interval buy, respectively sell criterion —
- and such that the batch quantities of buy, resp. sell offers
- either are equal or their difference is such that the stock exchange is itself willing to place a buy,
- respectively a sell offer for the difference.

```
value
  pclr(t,se) ≡ ∃ s:S,ss:Ofrs,bs:Ofrs,p:P • aplcr(s,ss,bs,p)(t,se)

  apclr: S×Ofrs×Ofrs×P → T×SE → Bool
  apclr(s,bs,ss,p)(t,se) ≡
    let buy = obs_Buy(se), sell = obs_Sell(se) in
    s ∈ obs_Ss(buy) ∩ obs_Ss(sell)
    ∧ bs ⊆ obs_Ofrs(s,buy) ∧ ss ⊆ obs_Ofrs(s,sell)
    ∧ buysell(p,bs,ss)(t)
    ∧ let (bq,sq) = (obs_Qs(bs),obs_Qs(ss)) in
        acceptable_difference(bq,sq,s,se) end end

  buysell: P×Ofrs×Ofrs → T → Bool
  buysell(p,bs,ss)(t) ≡
    ∀ ofr:Ofr • ofr ∈ bs ⇒
      let (lo,hi) = obs_lohi(ofr) in p ≤ hi end
      let (bt,et) = obs_TT(ofr) in bt ≤ t ≤ et end
  ∧ ∀ ofr:Ofr • ofr ∈ ss ⇒
      let (lo,hi) = obs_lohi(ofr) in p ≥ lo end
      let (bt,et) = obs_TT(ofr) in bt ≤ t ≤ et end
```

Next State Auxiliary Function

- We describe the result of a clearing of buy, respectively sell offered stocks by the properties of the stock exchange before and after the clearing.
- Before the clearing the stock exchange must have suitable batches of buy (bs), respectively sell (ss) offered stocks (of identity s) for which a common price (p) can be negotiated (apclr).
- After the clearing the stock exchange will "be in a different state".
- We choose to characterise here this "different state" buy first expressing that the cleared stocks must be removed as offers (rm_Ofrs).
- If the buy batch contained more stocks for offer than the sell batch then the stock exchange becomes a trader and places a new buy offer in order to make up for the difference.
- Similarly if there were more sell stocks than buy stocks. A
- t the same time the clearing is recorded (updClRm).

```
fclr(t,se) as se′
  pre pclr(t,se)
  post
    let s:S,bs:Ofrs,ss:Ofrs,p:P•apclr(s,ss,bs,p)(t,se) in
    let (bq,sq) = (obs_Qs(bs),obs_Qs(ss)),
        buy = obs_Buy(se), sell = obs_Sell(se) in
    let buy′ = rm_Ofrs(s,bs,buy), sell′ = rm_Ofrs(s,ss,sell) in
    obs_Buy(se′) = if bq > sq
        then updbs(buy′,s,bq−sq,tt_buy(s,bq−sq)(t,se))
        else buy′ end ∧
    obs_Sell(se′) = if bq < sq
        then updss(sell′,s,sq−bq,tt_sell(s,bq−sq)(t,se))
        else sell′ end ∧
    let clrm = obs_ClRm(se) in
    obs_ClRm(se′) = updClRm(s,p,t,bs,ss,clrm) end
    end end end
```

Many comments can be attached to the above predicate for clearability, respectively the clearing function:

- First we must recall that we are trying to model the domain.
- That is: we can not present too concrete a model of stock exchanges, neither what concerns its components, nor what concerns its actions.
- The condition, ie. the predicate for clearable batches of buy and sell stocks must necessarily be loosely defined — as many such batches can be found, and as the "final clinch", ie. the selection of exactly which batches are cleared and their (common) prices is a matter for "negotiation on the floor".
- We express this looseness in several ways:

  - ⋆ the batches are any subsets of those which could be cleared such that any possible difference in their two batch quantites is acceptable for the stock exchange itself to take the risk of obtaining a now guaranteed price (and if not, to take the loss — or profit!);
  - ⋆ the batch price should satisfy the lower/upper bound (buysell) criterion, and it is again loosely specified;
  - ⋆ and finally: Which stock (s) is selected, and that only exactly one stock is selected, again expresses some looseness, but does not prevent another stock (s≠s′) from being selected in a next "transition".
- There is no guarantee that the stock s buy and sell batches bs and ss and at the price p for which the clearable condition pclr holds, is also exactly the ones chosen — by apclr — for clearing (fclr), but that only could be said to reflect the "fickleness" of the "market"!
- Time was not a parameter in the clearing part of the next function.
- It is assumed that whatever the time is all stocks offered have valid time intervals that "surround" this time, ie. the current time is in their intervals.
- Then we must recall that we are modelling a number of stake-holder perspectives:
  - ⋆ buyers and sellers of stocks,
  - ⋆ their brokers and traders,
  - ⋆ the stock exchange and the securities commission.
- In the present model there is no clear expression, for example in the form of distinct formulas (distinct functions or lines) that reflect the concerns of precisely one subset of these stake-holders as contrasted with other formulas which then reflect the concerns of a therefrom distinct other subset of stake-holders.
- Now we have, at least, some overall "feel" for the domain of a stock exchange.
- We can now rewrite the formulas so as to reflect distinct sets of stake-holder concerns. We presently leave that as an exercise!

Auxiliary Generator Functions

value
  rm_Ofrs: S × Ofrs × (Buy|Sell) $\xrightarrow{\sim}$ (Buy|Sell)
  rm_Ofrs(s,os,busl) **as** busl′
    pre s ∈ obs_Ss(busl) ∧ subseteq(os,obs_Ofrs(s,busl))
    post if s ∈ obs_Ss(busl) **then** ∼∃ ... **else** ... **end**

A.4.2 Discussion

- We have detailed two "narrow" aspects of a financial industry: How banks may choose to process client (and own) transactions, and how securities are traded.

- The former model is chosen so as to reflect all possibilities as they may occur in the domain, ie. in actual situations.
- The latter model is sufficiently "loose" to allow a widest range of interpretations, yet it is also sufficiently precise in that it casts light on key aspects of securities trading.
- In this section of the talk we have not shown, as we did in several other sections, how the two infrastructure stake–holders: Banks and securities traders interact.

B

Methodology

We bring "condensed" excerpts of TripTych methodological issues.

B.1 On Software Development Processes

B.1.1 Processes, Process Specifications and Process Models

By a process we mean a set of related sequences (i.e., traces) where each sequence (trace) in the set is an ordered list of actions and events. Actions change a state. Events are like process inputs or outputs. A sequence, p_i, of actions, a_{i_k}, and events, e_{i_ℓ}, may relate to another such sequence, p_j, by sharing an event, e, in the form of the shared event being identical to one of the events, e_{i_m} (i.e., $e_{i_m} \equiv e$), in p_i and one of the events, e_{j_n} (i.e., $e_{j_n} \equiv e$), in p_j such that this event designates the communication between the two processes, that is, their synchronisation and the simultaneous exchange of a resource (or a possibly empty set of resources) between them. (Simple "assignment" actions may then bind these resources to appropriate names of the input process.) A simple process is just a single sequence whose events, if any, communicates with a further undefined environment. A single process is either a simple process or is a single sequence whose events communicates with other processes. Thus processes are, in general, composed from several processes.

Operations management is thus about the detailed monitoring and control of processes: development processes, marketing processes, sales processes, service processes, training processes, etcetera.

In order to manage in a meaningful way, including in a manner where 'management' can itself be monitored and controlled, that is, be evaluated and improved, the managed processes must be well understood. We take that as meaning that there must be a reasonably precise specification, that is, a model of the processes to be managed.

By a process specification we mean a syntactic entity: some text and/or diagram(s) that name and specify the actions to be performed and the events that may relate two (or more) processes, including a naming and specification of the resources being, or to be communicated.

By a process model we mean a semantic entity: the meaning of a process specification — with that meaning being a possibly infinite set of sets of processes (i.e., set of sets of traces).

By a process specification-based software development we mean either one of the process sets denoted by the process specification.

B.1.2 Software Development Process Descriptions

We have argued earlier that software development consists of three phases. As we shall soon see, carrying out these phases each result in quite distinct sets of documents. Figure B.1 shows the three phases as connected by directed DO and REDO labelled edges, i.e., arrows.

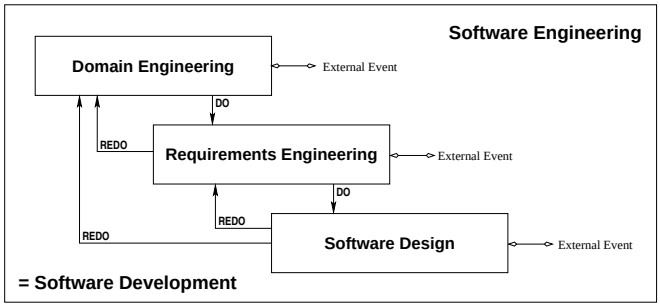

Fig. B.1. The TripTych phases of software development

The boxes (i.e., the phases) denote processes. DO labelled edges infix two boxes, the from and the to box as designated by the edge direction. The DO labelled arrows shall ideally mean that no activities of the processes of the to box can commence before all activities of the processes of the from box have completed. The DO labelled arrows thus designate "internal" events that synchronise the infixed process and which communicate the documents resulting from the 'from box' to the 'to box'. The REDO labelled edges can be said to infix one 'from box' with one or two 'to boxes', now in the reverse order of the DO labelled edge infixed boxes. The REDO labelled arrows (i.e., "internal events") shall ideally mean that all activities of the 'from box' must halt once one such activity requires that earlier work be redone. We do not here detail which 'to box' is selected nor how the development then proceeds.

The "dangling", but bi-directed ("external event") edges designate that box processes require input from or delivers output to an external world (an external process) — typically the human developers.

All directed or bi-directed edges also designate the communication of documents. We do not here detail how these documents are otherwise produced or consumed.

Phases consists of stages and stages of steps. The phases are logically well distinguished. "Boundaries" between stages or steps are pragmatically justified. Next we shall cover the stage and step concepts.

Domain Engineering

The top-left box of Fig. B.1 on the facing page is shown in detail in Fig. B.2. External edges designate the input of document information (including answers to clarifying question) from stakeholders and output of documents and questions to stakeholders including software development management.

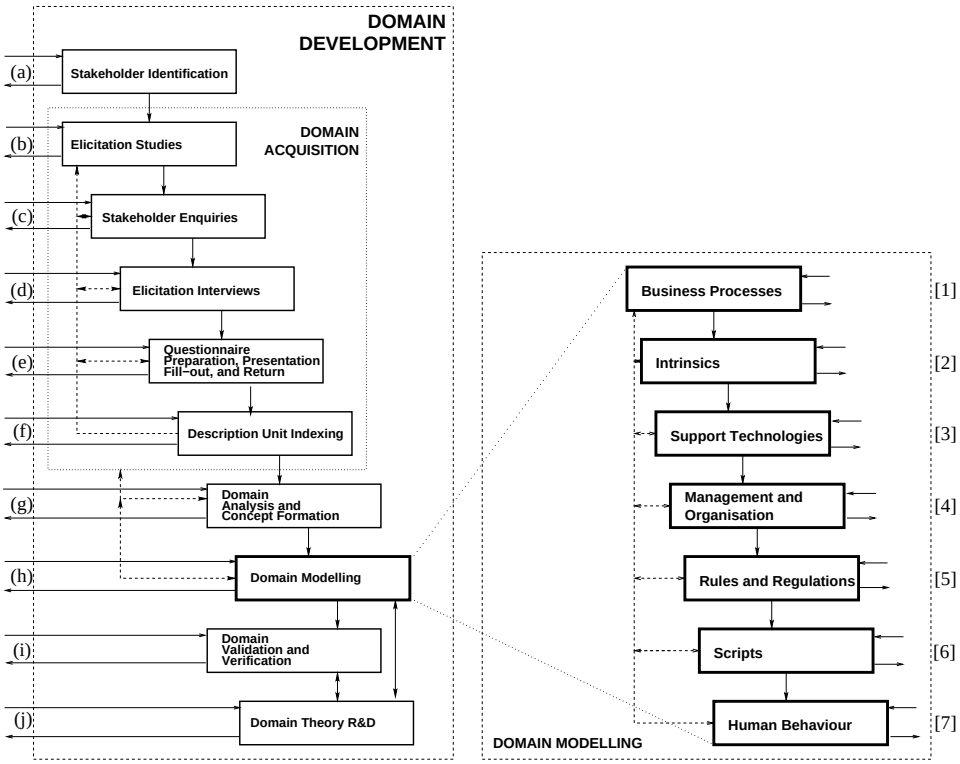

Fig. B.2. Stages and steps of the domain engineering phase

Figure B.2 on the preceding page expresses that domain engineering consists of many stages and that (some of) these stages consists of many steps. We now explain these stages. We cover the left part of Fig. B.2 on the previous page first. (a) First the stakeholders of the domain are identified and arrangements made for future liaison. (b–f) Knowledge about the domain is acquired in five steps; (b) by studying existing literature, the Web, observations in the domain, etc.; (c) contacts with the stakeholders; (d) talks with these; (e) helping them filling out questionnaires, that is, collecting domain description units; (f) and sorting these out. (g) Domain description units are then analysed and preliminary concepts may be formed. (h) Based on this analysis the major work on domain description is carried out. (i) The domain description is then analysed, verified and validated. (j) Finally, where relevant, properties not explicitly formulated in the domain description are established. [1–7] The major stage, (h), of engineering a domain description (that is, of domain modelling, right part of Fig. B.2 on the preceding page) consists of six steps [2–7], each covering a facet of the domain. [1] But first rough sketches are made of all the most pertinent business processes (that is, the entities, the set of functions and events over entities, and the behaviours) of the domain. [2] Based on such sketches the very basics, the entities, functions, events and behaviours that are common to all subsequent facets are described. [3] Then the support technologies of the domain, those which support entities, functions, events and behaviours of the domain are described. [4] The management functions and organisational structures are described. [5] And so are the rules and regulations which (ought) "govern" human behaviour in the domain. [6] Some specific structures (somehow ordered sets) of rules and regulations qualify as scripts, that is, as pseudo-programs, and these are described. [7] Finally the spectrum of possible or actual human behaviours are described: diligent, sloppy, negligent as well as near- or outright criminal behaviours.
Describing these facets usually involves trial-and-error descriptions, that is, iterations between steps. These iterations must be managed. Figure B.3 illustrates the possibilities of "endless thrashing" within just the domain modelling stage.

Requirements Engineering

We consider requirements to be analysable into three categories: domain, interface and machine requirements. Domain requirements are those requirements which can be expressed solely using terms from (or allowed in) the domain description. Machine requirements are those requirements which can be expressed without using terms from the domain description; instead terms are used from the machine (the hardware and the software to be designed). Interface requirements are those requirements which can be expressed using terms both from (or allowed in) the domain description and from (or allowed in) the machine specification. Where domain descriptions express *what there*

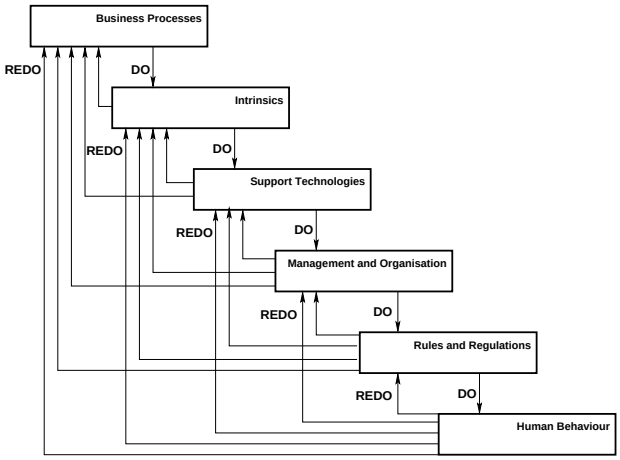

Fig. B.3. The domain modelling stage

is in the domain, the requirements prescriptions express *what there shall or must be in the machine*.

Once a domain description is considered complete work on requirements can commence. Figure B.4 on the next page records the six stages of requirements development. The major stage, requirements modelling, is detailed in Fig. B.5 on the following page. That figure shows domain requirements engineering actions in the top-left quadrant, interface requirements engineering actions in the lower-left quadrant, and machine requirements engineering actions in the right half of the diagram.

One aspect of domain requirements modelling stage (box 4 of Fig. B.4 on the next page) can be summarised as follows: the requirements engineer works with the requirements stakeholders and as follows: First the domain requirements are constructed illustratively by asking the stakeholders to identify (b) which parts of the domain description should be "carried over" into, i.e., projected onto the requirements prescription while (c) possibly instantiating these (now) prescriptions into special cases, and/or (d) making the prescriptions more deterministic, and/or (e) extending the domain description with descriptions of entities, functions, events and behaviours that were not feasible in the domain but are feasible with (the advent of) computing, and (f) finally fitting the emerging domain requirements prescriptions to those of related, other software development projects, if any. Step (a) deals with those requirements, business process re-engineering (BPR), which are not "implemented" as computing, but are relied upon in the correct functioning of the emerging software, that is, which reflect assumptions that must be made about the environment (humans and equipment, including other computing systems). The BPR must also be implemented and adhered to, but by the management and the staff of the user of the required software.

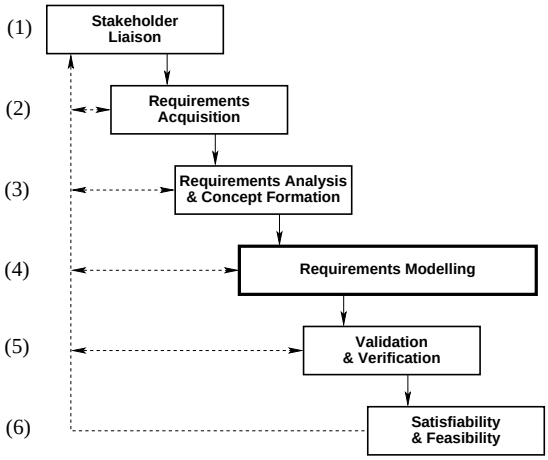

Fig. B.4. The requirements engineering phase. The modelling stage is detailed in Fig. B.5

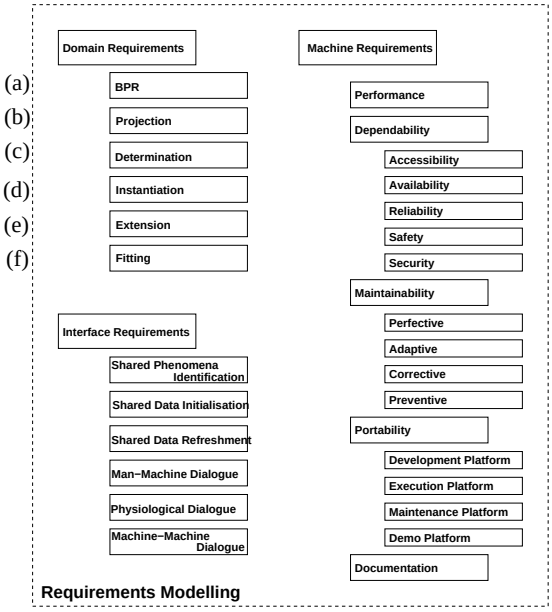

Fig. B.5. The requirements modelling stages

Figure B.5 also shows details of the interface and machine requirements steps — for which we refer the reader to [2, Sects. 19.5–.6].

Figure B.6 on the next page intends to indicate that a number of the machine requirements modelling steps can take place independent of one another, i.e., in parallel.

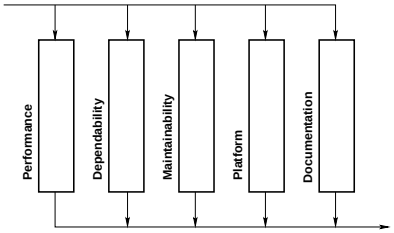

Fig. B.6. The machine requirements modelling stage: five concurrent steps

Software Design

Once a complete requirements prescription has been achieved one can start the software design. Figure B.7 on the following page shows a generic, summary process description for all three phases of software development while emphasising, in its lower three-quarter, the stages and steps of software design: from a possibly stepwise software architecture design via a stage of stepwise refinement of the software components identified by the architecture design, to the final coding step.
The i "stacks" of c_{ij} component boxes shall indicate that the software components may be stepwise refined ending up with executable code k_i. The component part of Fig. B.7 on the next page is rather idealised. One usually experiences that components can be shared across the software design, thus the component part of Fig. B.7 on the following page should be shown more realistically as a lattice of refinement steps.

B.1.3 Documents

Work within each step, stage and phase results in documents. Some are necessarily informal, others can both be formulated informally and formally. To each phase we can therefore attach a number of documents. Section B.2[1] give an overview of these phase documents. Bearing in mind the span and wealth of software related documents we can almost say: *"All we do, in software development, is writing documents — and a few can serve as the basis for computations by machines."*.

[1] Sects. B.2.1–B.2.3

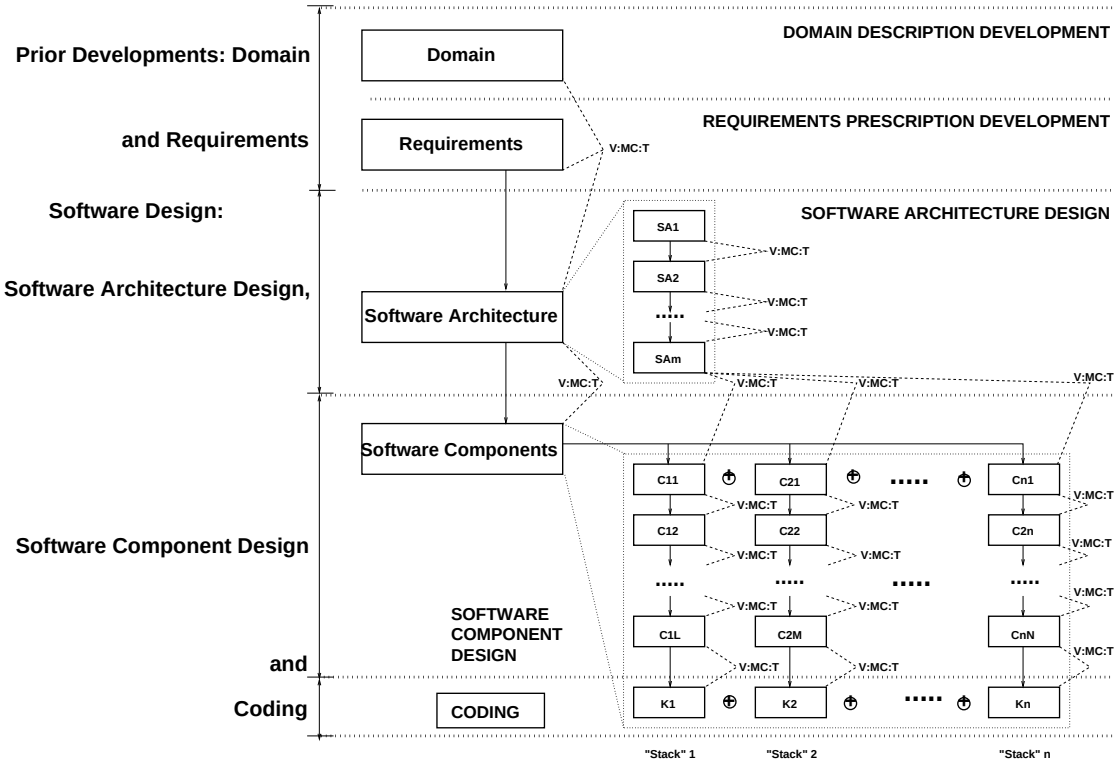

Fig. B.7. The software design phase

B.2 Software Development Documents

There are three kinds of documents: informative (Items 1. in the documents listings of Sects. B.2.1–B.2.3.), specificational (Items 2. in the documents listings of Sects. B.2.1–B.2.3.) and analytic (Items 3. in the documents listings of Sects. B.2.1–B.2.3.). Informative documents view software development projects as values. Analytic documents view specificational (description, prescription and specification) documents as values.

B.2.1 Domain Engineering Documents

We refer to Fig. B.2 on page 173.

1. Information
   (a) Name, Place and Date
   (b) Partners
   (c) Current Situation
   (d) Needs and Ideas
   (e) Concepts and Facilities
   (f) Scope and Span
   (g) Assumptions and Dependencies
   (h) Implicit/Derivative Goals
   (i) Synopsis

(j) Standards Compliance
(k) Contracts
(l) The Teams
  i. Management
  ii. Developers
  iii. Client Staff
  iv. Consultants
(m) Plans
  i. Project Graph
  ii. Budget
  iii. Funding
  iv. Accounts
(n) Management
  i. Assessment
  ii. Improvement
    A. Plans
    B. Actions

2. Descriptions
(a) Stakeholders
(b) The Acquisition Process
  i. Studies
  ii. Interviews
  iii. Questionnaires
  iv. Indexed Description Units
(c) Terminology
(d) Business Processes
(e) Facets:
  i. Intrinsics
  ii. Support Technologies
  iii. Management and Organisation
  iv. Rules and Regulations
  v. Scripts
  vi. Human Behaviour
(f) Consolidated Description

3. Analyses
(a) Domain Analysis and Concept Formation
  i. Inconsistencies
  ii. Conflicts
  iii. Incompleteness
  iv. Resolutions
(b) Domain Validation
  i. Stakeholder Walk-throughs
  ii. Resolutions
(c) Domain Verification
  i. Theorems and Proofs
  ii. Model Checking
  iii. Test Cases and Tests
(d) (Towards a) Domain Theory

B.2.2 Requirements Engineering Documents

We refer to Figs. B.4 and B.5 on page 176.

1. Information
(a) Name, Place and Date
(b) Partners
(c) Current Situation
(d) Needs and Ideas (Eurekas, I)
(e) Concepts & Facilities (Eurekas, II)
(f) Scope & Span
(g) Assumptions & Dependencies
(h) Implicit/Derivative Goals
(i) Synopsis (Eurekas, III)
(j) Standards Compliance
(k) Contracts, with Design Brief
(l) The Teams
  i. Management
  ii. Developers
  iii. Client Staff
  iv. Consultants
(m) Plans
  i. Project Graph
  ii. Budget
  iii. Funding
  iv. Accounts
(n) Management
  i. Assessment
  ii. Improvement
    A. Plans
    B. Actions

2. Prescriptions
(a) Stakeholders
(b) The Acquisition Process
  i. Studies
  ii. Interviews
  iii. Questionnaires
  iv. Indexed Description Units
(c) Rough Sketches (Eurekas, IV)
(d) Terminology
(e) Facets:

i. Business Process Re-engineering
   - Sanctity of the Intrinsics
   - Support Technology
   - Management and Organisation
   - Rules and Regulation
   - Human Behaviour
   - Scripting

ii. Domain Requirements
   - Projection
   - Determination
   - Instantiation
   - Extension
   - Fitting

iii. Interface Requirements
   - Shared Phenomena and Concept Identification
   - Shared Data Initialisation
   - Shared Data Refreshment
   - Man-Machine Dialogue
   - Physiological Interface
   - Machine-Machine Dialogue

iv. Machine Requirements
   - Performance
     - ⋆ Storage
     - ⋆ Time
     - ⋆ Software Size
   - Dependability
     - ⋆ Accessibility
     - ⋆ Availability
     - ⋆ Reliability
     - ⋆ Robustness
     - ⋆ Safety
     - ⋆ Security
   - Maintenance
     - ⋆ Adaptive
     - ⋆ Corrective
     - ⋆ Perfective
     - ⋆ Preventive
   - Platform
     - ⋆ Development Platform
     - ⋆ Demonstration Platform
     - ⋆ Execution Platform
     - ⋆ Maintenance Platform
   - Documentation Requirements
   - Other Requirements

v. Full Reqs. Facets Doc.

3. Analyses
   (a) Requirements Analysis and Concept Formation
      i. Inconsistencies
      ii. Conflicts
      iii. Incompleteness
      iv. Resolutions
   (b) Requirements Validation
      i. Stakeholder Walk-through and Reports
      ii. Resolutions
   (c) Requirements Verification
      i. Theorem Proofs
      ii. Model Checking
      iii. Test Cases and Tests
   (d) Requirements Theory
   (e) Satisfaction and Feasibility Studies
      i. Satisfaction: Correctness, unambiguity, completeness, consistency, stability, verifiability, modifiability, traceability
      ii. Feasibility: Technical, economic, BPR

B.2.3 Software Design Engineering Documents

We refer to Fig. B.7 on page 178.

1. Information
   (a) Name, Place and Date
   (b) Partners
   (c) Current Situation
   (d) Needs and Ideas
   (e) Concepts and Facilities and Facilities
   (f) Scope and Span
   (g) Assumptions and Dependencies
   (h) Implicit/Derivative Goals
   (i) Synopsis
   (j) Standards Compliance
   (k) Contracts
   (l) The Teams
      i. Management,
      ii. Developers,
      iii. Consultants
   (m) Plans
      i. Project Graph
      ii. Budget, Funding, Accounts
   (n) Management
      i. Assessment Plans & Actions
      ii. Improvement Plans & Actions
2. Software Specifications
   (a) Architecture Design ($S_{a_1} \ldots S_{a_n}$)
   (b) Component Design ($S_{c_{1_i}} \ldots S_{c_{n_j}}$)
   (c) Module Design ($S_{m_1} \ldots S_{m_m}$)
   (d) Program Coding ($S_{k_1}, \ldots, S_{k_n}$)
3. Analyses
   (a) Analysis Objectives and Strategies
   (b) Verification (S_{i_p}, $S_i \sqsupseteq_{L_i} S_{i+1}$)
      i. Theorems and Lemmas L_i
      ii. Proof Scripts $\wp_i$
      iii. Proofs Π_i
   (c) Model Checking ($S_i \sqsupseteq P_{i-1}$)
      i. Model Checkers
      ii. Propositions P_i
      iii. Model Checks $\mathcal{M}_i$
   (d) Testing ($S_i \sqsupseteq T_i$)
      i. Manual Testing
         - Manual Tests $M_{S_1} \ldots M_{S_\mu}$
      ii. Computerised Testing
         A. Unit (or Module) Tests C_u
         B. Component Tests C_c
         C. Integration Tests C_i
         D. System Tests $C_s \ldots C_{s_{i_{t_s}}}$
   (e) Evaluation of Adequacy of Analysis

Legend:
S Specification
L Theorem or Lemma
$\wp_i$ Proof Scripts
Π_i Proof Listings
P Proposition
$\mathcal{M}$ Model Check (run, report)
T Test Formulation
M Manual Check Report
C Computerised Check (run, report)
$\sqsupseteq$ "is correct with respect to (wrt.)"
$\sqsupseteq_\ell$ "is correct, modulo ℓ, wrt."

Items 3(b)–3(d) above have been detailed (i–iii, i–iii, i–ii, respectively) more than the corresponding Items 3((c))i–3((c))iii (Page 179, Sect. B.2.1) and Items 3((c))i–3((c))iii (Page 180, Sect. B.2.2). Naturally, also actions implied by these items need be pursued and documented as diligently as for software design.

B.3 RSL: The RAISE Specification Language

We bring an ultra–short introduction to RSL, the predominant formal specification language of these notes.

The survey is, alas, just an overview of the syntax of main aspects of RSL and an overview of some abstraction, i.e., model choices made possible by, for example, RSL.

A Section Table–of–Contents

B.3.1 [1] RSL Types

[1.1] Type Expressions

Let A, B, and C be any type names or type expressions, then:

type
- [1] **Bool**
- [2] **Int**
- [3] **Nat**
- [4] **Real**
- [5] **Char**
- [6] **Text**
- [7] A**-set**
- [8] A**-infset**
- [9] A × B × ... × C
- [10] A^*
- [11] A^ω
- [12] A $\overrightarrow{m}$ B
- [13] A → B
- [14] A $\xrightarrow{\sim}$ B
- [15] (A)
- [16] A | B | ... | C
- [17] mk_id(sel_a:A,...,sel_b:B)
- [18] sel_a:A ... sel_b:B

(save the [i] line numbers) exemply generic type expressions:

1. The Boolean type of truth values **false** and **true.**
2. The integer type on integers ..., -2, -1, 0, 1, 2, ...
3. The natural number type of positive integer values o, 1, 2, ...
4. The real number type of real values, i.e., valuse whose numerals can be written as an integer, followed by a priod (“.”), followed by a natural number (the fraction).
5. The character type of character values "a", "b", ...
6. The text type of character string values "aa", "aaa", ..., "abc", ...

7. The set type of finite set values, see below.
8. The set type of infinite set values.
9. The Cartesian type of Cartesian values, see below.
10. The list type of finite list values, see below.
11. The list type of infinite list values.
12. The map type of finite map values, see below.
13. The function type of total function values, see below.
14. The function type of partial function values.
15. In (A) A is constrained to be
    - either a Cartesian B × C × ... × D, in which case it is identical to type expression kind 9,
    - or not to be the name of a built–in type (cf., 1–6) or of a type, in which case the parentheses serve as simple delimiters, eg.: (A $\overrightarrow{m}$ B), or (A*)**-set**, or (A-**set**)list, or (A|B) $\overrightarrow{m}$ (C|D|(E $\overrightarrow{m}$ F)), etc.
16. The (postulated disjoint) union of types A, B, ..., and C.
17. The record type of mk_id–named record values mk_id(av,...,bv), where av, ..., and bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.
18. The record type of unnamed record values (av,...,bv), where av, ..., and bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

[1.2] Type Definitions

[1.2.1] Subtypes:

The set of elements b of type B which satisfy the predicate $\mathcal{P}$ is a sub–type (of type B):

type
A = {| b:B • $\mathcal{P}$(b) |}

[1.2.2] Sorts or Abstract Types:

Sorts (i.e., abstract types) A, B, ..., C are introduced when specifying:

type
A, B, ..., C

[1.2.3] Concrete Types:

Concrete types are introduced when specifying:

type
A = Type_expr

[1.2.4] BNF Rule Right–hand Sides for Concrete Type Definitions:

```
[1] Type_name =
        Type_expr /∗ without |s or sub−types ∗/
[2] Type_name =
        Type_expr_1 | Type_expr_2 | ... | Type_expr_n
[3] Type_name ==
        mk_id_1(s_a1:Type_name_a1,...,s_ai:Type_name_ai) |
        ... |
        mk_id_n(s_z1:Type_name_z1,...,s_zk:Type_name_zk)
[4] Type_name :: sel_a:Type_name_a  ...  sel_z:Type_name_z
[5] Type_name = {| v:Type_name′ • P(v) |}
```

where a form of [2–3] is provided by the combination:

```
Type_name = A | B | ... | Z
A == mk_id_1(s_a1:A_1,...,s_ai:A_i)
B == mk_id_2(s_b1:B_1,...,s_bj:B_j)
...
Z == mk_id_n(s_z1:Z_1,...,s_zk:Z_k)
```

B.3.2 [2] The RSL Predicate Calculus

[2.1] The RSL Proposional Expressions

Let identifiers (or propositional espressions) a, b, ..., c designate Boolean values. Then:

false, **true**
a, b, ..., c
∼a, a∧b, a∨b, a⇒b, a=b, a≠b

are propositional expressions, all having a Boolean value. ∼, ∧, ∨, ⇒, and = are Boolean connectives (i.e., operators) and "read" as not, and, or, if-then (or implies), equal and not-equal.

[2.2] The RSL Predicate Expressions

[2.2.1] Simple RSL Predicate Expressions

Let identifiers (or propositional espressions) a, b, ..., c designate Boolean values, and let x, y, ..., z (or term expressions) designate other than Boolean values, and let i, j, ..., k designate number values, then:

false, **true**
a, b, ..., c
∼a, a∧b, a∨b, a⇒b, a=b, a≠b
x=y, x≠y,
i<j, i≤j, i≥j, i>j, ...

are simple predicate expressions.

[2.2.2] Quantified RSL *Expressions*

Let X, Y, ..., C be type names or type expressions, and let $\mathcal{P}(x)$, $\mathcal{Q}(y)$ and $\mathcal{R}(z)$ designate predicate expressions in which z, y, and z are free. Then:

∀ x:X • $\mathcal{P}(x)$
∃ y:Y • $\mathcal{Q}(y)$
∃ ! z:Z • $\mathcal{R}(z)$

are quantified expressions, are also predicate expressions, and are "read" as: For all x (values in type X) the predicate $\mathcal{P}(x)$ holds; there exists (at least) one y (value in type Y) such that the predicate $\mathcal{Q}(y)$ holds; and: there exists a unique z (value in type Z) such that the predicate $\mathcal{R}(z)$ holds.

B.3.3 [3] RSL Sets, Cartesians, Lists, and Maps

[3.1] RSL Set, Cartesian, List, and Map Enumerations

[3.1.1] Sets:

Let the below *a*s denote values of type A, then the below designate simple set enumerations:

{{}, {a}, {a_1,a_2,...,a_m}, ...} ∈ A-**set**
{{}, {a}, {a_1,a_2,...,a_m}, ..., {a_1,a_2,...}} ∈ A-**infset**

The expression, last line below, to the right of the ≡, expresses set comprehension.

```
type
  A, B
  P = A → Bool
  Q = A ~→ B
value
  comprehend: A-infset × P × Q → B-infset
  comprehend(s,P,Q) ≡ { Q(a) | a:A • a ∈ s ∧ P(a) }
```


[3.1.2] Cartesians:

type
A, B, ..., C
A × B × ... × C
value
... (e1,e2,...,en) ...

[3.1.3] Lists:

Simple enumerations:

{⟨⟩, ⟨a⟩, ..., ⟨a1,a2,...,am⟩, ...} ∈ A*
{⟨⟩, ⟨a⟩, ..., ⟨a1,a2,...,am⟩, ..., ⟨a1,a2,...,am,... ⟩, ...} ∈ A^ω

⟨ ei .. ej ⟩

The last line above assumes e_i and e_j to be integer valued expressions. It then expresses the set of intergers from the value of e_i to and including the value of e_j. If the latter is smaller than the former then the list is empty.

The last line below expresses list comprehension.

type
A, B, P = A → **Bool**, Q = A $\xrightarrow{\sim}$ B
value
comprehend: A^ω × P × Q $\xrightarrow{\sim}$ B^ω
comprehend(lst,$\mathcal{P}$,$\mathcal{Q}$) ≡
⟨ $\mathcal{Q}$(lst(i)) | i **in** ⟨1..**len** lst⟩ • $\mathcal{P}$(lst(i)) ⟩

[3.1.4] Maps:

Simple map enumerations:

type
A, B
M = A $\xrightarrow[m]{}$ B
value
a,a1,a2,...,a3:A, b,b1,b2,...,b3:B

[ ], [a↦b], ..., [a1↦b1,a2↦b2,...,a3↦b3] ∀ ∈ M

The last line below expresses map comprehension:

```
type
  A, B, C, D
  M = A m~> B
  F = A ~> C
  G = B ~> D
  P = A → Bool
value
  comprehend: M×F×G×P → (C m~> D)
  comprehend(m,F,G,P) ≡
    [ F(a) ↦ G(m(a)) | a:A • a ∈ dom m ∧ P(a) ]
```

[3.2] RSL Set Operations

```
value                                    examples
  ∈: A × A-infset → Bool                   a ∈ {a,b,c}
  ∉: A × A-infset → Bool                   a ∉ {}, a ∉ {b,c}
  ∪: A-infset × A-infset → A-infset        {a,b,c} ∪ {a,b,d,e} = {a,b,c,d,e}
  ∪: (A-infset)-infset → A-infset          ∪{{a},{a,b},{a,d}} = {a,b,d}
  ∩: A-infset × A-infset → A-infset        {a,b,c} ∩ {c,d,e} = {c}
  ∩: (A-infset)-infset → A-infset          ∩{{a},{a,b},{a,d}} = {a}
  \: A-infset × A-infset → A-infset        {a,b,c} \ {c,d} = {a,b}
  ⊂: A-infset × A-infset → Bool            {a,b} ⊂ {a,b,c}
  ⊆: A-infset × A-infset → Bool            {a,b,c} ⊆ {a,b,c}
  =: A-infset × A-infset → Bool            {a,b,c} = {a,b,c}
  ≠: A-infset × A-infset → Bool            {a,b,c} ≠ {a,b}
  card: A-infset ~> Nat                    card {} = 0, card {a,b,c} = 3
```

- ∈: The membership operator (*is an element member of a set*, **true** *or* **false***?*);
- ∉: The non-membership operator (*is an element not a member of a set*, **true** *or* **false***?*);
- ∪: The infix union operator (when applied to two sets expresses the set whose members are in either or both of the two operand sets);
- ∪: The distributed prefix union operator (when applied to a set of sets expresses *the set whose members are in some of the sets of the operand set*);
- ∩: The infix intersection operator (expresses *the set whose members are in both of the two operand sets*);
- ∩: The distributed prefix intersection operator (when applied to a set of sets expresses *the set whose members are in all of the sets of the operand set*);
- \: The set complement (or set subtraction) operator (expresses *the set whose members are those of the first operand set which are not in the second operand set*);

- $\subset$: The proper subset operator (*are the members of the first operand set all members of the second operand set, and are there members of the second operand set which are not in the first operands set*, **true** *or* **false***?*);
- $\subseteq$: The subset operator (as for proper subset, but allows equality of the two operand set to be **true**);
- $=(\neq)$: The equal operator (*are the two operand sets the same (different),* **true** *or* **false***?*); and
- **card**: The cardinality operator (*"counts" the number of elements in the presumed finite operand set*).

```
value
  s′ ∪ s″ ≡ { a | a:A • a ∈ s′ ∨ a ∈ s″ }
  ∪ ss ≡ { a | a:A • ∃ s:A-set • s ∈ ss ⇒ a ∈ s }
  s′ ∩ s″ ≡ { a | a:A • a ∈ s′ ∧ a ∈ s″ }
  ∩ ss ≡ { a | a:A • ∀ s:A-set • s ∈ ss ⇒ a ∈ s }
  s′ \ s″ ≡ { a | a:A • a ∈ s′ ∧ a ∉ s″ }
  s′ ⊆ s″ ≡ ∀ a:A • a ∈ s′ ⇒ a ∈ s″
  s′ ⊂ s″ ≡ s′ ⊆ s″ ∧ ∃ a:A • a ∈ s″ ∧ a ∉ s′
  s′ = s″ ≡ ∀ a:A • a ∈ s′ ≡ a ∈ s″ ≡ s⊆s′ ∧ s′⊆s
  s′ ≠ s″ ≡ s′ ∩ s″ ≠ {}
  card s ≡
    if s = {} then 0 else
    let a:A • a ∈ s in 1 + card (s \ {a}) end end
    pre s /∗ is a finite set ∗/
  card s ≡ chaos /∗ tests for infinity of s ∗/
```

[3.3] RSL Cartesian Operations

```
type
  A, B, C
  g0: G0 = A × B × C
  g1: G1 = ( A × B × C )
  g2: G2 = ( A × B ) × C
  g3: G3 = A × ( B × C )

value
  va:A, vb:B, vc:C, vd:D

  (va,vb,vc):G0,
  (va,vb,vc):G1
  ((va,vb),vc):G2
  (va3,(vb3,vc3)):G3

decompostion expressions
  let (a1,b1,c1) = g0,
      (a1′,b1′,c1′) = g1 in .. end

  let ((a2,b2),c2) = g2 in .. end
  let (a3,(b3,c3)) = g3 in .. end
```

[3.4] RSL List Operations

value
  hd: $A^\omega \xrightarrow{\sim} A$
  tl: $A^\omega \xrightarrow{\sim} A^\omega$
  len: $A^\omega \xrightarrow{\sim}$ **Nat**
  inds: $A^\omega \to$ **Nat-infset**
  elems: $A^\omega \to$ **A-infset**
  .(.): $A^\omega \times$ **Nat** $\xrightarrow{\sim} A$
  $\frown$: $A^* \times A^\omega \to A^\omega$
  =: $A^\omega \times A^\omega \to$ **Bool**
  $\neq$: $A^\omega \times A^\omega \to$ **Bool**

examples /∗ the a, b, c, d: are values ∗/
  hd⟨a1,a2,...,am⟩=a1
  tl⟨a1,a2,...,am⟩=⟨a2,...,am⟩
  len⟨a1,a2,...,am⟩=m
  inds⟨a1,a2,...,am⟩={1,2,...,m}
  elems⟨a1,a2,...,am⟩={a1,a2,...,am}
  ⟨a1,a2,...,am⟩(i)=ai
  ⟨a,b,c⟩$\frown$⟨a,b,d⟩ = ⟨a,b,c,a,b,d⟩
  ⟨a,b,c⟩=⟨a,b,c⟩
  ⟨a,b,c⟩ ≠ ⟨a,b,d⟩

- **hd**: Head: Yield the head (i.e., first) element of non–empty lists.
- **tl**: Tail: Yield the list of list elements other than the head of the argument list (also only of non–empty lists) .
- **len**: Length: the length of a finite list.
- **inds**: Indices, or index set: Yield the index set, from 1 to the length of the list (which may be empty in which case the index set is also empty, or may be infinite, in which case the result is **chaos**).
- **elems**: Elements: Yield the possibly infinite set of all distinct elements of the list.
- $\ell(i)$: Indexing with a natural number, i, larger than 0 into a list ℓ larger than or equal to i yields its i'th element.
- $\frown$: Concatenate two operand lists into one list, first the elements of the first, finite length operand list, and then the elements of the second, possibly infinite length operand list, and in their respective order.
- = and ≠: Compare two operand lists for equality, element–by–element, respectively for the occurrence of at least one deviation!

```
value
  is_finite_list: A^ω → Bool

  len q ≡
    case is_finite_list(q) of
      true → if q = ⟨⟩ then 0 else 1 + len tl q end,
      false → chaos end

  inds q ≡
    case is_finite_list(q) of
      true → { i | i:Nat • 1 ≤ i ≤ len q },
      false → { i | i:Nat • i≠0 } end

  elems q ≡ { q(i) | i:Nat • i ∈ inds q }

  q(i) ≡
    if i=1
```

then if q≠⟨⟩ **then let** a:A,q′:Q • q=⟨a⟩⌢q′ **in** a **end else chaos end**
else q(i−1) **end**

fq ⌢ iq ≡
⟨ **if** 1 ≤ i ≤ **len** fq **then** fq(i) **else** iq(i − **len** fq) **end**
| i:**Nat** • **if len** iq≠**chaos then** i ≤ **len** fq+**len end** ⟩
pre is_finite_list(fq)

iq′ = iq″ ≡ **inds** iq′ = **inds** iq″ ∧ ∀ i:**Nat** • i ∈ **inds** iq′ ⇒ iq′(i) = iq″(i)

iq′ ≠ iq″ ≡ ∼(iq′ = iq″)

[3.5] RSL Map Operations

value
• (•): M → A $\xrightarrow{\sim}$ B, m(ai) = bi
dom: M → A-**infset** [domain of map]
dom [a1↦b1,a2↦b2,...,an↦bn] = {a1,a2,...,an}
rng: M → B-**infset** [range of map]
rng [a1↦b1,a2↦b2,...,an↦bn] = {b1,b2,...,bn}
†: M × M → M [override extension]
[a↦b,a′↦b′,a″↦b″] † [a′↦b″,a″↦b′] = [a↦b,a′↦b″,a″↦b′]
∪: M × M → M [merge ∪]
[a↦b,a′↦b′,a″↦b″] ∪ [a‴↦b‴] = [a↦b,a′↦b′,a″↦b″,a‴↦b‴]
\: M × A-**infset** → M [restriction by]
[a↦b,a′↦b′,a″↦b″]\{a} = [a′↦b′,a″↦b″]
/: M × A-**infset** → M [restriction to]
[a↦b,a′↦b′,a″↦b″]/{a′,a″} = [a′↦b′,a″↦b″]
=,≠: M × M → **Bool**
°: (A $\xrightarrow[m]{}$ B) × (B $\xrightarrow[m]{}$ C) → (A $\xrightarrow[m]{}$ C) [composition]
[a↦b,a′↦b′] ° [b↦c,b′↦c′,b″↦c″] = [a↦c,a′↦c′]

- •(•): Application: expresses that functions and maps can be applied to arguments.
- **dom**: Domain/Definition Set: denote "taking" the definition set values of a map (the a values for which the map is defined).
- **rng**: Range/Image: denote "taking" the range of a map (the corresponding b values for which the map is defined).
- †: Override/Extend: when applied to two operands denote the map which is like an override of the first operand map by all or some "pairings" of the second operand map,
- ∪: Merege: when applied to two operands denote the map which is the merge of two such maps,

- \: Restriction: the map which is a restriction of the first operand map to the elements that are not in the second operand set
- /: Restriction: the map which is a restriction of the first operand map to the elements of the second operand set.
- =, ≠: Equal, Not–Equal: when applied to two maps, compares these for equality, respectively inequality.
- °: Composition: The map from definition set elements of the first, left–operand map, m_1, to the range elements of the second, right–operand map, m_2, such that if a, in the definition set of m_1 and maps into b, and if b is in the definition set of m_2 and maps into c, then a, in the composition, maps into c.

```
value
  rng m ≡ { m(a) | a:A • a ∈ dom m }

  m1 † m2 ≡
    [ a↦b | a:A,b:B •
      a ∈ dom m1 \ dom m2 ∧ b=m1(a) ∨ a ∈ dom m2 ∧ b=m2(a) ]

  m1 ∪ m2 ≡ [ a↦b | a:A,b:B •
              a ∈ dom m1 ∧ b=m1(a) ∨ a ∈ dom m2 ∧ b=m2(a) ]

  m \ s ≡ [ a↦m(a) | a:A • a ∈ dom m \ s ]
  m / s ≡ [ a↦m(a) | a:A • a ∈ dom m ∩ s ]

  m1 = m2 ≡
    dom m1 = dom m2 ∧ ∀ a:A • a ∈ dom m1 ⇒ m1(a) = m2(a)
  m1 ≠ m2 ≡ ∼(m1 = m2)

  m°n ≡
    [ a↦c | a:A,c:C • a ∈ dom m ∧ c = n(m(a)) ]
    pre rng m ⊆ dom n
```

B.3.4 [4] RSL λ–Calculus and Functions

[4.1] The λ–Calculus Syntax

```
type /∗ A BNF Syntax: ∗/
    ⟨L⟩ ::= ⟨V⟩ | ⟨F⟩ | ⟨A⟩ | ( ⟨A⟩ )
    ⟨V⟩ ::= /∗ variables, i.e., identifiers ∗/
    ⟨F⟩ ::= λ⟨V⟩ • ⟨L⟩
    ⟨A⟩ ::= ( ⟨L⟩⟨L⟩ )

value /∗ Examples ∗/
    ⟨L⟩: e, f, a, ...
    ⟨V⟩: x, ...
    ⟨F⟩: λ x • e, ...
    ⟨A⟩: f a, (f a), f(a), (f)(a), ...
```

[4.2] Free and Bound Variables

Let x, y be variable names and e, f be λ-expressions.

- $\langle$V$\rangle$: Variable x is free in x
- $\langle$F$\rangle$: x is free in $\lambda y \bullet e$ if $x \neq y$ and x is free in e.
- $\langle$A$\rangle$: x is free in $f(e)$ if it is free in either f or e (i.e., also in both).

[4.3] Substitution

- **subst**([N/x]x) ≡ N
- **subst**([N/x]a) ≡ a
  for all variables a≠x.
- **subst**([N/x](P Q)) ≡ (**subst**([N/x]P) **subst**([N/x]Q)).
- **subst**([N/x](λx•P)) ≡ λy•P.
- **subst**([N/x](λy•P)) ≡ λy•**subst**([N/x]P)
  if x ≠ y and y is not free in N or x is not free in P.
- **subst**([N/x](λy•P)) ≡λz•**subst**([N/z]**subst**([z/y]P))
  if y ≠ x and y is free in N and x is free in P
  (where z is not free in (N P)).

[4.4] α–Renaming and β–Reduction

- α–renaming: λx•M
  If x, y are distinct variables then replacing x by y in λx•M results in λy•**subst**([y/x]M): We can rename the formal parameter of a λ-function expression provided that no free variables of its body M thereby become bound.
- β–reduction: (λx•M)(N)
  All free occurrences of x in M are replaced by the expression N provided that no free variables of N thereby become bound in the result.
  (λx•M)(N) ≡ **subst**([N/x]M)

[4.6] Function Signatures in RSL

For sorts we may want to postulate some functions:

```
type
  A, B, C
value
  obs_B: A → B,
  obs_C: A → C,
  gen_A: B×C → A
```

[4.7] Function Definitions in RSL

Functions can be defined explicitly:

value
f: A × B × C → D
f(a,b,c) ≡ Value_Expr

g: B-**infset** × (D $\overrightarrow{m}$ C-**set**) $\xrightarrow{\sim}$ A*
g(bs,dm) ≡ Value_Expr
pre $\mathcal{P}$(bs,dm)

comment: a, b, c, bs and dm are parameters of appropriate types

or implicitly:

value
f: A × B × C → D
f(a,b,c) **as** d
post $\mathcal{P}_1$(a,b,c,d)

g: B-**infset** × (D $\overrightarrow{m}$ C-**set**) $\xrightarrow{\sim}$ A*
g(bs,dm) **as** al
pre $\mathcal{P}_2$(bs,dm)
post $\mathcal{P}_3$(bs,dm,al)

comment: a, b, c, bs and dm are parameters of appropriate types

The symbol $\xrightarrow{\sim}$ indicates that the function is partial and thus not defined for all arguments. Partial functions should be assisted by preconditions stating the criteria for arguments to be meaningful to the function.

B.3.5 [5] Applicative Constructs of RSL

[5.1] The RSL let Constructs

[5.1.1] General:

Simple (i.e., non–recursive) let:

let a = $\mathcal{E}_d$ **in** $\mathcal{E}_b$(a) **end**

is an "expanded" form of:

(λa.$\mathcal{E}_b$(a))($\mathcal{E}_d$)

Recursive let:

let f = λa:A • E(f) **in** B(f,a) **end**

≡ ``the same'' **as**:

let f = **YF** **in** B(f,a) **end**

where:

F ≡ λg•λa•(E(g)) and YF = F(YF)

[5.1.2] Predicative **let***s:*

let a:A • $\mathcal{P}$(a) **in** $\mathcal{B}$(a) **end**

expresses the selection of an a value of type A which satisfies a predicate $\mathcal{P}$(a) for evaluation in the body $\mathcal{B}$(a).

[5.1.3] Patterns and Wild Cards:

Some indicative examples:

let {a} ∪ s = set **in** ... **end**
let {a,_} ∪ s = set **in** ... **end**

let (a,b,...,c) = cart **in** ... **end**
let (a,_,...,c) = cart **in** ... **end**

let ⟨a⟩⌢ℓ = list **in** ... **end**
let ⟨a,_,b⟩⌢ℓ = list **in** ... **end**

let [a↦b] ∪ m = map **in** ... **end**
let [a↦b,_] ∪ m = map **in** ... **end**

[5.2] The Applicative RSL Conditionals

if b_expr **then** c_expr **else** a_expr **end**

if b_expr **then** c_expr **end** ≡ /∗ same as: ∗/
if b_expr **then** c_expr **else skip end**

if b_expr_1 **then** c_expr_1
elsif b_expr_2 **then** c_expr_2
elsif b_expr_3 **then** c_expr_3
...

elsif b_exprt_n **then** c_expr_n **end**

case expr **of**
  choice_pattern_1 → expr_1,
  choice_pattern_2 → expr_2,
  ...
  choice_pattern_n_or_wild_card → expr_n
end

[5.3] Common Operator/Operand RSL Constructs

⟨Expr⟩ ::=
  ⟨Prefix_Op⟩ ⟨Expr⟩
  | ⟨Expr⟩ ⟨Infix_Op⟩ ⟨Expr⟩
  | ⟨Expr⟩ ⟨Suffix_Op⟩
  | ...
⟨Prefix_Op⟩ ::=
  − | ∼ | ∪ | ∩ | **card** | **len** | **inds** | **elems** | **hd** | **tl** | **dom** | **rng**
⟨Infix_Op⟩ ::=
  = | ≠ | ≡ | + | − | ∗ | ↑ | / | < | ≤ | ≥ | > | ∧ | ∨ | ⇒
  | ∈ | ∉ | ∪ | ∩ | \ | ⊂ | ⊆ | ⊇ | ⊃ | ⌢ | † | ◦
⟨Suffix_Op⟩ ::= !

B.3.6 [6] Imperative Constructs of RSL

[6.1] Variables, Assignments and the Unit Value

0. **variable** v:Type := expression
1. v := expr

[6.2] Statement Sequence and skip

2. **skip**
3. stm_1;stm_2;...;stm_n

[6.3] The Imperative RSL Conditionals

4. **if** expr **then** stm_c **else** stm_a **end**
5. **case** e **of**: p_1→S_1(p_1),...,p_n→S_n(p_n) **end**

[6.4] The Iterative RSL Conditionals

6. **while** expr **do** stm **end**
7. **do** stmt **until** expr **end**

[6.5] The Iterative RSL Sequencing

8. **for** b **in** list_expr • P(b) **do** S(b) **end**

[6.6] RSL Variable Expressions

9. v

B.3.7 [7] Parallel Constructs of RSL

[7.1] Process Channels

Let A, B and KIdx stand for a type of (channel) messages, respectively a (sort–like) index set over channels, then:

channel c:A
channel { k[i]:B • i:KIdx }

declare a channel, c, and a set of channels, k[i], which can communicate values of the designated types.

[7.2] Composition of Processes

Let P and Q stand for names of process functions, i.e., of functions which express willingness to engage in input and/or output events, i.e., in communication over channels.

Let P() and Q(i) stand for process expressions, then:

P() ‖ Q(i) Parallel composition
P() [] Q(i) Non−−deterministic External Choice (either/or)
P() ⌈⌉ Q(i) Non−−deterministic Internal Choice (either/or)

express the parallel of two processes, respectively the non–deterministic choice between two processes: Either external or internal.

[7.3] Process Input/Output

Let c, k[i] and e designate a channel, a channel and a type A, resp., type B valued expression. Then:

c ?, k[i] ? Input
c ! e, k[i] ! e Output

expresses the willing of a process to engage in an event that reads an input, respectively that writes an output.

[7.4] Process Signatures and Definitions

The below signatures are just examples. They emphasise that process functions must somehow express, in their signatyure via which channels they wish to engage in input and output events.

value
P: **Unit** → **in** c **out** k[i] **Unit**
Q: i:KIdx → **out** c **in** k[i] **Unit**

P() ≡ ... c ? ... k[i] ! e ...
Q(i) ≡ ... k[i] ? ... c ! e ...

The process function definitions (i.e., their bodies) express possible events.

B.3.8 [8] Simple RSL Specifications

Not using schemes, classes and objects an RSL specification is some sequence one or more below **type,** zero, one or more **variable,** zero, one or more **channel,** one or more **value,** and zero, one or more **axiom** clauses.

type
...
variable
...
channel
...
value
...
axiom
...

C

Indexes

D

Bibliographical Notes

[3, to appear] gives a concise overview of domain engineering; [1, to appear] relates domain and requirements engineering; [5] presents a number of domain engineering research challenges; [6, to appear] additionally presents a rather large example of the container line industry domain. [7, to appear] shows a generic, i.e., abstract domain model of road, rail, air and ship transport.

Finally [2, 8, 9], except for this, the management aspects of software engineering, present all the other issues of this Software Engineering Encyclopedia entry in "excruciating" details!

References

1. Dines Bjørner. From Domains to Requirements. In *Ugo'65 [tentative] Festschrift for Prof. Ugo Montanari*, volume ???? of *Lecture Notes in Computer Science (eds. Rocco de Nicola)*, pages 1–30, Heidelberg, May 2008. Springer.
2. Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design.* Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
3. Dines Bjørner. Domain Engineering. In *BCS FACS Seminars*, Lecture Notes in Computer Science, the BCS FAC Series (eds. Paul Boca and Jonathan Bowen), pages 1–42, London, UK, 2008. Springer. To appear.
4. Dines Bjørner. Believable Software Management. *Encyclopedia of Software Engineering*, 1(1):1–32, 2008. (This is a new journal, published by Taylor & Francis, New York and London, edited by Philip Laplante).
5. Dines Bjørner. Domain Theory: Practice and Theories, Discussion of Possible Research Topics. In *ICTAC'2007*, volume 4701 of *Lecture Notes in Computer Science (eds. J.C.P. Woodcock et al.)*, pages 1–17, Heidelberg, September 2007. Springer.
6. Dines Bjørner. Domain Engineering. In *The 2007 Lipari PhD Summer School*, volume ???? of *Lecture Notes in Computer Science (eds. E. Börger and A. Ferro)*, pages 1–102, Heidelberg, Germany, 2008. Springer. To appear.
7. Dines Bjørner. Development of Transportation Systems. In *2007 ISoLA Workshop On Leveraging Applications of Formal Methods, Verification and Validation;*

Special Workshop Theme: Formal Methods in Avionics, Space and Transport, ENSMA, Futuroscope, France, December 12–14 2007.
8. Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling.* Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
9. Dines Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages.* Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. Chapters 12–14 are primarily authored by Christian Krog Madsen.