The TSE Trading Rules

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

January 28, 2010

The reason why these notes are written is the appearance of [1].
I have taken the liberty of including that paper in this document, cf. Appendix C.
I had the good fortune of visiting Prof. Tetsuo Tamai, Tokyo Univ., 8–Dec.8, 2009.
I read [1] late November.
I then had wished that Tetsuo had given it to me upon my arrival.
I was, obviously ignorant of its publication some five months earlier.
I have now reread [1] (late January 2010).

I mentioned to Tetsuo that I would try my hand on a formalisation.
A description, both by a narrative, and by related formulas.

What you see here, in Chap. 1, is a first attempt[1].

At present (January 28, 2010) Chap. 2 is not written.
I have included some notes, Appendix A.
Their origin goes back to December 1996.
Appendix Sect. A.4 should be of particular relevance to Chaps. 1 and 2.
A few, smaller updates, were added till about 2004.

[1] Earlier versions of this document will have Chap. 1 being very incomplete.

Contents

Chapter 1

The Tokyo Stock Exchange

This chapter was begun on January 24. It is being released, first time, January 28.

1.1 Introduction

This chapter shall try describe: narrate and formalise some facets of the (now "old"[1]) stock trading system of the TSE: Tokyo Stock Exchange (especially the 'matching' aspects).

1.2 The Problem

The reason that I try tackle a description (albeit of the "old" system) is that Prof. Tetsuo Tamai published a delightful paper [1, IEEE Computer Journal, June 2009 (vol. 42 no. 6) pp. 58-65)], Social Impact of Information Systems, in which a rather sad story is unfolded: a human error key input: an offer for selling stocks, although "ridiculous" in its input data (*"sell 610 thousand stocks, each at one (1) Japanese Yen"*, whereas one stock at 610,000 JPY was meant), and although several immediate — within seconds — attempts to cancel this "order", could not be cancelled ! This lead to a loss for the selling broker at around 42 Billion Yen, at today's exchange rate, 26 Jan. 2010, 469 million US $s ![2] Prof. Tetsuo Tamai's paper gives a, to me, chilling account of what I judge as an extremely sloppy and irresponsible design process by TSE and Fujitsu. It also leaves, I think, a strong impression of arrogance on the part of TSE. This arrogance, I claim, is still there in the documents listed in Footnote 1.

So the problem is a threefold one of

[1] We write "old" since, as of January 4, 2010, that 'old' stock trading system has been replaced by the so-called arrowhead system. We refer to the following documents:

- `http://www.tse.or.jp/english/rules/equities/arrowhead/pamphlet.html`
- `http://www.tse.or.jp/english/rules/equities/arrowhead/pamphlet-e.pdf`
- `http://www.tse.or.jp/english/rules/equities/arrowhead/pamphlet1e.pdf`
  We have reproduced the "points to note when placing orders" in Appendix Sect. C.2 (Pages 71–74).
- `http://www.tse.or.jp/english/rules/equities/arrowhead/pamphlet2e.html`
  We have reproduced the "points to note when placing orders" in Appendix Sect. C.1 (Pages 67–70).

[2] So far three years of law court case hearing etc., has, on Dec. 4, 2009, resulted in complainant being awarded 10.7 billion Yen in damages. See `http://www.ft.com/cms/s/0/e9d89050-e0d7-11de-9f58--00144feab49a.html`.

- **Proper Requirements:** How does one (in this case a stock exchange) prescribe (to the software developer) what is required by an appropriate hardware/software system for, as in this case, stock handling: acceptance of buy bids and sell offers, the possible withdrawal (or cancellation) of such submitted offers, and their matching (i.e., the actual trade whereby buy bids are marched in an appropriate, clear and transparent manner).
- **Correctness of Implementation:** Hhow does one make sure that the software/hardware system meets customers' expectations.
- **Proper Explanation to Lay Users:** How does one explain, to the individual and institutional customers of the stock exchange, those offering stocks for sale of bids for buying stocks – how does one explain – in a clear and transparent manner the applicable rules governing stock handling.[3]

I shall only try contribute, in this document, to a solution to the first of these sub-problems.

1.3 A Domain Description

1.3.1 Market and Limit Offers and Bids

1. A market sell offer or buy bid specifies
   (a) the unique identification of the stock,
   (b) the number of stocks to be sold or bought, and
   (c) the unique name of the seller.
2. A limit sell offer or buy bid specifies the same information as a market sell offer or buy bid (i.e., Items 1a–1c), and
   (d) the price at which the identified stock is to be sold or bought.
3. A trade order is either a (mkMkt marked) market order or (mkLim marked) a limit order.
4. A trading command is either a sell order or a buy bid.
5. The sell orders are made unique by the mkSell "make" function.
6. The buy orders are made unique by the mkBuy "make" function.

type
1 Market = Stock_id × Noumber_of_Stocks × Name_of_Customer
1a Stock_id
1b Number_of_Stocks = {|n•n:**Nat**∧n>1|}
1c Name_of_Customer
2 Limit = Market × Price
2d Price = {|n•n:**Nat**∧n>1|}

[3]The rules as explained in the Footnote 1 on the previous page listed documents are far from clear and transparent: they are full of references to fast computers, overlapping processing, etc., etc.: matters with which these buying and selling customers should not be concerned — so, at least, thinks this author !

3 Trade == mkMkt(m:Market) | mkLim(l:Limit)
4 Trading_Command = Sell_Order | Buy_Bid
5 Sell_Order == mkSell(t:Trade)
6 Buy_Bid == mkBuy(t:Trade)

1.3.2 Order Books

7. We introduce a concept of linear, discrete time.
8. For each stock the stock exchange keeps an order book.
9. An order book for stock $s_{id} : SI$ keeps track of limit buy bids and limit sell offers (for the *id*entified *s*tock, s_{id}), as well as the market buy bids and sell offers; that is, for each price
   (d) the number stocks, by unique order number, offered for sale at that price, that is, limit sell orders, and
   (e) the number of stocks, by unique order number, bid for buying at that price, that is, limitbuy bid orders;
   (f) if an offer is a market sell offer, then the number of stocks to be sold is recorded, and if an offer is a market buy bid (also an offer), then the number of stocks to be bought is recorded,
10. Over time the stock exchange displays a series of full order books.
11. A trade unit is a pair of a unique order number and an amount (a number larger than 0) of stocks.
12. An amount designates a number of one or more stocks.

type
7 T
8 All_Stocks_Order_Book = Stock_Id $\xrightarrow{m}$ Stock_Order_Book
9 Stock_Order_Book = (Price $\xrightarrow{m}$ Orders) × Market_Offers
9 Orders:: so:Sell_Orders × bo:Buy_Bids
9d Sell_Orders = On $\xrightarrow{m}$ Amount
9e Buy_Bids = On $\xrightarrow{m}$ Amount
9f Market_Offers :: mkSell(n:**Nat**) × mkBuy(n:**Nat**)
10 TSE = T $\xrightarrow{m}$ All_Stocks_Order_Book
11 TU = On × Amount
12 Amount = {|n•**Nat**∧n≥1|}

1.3.3 Aggregate Offers

13. We introduce the concepts of aggregate sell and buy orders for a given stock at a given price (and at a given time).
14. The aggregate sell orders for a given stock at a given price is

(g) the stocks being market sell offered and

(h) the number of stocks being limit offered for sale at that price or lower

15. The aggregate bur bids for a given stock at a given price is

(i) including the stocks being market bid offered and

(j) the number of stocks being limit bid for buying at that price or higher

```
value
  14 aggr_sell: All_Stocks_Order_Book × Stock_Id × Price → Nat
  14 aggr_sell(asob,sid,p) ≡
  14   let ((sos,_),(mkSell(ns),_)) = asob(sid) in
  14g    ns +
  14h    all_sell_summation(sos,p) end
  15 aggr_buy: All_Stocks_Order_Book × Stock_Id × Price → Nat
  15 aggr_buy(asob,sid,p) ≡
  15   let ((_,bbs),(_,mkBuy(nb))) = asob(sid) in
  15i    nb +
  15j    nb + all_buy_summation(bbs,p) end

  all_sell_summation: Sell_Orders × Price → Nat
  all_sell_summation(sos,p) ≡
    let ps = {p′|p′:Prices • p′ ∈ dom sos ∧ p′ ≥ p} in accumulate(sos,ps)(0) end

  all_buy_summation: Buy_Bids × Price → Nat
  all_buy_summation(bbs,p) ≡
    let ps = {p′|p′:Prices • p′ ∈ dom bos ∧ p′ ≤ p} in accumulate(bbs,ps)(0) end
```

The auxiliary accumulate function is shared between the all_sell_summation and the all_buy_summation functions. It sums the amounts of limit stocks in the price range of the accumulate function argument ps. The auxiliary sum function sums the amounts of limit stocks — "pealing off" the their unique order numbers.

```
value
  accumulate: (Price m→ Orders) × Price-set → Nat → Nat
  accumulate(pos,ps)(n) ≡
    case ps of {} → n, {p}∪ ps′ → accumulate(pos,ps′)(n+sum(pos(p)){dom pos(p)}) end

  sum: (Sell_Orders|Buy_Bids) → On-set → Nat
  sum(ords)(ns) ≡
    case ns of {} → 0, {n}∪ ns′ → ords(n)+sum(ords)(ns′) end
```

To handle the sub_limit_sells and sub_limit_buys indicated by Item 17c on the facing page of the Itayose "algorithm" we need the corresponding sub_sell_summation and sub_buy_summation functions:

```
value
```

```
sub_sell_summation: Stock_Order_Book × Price → Nat
sub_sell_summation(((sos,_),(ns,_)),p) ≡ ns +
  let ps = {p′|p′:Prices • p′ ∈ dom sos ∧ p′ > p} in accumulate(sos,ps)(0) end

sub_buy_summation: Stock_Order_Book × Price → Nat
sub_buy_summation(((_,bbs),(_,nb)),p) ≡ nb +
  let ps = {p′|p′:Prices • p′ ∈ dom bos ∧ p′ < p} in accumulate(bbs,ps)(0) end
```

1.3.4 The TSE Itayose "Algorithm"

16. The TSE practices the so-called Itayose "algorithm" to decide on opening and closing prices[4]. That is, the Itayose "algorithm" determines a single so-called 'execution' price, one that matches sell and buy orders[5]:

17. The "matching sell and buy orders" rules:

    (a) *All market orders must be 'executed'*[6].

    (b) *All limit orders to sell/buy at prices lower/higher than the 'execution price'*[7] *must be executed.*

    (c) *The following amount of limit orders to sell or buy at the execution prices must be executed: the entire amount of either all sell or all buy orders, and at least one 'trading unit'*[8] *from 'the opposite side of the order book'*[9].

```
value
  17 match: All_Stocks_Order_Book × Stock_Id → Price-set
  17 match(asob,sid) as ps
  17   pre: sid ∈ dom asob
  17   post: ∀ p′:Price • p′ ∈ ps ⇒
  17′          ∃ os:On-set •
  17a′                    market_buys(asob(sid))
  17b′                  + sub_limit_buys(asob(sid))(p′)
  17c′                  + all_priced_buys(asob(sid))(p′)
  17a′                =   market_sells(asob(sid))
  17b′                  + sub_limit_sells(asob(sid))(p′)
  17c′                  + some_priced_buys(asob(sid))(p′)(os) ∨
  17″          ∃ os:On-set •
  17a″                    market_buys(asob(sid))
  17b″                  + sub_limit_buys(asob(sid))(p′)
  17c″                  + some_priced_buys(asob(sid))(p′)(os)
  17a″                =   market_sells(asob(sid))
```

[4] [1, pp 59, col. 1, lines 4-3 from bottom, cf. Page 59]

[5] [1, pp 59, col. 2, lines 1–3 and Items 1.–3. after yellow, four line 'insert', cf. Page 59] These items 1.–3. are reproduced as "our" Items 17a–17c.

[6] To execute an order:

[7] Execution price:

[8] Trading unit:

[9] The opposite side of the order book:

17b″ + sub_limit_sells(asob(sid))(p′)
17c″ + all_priced_buys(asob(sid))(p′) ∨

The match function calculates a set of prices for each of which a match can be made. The set may be empty: there is no price which satisfies the match rules (cf. Items 17a–17c below). The set may be a singleton set: there is a unique price which satisfies match rules Items 17a–17c. The set may contain more than one price: there is not a unique price which satisfies match rules Items 17a–17c. The single (′) and the double (″) quoted (17a–17c) group of lines, in the match formulas above, correspond to the Itayose "algorithm"s Item 17c *'opposite sides of the order book'* description. The existential quantification of a set of order numbers of lines 17′ and 17″ correspond to that "algorithms" (still Item 17c) point of *at least one 'trading unit'*. It may be that the **post** condition predicate is only fullfilled for all trading units – so be it.

```
value
  market_buys: Stock_Order_Book → Amount
  market_buys((_,(_,mkBuys(nb))),p) ≡ nb

  market_sells: Stock_Order_Book → Amount
  market_sells((_,(mkSells(ns),_)),p) ≡ ns

  sub_limit_buys: Stock_Order_Book → Price → Amount
  sub_limit_buys(((,bbs),_))(p) ≡ sub_buy_summation(bbs,p)

  sub_limit_sells: Stock_Order_Book → Price → Amount
  sub_limit_sells((sos,_))(p) ≡ sub_sell_summation(sos,p)

  all_priced_buys: Stock_Order_Book → Price → Amount
  all_priced_buys((_,bbs),_)(p) ≡ sum(bbs(p))

  all_priced_sells: Stock_Order_Book → Price → Amount
  all_priced_sells((sos,_),_)(p) ≡ sum(sos(p))

  some_priced_buys: Stock_Order_Book → Price → On-set → Amount
  some_priced_buys((_,bbs),_)(p)(os) ≡
    let tbs = bbs(p) in if {}≠os∧os⊆dom tbs then sum(tbs)(os) else 0 end end

  some_priced_sells: Stock_Order_Book → Price → On-set → Amount
  some_priced_sells((sos,_),_)(p)(os) ≡
    let tss = sos(p) in if {}≠os∧os⊆dom tss then sum(tss)(os) else 0 end end
```

The formalisation of the Itayise "algorithm", as well as that "algorithm" [itself], does not guarantee a match where a match "ought" be possible. The "stumbling block" seems to be the Itayose "algorithm"s Item 17c. There it says: *'at least one trading unit'*. We suggest that a match could be made in which some of the stocks of a candidate trading unit be matched with the remaining stocks also being traded, but now with the stock exchange being the buyer and with the stock exchange immediately "turning around" and posting those remaining stocks as a TSE marked trading unit for sale.

Much more to come: essentially I have only modelled column 2, rightmost column, Page 59 of [1, Tetsuo Tamai, "TSE"]. Next to be modelled is column 1, leftmost column, Page 60 of [1]. See these same page numbers of the present document !

1.3.5 Match Executions

```
to come
```

1.3.6 Order Handling

```
to come
```

Chapter 2

The New Tokyo Stock Exchange

to come

Chapter 3

Bibliographical Notes

Bibliography

[1] T. Tamai. Social Impact of Information System Failures. *Computer, IEEE Computer Society Journal*, 42(6):58–65, June 2009.

Appendix A

Some 1996–1999 Models of "Abstracted" 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. ∎

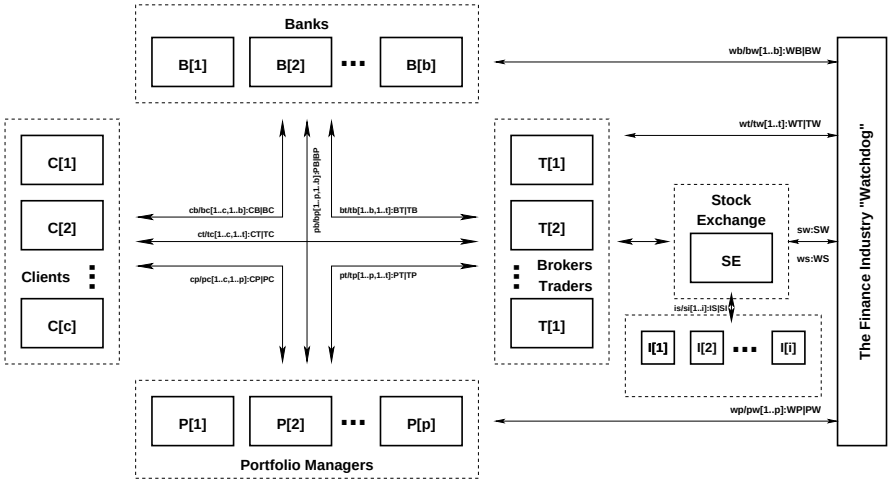


Figure A.1: A financial behavioural system abstraction

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.

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.

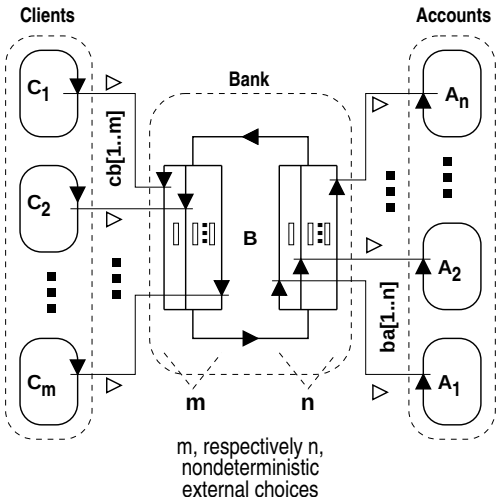


Figure A.2: A fifth schematic "rendezvous" class

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.

```
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)


  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()
```

[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.

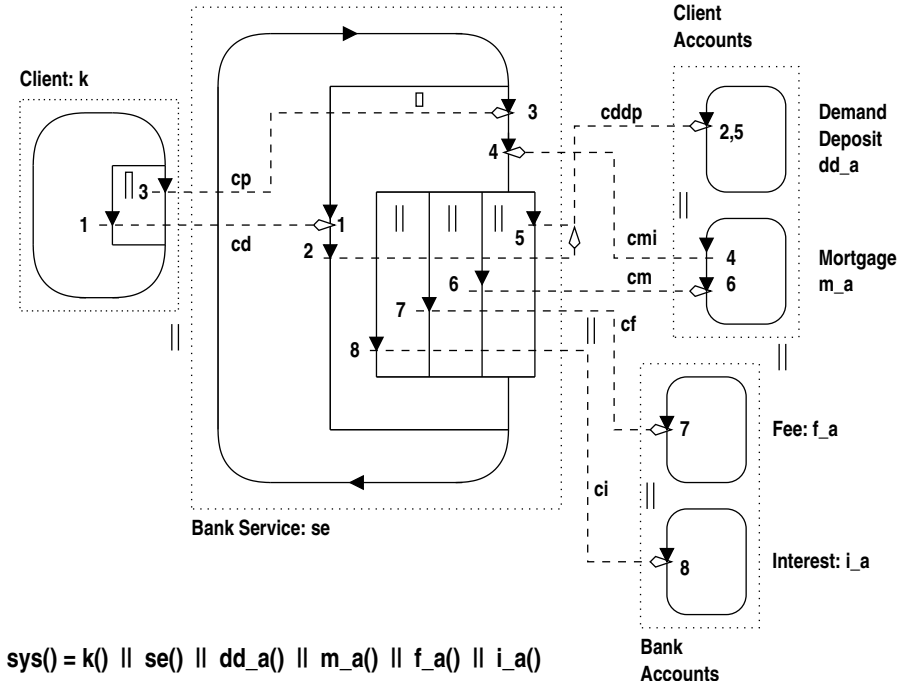


Figure A.3: A loan repayment scenario

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. ▪

[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.

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. 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

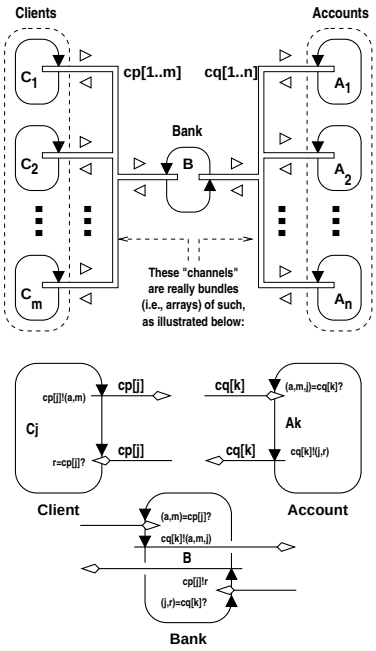


Figure A.4: Two-way component interaction

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 account. 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 = Γ × Σ
[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.

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

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

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.

18. *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".

19. *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.

20. *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

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 $\overrightarrow{m}$ A-**set**
  Accounts = A $\overrightarrow{m}$ Balance

M_Register = C $\overrightarrow{m}$ M-**set**
Loans = M $\overrightarrow{m}$ (Loan × Date)
Loan,Balance = P
P = **Nat**

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

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.

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.

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.

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.

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.

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
```

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.

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.

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

```
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

```
  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

```
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

```
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

```
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
```

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

```
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

```
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 32, 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.

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:*

Semantics of Bank Script Language

We now give semantics to the bank script language of Example A.10 on page 36.

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 $\overrightarrow{m}$ A-**set**
  Accounts = A $\overrightarrow{m}$ Balance
  M_Register = C $\overrightarrow{m}$ M-**set**
  Loans = M $\overrightarrow{m}$ (Loan × Date)
  Loan,Balance = P
  P = **Nat**
  Σ = (V $\overrightarrow{m}$ (C|A|M|P|I)) $\bigcup$ (Fn $\overrightarrow{m}$ 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: (Σ×Check×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) ≡
  case t of
    mkAN(a) → (σ,true,(ρ,α†[ a↦α(σ(a))+σ(p) ],μ,ℓ))
    mkMN(m) → (σ,true,(ρ,α,μ,ℓ†[ σ(m)↦ℓ(σ(m))+σ(p) ]))
    bank_i → (σ,true,(ρ,α†[ a_i↦α(a_i)+σ(p) ],μ,ℓ))
    bank_c →  (σ,true,(ρ,α†[ a_ℓ↦α(a_ℓ)+σ(p) ],μ,ℓ))
  end
```

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

```
Int_Clause(mkS(p,t))(σ)(tf)(ρ,α,μ,ℓ) ≡
  case t of
    mkAN(a) → (σ,true,(ρ,α†[ σ(a)↦α(σ(a))−σ(p) ],μ,ℓ))
    mkMN(m) → (σ,true,(ρ,α,μ,ℓ†[ σ(m)↦ℓ(σ(m))−σ(p) ]))
    bank_i → (σ,true,(ρ,α†[ a_i↦α(a_i)−σ(p) ],μ,ℓ))
    bank_c →  (σ,true,(ρ,α†[ a_ℓ↦α(a_ℓ)−σ(p) ],μ,ℓ))
  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) ≡
  let pi = σ(p)−σ(i) in
  case t of
    mkAN(a) → (σ,true,(ρ,α†[σ(a)↦α(σ(a))−pi],μ,ℓ))
    mkMN(m) → (σ,true,(ρ,α,μ,ℓ†[σ(m)↦ℓ(σ(m))−pi]))
    bank_i → (σ,true,(ρ,α†[a_i↦α(a_i)−pi],μ,ℓ))
    bank_c → (σ,true,(ρ,α†[a_ℓ↦α(a_ℓ)−pi],μ,ℓ))
  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) ≡
  (σ\{a},true,(ρ†[σ(c)↦α(σ(c))\{σ(a)}],α\{σ(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.13. ∎

A.2.5 A Student Exercise

Example A.13 *Financial Service Industry Business Processes: 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 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

[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.

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.
- **Support technology** is here seen as the various means whereby transactions are processed and their effect recorded.

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

- 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:

$[\,(t,t) \mapsto [\,c \mapsto tsk\,]\,]$

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 ~→ B ~→ 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 ~→ VAL* ~→ (T×T×T) ~→ B ~→ 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 $\overrightarrow{m}$ Ofrs
Ofrs = O $\overrightarrow{m}$ Ofr !
Ofr = (T×T) $\overrightarrow{m}$ (Q × (lo:P×hi:P) × ...)
ClRm = O $\overrightarrow{m}$ Clrd | Rmvd
Clrd = S × P × T × Ofrs × Ofrs
Rmvd = S × T × O × Ofr
Market = T → SE

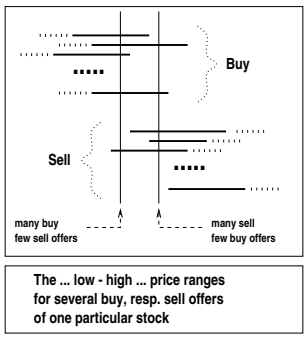


Figure A.5: A "Snapshot" Stock Exchange View of Current Offers of a Single Stock

- 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

[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.

- 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!

```
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.

Appendix B

Tetsuo Tamai's Paper

For private, limited circulation only, I take the liberty of enclosing Tetsuo Tamai's IEEE Computer Journal paper.

COVER FEATURE

SOCIAL IMPACT OF INFORMATION SYSTEM FAILURES

Tetsuo Tamai, *University of Tokyo*

The social impact of information systems becomes visible when serious system failures occur. A case of mistyping in entering a stock order by Mizuho Securities and the following lawsuit between Mizuho and the Tokyo Stock Exchange sheds light on the critical role of software in society.

Almost daily, we hear news of system failures that have had a serious impact on society. The ACM Risks Forum moderated by Peter Neumann is an informative source that compiles various reported instances of computer-related risks (http://catless.ncl.ac.uk/risks).

One of journalism's shortcomings is that it makes a loud outcry when trouble occurs with a computer-based system, but it remains silent when nothing goes wrong. This gives the general public the wrong impression that computer systems are highly unreliable. Indeed, as software is invisible and not easy for ordinary people to understand, they generally perceive software to be something unfathomable and undependable.

Another problem is that when a system failure occurs, news sources offer no technical details. Reporters usually don't have the knowledge about software and information systems needed to report technically significant facts, and the stakeholders are generally reluctant to disclose details. The London Ambulance Service failure case is often cited in software engineering literature because its detailed inquiry report is open to the public, which only emphasizes how rare such cases are (www.cs.ucl.ac.uk/staff/a.finkelstein/las/lascase0.9.pdf).

MIZUHO SECURITIES VERSUS THE TOKYO STOCK EXCHANGE

The case of Mizuho Securities versus the Tokyo Stock Exchange (TSE) is archived in the 12 December 2005 issue of the *Risks Digest* (http://catless.ncl.ac.uk/risks/24.12.html), and additional information can be obtained from sources such as the *Times* (www.timesonline.co.uk/tol/news/world/asia/article755598.ece) and the *New York Times* (www.nytimes.com/2005/12/13/business/worldbusiness/13glitch.html?_r=1), among others.

The incident started with the mistyping of an order to sell a share of J-Com, a start-up recruiting company, on the day its shares were first offered to the public. An employee at Mizuho Securities, intending to sell one share at 610,000 yen, mistakenly typed an order to sell 610,000 shares at 1 yen.

 Published by the IEEE Computer Society 0018-9162/09/$25.00 © 2009 IEEE

What happened after that was beyond imagination. The order went through and was accepted by the Tokyo Stock Exchange Order System. Mizuho noticed the blunder and tried to withdraw the order, but the cancel command failed repeatedly. Thus, it was obliged to start buying back the shares itself to cut the loss. In the end, Mizuho's total loss amounted to 40 billion yen ($225 million). Four days later, TSE called a news conference and admitted that the cancel command issued by Mizuho failed because of a program error in the TSE system. Mizuho demanded compensation for the loss, but TSE refused. Then, Mizuho sued TSE for damages.

When such a case goes to court, we can gain access to documents presented as evidence, which provides a rare opportunity to obtain information about the technical details behind system failures. Still, requesting and acquiring documents from the court requires considerable effort by the third party. As it happened, Mizuho contacted me to give an expert opinion, thus I had access to all materials presented to the court. Admittedly, there is always the possibility of bias, but as a scientist, I have endeavored to report this case as impartially as possible.

Another reason for examining this case is that it involved several typical and interesting software engineering issues including human interface design, fail-safety issues, design anomalies, error injection by fixing code, ambiguous requirements specification, insufficient regression testing, subcontracting, product liability, and corporate governance.

WHAT HAPPENED

J-Com was initially offered on the Tokyo Stock Exchange Mother Index on 8 December 2005. On that day, a Mizuho employee got a call from a client telling him to sell a single share of J-Com at 610,000 yen. At 9:27 a.m., the employee entered an order to sell 610,000 shares at 1 yen through a Fidessa (Mizuho's securities ordering system) terminal. Although a "Beyond price limit" warning appeared on the screen, he ignored it (pushing the Enter key twice meant "ignore warning" by the specification), and the order was sent to the TSE Stock Order System. J-Com's outstanding shares totaled 14,500, which means the erroneous order was to sell 42 times the total number of shares.

At 9:28 a.m., this order was displayed on the TSE system board, and the initial price was set at 672,000 yen.

Price determination mechanism

TSE stock prices are determined by two methods: *Itayose* (matching on the board) and *Zaraba* (regular market). The Itayose method is mainly used to decide opening and closing prices; the Zaraba method is used during continuous auction trading for the rest of the trading session. In the J-Com case, the Itayose method was used as it was the first day of determining the J-Com stock price.

There are two order types for selling or buying stocks: *market orders* and *limit orders*. Market orders do not specify the price to buy or sell and accept the price the market determines, while limit orders specify the price. When sell and buy orders are matched to execute trading, market orders of both sell and buy are always given the first priority.

Market participants generally want to buy low and sell high. But when the Itayose method is applied, there is no current market price to refer to, and thus there can be a variety of sell/buy orders, resulting in a wide range of prices. With the Itayose method, a single execution price is determined that matches sell and buy orders by satisfying the following rules:

An employee at Mizuho Securities, intending to sell one share at 610,000 yen, mistakenly typed an order to sell 610,000 shares at 1 yen.

1. All market orders must be executed.
2. All limit orders to sell/buy at prices lower/higher than the execution price must be executed.
3. The following amount of limit orders to sell or buy at the execution price must be executed: the entire amount of either all sell or all buy orders, and at least one trading unit from the opposite side of the order book.

The third rule is complicated but functions as a tie-breaker when the first two rules do not determine a unique price. Looking at an example helps to understand how the rules work.

Table 1 represents an instance of the order book. The center column gives the prices. The left center column shows the volume of sell offers at the corresponding price, while the right center column shows the volume of the buy bids. The volume of the market sell orders and the market buy orders is displayed at the bottom line and at the top line, respectively. The leftmost column shows the aggregate volume of sell offers (working from the bottom to the top in the order of priority), and the rightmost column gives the aggregate volume of buy bids (working from the top to the bottom in the order of priority).

We start by focusing on rules (1) and (2) to determine the opening price. First, the price level is searched where the amounts of the aggregated sell and the aggregated buy cross over. In this case, the line is between 500 yen and 499

COVER FEATURE

Table 1. Order book example illustrating Itayose method.

Sell offer			Buy bid	
Aggregate sell orders	Shares offered at bid	Price (yen)	Buy offers at bid	Aggregate buy orders
		Market	4,000	
48,000	8,000	502	0	4,000
40,000	20,000	501	2,000	6,000
20,000	5,000	500	3,000	9,000
15,000	6,000	499	15,000	24,000
9,000	3,000	498	8,000	32,000
6,000	0	497	20,000	52,000
	6,000	Market		

yen. These two prices satisfy conditions (1) and (2), so they are the opening price candidates. Then, applying rule (3), the price is finally determined as 499 yen.

Of course, this algorithm does not always determine the price. For example, if the orders are all buy and no sell, there is no solution that satisfies all three rules. An additional mechanism that holds back transactions even if the matching price is found by the Itayose method is a measure to prevent sudden price leaps or drops. On the TSE, an immediate execution only takes place if the next execution price is within a certain range from the previous execution price. The price level determines the range. For example, if the most recently executed price was 500 yen, the next execution price must be within the range of 490-510 yen. In other words, it can only fluctuate up to 10 yen in either direction.

Suppose the matched price is beyond this range—for example, 550 yen when the previous price was 500 yen. Then, execution does not take place; instead a special bid quote of 510 yen is indicated to call for offers at this price. If no offers at this price are received, the special bid quote will be raised to 520 yen after 5 minutes, and so on until equilibrium is achieved. This mechanism is intended to make a smooth transition between widely divergent prices.

But on the morning of 8 December, J-Com had no previous price. In such cases, the publicly assessed value is used in place of the previous price, which was 610,000 yen. Because the matched price was much higher, a special bid quote of 610,000 yen was shown at 9:00 a.m., then raised to 641,000 yen at 9:10 a.m., which means the range was ± 31,000 yen, and raised again to 672,000 yen at 9:20 a.m. Table 2 shows the order book at that moment, when the 1-yen sell offer came in.

Initial price determination

The term "reverse special quote" denotes this particularly rare event. It means that when a special buy bid quote is displayed, a sell order of low price with a significant amount that reverses the situation to a special sell offer quote comes in (or conversely a special sell offer quote is reversed to a special buy bid quote). TSE has another rule that applies to such a case. This rule stipulates that the previous special quote is fixed as the execution price, and the transaction proceeds. Thus, the initial price of J-Com was now determined to be 672,000 yen. In addition to the step price range set for reducing sudden price change, there is also a price limit range for a day. The upper and lower limits of the price for each stock are defined based on the initial price of the day. In the J-Com case, the limits were defined at the moment when the initial price was determined: The upper limit was 772,000 yen, and the lower limit was 572,000 yen.

In regular trading, the price limits are fixed at the start of the market day, and orders with prices exceeding the limit (either upper or lower) are rejected. But when the initial price is determined during the market time, as in the J-Com case, orders received before the price limits are set are not ignored. Instead, the price of an order exceeding the upper limit is adjusted to the upper price limit, and an order under the lower limit is adjusted to the lower price limit. Thus, the 1-yen order by Mizuho was adjusted to 572,000 yen.

Noticing the mistake, Mizuho entered a cancel command through a Fidessa terminal at 9:29:21, but it failed. Between 9:33:17 and 9:35:40, Mizuho tried to cancel the order several times through TSE system terminals that are installed at the Mizuho site, but the cancellations failed. Mizuho called TSE asking for a cancellation on the TSE side, but the answer was no.

At 9:35:33, Mizuho started to buy back J-Com shares. In the end, it could only buy back 510,000 shares; nearly 100,000 shares were bought by others and never restored.

Aftermath

On 12 December, four days after the incident, TSE president Takuo Tsurushima held a press conference and admitted that the order cancellation by Mizuho failed because of a defect in the TSE Stock Order System.

Table 2. Order book for J-Com stock at 9:20 a.m.

Sell offer			Buy bid	
Aggregate	Amount	Price (yen)	Amount	Aggregate
		Market	253	
1,432	695	OVR*	1,479	1,732
737		6,750	4	1,736
737		6,740	6	1,742
737		6,730	6	1,748
737	3	6,720**	28	1,776
734		6,710	2	1,778
734		6,700	3	1,781
734	1	6,690	1	1,782
733		6,680	1	1,783
733	114	UDR***	120	1,903
	619	Market		

*More than 675,000 yen **Special buy quote ***Less than 668,000 yen

Mizuho could not buy back 96,236 shares, and it was impossible for Mizuho to deliver real shares to those who had bought them. An exceptional measure was taken to settle trading by paying 912,000 yen per share in cash. The result was a 30-billion-yen loss to Mizuho. Mizuho had already suffered a loss of 10 billion yen by buying back 510,000 shares, thus the total loss amounted to 40 billion yen.

Mizuho and TSE started negotiations on compensation for damages in March 2006, but they failed to reach an agreement. Mizuho sent a formal letter to TSE in August 2006 requesting compensation, which TSE declined by sending a letter of refusal.

Mizuho filed a suit against TSE in the Tokyo District Court on 27 October 2006, demanding compensation of 41.5 billion yen. The first oral pleadings took place on 15 December 2006, and trials were held 13 times in two years, the last on 19 December 2008. The court's decision in that trial was scheduled to be given on 27 February 2009, but the court decided to postpone the decision.

In the contract between TSE and each user of the TSE Stock Order System, including Mizuho, there is a clause on exemption from responsibility on the TSE side except when a serious mistake is attributed to TSE. The crucial issue was whether the damage caused by the system defect was due to a serious mistake beyond the range of exemption. TSE also argued that as the incident started with a mistake on the Mizuho side, the mistakes and the resulting damages should be canceled out.

PROBABLE CAUSE

The TSE system unduly rejected the Mizuho order cancellation because the module for processing order cancellation erroneously judged that the J-Com target sell order had been completely executed, thus leaving no transactions to be canceled. This bug had been hiding for five years.

Fujitsu developed the system under contract with TSE and released it for use in May 2000. An evidence document submitted to the court reported that a similar error was found during integration testing in February 2000 and that the current fault occurred as a result of fixing that error.

But there are several mysteries surrounding this apparently simple failure case. Initially, TSE maintained that the target cancellation order could not be found because its price had been changed from 1 yen to the adjusted price of 572,000 yen, whereas the designated cancel price command was the original 1 yen. This explanation is bizarre as it implies that the order data is searched in the database using price as a key when it is obvious that price cannot be a key because there can be multiple orders with the same price. In addition, as this case shows, the price of the same order can be modified during the transaction. This explanation turned out to be wrong, but it came from the fact that there was indeed a logic in the procedure that partly used price to search order data. TSE also maintained that if buy orders did not flow in continuously and thus the target sell orders were not always being matched to buy orders, the order cancel module would not have been invoked within the order matching module but instead invoked in the order entry module, and then the cancellation would have succeeded. However, this explanation implies that different cancel modules are called or the same module behaves differently according to when it is invoked.

The third question, and probably the most crucial one with respect to the direct cause of the error, is how data handling identifies orders causing a reverse special quote. That information is written into a database containing

the order book data, but once the information is used in determining the execution price, it is immediately cleared. The rationale behind this design decision is mysterious. The programmer who was charged with fixing the February 2000 bug intended to use this data to judge the type of order to be canceled but he did not know that the data no longer existed.

TSE and Fujitsu claimed that this incident occurred in a highly exceptional situation when the following seven conditions held at the same time:

1. The daily price limits have not been determined.
2. The special quote is displayed.
3. The reverse special quote occurs.
4. The price of the order that has caused the reverse special quote is out of the newly defined daily price limits.
5. The target order of cancellation caused the reverse special quote.
6. The target cancellation order is in the process of sell and buy matching, which forces the cancellation process to wait.
7. The target order is continually being matched.

The order cancellation module appears to have insufficient cohesion as different functions are overloaded.

A general procedure for the order cancellation module would be as follows:

1. Find the order to be canceled.
2. Determine if the order satisfies conditions for cancellation.
3. Execute cancellation if the conditions are met.

Because each order has a few simple attributes—stock name, sell or buy, remaining number of shares to be processed (if 0, the order is completed), and price—the condition that an order can be canceled is straightforward: "the remaining number of shares to be processed is greater than zero." There is only one other condition that cannot be determined by the order attribute data but can be determined by its execution state: If the target order is in the process of matching, the cancel process must wait.

A remarkable point to note is that factors such as undefined limit price, display of special quote, reversing special quote, price adjustment to the limit, and so forth have no influence on the cancellation judgment. Thinking in this way, it seems that the system design artificially introduced the seven complicated conditions listed by TSE and Fujitsu.

DESIGN ANOMALY

Figure 1 shows a flowchart of the module that handles order cancellation. Because order cancellation and order change are processed in the same way, the two functions are overloaded in this same module, but for the sake of simplicity I only deal with order cancellation.

The flowchart is not shown to provide details but to illustrate the kind of documents presented to the court. It is extracted and modified from a document submitted as evidence by the defendant, which was an analysis of the error reported by a TSE system engineer. The plaintiff required the defendant to provide the entire design specification and source code, but the defendant refused and the judge did not force the issue, being reluctant to go into technical details in court.

Part A of the flowchart deals with the logic of price adjustment to limit if necessary. The decision logic is as follows:

- if the order to cancel is sell and the price is lower than the lower limit, it is adjusted to the lower limit; and
- if the order to cancel is buy and the price is higher than the upper limit, it is adjusted to the upper limit.

Part B of the flowchart is the logic inserted in February 2000 when an error was found during testing and caused a failure in December 2005. Its logic is as follows: If called in the order matching process; and limit prices are already set; and the order to cancel is a buy over the limit price or a sell under the limit price and is not a reverse special quote order; then a cancellation is infeasible because all shares are already executed. Although this logic is unduly complicated, it is sound only if all the if-conditions are correctly judged. Unfortunately, the judgment on "if not a reverse special quote" gave a wrong answer of "true" in this Mizuho case, and the decision erroneously judged that the cancellation was infeasible.

Insufficient information is available to allow capturing details of the system design, but from what is available we can infer the following design flaws.

Problems in database design

Three databases are related to the problem in this case: Order DB, Sell/Buy Price DB, and Stock Brand DB. The Order DB stores data of all entered orders. This database should include the current attributes of each order, including those necessary for judging whether the designated order can be canceled. For example, because there is a record field for the executed shares in this database, determining if all the shares of the order have been executed or not should be a trivial process. However, due to the time gap between usage and update of the data, the process is much more complicated. If the principle of database integrity is respected, the logic would be much clearer, but performance seems to be given higher priority than integrity.

Part A of the flowchart in Figure 1 calculates price adjustment within the cancellation handling module, which implies that the price data in the Order DB does not reflect the current status.

The Sell/Buy Price DB sorts sell/buy orders by price for each stock brand. This is by nature a secondary database constructed from the Order DB. The secondary index is price, but identifying an order uniquely in the database requires the order ID. The explanation that price is used to search the database must refer to search in this database, and the price adjustment logic embedded in the order cancel module should be related to it. The data handling over the Price DB and the Order DB appears to be unduly complicated.

The Stock Brand DB corresponds to a physical order book for each stock, but its substantial data is stored in the Sell/Buy Price DB and only some specific data for each stock brand is kept here. However, to implement a rule that an order that has caused a reverse special quote has an exceptional priority in matching—lower than the regular case—the customer ID and order ID of such a stock is written in this database, and they are cleared as soon as the matching is done. This kind of temporary usage of a database goes against the general principle that a database should save persistent data accessed by multiple modules.

Problems in module design.

The part of the system that handles order cancellation appears to have low modularity. The logic in part B of the flowchart made a wrong judgment because the information telling it that the target order had induced the reverse special quote had been temporarily written on the Stock Brand DB by the order matching module and had already been cleared. This implies an accidental module coupling between the order matching and order cancelling modules.

The order cancellation module appears to have insufficient cohesion as different functions are overloaded. It is not clear how the tasks of searching the target order to be canceled, determining cancellability, executing cancellation, and updating the database are this module's responsibility.

LESSONS LEARNED

In addition to the insights into the associated software design problems, this case provides lessons learned with regard to software engineering technologies, processes, and social aspects.

Safety and human interface

If the order entry system on either the Mizuho or TSE side had been equipped with more elaborate safety measures, the accident could have been avoided. It was not the first time that the mistyping of a stock order resulted in a big loss. For instance, in December 2001, a trader at UBS

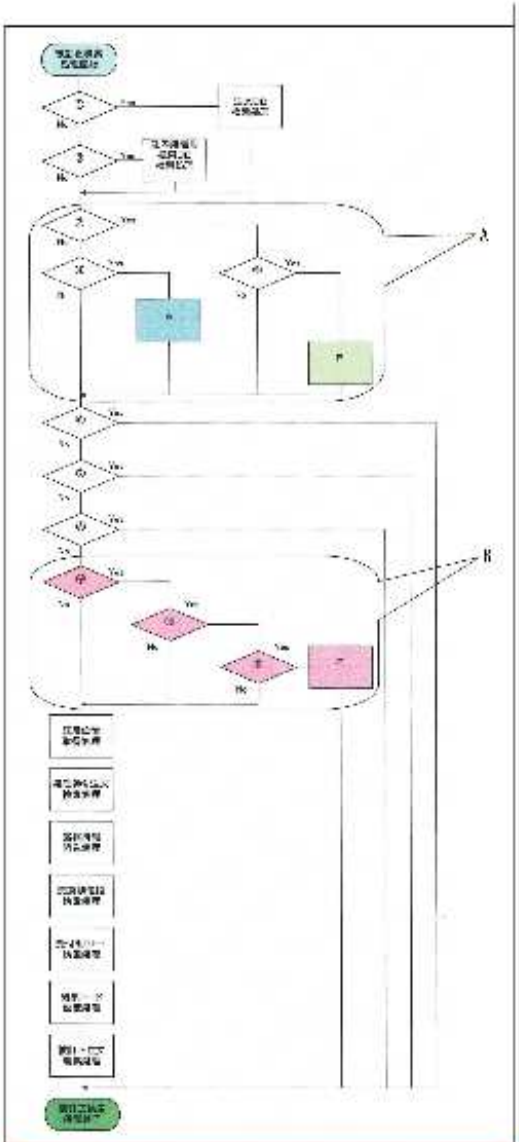


Figure 1. Flowchart of the order cancellation module.

Warburg, the Swiss investment bank, lost more than 10 billion yen while trying to sell 16 shares of the Japanese advertising company Dentsu at 610,000 yen each. He sold 610,000 shares at six yen each. (The similarity between these two cases, including the common figure of 610,000, is remarkable.)

Table 3. Associations between the Software Engineering Code of Ethics and Professional Practice and the TSE-Mizuho case.

Engineering issue	Applicable ACM/IEEE-CS Principle	Ethics clause
Design anomaly	3.01	Strive for high quality, acceptable cost and a reasonable schedule, ensuring significant tradeoffs are clear to and accepted by the employer and the client, and are available for consideration by the user and the public.
	3.14	Maintain the integrity of data, being sensitive to outdated or flawed occurrences.
Safety and human interface	1.03	Approve software only if they have a well-founded belief that it is safe, meets specifications, passes appropriate tests, and does not diminish quality of life, diminish privacy or harm the environment. The ultimate effect of the work should be to the public good.
Requirements specification	3.07	Strive to fully understand the specifications for software on which they work.
	3.08	Ensure that specifications for software on which they work have been well documented, satisfy the users' requirements and have the appropriate approvals.
Verification and validation	3.10	Ensure adequate testing, debugging, and review of software and related documents on which they work.
Role of user and developer	4.02	Only endorse documents either prepared under their supervision or within their areas of competence and with which they are in agreement.
	5.01	Ensure good management for any project on which they work, including effective procedures for promotion of quality and reduction of risk.
Chain of subcontracting	2.01	Provide service in their areas of competence, being honest and forthright about any limitations of their experience and education.
	3.04	Ensure that they are qualified for any project on which they work or propose to work by an appropriate combination of education and training, and experience.

The habit of ignoring warning messages is common, but it was a critical factor in these cases. It raises the question of how to design a safe—but not clumsy—human interface.

Requirements specification

Development of the current TSE Stock Order System started with the request for proposal (RFP) that TSE presented to the software industry in January 1998. Two companies submitted proposals, and TSE selected Fujitsu as the vendor with which to contract. After several discussions between TSE and Fujitsu, Fujitsu wrote the requirements specification, which TSE approved.

With respect to the order cancellation requirement, it is only mentioned as a function to "Cancel order" in the RFP, and no further details are given there. In the requirements specification, six conditions are listed when cancel (or change) orders are not allowed, but none of them fit the Mizuho case. The document also states that "in all the other cases, change/cancel condition checking should be the same as the current system." Here, "current system" refers to the prior version of the TSE Stock Order System, also developed by Fujitsu, which had been in use until May 2000.

The phrase "the same as the current system" frequently appears in this requirements specification, which was criticized by software experts after the Mizuho incident was publicized. The phrase may be acceptable if there is a consensus between the user and the developer on what it means in each context, but when things go wrong, the question arises whether the specification descriptions were adequate.

Verification and validation

The fact that an error was injected while fixing a bug found in testing is so typical that every textbook on testing warns about this possibility. It is obvious that regression testing was not properly done. It is perhaps too easy to criticize this oversight, but it would be worthwhile to study why it happened in this particular case. So far, not many details have been disclosed.

Role of user and developer

It is conceivable that communication between the user and the developer was inadequate during the TSE system development. The user, TSE, basically did not participate in the process of design and implementation. More involvement of the user during the entire development process would have promoted deeper understanding of the requirements by the developer, and the defect injected during testing might have been avoided.

Subcontracting chain

As in many large-scale information system development projects, the TSE system project was organized in a hierarchical subcontracting structure. The engineer who was in charge of fixing the code in question had a low position in the subcontracting chain. This organizational structure was the likely cause of the misunderstanding about database usage. Such a subcontract structure has often been studied from the industry and labor problem point of view, but it is also important to examine it from the engineering point of view.

Product liability

The extent to which software is regarded as a product amenable to product liability laws may depend on legal and cultural boundaries, but there is a general worldwide trend demanding stricter liability for software. More lawsuits are being filed, and thus software engineers must be more knowledgeable about software product liability issues.

ETHICAL ISSUES

This case raises several questions about professional ethics. However, we should be careful in relating ethical issues and legal matters. Illegal conduct and unethical conduct are of course not equivalent. Moreover, the Mizuho incident is a civil case, not a criminal case.

The Software Engineering Code of Ethics and Professional Practice developed by an ACM and IEEE Computer Society joint task force provides a good framework for discussing ethical issues. The Code comprises eight principles, and each clause is numbered by its principle category and the sequence within the principle. The principles are numbered as 1: Public, 2: Client and Employer, 3: Product, 4: Judgment, 5: Management, 6: Profession, 7: Colleagues, and 8: Self.

As Table 3 shows, some clauses in the Code have relatively strong associations with various aspects of the TSE-Mizuho case. However, this discussion is by no means intended to blame the software engineers who participated in planning, soliciting requirements, designing, implementing, testing, or maintaining the TSE system or other related activities, or to suggest negligence of ethical obligations. First, the Code was not intended to be used in this fashion. Second, the collected facts and disclosed materials are insufficient to precisely judge what kind of specific conduct caused the unfortunate result. However, linking the problems in this case with plausibly related ethical obligation clauses as shown in Table 3 can provide a basis for considering the ethical aspects of this incident and other similar cases.

In addition to individual ethical conduct, the Mizuho-TSE case raises issues pertaining to corporate governance. Why did such an erroneous order by a trader go through unnoticed at Mizuho? Did the TSE staff respond appropriately when they were consulted about the order cancellation? How did Fujitsu manage subcontractors? Corporate governance is another domain where software engineering must deal with social and ethical issues.

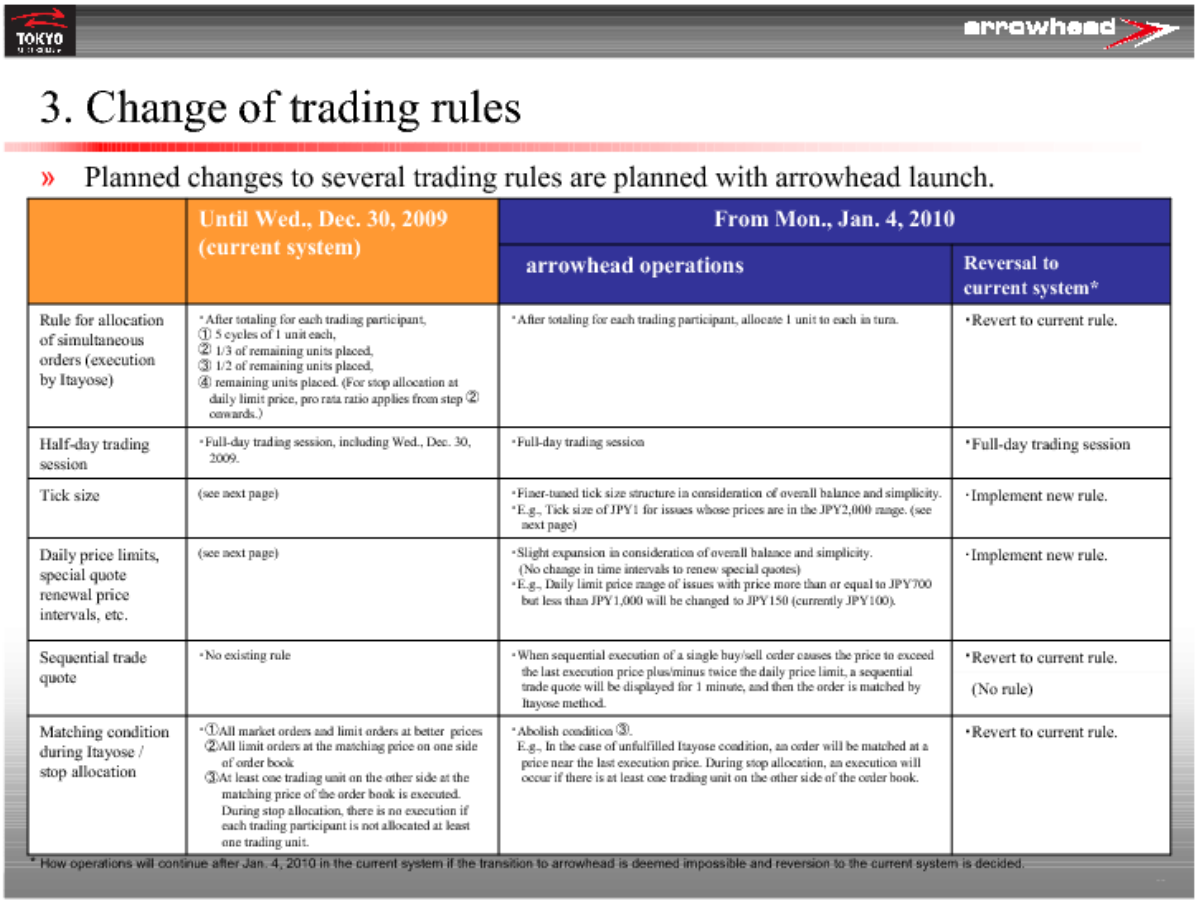
If we can learn valuable lessons from this unfortunate incident, it would be beneficial. We should also encourage people who have access to information about similar system failures having significant social impact to analyze and report those cases.

Tetsuo Tamai is a professor in the Graduate School of Arts and Sciences at the University of Tokyo. His research interests include requirements engineering and formal and informal approaches to domain modeling. He received a DrS in mathematical engineering from the University of Tokyo. He is a member of the IEEE Computer Society, the ACM, the Information Processing Society of Japan, and the Japan Society for Software Science and Engineering. Contact him at tamai@graco.c.u-tokyo.ac.jp.

Appendix C

Tokyo Stock Exchange **arrowhead** Announcements

C.1 Change of trading rules



TOKYO

arrowhead

3. Change of trading rules

» Planned changes to several trading rules are planned with arrowhead launch.

	Until Wed., Dec. 30, 2009 (current system)	From Mon., Jan. 4, 2010	
		arrowhead operations	Reversal to current system*
Rule for allocation of simultaneous orders (execution by Itayose)	・After totaling for each trading participant, ① 5 cycles of 1 unit each, ② 1/3 of remaining units placed, ③ 1/2 of remaining units placed, ④ remaining units placed. (For stop allocation at daily limit price, pro rata ratio applies from step ② onwards.)	・After totaling for each trading participant, allocate 1 unit to each in turn.	・Revert to current rule.
Half-day trading session	・Full-day trading session, including Wed., Dec. 30, 2009.	・Full-day trading session	・Full-day trading session
Tick size	(see next page)	・Finer-tuned tick size structure in consideration of overall balance and simplicity. ・E.g., Tick size of JPY1 for issues whose prices are in the JPY2,000 range. (see next page)	・Implement new rule.
Daily price limits, special quote renewal price intervals, etc.	(see next page)	・Slight expansion in consideration of overall balance and simplicity. (No change in time intervals to renew special quotes) ・E.g., Daily limit price range of issues with price more than or equal to JPY700 but less than JPY1,000 will be changed to JPY150 (currently JPY100).	・Implement new rule.
Sequential trade quote	・No existing rule	・When sequential execution of a single buy/sell order causes the price to exceed the last execution price plus/minus twice the daily price limit, a sequential trade quote will be displayed for 1 minute, and then the order is matched by Itayose method.	・Revert to current rule. (No rule)
Matching condition during Itayose / stop allocation	・①All market orders and limit orders at better prices ②All limit orders at the matching price on one side of order book ③At least one trading unit on the other side at the matching price of the order book is executed. During stop allocation, there is no execution if each trading participant is not allocated at least one trading unit.	・Abolish condition ③. E.g., In the case of unfulfilled Itayose condition, an order will be matched at a price near the last execution price. During stop allocation, an execution will occur if there is at least one trading unit on the other side of the order book.	・Revert to current rule.

* How operations will continue after Jan. 4, 2010 in the current system if the transition to arrowhead is deemed impossible and reversion to the current system is decided.

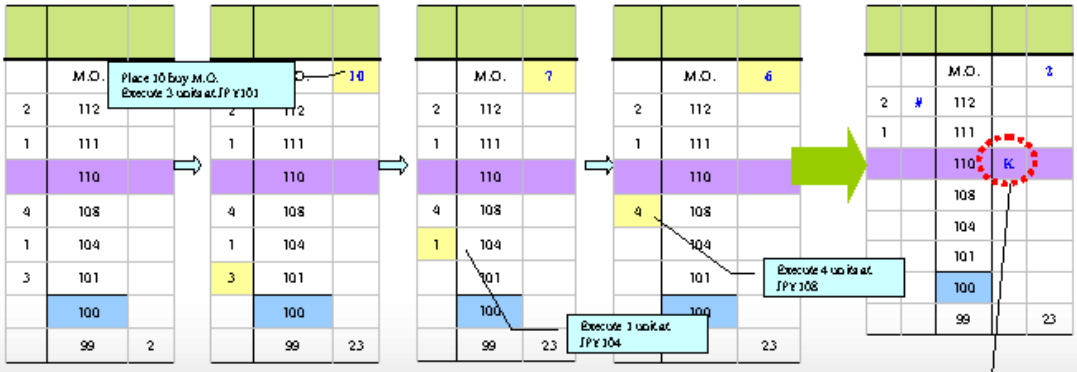
TOKYO arrowhead

3. Change of trading rules

»Sequential Trade Quote

New rule to be introduced with arrowhead launch.

If a single order that causes a series of executions arrives in the order book, and such executions cause the price to exceed twice the special quote renewal price interval from the last execution price, a sequential trade quote will be displayed at this price for 1 minute and conduct matching using Itayose.



TOKYO arrowhead

3. Change of trading rules

» Revision of tick sizes

Stock price (JPY)					Current (JPY)	Revised (JPY)
		~	2,000	or less	1	1
above	2,000	~	3,000	〃	5	**1**
〃	3,000	~	5,000	〃	10	**5**
〃	5,000	~	30,000	〃	10	10
〃	30,000	~	50,000	〃	50	50
〃	50,000	~	300,000	〃	100	100
〃	300,000	~	500,000	〃	1,000	**500**
〃	500,000	~	3,000,000	〃	1,000	1,000
〃	3,000,000	~	5,000,000	〃	10,000	**5,000**
〃	5,000,000	~	20,000,000	〃	10,000	10,000
〃	20,000,000	~	30,000,000	〃	50,000	**10,000**
〃	30,000,000	~	50,000,000	〃	100,000	**50,000**
〃	50,000,000				100,000	100,000

TOKYO arrowhead

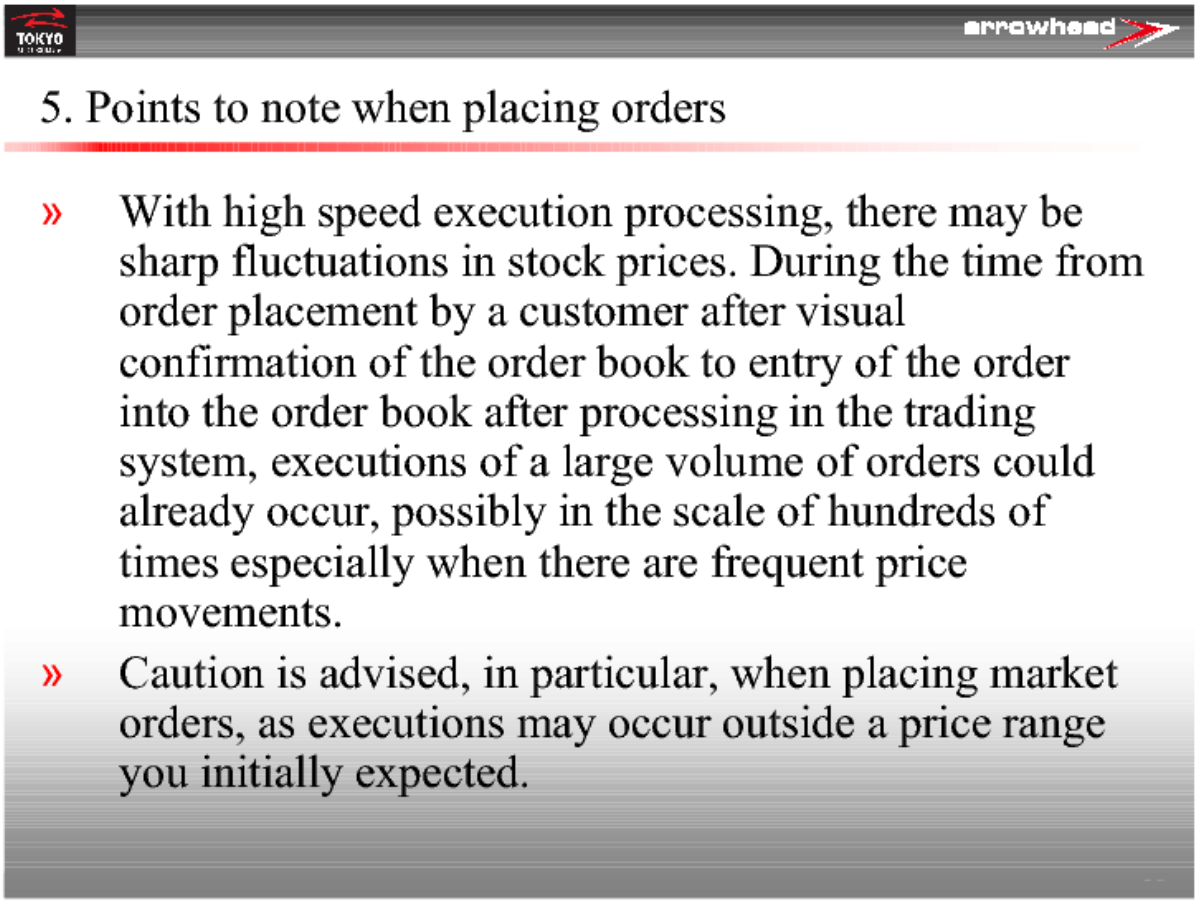
3. Change of trading rules

» Revision of daily price limits and special quote renewal price intervals

Price (JPY)					Price limit		Renewal price interval	
					Current	Revised	Current	Revised
			Less than	100	50	50	5	5
100	and above	~	less than	200	50	50	5	5
200	~	~	less than	500	80	80	5	8
500	~	~	less than	700	100	100	10	10
700	~	~	less than	1,000	100	120	10	15
1,000	~	~	less than	1,500	200	300	20	30
1,500	~	~	less than	2,000	300	400	30	40
2,000	~	~	less than	3,000	400	600	40	60
3,000	~	~	less than	5,000	500	700	50	70
5,000	~	~	less than	7,000	1,000	1,000	100	100
7,000	~	~	less than	10,000	1,000	1,500	100	150
10,000	~	~	less than	15,000	2,000	3,000	200	300
15,000	~	~	less than	20,000	2,000	4,000	200	400
20,000	~	~	less than	30,000	3,000	5,000	300	500
30,000	~	~	less than	50,000	4,000	7,000	400	700
50,000	~	~	less than	70,000	5,000	10,000	500	1,000
70,000	~	~	less than	100,000	10,000	15,000	1,000	1,500

Price (JPY)					Price limit		Renewal price interval	
					Current	Revised	Current	Revised
100,000	and above	~	less than	150,000	20,000	30,000	2,000	3,000
150,000	~	~	less than	200,000	30,000	40,000	3,000	4,000
200,000	~	~	less than	300,000	40,000	50,000	4,000	5,000
300,000	~	~	less than	500,000	50,000	70,000	5,000	7,000
500,000	~	~	less than	700,000	100,000	100,000	10,000	10,000
700,000	~	~	less than	1,000,000	100,000	150,000	10,000	15,000
1,000,000	~	~	less than	1,500,000	200,000	300,000	20,000	30,000
1,500,000	~	~	less than	2,000,000	300,000	400,000	30,000	40,000
2,000,000	~	~	less than	3,000,000	400,000	600,000	40,000	60,000
3,000,000	~	~	less than	5,000,000	500,000	700,000	50,000	70,000
5,000,000	~	~	less than	7,000,000	1,000,000	1,000,000	100,000	100,000
7,000,000	~	~	less than	10,000,000	1,000,000	1,500,000	100,000	150,000
10,000,000	~	~	less than	15,000,000	2,000,000	3,000,000	200,000	300,000
15,000,000	~	~	less than	20,000,000	3,000,000	4,000,000	300,000	400,000
20,000,000	~	~	less than	30,000,000	4,000,000	5,000,000	400,000	500,000
30,000,000	~	~	less than	50,000,000	5,000,000	7,000,000	500,000	700,000
50,000,000	~				10,000,000	10,000,000	1,000,000	1,000,000

C.2 Points to note when placing orders



TOKYO arrowhead

5. Points to note when placing orders

(Case) Current price in the order book is JPY100.
An order for 10 units of market buy orders is placed.

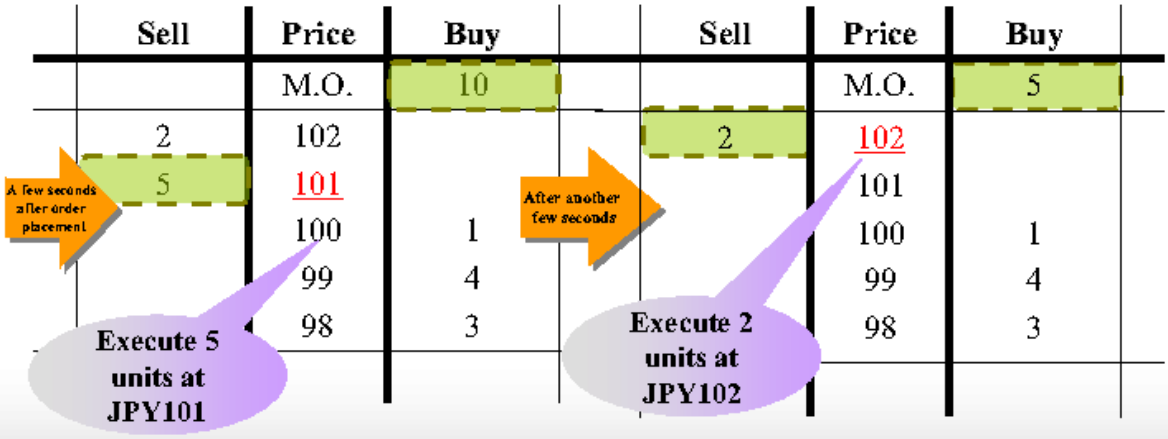
Cumulative Sell	Sell	Price	Buy	Cumulative Buy
7		M.O.	10	10
7	2	102		
5	5	101		10
		100	1	11
		99	4	15
		98	3	18

Place 10 units of buy M.O.

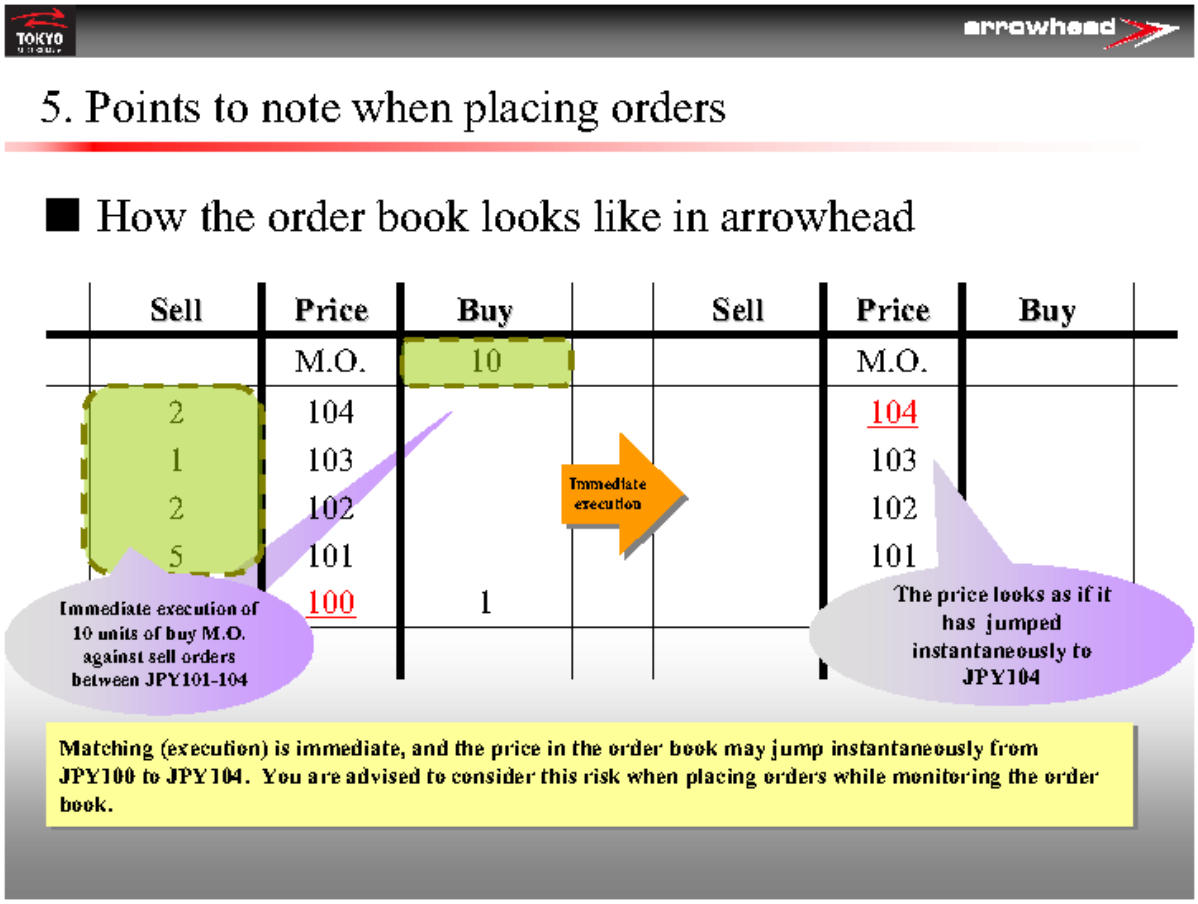
What is the difference in execution speed between the current system and arrowhead?

5. Points to note when placing orders

■How the order book looks like in the current system



Matching (execution) is performed every few seconds, so placing orders while monitoring the order book was feasible.


TOKYO
arrowhead
5. Points to note when placing orders
How the order book looks like in arrowhead
Sell
Price
Buy
M.O.
10
2
1
2
5
104
103
102
101
100
1
Immediate execution
Sell
Price
Buy
M.O.
104
103
102
101
Immediate execution of 10 units of buy M.O. against sell orders between JPY101-104
The price looks as if it has jumped instantaneously to JPY104
Matching (execution) is immediate, and the price in the order book may jump instantaneously from JPY100 to JPY104. You are advised to consider this risk when placing orders while monitoring the order book.