A Credit Card System: Uppsala Draft

Dines Bjørner
Fredsvej 11, DK-2840 Holte, Danmark
E–Mail: bjorner@gmail.com, URL: www.imm.dtu.dk/˜dibj

May 31, 2016: 10:31 am

Abstract

This report presents a first attempt at a model of a credit card system. A first version of this document, marked Uppsala Draft, was worked out for my Uppsala University lectures 18–20 April 2016. *Remarks in this type-font refers to my paper: [Bjø16, Manifest Domains: Analysis & Description].* Appendix A presents a primer of RSL, the Raise Specification Language.

Contents

1 Introduction

We present a domain description of an abstracted credit card system. The narrative part of the description is terse, perhaps a bit too terse. I might "repair" this shortness if told so. A reference is made to my paper: [Bjø16, Manifest Domains: Analysis & Description]. That paper can be found on the Internet: http://www2.compute.dtu.dk/˜dibj/2015/faoc/faoc-bjorner.pdf.

Credit cards are moving from simple plastic cards to smart phones. Uses of credit cards move from their mechanical insertion in credit card terminals to being swiped. Authentication (hence not modelled) moves from keying in security codes to eye iris "prints", and/or finger prints or voice prints or combinations thereof.

This document abstracts from all that in order to understand a bare, minimum essence of credit cards and their uses. Based on a model, such as presented here, the reader should be able to extend/refine the model into any future technology – for requirements purposes.

2 Endurants

2.1 Credit Card Systems

[Bjø16, Sect.3.1.6, pg.11:]: observe_part_sorts

1 Credit card systems consists of three kinds of parts:

   a. a part, *cs*:*CS*, of credit cards[1],

   b. a part, *bs*:*BS*, of banks, and

   c. a part, *ss*:*SS*, of shops.

type
1. CCS
1a.. CS, C, CSI
1b.. BS, B, BSI
1c.. SS, S, SSI
value
1a.. obs_CS: CCS → CS, uid_CS: CS → CSI,
1b.. obs_BS: CCS → BS, uid_BS: BS → BSI,
1c.. obs_SS: CCS → SS, uid_SS: SS → SSI

The composite part CS can be thought of as a credit card company, say VISA[2]. The composite part BS can be thought of as a bank society, say BBA: British Banking Association. The composite part SS can be thought of as the association of retailers, say bira: British Independent Retailers Association[3].
[Bjø16, Sect.3.1.7, pg.13]: observe_part_type

[1] We "equate" credit cards with their holders.

[2] Our simple model allows for only one credit card company. But that model can easily be extended to model a set of credit card companies, viz.: VISA, MasterCard, American Express, Diner's Club, etc..

[3] The model does not prevent "shops" from being airlines, or car rental agencies, or dentists, or consultancy firms. In this case *SS* would be some appropriate association.

2 The credit card part, *cs*:*CS*, abstracts a set, *socs*:*Cs*, of credit cards.

3 The bank part, *bs*:*BS*, abstracts a set, *sobs*:*Bs*, of banks.

4 The shop part, *ss*:*SS*, abstracts a set, *soss*:*Sc*, of shops.

```
type
2.  Cs = C-set, C
3.  Bs = B-set, B
4.  Ss = S-set, S
value
2.  obs_CS: CS → Cs, obs_Cs: CS → Cs
3.  obs_BS: BS → Bs, obs_Bs: BS → Bs
4.  obs_SS: SS → Ss, obs_Ss: SS → Ss
```

[Bjø16, Sect.3.2, pg.16]: observe_unique_identifier

5 Each credit card, bank and shop has a unique identifier, *ci*:*CI*, *bi*:*BI*, respectively *si*:*SI*.

6 One can define functions which extract all the

a. unique credit card,

b. bank and

c. shop identifiers

from a credit card system.

```
type
5.  CI, BI, SI
value
5.   uid_C: C → CI
5.   uid_B: B → BI
5.   uid_S: S → SI
6a.. xtr_CIs: CCS → CI-set
6a.. xtr_CIs(ccs) ≡ {uid_C(c)|c:C•c ∈ obs_Cs(obs_CS(ccs))}
6b.. xtr_BIs: CCS → BI-set
6b.. xtr_BIs(ccs) ≡ {uid_B(s)|b:B•b ∈ obs_Bs(obs_BS(ccs))}
6c.. xtr_SIs: CCS → SI-set
6c.. xtr_SIs(ccs) ≡ {uid_S(s)|s:S•s ∈ obs_Ss(obs_SS(ccs))}
```

7 For all credit card systems it is the case that

a. all credit card identifiers are distinct from bank identifiers,

b. all credit card identifiers are distinct from shop identifiers,

c. all shop identifiers are distinct from bank identifiers,

```
axiom
7.    ∀ ccs:CCS •
7.        let cis=xtr_CIs(ccs), bis=xtr_BIs(ccs), sis = xtr_SIs(ccs) in
7a..        cis ∩ bis = {}
7b..      ∧ cis ∩ sis = {}
7c..      ∧ sis ∩ bis = {} end
```

2.2 Credit Cards

We “equate” credit cards with their holders.

8 A credit card (besides a unique identification) has

[Bjø16, Sect.3.1.2, pg.17]: observe_mereology

a. a mereology which “connects” it to any of the shops of the system and to exactly one bank of the system,

b. and some attributes — which we shall disregard.

c. The wellformedness of a credit card system includes the wellformedness of credit card mereologies with respect to the system of banks and shops:

i The unique shop identifiers of a credit card mereology must be those of the shops of the credit card system; and

ii the unique bank identifier of a credit card mereology must be of one of the banks of the credit card system.

```
type
8.        C
8a..        CM = SI-set × BI
value
8a..        mereo_CM: C → CM
8c..        wf_CM_of_C: CCS → Bool
8c..        wf_CM_of_C(ccs) ≡
8a..            let bis=xtr_BIs(ccs), sis=xtr_SIs(ccs) in
8a..             ∀ c:C•c ∈ obs_Cs(obs_CS(ccs)) ⇒
8a..                 let (ccsis,bi)=mereo_CM(c) in
8(c.)i.              ccsis ⊆ sis
8(c.)ii.           ∧ b ∈ bis
8a..             end end
```

Constraint 8(c.)i limits a credit card to potentially be used only in a proper subset of all shops. To allow for all shops one must change the wording to *‘be **all** those of the shops ...’*, and change ⊆ in formula line 8(c.)i to ‘=’.

2.3 Banks

Our model of banks is very limited.

9 A bank has

  a. a unique bank identifier,
  b. a mereology which “connects” it to a subset of all credit cards and a subset of all shops,
  c. and, as attributes:
    i a cash register, and
    ii a ledger.

9(c.)ii. The ledger records

  a. for every card, by unique credit card identifier,
  b. the current balance: how much money, credit or debit, i.e., plus or minus, that customer is owed, respectively has borrowed from the bank,
  c. the dates-of-issue and -expiry of the credit card, and
  d. the name, address, and other information about the credit card holder.

10 The wellformedness of the credit card system includes the wellformedness of the banks with respect to the credit cards and shops:

  a. the bank mereology’s
  b. must list a subset of the credit card identifiers and a subset of the shop identifiers.

```
type
9.        B
9b..      BM = CI-set × SI-set
9(c.)i.   CR = Bal
9(c.)ii.  LG = CI $\underset{m}{\overrightarrow{\ }}$ (Bal×DoI×DoE×...)
9b..      Bal = Int
value
9b..      mereo_B: B → BM
9(c.)i.   attr_CR: B → CR
9(c.)ii.  attr_LG: B → LG
10.       wf_BM_B: CCS → Bool
10.       wf_BM_B(ccs) ≡
10.           let allcis = xtr_CIs(ccs), allsis = xtr_SIs(ccs) in
10.           ∀ b:B • b ∈ obs_Bs(obs_BS(ccs)) in
10a..             let (cis,sis) = mereo_B(b) in
10b..             cis ⊆ ∀ cis ∧ sis ⊆ allsis
10.           end end
```

2.4 Shops

11 A shop (besides a unique shop identifier) has a

a. mereology and some attributes.

11a.. The mereology of a shop is a pair:

a. a unique bank identifiers, and

b. a set of unique credit card identifiers.

11a.. We omit treatment of shop attributes.

11a.. The mereology of a shop

a. must list a bank of the credit card system,

b. and a subset (or all) of the unique credit identifiers.

```
type
11a..   SM = CI-set × BI
value
11a..   mereo_S: S → SM
11a..   wf_SM_S: CCS → Bool
11a..   wf_SM_S(ccs) ≡
11a..       let allcis = xtr_CIs(ccs), allbis = xtr_BIs(ccs) in
11a..       ∀ s:S • s ∈ obs_Ss(obs_SS(ccs)) ⇒
11a..           let (cis,bi) mereo_S(s) in
11a..           cis ⊆ allcis
11b..        ∧ bi ∈ allbis
11a..       end end
```

3 Perdurants

3.1 Behaviours

3.1.1 System

[Bjø16, Sect.4.11.2, pg.35]: Process Schema I: Abstract is_composite(p) and Process Schema II: Concrete is_concrete(p).

12 We ignore the behaviours related to the *CCS*, *CS*, *HS* and *SS* parts.

13 We therefore only consider the behaviours related to the *Cs*, *Hs* and *Ss* parts.

14 And we therefore compile the credit card system into the parallel composition of the parallel compositions of all the credit card, *crd*, all the bank, *bnk*, and all the shop, *shp*, behaviours.

value
12. ccs:CCS
12. cs:CS = obs_CS(ccs), uics:CSI = uid_CS(cs),
12. bs:BS = obs_BS(ccs), uibs:BSI = uid_BS(bs),
12. ss:SS = obs_SS(ccs), uiss:SSI = uid_SS(ss),
13. socs:Cs = obs_Cs(cs),
13. sobs:Bs = obs_Bs(bs),
13. soss:Ss = obs_Ss(ss),
14. sys: **Unit** → **Unit**,
12. sys() ≡
14.   cards$_{uics}$(mereo_CS(cs),...) ∥ ∥{crd$_{uid\_C(c)}$(mereo_C(c))|c:C•c ∈ socs}
14.   ∥ banks$_{uibs}$(mereo_BS(bs),...) ∥ ∥{bnk$_{uid\_B(b)}$(mereo_B(b))|b:B•b ∈ sobs}
14.   ∥ shops$_{uiss}$(mereo_SS(ss),...) ∥ ∥{shp$_{uid\_S(s)}$(mereo_S(s))|s:S•s ∈ soss},
12. cards$_{uics}$(...) ≡ **skip**,
12. banks$_{uibs}$(...) ≡ **skip**,
12. shops$_{uiss}$(...) ≡ **skip**
axiom **skip** ∥ behaviour(...) ≡ behaviour(...)

3.1.2 Channels

[Bjø16, Sect.4.5.1, pg.30]: Channels and Communication

15 Credit card behaviours interact with

a. many bank (each with one) and

b. many shop behaviours.

16 Shop behaviours interact with

a. many bank (each with one) and

15b.. many credit card behaviours.

15a..,16a.. Bank behaviours interact with many credit card and many shop behaviours.

17 The inter-behaviour interactions concern:

a. between credit cards and banks: withdrawal requests as to a sufficient, mk_Wdrw(am), balance on the credit card account for buying *am*:*aM* amounts of goods or services, with the bank response of either is_OK() or is_NOK(), or the revoke of a card;

b. between credit cards and shops: the buying, for an amount, *am*:*aM*, of goods or services: mk_Buy(am), or the refund of an amount; and

c. between shops and banks: the deposit of an amount, *am*:*aM*, in the shops' bank account: mk_Dep(am).

channel
15a.. {ch_cb[ci,bi]|ci:CI,bi:BI•ci ∈ cis ∧ bi ∈ bis}:CB_Msg
15b.. {ch_cs[ci,si]|ci:CI,si:SI•ci ∈ cis ∧ si ∈ sis}:CS_Msg
16a. {ch_sb[si,bi]|si:SI,bi:BI•si ∈ sis ∧ bi ∈ bis}:SB_Msg
type
17a.. CB_Msg == mk_Wdrw(am:aM) | is_OK() | is_NOK() | mk_Rev()
17b.. CS_Msg == mk_Buy(am:aM) | mk_Ref(am:aM)
17c.. SB_Msg == mk_Dep((ci:CI|si:SI),am:aM)

3.1.3 Behaviour Interactions

18 The credit card initiates

a. *buy* transactions

i [1.Buy] by inquiring with its bank as to sufficient purchase funds;

ii [2.Buy] if NOK then there are presently no further actions; if OK

iii [3.Buy] the credit card behaviour requests the purchase from the shop – handing it an appropriate amount *am*:*aM*;

iv [4.Buy] finally the shop requests its bank to deposit the purchase amount in the shop's bank account.

b. *refund* transactions

i [1.Refund] by requesting such refunds, in the amount of *am*:*aM*, from a[ny] shop;

ii [2.Refund] whereupon the shop requestss its bank to move the amount *am*:*aM* from the shop's bank account to the credit card's account.

Thus the three sets of behaviours, *crd*, *bnk* and *shp* interact as sketched in Fig. 1 on the following page.

[1.Buy] Item 23 on the next page: ch_cb[ci,bi]!mk_Wdrw(am);
    Item 33 on Page 12: **let** mk_Wdrw(ci,am) = ⌈⌉ {ch_cb[bi,bi]?|ci:CI•ci ∈ cis}
[2.Buy] Item 35 on Page 12: ch_cb[ci,bi]!is_[N]OK()
    Item 33 on Page 12: **let** mk_Wdrw(ci,am) = ⌈⌉ {ch_cb[ci,bi]?|ci:CI•ci ∈ cis} **in**
[3.Buy] Item 25 on the next page: ch_cs[ci,si]!mk_Buy(am)
    Item 45 on Page 13: **let** mk_Buy(am) = ⌈⌉{ch_cs[ci,si]?|ci:CI•ci ∈ cis} **in**
[4.Buy] Item 45 on Page 13: ch_sb[si,bi]!mk_Dep(si,am)
    Item 39 on Page 12: **let** mk_Dep(si,am)=⌈⌉ {ch_cs[ci,si]?|si:SI•si ∈ sis} **in**
[1.Refund] Item 27 on Page 11: ch_cs[ci,si]!mk_Ref((ci,si),am)
    Item 46 on Page 13: **let** (si,mk_Ref(ci,am))=⌈⌉{(si′,ch_sb[si,bi]?)|si,si′:SI•{si,si′}⊆sis∧si=si′} **in**
[2.Refund] Item 41 on Page 13: ch_sb[ci,bi]!mk_Ref(ci,am)

3.1.4 Credit Card

19 The credit card behaviour, *crd*, takes the credit card unique identifier, the credit card mereology, and attribute arguments (omitted). The credit card behaviour, *crd*, accepts inputs from and offers outputs to the bank, *bi*, and any of the shops, *si*∈*sis*.

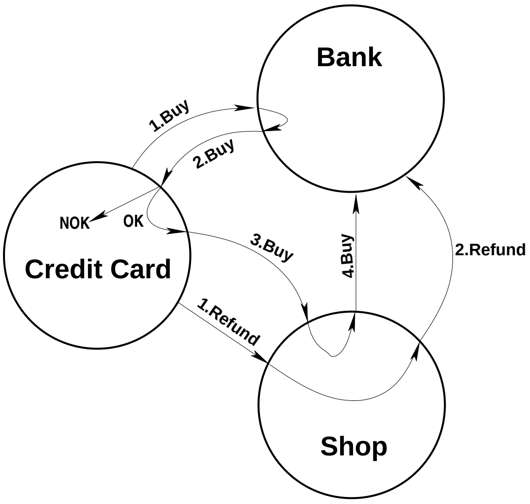

Figure 1: Credit Card, Bank and Shop Behaviours

20 The credit card behaviour, *crd*, non-deterministically, internally "cycles" between

a. *buy*ing,

b. getting *ref*unds.

```
value
19.   crd_{ci:CI}: (bi,sis):CM → in,out ch_cb[ci,bi],{ch_cs[ci,si]|si:SI•si ∈ sis}  Unit
19.   crd_{ci}(bi,sis) ≡
20a..       (buy(ci,(bi,sis))
20.          ⨅
20b..        rfu(ci,(bi,sis))) ;
19.         crd_{ci}(ci,(bi,sis))
```

21 By *am*:*AM* we mean an amount of money, and by *si*:*SI* we refer to a shop in which we have selected a number or goods or services (not detailed) costing *am*:*AM*.

22 The amount for which to buy and the shop from which to buy are selected (arbitrarily).

23 The credit card (holder) withdraws *am*:*AM* from the bank, if sufficient funds are available[4].

24 The response from the bank

25 is either OK and the credit card [holder] completes the purchase by buying the goods or services offered by the selected shop,

[4] First the credit card [holder] requests a withdrawal. If sufficient funds are available, then the withdrawal takes place, otherwise not – and the credit card holder is informed accordingly.

26 or the response is “not OK”, and the transaction is skipped.

```
type
21.   AM = Int
value
20a..  buy: ci:CI × (bi,sis):CM → in,out ch_cb[ci,bi] out {ch_cs[ci,si]|si:SI•si ∈ sis}  Unit
20a..  buy(ci,(bi,sis)) ≡
22.      let am:aM • am>0, si:SI • si ∈ sis in See Discussion note 49a. on Page 15
23.      let msg = (ch_cb[ci,bi]!mk_Wdrw(am);ch_cb[ci,bi]?) in
24.      case msg of
25.          is_OK() → ch_cs[ci,si]!mk_Buy(am),
26.          is_NOK() → skip
20a..      end end end
```

27 The credit card [handler] requests a refund *am*:*aM* from shop *si*:*SI*.

This request is handled by the shop behaviour’s sub-action *ref*, see lines 43.–47. page 13.

```
value
20b..  rfu: ci:CI × (bi,sis):CM → out {ch_cs[ci,si]|si:SI•si ∈ sis}  Unit
20b..  rfu(ci,(bi,sis)) ≡
27.        let am:AM • am>0, si:SI • si ∈ sis in   See Discussion note 49b. on Page 15
27.        ch_cs[ci,si]!mk_Ref((ci,si),am)
19.        end
```

3.1.5 Banks

28 The bank behaviour, *bnk*, takes the bank’s unique identifier, the bank mereology, and the programmable attribute arguments: the ledger and the cash register. The bank behaviour, *bnk*, accepts inputs from and offers outputs to the any of the credit cards, *ci*∈*cis*, and any of the shops, *si*∈*sis*.

29 The bank behaviour non-deterministically internally chooses to accept

30 either withdrawal requests from credit cards

31 or deposit requests from shops or

32 or refund requests from credit cards.

```
value
28.   bnk_bi:BI: (cis,sis):BM → (LG×CR) →
28.        in,out {ch_cb[ci,bi]|ci:CI•ci ∈ cis} {ch_sb[si,bi]|si:SI•si ∈ sis}  Unit
28.   bnk_bi((cis,sis))(lg:(bal,doi,doe,...),cr) ≡
30.       withdraw(ci,(cis,sis))(lg,cr)
29.       ⌈⌉
```

```
31.       deposit(bi,(cis,sis))(lg,cr)
29.       ⊓
32.       refund(bi,(cis,sis))(lg,cr)
```

33 The withdraw request (an action) non-deterministically, externally offers to accept input from a credit card behaviour and marks the only possible form of input from credit cards, $(mk\_Wdrw(am))$, with the identity of the credit card.

34 If the requested amount (to be withdrawn) is not within balance on the account

35 then we, at present, refrain from defining an outcome (**chaos**),

36 otherwise the bank behaviour informs the credit card behaviour that the amount can be withdrawn.

37 Whereupon the bank behaviour is resumed notifying a lower balance and "withdraws" the monies from the cash register.

value

```
30.  withdraw: bi:BI × (cis,sis):BM → (LG×CR) → in,out {ch_cb[bi,ci]|ci:CI•ci ∈ cis} Unit
30.  withdraw(bi,(cis,sis))(lg,cr) ≡
33.      let mk_Wdrw(ci,am) = [] {ch_cb[ci,bi]?|ci:CI•ci ∈ cis} in
37.      let (bal,doi,doe) = lg(ci) in
34.      if am>bal
35.          then ch_cb[ci,bi]!is_NOK()
36.          else ch_cb[ci,bi]!is_OK() end ;
37.      bnk$_{bi}$(cis,sis)(lg†[ci↦(bal−am,doi,doe)],cr−am)
28.      end end
```

38 The deposit action is invoked, buy a shop behaviour, when a credit card [holder] buy's for a certain amount, *am*:*aM*, or requests a refund of that amount. The deposit is made by shop behaviours, either on behalf of themselves, hence *am*:*aM*, is to be inserted into the shops' bank account, *si*:*SI*, or on behalf of a credit card [i.e., a customer], hence *am*:*aM*, is to be inserted into the credit card holder's account, *ci*:*CI*.

39 The message, ch_cs[ci,si]?, received from a credit card behaviour is either concerning a buy [in which case *i* is a *ci*:*CI*, hence sale, or a refund order [in which case *i* is a *si*:*SI*].

40 In either case, the respective bank account is "upped" by *am*:*aM* – and the bank behaviour is resumed.

value

```
31.  deposit: bi:BI × (cis,sis):BM → (LG×CR) → in,out {ch_sb[bi,si]|si:SI•si ∈ sis} Unit
31.  deposit(bi,(cis,sis))(lg,cr) ≡
39.      let mk_Dep(si,am) = [] {ch_cs[ci,si]?|si:SI•si ∈ sis} in
37.      let (bal,doi,doe) = lg(si) in
40.      bnk$_{bi}$(cis,sis)(lg†[si↦(bal+am,doi,doe)],cr+am)
38.      end end
```

41 The *refund* action non-deterministically externally offers to accept a mk_Ref(ci,am) request from a shop behaviour, *si*.

42 The bank behaviour is then resumed with the credit card balance incremented by *am* and the shop balance decremented by that same amount.

```
value
31.  refund: bi:BI × (cis,sis):BM → (LG×CR) → in,out {ch_sb[bi,si]|si:SI•si ∈ sis} Unit
31.  refund(bi,(cis,sis))(lg,cr) ≡
41.       let (si,mk_Ref(ci,am)) = ⌷ {(si′,ch_sb[si,bi]?)|si,si′:SI•{si,si′}⊆sis∧si=si′} in
37.       let (balc,doic,doec) = lg(ci), (sbal,sdoi,sdoe) = lg(si) in
42.       bnk_bi(cis,sis)(lg†[ci↦(bcal+am,cdoi,cdoe)]†[si↦(sbal−am,sdoi,sdoe)],cr)
31.       end end
```

3.1.6 Shops

43 The shop behaviour, *shp*, takes the shop's unique identifier, the shop mereology, and attribute arguments (omitted). The shop behaviour, *shp*, accepts inputs from and offers outputs to the any of the credit cards, *ci*∈*cis*, and any of the shops, *si*∈*sis*.

44 The shop behaviour non-deterministically, externally

45 either offers to accept a Buy request from a credit card behaviour.
This input is (via the *sale* action) of the form *mk_Buy*(*am*),

or

46 offers to accept a refund request in this amount, *am* from a credit card [holder].
This input is (via the *refe* action) of the form *mk_Ref*(*am*),

47 Whereupon the shop behaviour resumes being a shop behaviour.

```
value
43.  shp_si:SI: sm:(cis:CI-set×bi:BI) × ... → in,out: {cs[ci,si]|ci:CI•ci ∈ cis},sb[si,bi] Unit
43.  shp_si((cis,bi),...) ≡
45.     (sal(si,(bi,cis),...)
44.       ⌷
46.      ref(si,(cis,bi),...)):

45.  sal: sm:(cis:CI-set×bi:BI) × ... → in,out: {cs[ci,si]|ci:CI•ci ∈ cis},sb[si,bi] Unit
45.  sal(si,(cis,bi),...) ≡
45.     let mk_Buy(am) = ⌷{ch_cs[ci,si]?|ci:CI•ci ∈ cis} in
45.     ch_sb[si,bi]!mk_Dep(si,am) end
47.     shp_si((cis,bi),...)

46.  ref: sm:(cis:CI-set×bi:BI) → in,out: {cs[ci,si]|ci:CI•ci ∈ cis},sb[si,bi] Unit
46.  ref(si,(cis,bi)) ≡
```

```
46.    let mk_Ref((ci,si),am) = ⌷{ch_cs[ci,si]?|ci:CI•ci ∈ cis} in
46.    ch_sb[ci,bi]!mk_Ref(ci,am) end
47.    shp$_{si}$((cis,bi),...)
```

4 Discusssion

48 The credit card system narrated and formalised in this document is an abstraction. We claim that it portrays an essence of credit cards.

49 The reader may object to certain things:

a. We do not model how a credit card holder selects services from a service provider (here modelled as shops) or products in a shop. Nor do we model that the card holder actually obtains those services or products.

   All this is summarised in Item 3.1.4 on Page 11: **let** am:aM • am>0, si:SI • si ∈ sis **in**.

   In other words: this is not considered an element of "an essence" of credit cards.

b. We, "similarly" do not model how the refund request is arrived at.

   All this is summarised in Item 3.1.4 on Page 11: **let** am:AM • am>0, si:SI • si ∈ sis **in**.

   In other words: this is not considered an element of "an essence" of credit cards.

c. Also: we do not model whether the balance of the shop's bank account is sufficient to refund a card holder.

d. Etcetera.

The present credit card system model can "easily" be extended to incorporate these and other matters.

50 Without showing explicit evidence we claim that present domain description can serve as a basis for both domain and requirements modelling standard as well as current and future credit/pay/etc. card systems.

51 Etcetera.

5 Bibliography

5.1 Some Remarks

We refer to texts on RSL and Software Engineering: [Bjø06a, Bjø06b, Bjø06c, Bjø08b, Bjø08c, Bjø08d, Bjø10a, Bjø10b, Bjø10c, Bjø09b]

5.2 Other Domain Descriptions

We list a number of reports all of which document descriptions of domains. These descriptions were carried out in order to research and develop the domain analysis and description concepts now summarised in the present paper. These reports ought now be revised, some slightly, others less so, so as to follow all of the prescriptions of the current paper. Except where a URL is given in full, please prefix the web reference with: `http://www2.compute.dtu.dk/~dibj/`.

1 *A Railway Systems Domain:* `http://euler.fd.cvut.cz/railwaydomain/` (2003)

2 *Models of IT Security. Security Rules & Regulations:* `it-security.pdf` (2006)

3 *A Container Line Industry Domain:* `container-paper.pdf` (2007)

4 *The "Market": Consumers, Retailers, Wholesalers, Producers:* `themarket.pdf` (2007)

5 *What is Logistics ?:* `logistics.pdf` (2009)

6 *A Domain Model of Oil Pipelines:* `pipeline.pdf` (2009)

7 *Transport Systems:* `comet/comet1.pdf` (2010)

8 *The Tokyo Stock Exchange:* `todai/tse-1.pdf` and `todai/tse-2.pdf` (2010)

9 *On Development of Web-based Software. A Divertimento:* `wfdftp.pdf` (2010)

10 *Documents (incomplete draft):* `doc-p.pdf` (2013)

5.2.1 Published Papers

- Web page www.imm.dtu.dk/˜dibj/domains/ lists the published papers and reports mentioned below.

- I have thought about domain engineering for more than 25 years.

- But serious, focused writing only started to appear since [Bjø06c, Part IV] — with [Bjø03, Bjø97] being exceptions:

  - [Bjø07, 2007] suggests a number of domain science and engineering research topics;
  - [Bjø10d, 2008] covers the concept of domain facets;
  - [BE10, 2008] explores compositionality and Galois connections.
  - [Bjø08a, Bjø10f, 2008,2009] show how to systematically, but, of course, not automatically, "derive" requirements prescriptions from domain descriptions; 1
  - [Bjø11a, 2008] takes the triptych software development as a basis for outlining principles for believable software management;
  - [Bjø09a, Bjø14a, 2009,2013] presents a model for Stanisław Leśniewski's [**?**] concept of mereology;
  - [Bjø10e, Bjø11b] present an extensive example and is otherwise a precursor for the present paper;
  - [Bjø11c, 2010] presents, based on the `TripTych` view of software development as ideally proceeding from domain description via requirements prescription to software design, concepts such as software demos and simulators; 2
  - [Bjø13, 2012] analyses the `TripTych`, especially its domain engineering approach, with respect to Maslow's [5] and Peterson's and Seligman's [6] notions of humanity: how can computing relate to notions of humanity;

[5] *Theory of Human Motivation.* Psychological Review 50(4) (1943):370-96; and *Motivation and Personality*, Third Edition, Harper and Row Publishers, 1954.

[6] *Character strengths and virtues: A handbook and classification.* Oxford University Press, 2004

- the first part of [Bjø14b, 2014] is a precursor for the present paper with its second part presenting a first formal model of the elicitation process of analysis and description based on the prompts more definitively presented in the current paper; and
- [Bjø14c, 2014] focus on domain safety criticality.

The present paper basically replaces the domain analysis and description section of all of the above reference — including [Bjø06c, Part IV, 2006].

5.3 References

[BE10] Dines Bjørner and Asger Eir. Compositionality: Ontology and Mereology of Domains. Some Clarifying Observations in the Context of Software Engineering in July 2008, eds. Martin Steffen, Dennis Dams and Ulrich Hannemann. In *Festschrift for Prof. Willem Paul de Roever Concurrency, Compositionality, and Correctness*, volume 5930 of *Lecture Notes in Computer Science*, pages 22–59, Heidelberg, July 2010. Springer.

[Bjø97] Dines Bjørner. Michael Jackson's Problem Frames: Domains, Requirements and Design. In Li ShaoYang and Michael Hinchley, editors, *ICFEM'97: International Conference on Formal Engineering Methods*, Los Alamitos, November 12–14 1997. IEEE Computer Society. Final Version.

[Bjø03] Dines Bjørner. Domain Engineering: A "Radical Innovation" for Systems and Software Engineering ? In *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, Heidelberg, October 7–11 2003. Springer–Verlag. The Zohar Manna International Conference, Taormina, Sicily 29 June – 4 July 2003. .

[Bjø06a] Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. .

[Bjø06b] 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.

[Bjø06c] Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.

[Bjø07] 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.

[Bjø08a] Dines Bjørner. From Domains to Requirements. In *Montanari Festschrift*, volume 5065 of *Lecture Notes in Computer Science (eds. Pierpaolo Degano, Rocco De Nicola and José Meseguer)*, pages 1–30, Heidelberg, May 2008. Springer.

[Bjø08b] Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Qinghua University Press, 2008.

[Bjø08c] Dines Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Qinghua University Press, 2008.

[Bjø08d] Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Qinghua University Press, 2008.

[Bjø09a] Dines Bjørner. On Mereologies in Computing Science. In *Festschrift: Reflections on the Work of C.A.R. Hoare*, History of Computing (eds. Cliff B. Jones, A.W. Roscoe and Kenneth R. Wood), pages 47–70, London, UK, 2009. Springer.

[Bjø09b] Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering*. A JAIST Press Research Monograph # 4, 536 pages, March 2009.

[Bjø10a] Dines Bjørner. ***Chinese:*** *Software Engineering, Vol. 1: Abstraction and Modelling*. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.

[Bjø10b] Dines Bjørner. ***Chinese:*** *Software Engineering, Vol. 2: Specification of Systems and Languages*. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.

[Bjø10c] Dines Bjørner. ***Chinese:*** *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.

[Bjø10d] Dines Bjørner. Domain Engineering. In Paul Boca and Jonathan Bowen, editors, *Formal Methods: State of the Art and New Directions*, Eds. Paul Boca and Jonathan Bowen, pages 1–42, London, UK, 2010. Springer.

[Bjø10e] Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics, Part I of II: The Engineering Part*. *Kibernetika i sistemny analiz*, (4):100–116, May 2010.

[Bjø10f] Dines Bjørner. The Rôle of Domain Engineering in Software Development. Why Current Requirements Engineering Seems Flawed! In *Perspectives of Systems Informatics*, volume 5947 of *Lecture Notes in Computer Science*, pages 2–34, Heidelberg, Wednesday, January 27, 2010. Springer.

[Bjø11a] Dines Bjørner. Believable Software Management. *Encyclopedia of Software Engineering*, 1(1):1–32, 2011.

[Bjø11b] Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics Part II of II: The Science Part*. *Kibernetika i sistemny analiz*, (2):100–120, May 2011.

[Bjø11c] Dines Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. In *Rainbow of Computer Science, Festschrift for Hermann Maurer on the Occasion of His 70th Anniversary.*, Festschrift (eds. C. Calude, G. Rozenberg and A. Saloma), pages 167–183. Springer, Heidelberg, Germany, January 2011.

[Bjø13] Dines Bjørner. *Domain Science and Engineering as a Foundation for Computation for Humanity*, chapter 7, pages 159–177. Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC [Francis & Taylor], 2013. (eds.: Justyna Zander and Pieter J. Mosterman).

[Bjø14a] Dines Bjørner. *A Rôle for Mereology in Domain Science and Engineering*. Synthese Library (eds. Claudio Calosi and Pierluigi Graziani). Springer, Amsterdam, The Netherlands, October 2014.

[Bjø14b] Dines Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model. In Shusaku Iida, José Meseguer, and Kazuhiro Ogata, editors, *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi*. Springer, May 2014.

[Bjø14c] Dines Bjørner. Domain Engineering – A Basis for Safety Critical Software. Invited Keynote, ASSC2014: Australian System Safety Conference, Melbourne, 26–28 May, December 2014.

[Bjø16] Dines Bjørner. Manifest Domains: Analysis & Description. *Expected published by Formal Aspects of Computing*, 2016.

A RSL: The Raise Specification Language

A.1 Type Expressions

- Type expressions are expressions whose value are of type type, that is,
- possibly infinite sets of values (of "that" type).

A.1.1 Atomic Types

- Atomic types have (atomic) values.
- That is, values which we consider to have no proper constituent (sub-)values,
- i.e., cannot, to us, be meaningfully "taken apart".

RSL has a number of *built-in* atomic types. There are the Booleans, integers, natural numbers, reals, characters, and texts.

```
type
  [1] Bool    true, false
  [2] Int     ... , −2, −2, 0, 1, 2, ...
  [3] Nat      0, 1, 2, ...
  [4] Real    ..., −5.43, −1.0, 0.0, 1.23···, 2,7182···, 3,1415···, 4.56, ...
  [5] Char     "a", "b", ..., "0", ...
  [6] Text    "abracadabra"
```

A.1.2 Composite Types

- Composite types have composite values.
  - That is, values which we consider to have proper constituent (sub-)values,
  - i.e., can be meaningfully "taken apart".
- There are two ways of expressing composite types:
  - either explicitly, using concrete type expressions,
  - or implicitly, using sorts (i.e., abstract types) and observer functions.

Concrete Composite Types From these one can form type expressions: finite sets, infinite sets, Cartesian products, lists, maps, etc.

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

```
  [7] A-set
  [8] A-infset
  [9] A × B × ... × C
  [10] A*
```

[11] A^{ω}
[12] $\mathsf{A} \underset{m}{\overrightarrow{\;}} \mathsf{B}$
[13] $\mathsf{A} \rightarrow \mathsf{B}$
[14] $\mathsf{A} \overset{\sim}{\rightarrow} \mathsf{B}$
[15] (A)
[16] A | B | ... | C
[17] mk_id(sel_a:A,...,sel_b:B)
[18] sel_a:A ... sel_b:B

The following are 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 0, 1, 2, ...

4 The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period ("."), 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 cardinality set values.

8 The set type of infinite and finite cardinality set values.

9 The Cartesian type of Cartesian values.

10 The list type of finite length list values.

11 The list type of infinite and finite length list values.

12 The map type of finite definition set map values.

13 The function type of total function values.

14 The function type of partial function values.

15 In (A) A is constrained to be:

- either a Cartesian $\mathsf{B} \times \mathsf{C} \times \ldots \times \mathsf{D}$, 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, e.g., $(\mathsf{A} \underset{m}{\overrightarrow{\;}} \mathsf{B})$, or (A^*)**-set**, or (A-**set**)list, or $(\mathsf{A}|\mathsf{B}) \underset{m}{\overrightarrow{\;}} (\mathsf{C}|\mathsf{D}|(\mathsf{E} \underset{m}{\overrightarrow{\;}} \mathsf{F}))$, 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, ..., 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, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

Sorts and Observer Functions

```
type
   A, B, C, ..., D
value
   obs_B: A → B, obs_C: A → C, ..., obs_D: A → D
```

- The above expresses
  - that values of type A
  - are composed from at least three values —
  - and these are of type B, C, ..., and D.
- A concrete type definition corresponding to the above
  - presupposing material of the next section

```
type
   B, C, ..., D
   A = B × C × ... × D
```

A.2 Type Definitions

A.2.1 Concrete Types

- Types can be concrete
- in which case the structure of the type is specified by type expressions:

```
type
   A = Type_expr
```

- Some schematic type definitions are:

```
[1]  Type_name = Type_expr /∗ without |s or subtypes ∗/
[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 combining the types:

```
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)
```

Types A, B, ..., Z are disjoint, i.e., shares no values, provided all mk_id_k are distinct and due to the use of the disjoint record type constructor ==.

```
axiom
   ∀ a1:A_1, a2:A_2, ..., ai:Ai •
      s_a1(mk_id_1(a1,a2,...,ai))=a1 ∧ s_a2(mk_id_1(a1,a2,...,ai))=a2 ∧
      ... ∧ s_ai(mk_id_1(a1,a2,...,ai))=ai ∧
   ∀ a:A • let mk_id_1(a1′,a2′,...,ai′) = a in
      a1′ = s_a1(a) ∧ a2′ = s_a2(a) ∧ ... ∧ ai′ = s_ai(a) end
```

A.2.2 Subtypes

- In RSL, each type represents a set of values. Such a set can be delimited by means of predicates.
- The set of values b which have type B and which satisfy the predicate 𝒫, constitute the subtype A:

```
type
   A = {| b:B • 𝒫(b) |}
```

A.2.3 Sorts — Abstract Types

- Types can be (abstract) sorts
- in which case their structure is not specified:

```
type
   A, B, ..., C
```

A.3 The RSL Predicate Calculus

A.3.1 Propositional Expressions

- Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values (**true** or **false** [or **chaos**]).

- Then:

false, **true**
a, b, ..., c $\sim$a, a$\wedge$b, a$\vee$b, a$\Rightarrow$b, a$=$b, a$\neq$b

- are propositional expressions having Boolean values.
- $\sim$, $\wedge$, $\vee$, $\Rightarrow$, $=$ and $\neq$ are Boolean connectives (i.e., operators).
- They can be read as: *not*, *and*, *or*, *if then* (or *implies*), *equal* and *not equal*.

A.3.2 Simple Predicate Expressions

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

false, **true**
a, b, ..., c
$\sim$a, a$\wedge$b, a$\vee$b, a$\Rightarrow$b, a$=$b, a$\neq$b
x$=$y, x$\neq$y,
i$<$j, i$\leq$j, i$\geq$j, i$\neq$j, i$\geq$j, i$>$j

- are simple predicate expressions.

A.3.3 Quantified Expressions

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

$\forall$ x:X • $\mathscr{P}(x)$
$\exists$ y:Y • $\mathscr{Q}(y)$
$\exists$! z:Z • $\mathscr{R}(z)$

- are quantified expressions — also being predicate expressions.

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

A.4 Concrete RSL Types: Values and Operations

A.4.1 Arithmetic

```
type
   Nat, Int, Real
value
   +,−,∗: Nat×Nat→Nat | Int×Int→Int | Real×Real→Real
   /: Nat×Nat→̃Nat | Int×Int→̃Int | Real×Real→̃Real
   <,≤,=,≠,≥,> (Nat|Int|Real) → (Nat|Int|Real)
```

A.4.2 Set Expressions

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

$$\{\{\}, \{a\}, \{e_1,e_2,...,e_n\}, ...\} \in A\textbf{-set}$$
$$\{\{\}, \{a\}, \{e_1,e_2,...,e_n\}, ..., \{e_1,e_2,...\}\} \in A\textbf{-infset}$$

Set Comprehension

- The expression, last line below, to the right of the ≡, expresses set comprehension.
- The expression "builds" the set of values satisfying the given predicate.
- It is abstract in the sense that it does not do so by following a concrete algorithm.

```
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)}
```

A.4.3 Cartesian Expressions

Cartesian Enumerations

- Let *e* range over values of Cartesian types involving *A*, *B*, ..., *C*,
- then the below expressions are simple Cartesian enumerations:

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

A.4.4 List Expressions

List Enumerations

- Let a range over values of type A,
- then the below expressions are simple list enumerations:

{⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ...} ∈ A^*
{⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ..., ⟨e1,e2,...,en,... ⟩, ...} ∈ A^ω

⟨ a_i .. a_j ⟩

- The last line above assumes a_i and a_j to be integer-valued expressions.
- It then expresses the set of integers 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.

List Comprehension

- The last line below expresses list comprehension.

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

A.4.5 Map Expressions

Map Enumerations

- Let (possibly indexed) u and v range over values of type $T1$ and $T2$, respectively,
- then the below expressions are simple map enumerations:

type
  T1, T2
  M = T1 $\overrightarrow{m}$ T2
value
  u,u1,u2,...,un:T1, v,v1,v2,...,vn:T2
  [], [u↦v], ..., [u1↦v1,u2↦v2,...,un↦vn] ∀ ∈ M

Map Comprehension

- The last line below expresses map comprehension:

```
type
    U, V, X, Y
    M = U $\overrightarrow{m}$ V
    F = U $\overset{\sim}{\rightarrow}$ X
    G = V $\overset{\sim}{\rightarrow}$ Y
    P = U → Bool
value
    comprehend: M×F×G×P → (X $\overrightarrow{m}$ Y)
    comprehend(m,F,G,P) ≡
        [ F(u) ↦ G(m(u)) | u:U • u ∈ dom m ∧ P(u)]
```

A.4.6 Set Operations

Set Operator Signatures

```
value
    19 ∈: A × A-infset → Bool
    20 ∉: A × A-infset → Bool
    21 ∪: A-infset × A-infset → A-infset
    22 ∪: (A-infset)-infset → A-infset
    23 ∩: A-infset × A-infset → A-infset
    24 ∩: (A-infset)-infset → A-infset
    25 \: A-infset × A-infset → A-infset
    26 ⊂: A-infset × A-infset → Bool
    27 ⊆: A-infset × A-infset → Bool
    28 =: A-infset × A-infset → Bool
    29 ≠: A-infset × A-infset → Bool
    30 card: A-infset $\overset{\sim}{\rightarrow}$ Nat
```

Set Examples

```
examples
    a ∈ {a,b,c}
    a ∉ {}, a ∉ {b,c}
    {a,b,c} ∪ {a,b,d,e} = {a,b,c,d,e}
    ∪{{a},{a,b},{a,d}} = {a,b,d}
    {a,b,c} ∩ {c,d,e} = {c}
    ∩{{a},{a,b},{a,d}} = {a}
    {a,b,c} \ {c,d} = {a,b}
    {a,b} ⊂ {a,b,c}
```

$\{a,b,c\} \subseteq \{a,b,c\}$
$\{a,b,c\} = \{a,b,c\}$
$\{a,b,c\} \neq \{a,b\}$
card $\{\} = 0$, **card** $\{a,b,c\} = 3$

Informal Explication

19 $\in$: The membership operator expresses that an element is a member of a set.

20 $\notin$: The nonmembership operator expresses that an element is not a member of a set.

21 $\cup$: The infix union operator. When applied to two sets, the operator gives the set whose members are in either or both of the two operand sets.

22 $\cup$: The distributed prefix union operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.

23 $\cap$: The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.

24 $\cap$: The prefix distributed intersection operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.

25 $\setminus$: The set complement (or set subtraction) operator. When applied to two sets, the operator gives the set whose members are those of the left operand set which are not in the right operand set.

26 $\subseteq$: The proper subset operator expresses that all members of the left operand set are also in the right operand set.

27 $\subset$: The proper subset operator expresses that all members of the left operand set are also in the right operand set, and that the two sets are not identical.

28 $=$: The equal operator expresses that the two operand sets are identical.

29 $\neq$: The nonequal operator expresses that the two operand sets are *not* identical.

30 **card**: The cardinality operator gives the number of elements in a finite set.

Set Operator Definitions The operations can be defined as follows ($\equiv$ is the definition symbol):

value
$s' \cup s'' \equiv \{\ a \mid a{:}A \bullet a \in s' \vee a \in s''\ \}$
$s' \cap s'' \equiv \{\ a \mid a{:}A \bullet a \in s' \wedge a \in s''\ \}$
$s' \setminus s'' \equiv \{\ a \mid a{:}A \bullet a \in s' \wedge a \notin s''\ \}$
$s' \subseteq s'' \equiv \forall\ a{:}A \bullet a \in s' \Rightarrow a \in s''$
$s' \subset s'' \equiv s' \subseteq s'' \wedge \exists\ a{:}A \bullet a \in s'' \wedge a \notin s'$
$s' = s'' \equiv \forall\ a{:}A \bullet a \in s' \equiv a \in s'' \equiv s{\subseteq}s' \wedge s'{\subseteq}s$

```
    $s' \neq s'' \equiv s' \cap s'' \neq \{\}$
    card s $\equiv$
        if s = {} then 0 else
        let a:A • $a \in s$ in 1 + card (s \ {a}) end end
        pre s /* is a finite set */
    card s $\equiv$ chaos /* tests for infinity of s */
```

A.4.7 Cartesian Operations

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

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

decomposition 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
```

A.4.8 List Operations

List Operator Signatures

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

List Operation Examples

```
examples
    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⟩⁀⟨a,b,d⟩ = ⟨a,b,c,a,b,d⟩
    ⟨a,b,c⟩=⟨a,b,c⟩
    ⟨a,b,c⟩ ≠ ⟨a,b,d⟩
```

Informal Explication

- **hd**: Head gives the first element in a nonempty list.
- **tl**: Tail gives the remaining list of a nonempty list when Head is removed.
- **len**: Length gives the number of elements in a finite list.
- **inds**: Indices give the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems**: Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$: Indexing with a natural number, i larger than 0, into a list ℓ having a number of elements larger than or equal to i, gives the ith element of the list.
- $\frown$: Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.
- $=$: The equal operator expresses that the two operand lists are identical.
- $\neq$: The nonequal operator expresses that the two operand lists are *not* identical.

The operations can also be defined as follows:

List Operator Definitions

```
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″)
```

A.4.9 Map Operations

Map Operator Signatures and Map Operation Examples

```
value
    m(a): M → A →̃ B, m(a) = b

    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‴]
```

$$\backslash:\ \mathsf{M} \times \mathsf{A}\textbf{-infset} \rightarrow \mathsf{M}\ [\mathsf{restriction\ by}]$$
$$[\mathsf{a}\mapsto\mathsf{b},\mathsf{a}'\mapsto\mathsf{b}',\mathsf{a}''\mapsto\mathsf{b}'']\backslash\{\mathsf{a}\} = [\mathsf{a}'\mapsto\mathsf{b}',\mathsf{a}''\mapsto\mathsf{b}'']$$

$$/:\ \mathsf{M} \times \mathsf{A}\textbf{-infset} \rightarrow \mathsf{M}\ [\mathsf{restriction\ to}]$$
$$[\mathsf{a}\mapsto\mathsf{b},\mathsf{a}'\mapsto\mathsf{b}',\mathsf{a}''\mapsto\mathsf{b}'']/\{\mathsf{a}',\mathsf{a}''\} = [\mathsf{a}'\mapsto\mathsf{b}',\mathsf{a}''\mapsto\mathsf{b}'']$$

$$=,\neq:\ \mathsf{M} \times \mathsf{M} \rightarrow \textbf{Bool}$$

$$^\circ:\ (\mathsf{A} \xrightarrow[m]{} \mathsf{B}) \times (\mathsf{B} \xrightarrow[m]{} \mathsf{C}) \rightarrow (\mathsf{A} \xrightarrow[m]{} \mathsf{C})\ [\mathsf{composition}]$$
$$[\mathsf{a}\mapsto\mathsf{b},\mathsf{a}'\mapsto\mathsf{b}'] \circ [\mathsf{b}\mapsto\mathsf{c},\mathsf{b}'\mapsto\mathsf{c}',\mathsf{b}''\mapsto\mathsf{c}''] = [\mathsf{a}\mapsto\mathsf{c},\mathsf{a}'\mapsto\mathsf{c}']$$

Map Operation Explication

- $m(a)$: Application gives the element that a maps to in the map m.
- **dom**: Domain/Definition Set gives the set of values which *maps to* in a map.
- **rng**: Range/Image Set gives the set of values which *are mapped to* in a map.
- †: Override/Extend. When applied to two operand maps, it gives the map which is like an override of the left operand map by all or some "pairings" of the right operand map.
- ∪: Merge. When applied to two operand maps, it gives a merge of these maps.
- \: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements that are not in the right operand set.
- /: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements of the right operand set.
- =: The equal operator expresses that the two operand maps are identical.
- ≠: The nonequal operator expresses that the two operand maps are *not* identical.
- °: Composition. When applied to two operand maps, it gives the map from definition set elements of the left operand map, m_1, to the range elements of the right operand map, m_2, such that if a is 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.

Map Operation Redefinitions The map operations can also be defined as follows:

value
$$\textbf{rng}\ \mathsf{m} \equiv \{\ \mathsf{m(a)} \mid \mathsf{a{:}A} \bullet \mathsf{a} \in \textbf{dom}\ \mathsf{m}\ \}$$

$$\mathsf{m1} \dagger \mathsf{m2} \equiv$$
$$[\ \mathsf{a}\mapsto\mathsf{b} \mid \mathsf{a{:}A,b{:}B} \bullet$$
$$\mathsf{a} \in \textbf{dom}\ \mathsf{m1} \setminus \textbf{dom}\ \mathsf{m2} \wedge \mathsf{b{=}m1(a)} \vee \mathsf{a} \in \textbf{dom}\ \mathsf{m2} \wedge \mathsf{b{=}m2(a)}\ ]$$

```
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
```

A.5 λ-Calculus + Functions

A.5.1 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), ...
```

A.5.2 Free and Bound Variables

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

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

A.5.3 Substitution

In RSL, the following rules for substitution apply:

- **subst**([N/x]x) ≡ N;

- $\mathbf{subst}([N/x]a) \equiv a$,

  for all variables $a \neq x$;
- $\mathbf{subst}([N/x](P\ Q)) \equiv (\mathbf{subst}([N/x]P)\ \mathbf{subst}([N/x]Q))$;
- $\mathbf{subst}([N/x](\lambda x \bullet P)) \equiv \lambda\ y \bullet P$;
- $\mathbf{subst}([N/x](\lambda\ y \bullet P)) \equiv \lambda y \bullet\ \mathbf{subst}([N/x]P)$,

  if $x \neq y$ and y is not free in N or x is not free in P;
- $\mathbf{subst}([N/x](\lambda y \bullet P)) \equiv \lambda z \bullet \mathbf{subst}([N/z]\mathbf{subst}([z/y]P))$,

  if $y \neq x$ and y is free in N and x is free in P

  (where z is not free in (N P)).

A.5.4 α-Renaming and β-Reduction

- α-renaming: $\lambda x \bullet M$

  If x, y are distinct variables then replacing x by y in $\lambda x \bullet M$ results in $\lambda y \bullet \mathbf{subst}([y/x]M)$. 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: $(\lambda x \bullet M)(N)$

  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. $(\lambda x \bullet M)(N) \equiv \mathbf{subst}([N/x]M)$

A.5.5 Function Signatures

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
```

A.5.6 Function Definitions

Functions can be defined explicitly:

```
value
   f: Arguments → Result
   f(args) ≡ DValueExpr
```

```
  g: Arguments ⥲ Result
  g(args) ≡ ValueAndStateChangeClause
  pre P(args)
```

Or functions can be defined implicitly:

```
value
  f: Arguments → Result
  f(args) as result
  post P1(args,result)

  g: Arguments ⥲ Result
  g(args) as result
  pre P2(args)
  post P3(args,result)
```

The symbol $\overset{\sim}{\to}$ 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.

A.6 Other Applicative Expressions

A.6.1 Simple let Expressions

Simple (i.e., nonrecursive) **let** expressions:

$$\textbf{let}\ \mathsf{a} = \mathscr{E}_d\ \textbf{in}\ \mathscr{E}_b(\mathsf{a})\ \textbf{end}$$

is an "expanded" form of:

$$(\lambda \mathsf{a}.\mathscr{E}_b(\mathsf{a}))(\mathscr{E}_d)$$

A.6.2 Recursive let Expressions

Recursive **let** expressions are written as:

$$\textbf{let}\ \mathsf{f} = \lambda \mathsf{a}{:}\mathsf{A} \bullet \mathsf{E}(\mathsf{f})\ \textbf{in}\ \mathsf{B}(\mathsf{f},\mathsf{a})\ \textbf{end}$$

is "the same" as:

$$\textbf{let}\ \mathsf{f} = \textbf{Y}\mathsf{F}\ \textbf{in}\ \mathsf{B}(\mathsf{f},\mathsf{a})\ \textbf{end}$$

where:

$$\mathsf{F} \equiv \lambda \mathsf{g}\bullet\lambda \mathsf{a}\bullet(\mathsf{E}(\mathsf{g}))\ \text{and}\ \textbf{Y}\mathsf{F} = \mathsf{F}(\textbf{Y}\mathsf{F})$$

A.6.3 Predicative let Expressions

Predicative **let** expressions:

```
let a:A • $\mathscr{P}$(a) in $\mathscr{B}$(a) end
```

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

A.6.4 Pattern and "Wild Card" let Expressions

Patterns and *wild cards* can be used:

```
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
```

A.6.5 Conditionals

Various kinds of conditional expressions are offered by RSL:

```
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_expr_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
```

A.6.6 Operator/Operand Expressions

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

A.7 Imperative Constructs

A.7.1 Statements and State Changes

Often, following the RAISE method, software development starts with highly abstract-applicative constructs which, through stages of refinements, are turned into concrete and imperative constructs. Imperative constructs are thus inevitable in RSL.

```
    Unit
value
    stmt: Unit → Unit
    stmt()
```

- Statements accept no arguments.
- Statement execution changes the state (of declared variables).
- **Unit** → **Unit** designates a function from states to states.
- Statements, stmt, denote state-to-state changing functions.
- Writing () as “only” arguments to a function “means” that () is an argument of type **Unit**.

A.7.2 Variables and Assignment

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

A.7.3 Statement Sequences and skip

Sequencing is expressed using the ';' operator. **skip** is the empty statement having no value or side-effect.

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

A.7.4 Imperative 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**

A.7.5 Iterative Conditionals

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

A.7.6 Iterative Sequencing

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

A.8 Process Constructs

A.8.1 Process Channels

Let A and B stand for two types of (channel) messages and i:KIdx for channel array indexes, then:

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

declare a channel, c, and a set (an array) of channels, k[i], capable of communicating values of the designated types (A and B).

A.8.2 Process Composition

- 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,
- thereby communicating over declared channels.
- Let P() and Q stand for process expressions, then:

P ∥ Q	Parallel composition
P ⌷ Q	Nondeterministic external choice (either/or)
P ⊓ Q	Nondeterministic internal choice (either/or)
P ╫ Q	Interlock parallel composition

express the parallel (∥) of two processes, or the nondeterministic choice between two processes: either external (⌷) or internal (⊓). The interlock (╫) composition expresses that the two processes are forced to communicate only with one another, until one of them terminates.

A.8.3 Input/Output Events

Let c, k[i] and e designate channels of type A and B, then:

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

- expresses the willingness of a process to engage in an event that
  - “reads” an input, respectively
  - “writes” an output.

A.8.4 Process Definitions

The below signatures are just examples. They emphasise that process functions must somehow express, in their signature, 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.

A.9 Simple RSL Specifications

Often, we do not want to encapsulate small specifications in schemes, classes, and objects, as is often done in RSL. An RSL specification is simply a sequence of one or more types, values (including functions), variables, channels and axioms:

type
  ...
variable
  ...

```
channel
    ...
value
    ...
axiom
    ...
```

In practice a full specification repeats the above listings many times, once for each "module" (i.e., aspect, facet, view) of specification. Each of these modules may be "wrapped" into scheme, class or object definitions.