

# Building Ground Models

from natural language and use case descriptions

(Simple Exercises)

Egon Börger

Dipartimento di Informatica, Università di Pisa  
<http://www.di.unipi.it/~boerger>

For details see Chapter 3.1 (Requirements Capture by Ground Models) of:

**E. Börger, R. Stärk**

**Abstract State Machines**

**A Method for High-Level System Design and Analysis**

**Springer-Verlag 2003**

**For update info see AsmBook web page:**

**<http://www.di.unipi.it/~AsmBook>**

- The Formalization Problem
  - Fundamental Questions to be Asked
- Modeling Use Cases
  - ATM (Cash Machine)
  - Password Change with Line Editor
  - Alarm Clock
- Incremental Design (Refinement)
  - Invoice System
- Managing Teams of Agents
  - Telephone Exchange

# Invoicing Orders: Informal Description

H.Habrias et al, Comparing Systems Specifications Techniques, 1998

- R0.1 The subject is to invoicing orders.
- R0.2 To invoice is to change the state of an order (to change it from the state “pending” to “invoiced”).
- R0.3 On an order, we have one and only one reference to an ordered product of a certain quantity. The quantity can be different from other orders.
- R0.4 The same reference can be ordered on several different orders.
- R0.5 The state of the order will be changed to “invoiced” if the ordered quantity is either less than or equal to the quantity which is in stock according to the reference of the ordered product.

# Invoicing Orders: Informal Description Cont'd

- Case 1
  - R1.1 All the ordered references are in stock.
  - R1.2 The stock or the set of the orders may vary due to the entry of new orders or cancelled orders, or due to having a new entry of quantities of products in stock at the warehouse. But we do not have to take these entries into account.
  - R1.3 This means that you will not receive two entry flows (orders, entries in stock). The stock and the set of orders are always given to you in an up-to-date state.
- Case 2
  - You do have to take into account the entry of new orders, cancellation of orders, and entries of quantities in the stock.
- The **Formalization Problem** consists in providing for such a (often natural language) description a rigorous representation, detailed enough to be unambiguous.

# Formalization : Fundamental Questions to be Asked

- Who are the system's **agents** and what are their relations?
  - In particular, what is the relation bw the system & its env?
- What are the system's **states**?
  - What are the domains, functions, predicates?
  - What are the static/dynamic parts (including in/output) of states?
- How do system states evolve (what are system **transitions**)?
  - Under which conditions do the state transition rules apply?
  - What is supposed to happen if those conditions are not satisfied?
    - reporting erroneous use, exception handling, robustness features?
- What is the **initialization** of the system & who provides it?
- Is the system description **complete** and **consistent**?
- What are the system **assumptions** & the **desired properties**?

# Analysing & formalizing the **state** of invoicing orders

What are the system's **states** (static/dynamic **sets**, **fcts**)?

- By R0.1 there is a set **ORDER** : static in Case 1 (R1.3), dynamic in Case 2, **no initialization and no bound are specified**
- By R0.2 there is a dyn fct **state: ORDER**  $\rightarrow$  {**pending, invoiced**}. R0.1/2 seem to imply that initially state (O) = pending for all O.
- By R0.3/5 there are two fcts, both static in Case 1 (R1.3), maybe dynamic in Case 2 **but initializn & dynamics unspecified**
  - **product: ORDER**  $\rightarrow$  **PRODUCT** representing THE (?) ordered prod in stock
  - **order-quantity: ORDER**  $\rightarrow$  **QUANTITY** by R0.3/4 not injective, not const
- By R0.5 there is a fct **stock-quantity: PRODUCT**  $\rightarrow$  **QUANTITY** apparently dynamic - static interpretation is not reasonable - but nothing is specified when & by whom it should be updated

## Analysis & formalizn of “invoicing orders” **transitions** (1)

What are the system’s **transitions**? By R0.1/2 there is only 1 transition

- The spec in R0.5 does not mention the **update of stock-quantity**
- The spec in R0.5 leaves open whether the invoicing is done
  - **simultaneously for all** (or a subset of) orders in ORDER
    - with what synchronization for concurrent access of the same product by diff orders
  - **for one order at a time**
    - in which succession
    - with what successful termination or abruption
- The spec in R0.5 leaves the **time model** (duration of invoicing) open
- The spec in R0.5 leaves **error/exception/robustness** handling open

## Analysis & formalizn of “invoicing orders” **transitions** (2)

What are the system’s **transitions**? By R0.1/2 there is only 1 transition

- Additionally for Case 2, the informal description does not specify
  - the **agents** for dynamic manipulation of orders & stock products
  - how they **interact for shared data** (ORDER, stock-quantity (Prod)),
  - whether they act independently or following a **schedule** (interleaving?)
  - whether **arrival/cancellation sequence** of orders is to be reflected by invoicing
  - whether also **PRODUCT** is dynamic, etc.

## Invoicing orders, static case: An ASM Formalization (1)

- **single-order rule** (per step at most 1 order is invoiced, the schedule is unspecified)

```
choose Order s.t. state (Order) = pending
    & order-quantity(Order) ≤ stock-quantity(product(Order))
state (Order) := invoiced
subtract order-quantity(Order)
    from stock-quantity(product(Order))
```

NB. The rule disregards a possible “arrival time” of orders

## Invoicing orders, static case: An ASM Formalization (2)

- **all-or-none rule** (simultaneously all orders for one Product are invoiced, or none if the stock cannot satisfy the request)

choose Product  $\in$  PRODUCT

Let Pending(Product) = {O | state (O) = pending & product(O) = Product}

Let Total (Product) =  $\sum_{O \in \text{Pending}(\text{Product})} \text{order-quantity}(O)$

If Total (Product)  $\leq$  stock-quantity(Product)

then forall Order  $\in$  Pending(Product)

state(Order) := invoiced

subtract Total (Product) from stock-quantity(Product)

else report “ there are not enough items in stock to satisfy all Orders of Product ”

NB. The rule disregards a possible “arrival time” of orders

## Invoicing orders, static case: ASM Formalizn (3)

- **maximal-number-of-orders rule** choosing a maximal satisfiable subset of simultaneously invoiced pending orders for a same product

choose Product  $\in$  PRODUCT

choose maximal MaxOrd  $\subseteq$  ORDER s.t.  $\exists$  Product  $\in$  PRODUCT

s.t. for all O  $\in$  MaxOrd: state (O) = pending &  
product(O) = Product

& Total(MaxOrd)  $\leq$  stock-quantity(Product)

forall O  $\in$  MaxOrd

state(Order):= invoiced

subtract Total (MaxOrd) from stock-quantity(Product)

where Total(MaxOrd) =  $\sum_{O \in \text{MaxOrd}} \text{order-quantity}(O)$

NB. Similarly for other strategies, e.g. **maximal-sale rule** heading for maximal quantity of sold items: choose MaxOrd  $\subseteq$  ORDER s.t. ...

MaxOrd has maximal  $\sum_{O \in \text{MaxOrd}} \text{order-quantity}(O)$

## Invoicing orders, dynamic case: An ASM Formalization

- **enter-orders**  $\equiv$

If  $in = (\text{Prod}_1 \text{Qty}_1 \dots \text{Prod}_n \text{Qty}_n)$  then

let  $o_1 \dots o_n = \text{new}(\text{ORDER})$  in (providing fresh orders)

forall  $1 \leq i \leq n$

order-quantity ( $o_i$ ) :=  $\text{Qty}_i$

product ( $o_i$ ) :=  $\text{Prod}_i$

state ( $o_i$ ) := pending

NB. let  $x = \text{new}(\text{Domain})$  in Rule abbreviates

let  $x = \text{new}$  in (  $\text{Domain}(x) := \text{true}, \text{Rule}$  )

Input  $in$  is assumed to be consumed by firing the rule.

No error handling if input is not legal.

Similarly for cancellation rules and stock increase rules.

# Formalization : Reassuming the used language elems

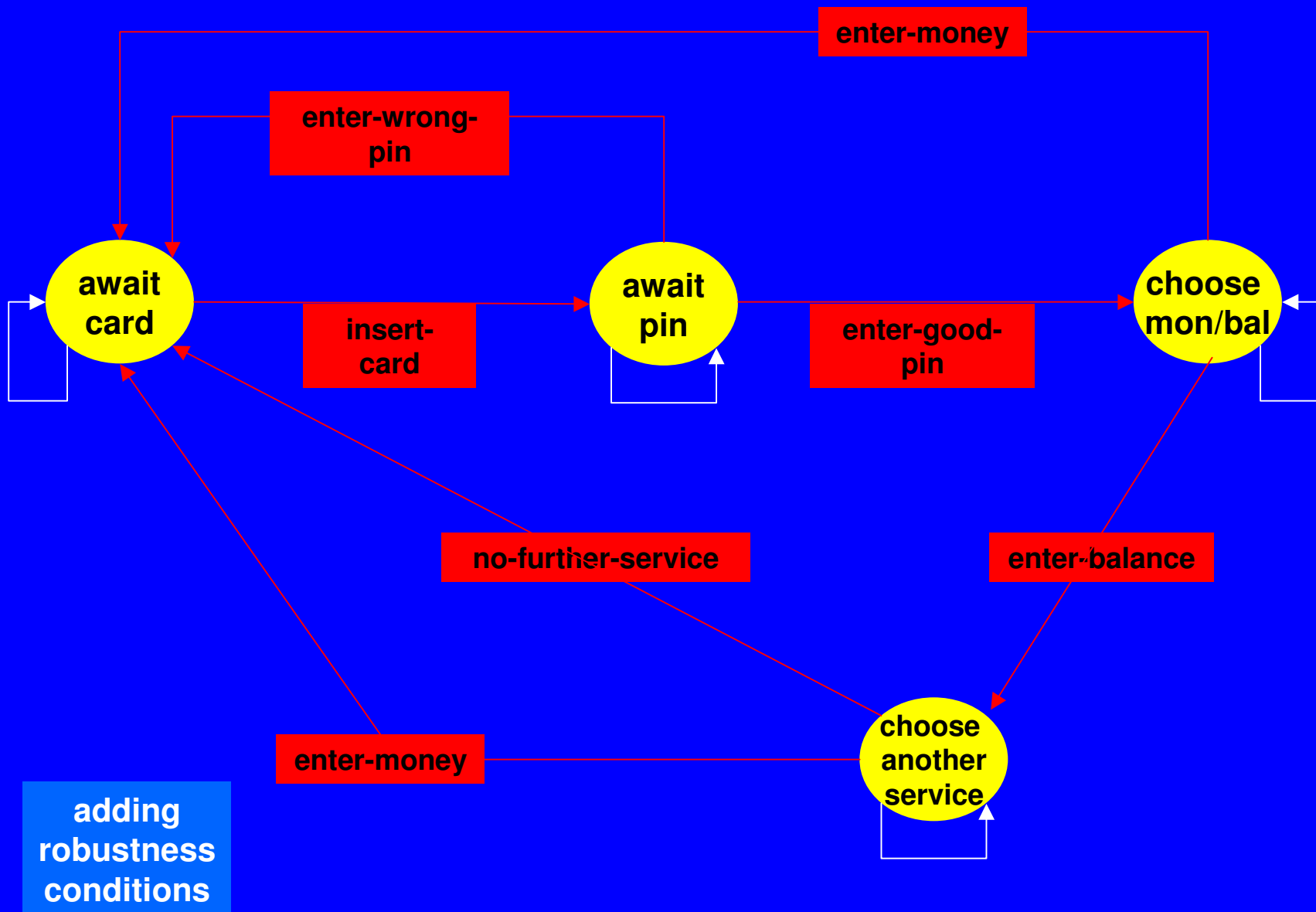
- Guarded updates
  - If Condition then  $f(s, \dots) := t$   
with arbitrary (first-order) Condition, arbitrary expressions  $s, t$ , arbitrary function  $f$  to directly reflect arbitrary states
- Simultaneous execution of multiple updates/rules
  - enhanced by forall construct
- Special constructs for **choosing** (selection functions) to reflect unspecified scheduling (non-determinism)
- Important
  - to avoid details which are irrelevant for the problem domain
    - like detailed type or procedure declarations, or structuring of classes or modules, which belong to the implementation, not to the problem

- The Problem of Formalization
  - Fundamental Questions to be Asked
- Modeling Use Cases
  - ATM (Cash Machine)
  - Password Change
  - Alarm Clock
- Incremental Design (Refinement)
  - Invoice System
- Managing Teams of Agents
  - Telephone Exchange

# Simple ATM (Cash Machine): Problem Description

- Design an ATM, and show it to be well functioning, where via a GUI the customer can perform the following ops:
  - Enter the ID.
    - Only 1 attempt allowed per session, upon failure the card is withdrawn.
  - Withdraw money from the account.
    - Only one attempt is allowed per session. A warning is issued if the amount required exceeds the balance of the account.
  - Ask for the balance of the account.
    - Allowed only once and only before attempting to withdraw money.
- The central system (supposed to be designed separately) receives the information about every withdrawal and updates the account balance correspondingly.
  - The customer is not allowed to do any other transaction before this update has become effective.
- Refine the ATM to get out-of-service when not enough money is left

# ATM: Use Case Description sequencing of user ops



## Refining Operations for Cash ASM

**insert-card** = if inserted(curr-card) then out := enter-pin-msg

**enter-good-pin** = if curr-in  $\in$  ID & curr-in = pin(curr-card)  
& **accessible**(**account**(curr-card))  
then out := choose-service-msg

**enter-wrong-pin** = if curr-in  $\in$  ID &  
& (curr-in  $\neq$  pin(curr-card) or not **accessible**(**account**(curr-card)))  
then out := wrong-pin-msg (NB. curr-card is not returned)

**enter-money** = if curr-in  $\in$  MONEY then  
process-money-request (curr-in)

**process-money-request** (curr-in) = if **allowed** (curr-in, curr-card)  
then out := exit-msg # **money**(curr-in) # curr-card  
**call deletion** of curr-in from **account**(curr-card)  
**accessible** (**account**(curr-card)) := false

else out := not-allowed-msg # curr-card

# Signature/Constraints/Robustness Rules for Cash ASM

- shared function bw ATM and environment
  - **accessible**: updated by ATM from true to false, supposed to be updated by env (central system) from false to true
- monitored functions representing user input
  - **curr-card, curr-in** (assumed to be consumed by rule firing)
- static functions
  - account, pin, allowed (**may be updatable by the env**) , money
- Robustness refinement for rule If Cond then r :

## If not Cond then robustness-rule

NB. A robustness-rule usually has the form If Cond' then ..., where Cond' selects the relevant ways in which Cond is violated (preventing r from being fired) without other rules of the machine being enabled.

Example: Ignoring input in a given control state

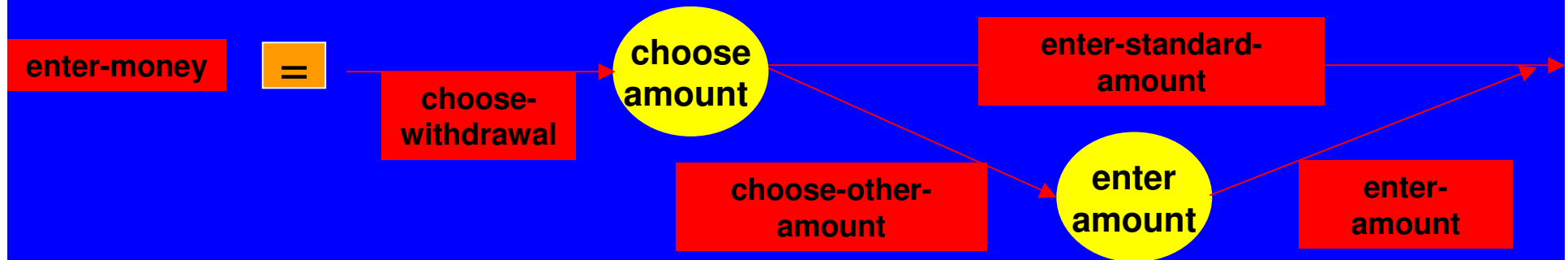
NB. CARD, MONEY kept abstract, not implemented (by numbers/units)

## Justifying the Well Functioning of the Cash ASM

- Per session only one attempt is allowed to enter a correct pin number, upon failure the card is withdrawn.
  - Follows from rule **enter-wrong-pin** (whereby the ATM is brought back to the await-card state, informing by msg3 without giving back the card)
- Per session only one attempt is allowed to withdraw money, a warning is issued if the amount required exceeds the balance.
  - Follows from rule **enter-money** (which brings the ATM back to the await-card state, giving back the card & in case warning by msg6)
- Asking for the balance of the account is allowed only once and only before attempting to withdraw money.
  - Follows from sequencing of operations in the control state diagram
- The central system receives the information about every withdrawal and updates the account balance correspondingly.
  - guaranteed by the (assumption on) opn **delete** in rule enter-money
- Any further transaction disallowed until account update.
  - guaranteed by shared predicate **accessible** in rules enter-good/wrong-pin

## Refining enter-money of Cash ASM by choice of money amounts

- For entering money, the choice bw standard amounts (say from a set Std) and a user-defined amount should be offered



**choose-withdrawal** = if curr-in = withdrawal then skip

• **enter-standard-amount** =

if curr-in  $\in$  Std then process-money-request (curr-in)

• **choose-other-amount** = (if curr-in = other- amount then skip)

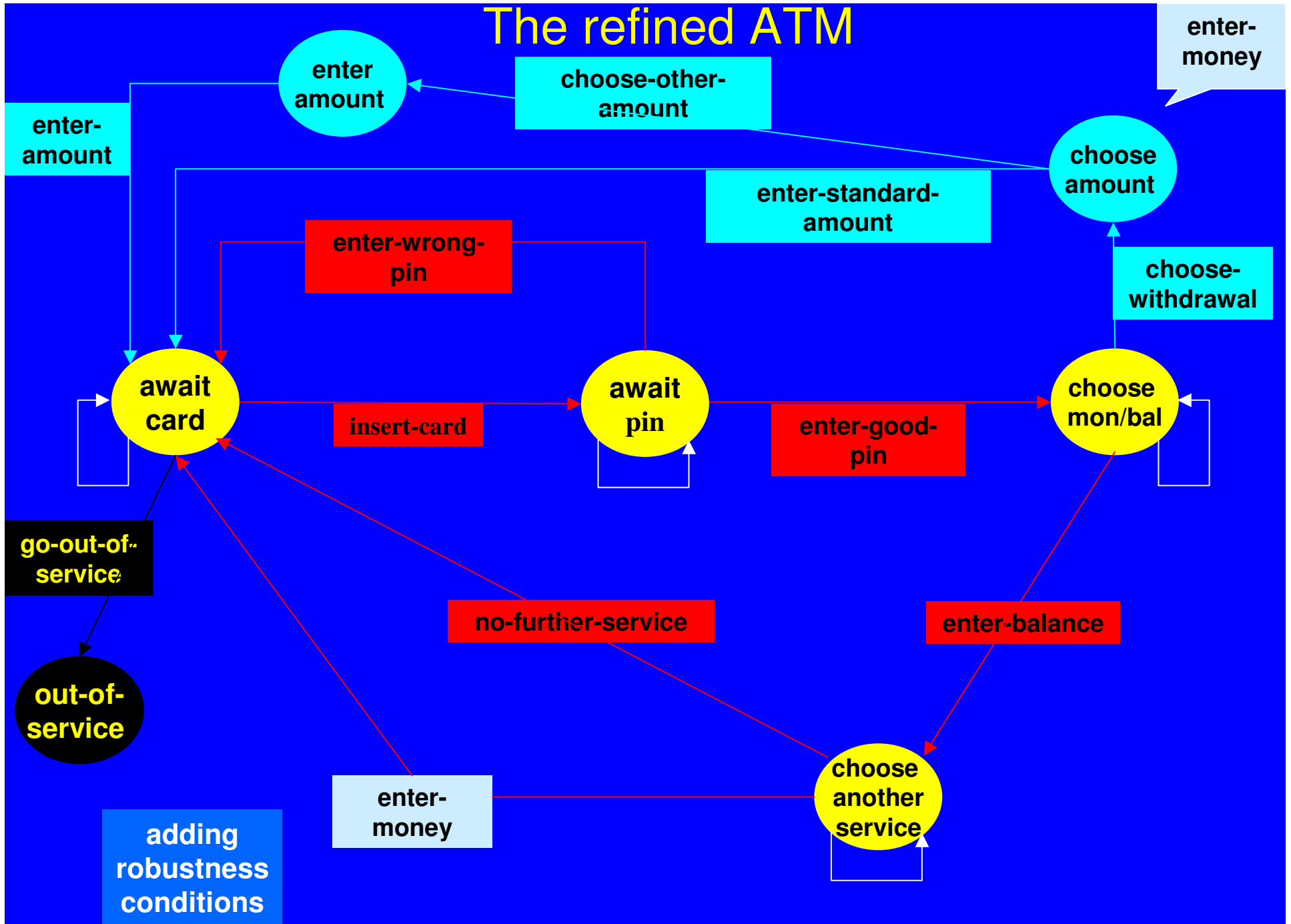
• **enter-amount** =

if curr-in  $\in$  MONEY then process-money-request (curr-in)

## Adding out-of-service when not enough money left

- Refine process-money-request (curr-in) by
    - adding  $\text{money-left} := \text{money-left} - \text{curr-in}$  to the then-branch
    - guarding rule by  $\text{money-left} - \text{curr-in} \geq 0$
    - issuing in the else-case a not-enough-money-left-msg
- $\text{process-money-request}(\text{curr-in}) = \text{if } \text{money-left} - \text{curr-in} \geq 0$   
 $\text{then } \text{if allowed}(\text{curr-in}, \text{curr-card})$   
     $\text{then out} := \text{exit-msg} \# \text{money}(\text{curr-in}) \# \text{curr-card}$   
         $\text{call deletion of curr-in from account}(\text{curr-card})$   
         $\text{accessible}(\text{account}(\text{curr-card})) := \text{false}$   
         $\text{money-left} := \text{money-left} - \text{curr-in}$   
     $\text{else out} := \text{not-allowed-msg} \# \text{curr-card}$   
 $\text{else out} := \text{not-enough-money-left-msg} \# \text{curr-card}$
- Add new ctl state out-of-service & new rule go-out-of-service
- $\text{go-out-of-service} = \text{if } \text{money-left} \leq \text{min} \text{ then } \text{ctl} := \text{out-of-service}$

# The refined ATM



## Implementing ASM control states by ATM-GUI

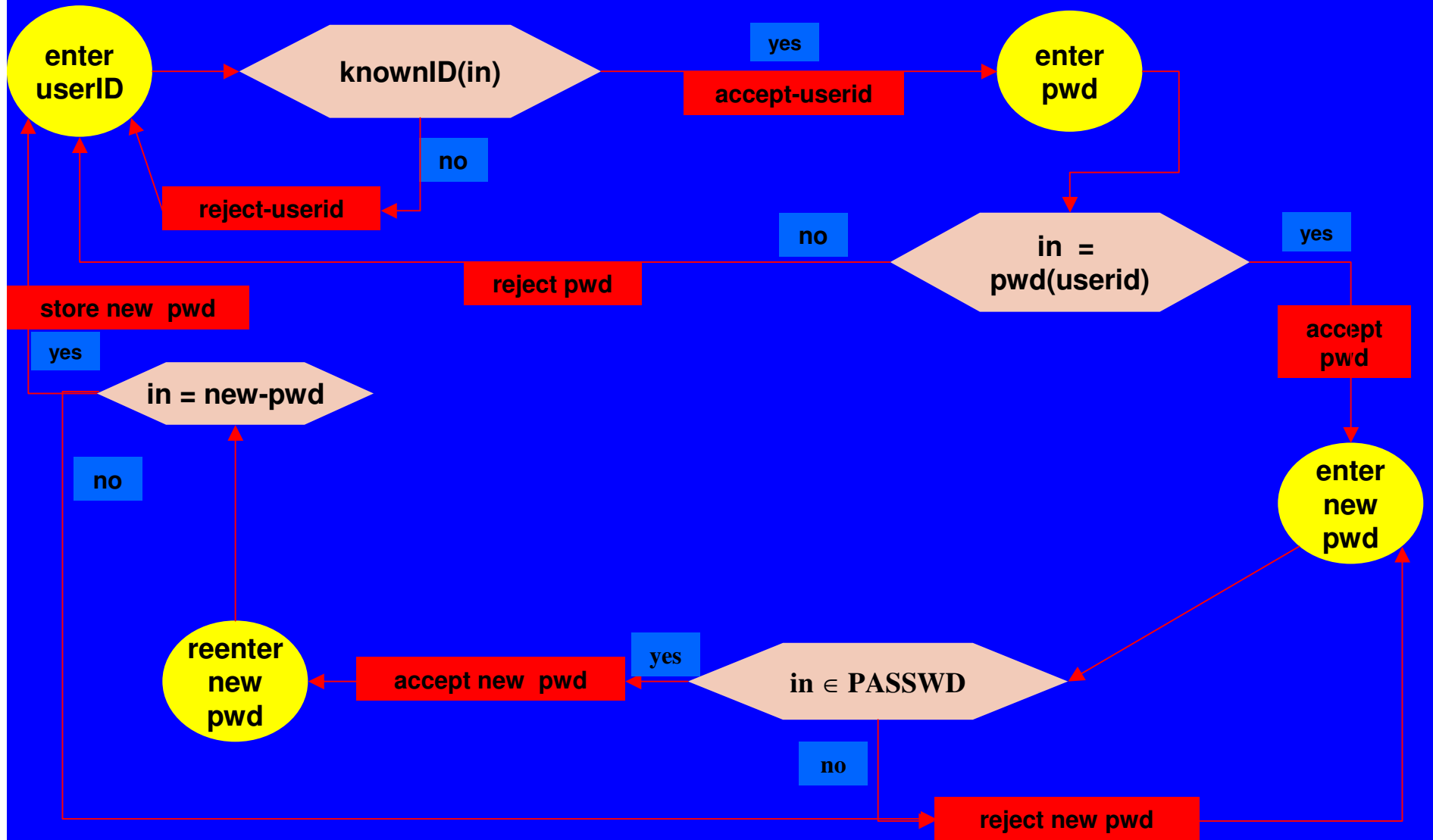
- **await card**  $\equiv$  customer screen “enter your card”
- **await pin**  $\equiv$  customer screen “enter your pin number & click ok”
- **choose mon/bal**  $\equiv$  customer screen “choose a service by clicking on one of - withdrawal - get an account statement”
- **choose another service**  $\equiv$  customer screen “choose another service by clicking on one of - no other service – withdraw money ”
- **choose amount**  $\equiv$  customer screen “choose desired amount by clicking on one of  $std_1$  ...  $std_n$  other amount ”
- **enter amount**  $\equiv$  customer screen “enter desired amount & click ok”
- **ctl = c refined as currscreen = screen (c)**

- The Formalization Problem
  - Fundamental Questions to be Asked
- Modeling Use Cases
  - ATM (Cash Machine)
  - Password Change
  - Alarm Clock
- Incremental Design (Refinement)
  - Invoice System
- Managing Teams of Agents
  - Telephone Exchange

## Password Change with Line Editor: Problem Description

- Design a pgm, and show it to be well functioning, which allows a customer to change his password using the following sequence of operations:
  - Enter the user ID and the (current) password.
    - Access should be refused if the ID or the password is incorrect.
  - Enter twice the new password.
    - The new password request should be rejected if the new password is syntactically incorrect or not correctly repeated. Otherwise the new password should be stored as valid password from now on.
- Refine the machine by a line editor for inputting passwords
- Refine the machine by restricting no. of attempts to define new pwd

# Password Change Use Case Description: Sequing of User/System Ops



## Signature & Refining actions of Password Change ASM

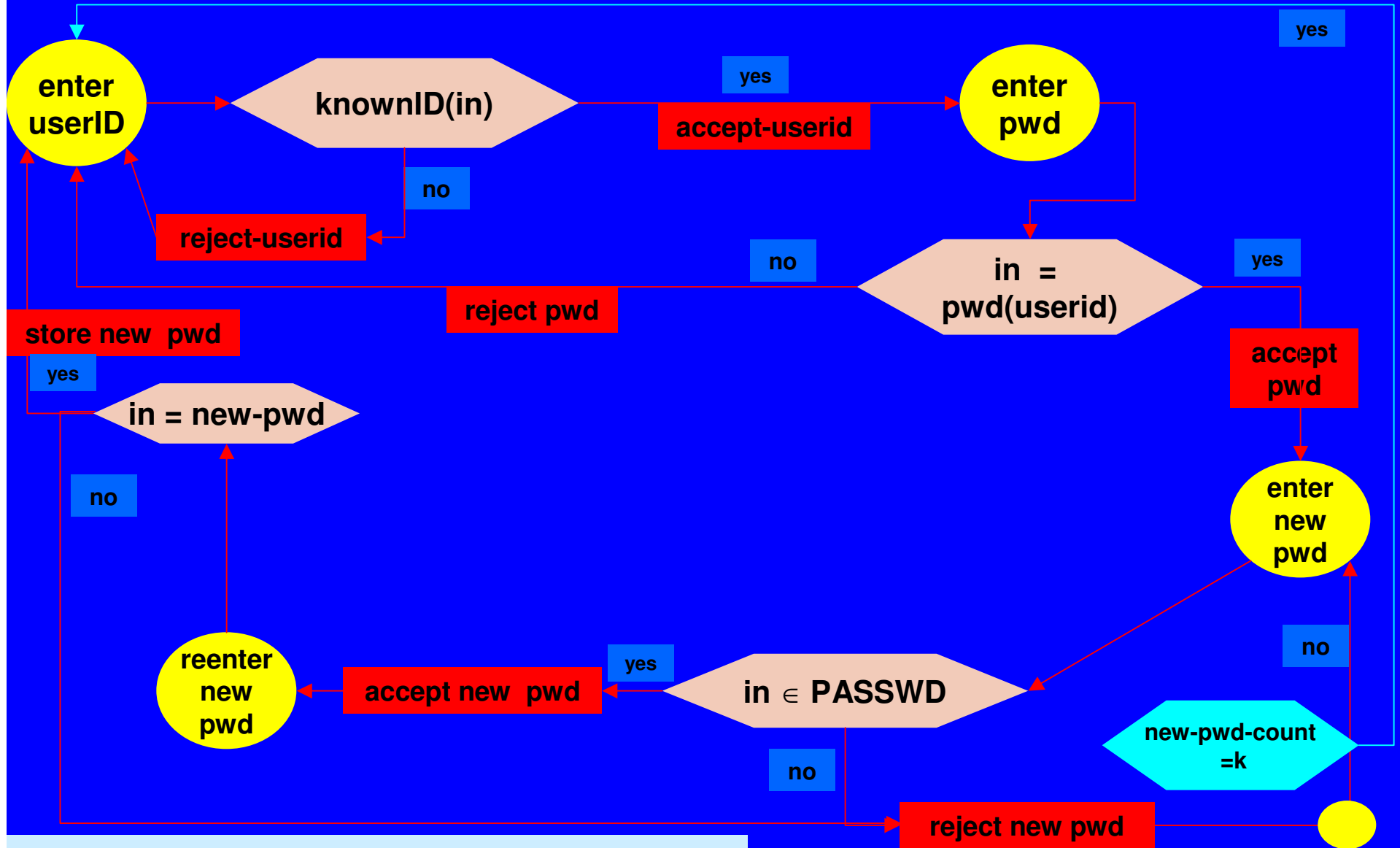
- static function knownID
- dynamic function pwd
- accept-userid = userid: = in
- accept pwd = reject pwd = skip
- accept new pwd = new-pwd := in
- reject-userid = msg:= in is not a legal user ID
  - similarly for reject pwd and reject new pwd
- store new pwd = pwd(userid) := new-pwd



# Line Editor Exercise

- Define a buffer with operations
  - InsertChar, DeleteChar
  - ForwardCursor, BackwordCursorfrom given operations
  - insert(String, Character, Position)
  - delete (String, Position)
- Add display-oriented features, distinguishing between printable and unprintable characters, using operations
  - prefix(String,l) providing the prefix of length l of String
  - blanks(Length) yielding a string of blanks of given Length
  - fill(String, l) filling String with blanks to reach at least length l

# Refining Password Change by restricting attempts to define new pwd



adding to reject new pwd  
 new-pwd-count := new-pwd-count + 1

accept pwd = new-pwd-count:=0

# Justifying the Well Functioning of the Password Machine

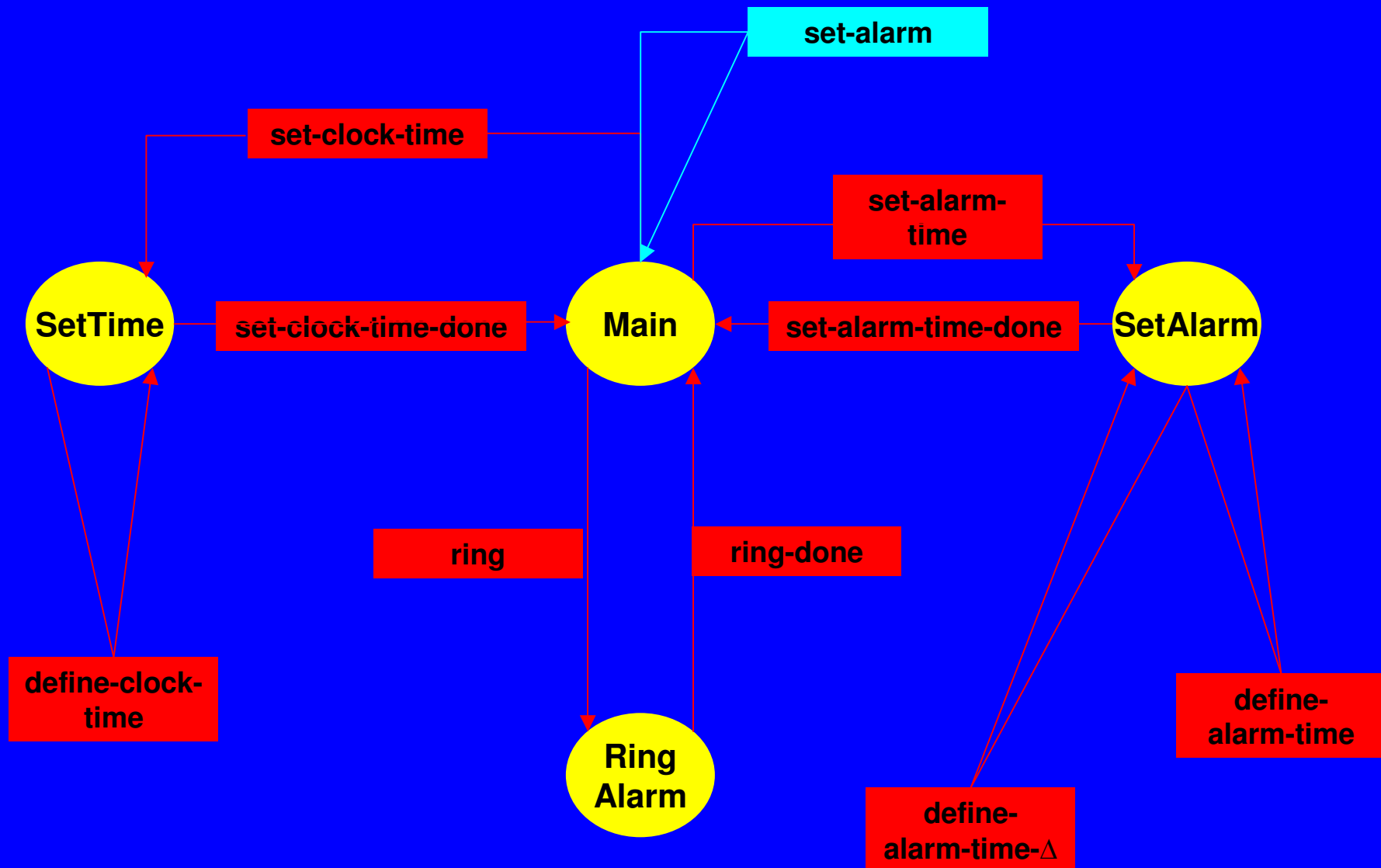
- Access should be refused if the ID or the password is incorrect.
  - Follows from rules **reject-userid**, **reject pwd** (whereby the machine is brought back to its initial control state `enter usrID` )
- The new password request should be rejected if the new password is syntactically incorrect or not correctly repeated.
  - Follows from rule **reject new pwd**, which is applicable only in control states `enter new pwd` and `reenter new pwd`. In the second case note the update of `new-pwd` in **accept new pwd**.
- Otherwise the new password should be stored as valid password from now on.
  - Follows from rule **store new pwd**

- The Formalization Problem
- Modeling Use Cases
  - ATM (Cash Machine)
  - Password Change
  - Alarm Clock
- Incremental Design (Refinement)
  - Invoice System
- Managing Teams of Agents
  - Telephone Exchange

## Simple Alarm Clock: Problem Description

- Design an alarm clock, which automatically every second updates & displays currtime, and allows setting of currtime or alarm time (hour, minute) by the user.
  - currtime is displayed at each update
  - to initiate (re)setting curr/alarm time, the user pushes a “time”/”alarm” button
  - the clock, upon reaching the alarm time, rings until
    - either the user cancels the alarm by pressing the “alarm stop” button
    - or 30 seconds have elapsedexecution of ring may be made dependable upon the position of an alarm-on/off-button, to be set by the user
- Refine the alarm to provide for 3 consecutive ringings with an interval of 5 minutes between them

# Alarm Clock Use Case Description: Sequencing ops



# Simple Alarm Clock: Signature & Constraints

- **curtime** a shared fct, updated by either
  - **env**, each second, as long as the action `set-clock-time-done` is not executed
    - Variant: `ctlState`  $\neq$  `SetTime` in case there is only one display, to be used also for the newly set time
  - or **Alarm Clock ASM (rule `set-clock-time-done`)**  
othw
- **ring** a spontaneous (not input triggered) action of the machine

## Simple Alarm Clock: Defining clock-time Actions

- **set-clock-time** = if in = set-time-button-pushed  
then skip
- **define-clock-time** = if in  $\in$  TIME  
then new-time := **convert** (in)

NB. new-time may be displayed (see refinement below)

- **set-clock-time-done** =  
if in = set-time-button-pushed then  
if new-time  $\neq$  undef then currtime: = new-time  
new-time := undef  
else skip

(Leaving currtime unchanged if the user did not set new-time)

## Simple Alarm Clock: Defining alarm-time Actions

- **set-alarm-time = set-alarm-time-done =**  
if in = set-alarm-button-pushed then skip
- **define-alarm-time =**  
if in  $\in$  TIME then alarm-time := in
- **A Variant (requesting alarm at  $\Delta$  from now on):**  
**define-alarm-time- $\Delta$  =**  
if in  $\in$  TIME then alarm-time := currtime + in

# Simple Alarm Clock: Ringing Actions

- **ring** = if currtime = alarm-time  
then ring-time:=currtime
- **ring-done** = if in = alarm-on/off-button-pushed  
or currtime = ring-time + 30 seconds  
then skip

Refining to 3 consecutive ringings at interval  $\varepsilon$ :

add the following updates to **ring**:

if ring-no  $\in$  {1,2} (ring-no initialized by 1)

then alarm-time := currtime +  $\varepsilon$

ring-no := ring-no + 1

else ring-no := 1

## Simple Alarm Clock: Refining Ringing Actions

- **Refining ring** making it dependable on the position of the alarm-on/off-button, to be pulled/pushed by the user

**ring** = if currtime = alarm-time &

    alarm-on/off-button = on then ring-time:=currtime

**ring-done** = **set-alarm-off** =

    if in = alarm-on/off-button-pushed

    then alarm-on/off-button := off

- **set-alarm-on** = if in = alarm-on/off-button-pulled  
    then alarm-on/off-button := on

# Simple Alarm Clock: Data Refinements

- derived fct **display**: TIME (say for currtime)

display =

- currtime | MINUTE for 24-hrs-display
  - currtime a.m. if currtime  $\leq$  12.00.00
  - (currtime – 12.00.00) p.m. othw for 12-hrs-display

- **refinement to using display also for time setting**

display =

- currtime if ctlState = Main or ctlState = RingAlarm
- new-time if ctlState = Set Time
- alarm-time if ctlState = SetAlarm

- The Formalization Problem
  - Fundamental Questions to be Asked
- Modeling Use Cases
  - ATM (Cash Machine)
  - Password Change
  - Alarm Clock
- Incremental Design (Refinement)
  - Invoice System
- Managing Teams of Agents
  - Telephone Exchange

# Incremental Machine Design: Invoice System

- Specify a one-user system to handle **invoices** for orders from **clients**, and to accordingly maintain the record of **products** (prices and availability in stock), showing also how to incorporate subsystems for **error detection** and **statistics**. Refine the spec by specifying also an appropriate **GUI**.
  - The system should provide a (“for change” ) extendable set of operations including the following ones (details below) :
    - creating/modifying products, clients, invoices
    - reporting errors in handling products, invoices, clients
    - retrieving information on clients, products, invoices and their past
  - For customers’ inspection, use abstract & application domain oriented ops, refinable by more detailed equiv ops
- Illustrate the restrictions of the “one-user one-operation per time system” wrt a distributed multiple-user version

## Invoice System: Informal Description (cont'd)

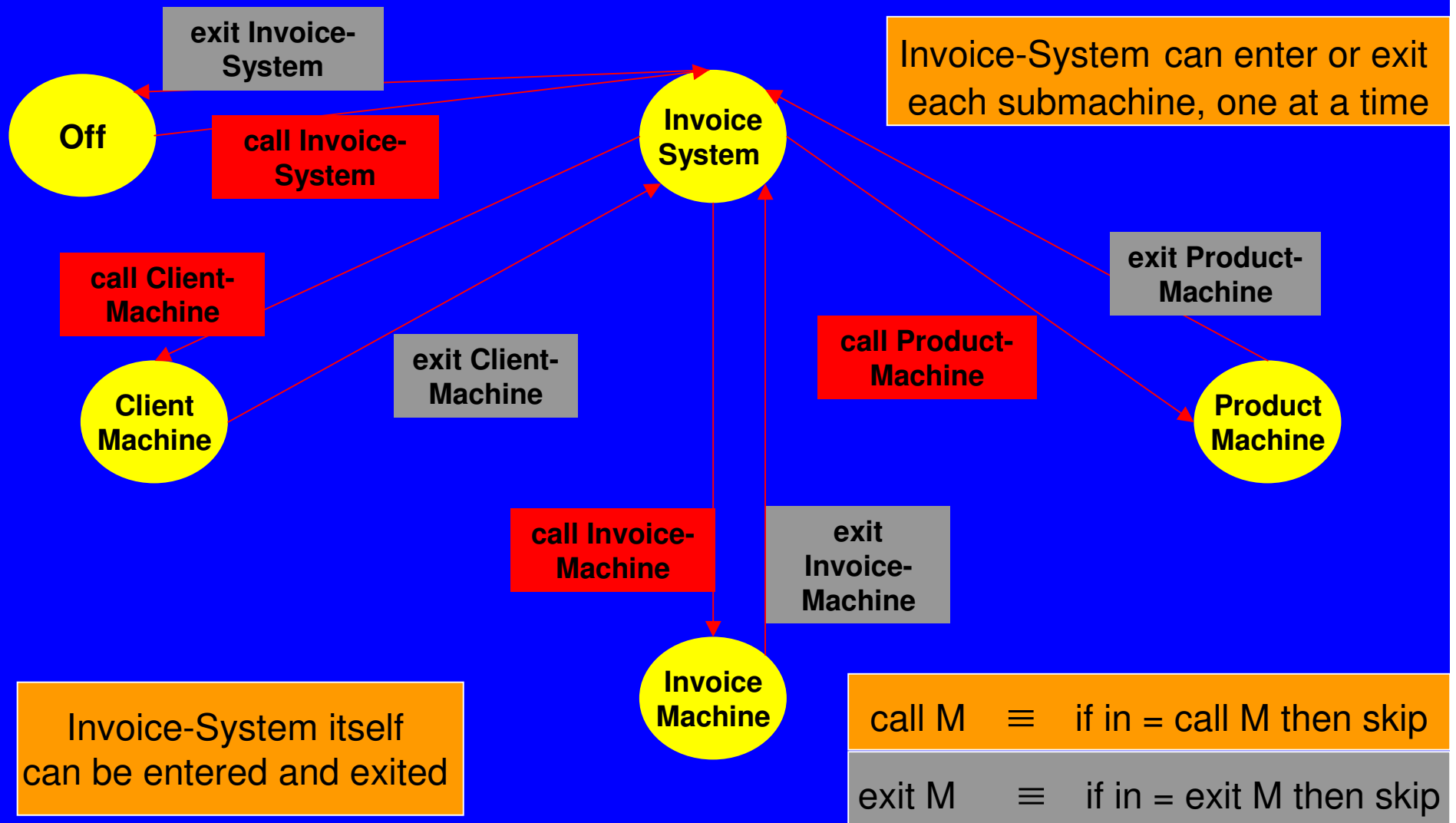
- **Invoices** are issued to clients, have a discount (percentage of the total) & a maximum allowance, both defined upon invoice creation from corresponding client attributes. Invoices have a set of positions (e.g. lines), one per ordered product, holding the product's unit price (taken upon creation of the position from the price of the product) & the ordered quantity.
- **Clients** are classified (e.g. into normal, friend, dubious), yielding a discount (per category), and have a maximum allowance which is applied to each of their invoices.
- **Products** have price, status (available, sold out), possibly a substitute (another product guaranteed not to be sold out).

# Invoice System: Required Properties

- Provide an **error report** when, upon inserting a new product (or its substitute) or upon incrementing its quantity in an invoice:
  - the product is sold out and there is no available substitute
  - the maximum invoice allowance (of the discounted total) would be exceeded by adding the product (or its substitute) to the inv
- Report parameter type errors which may occur in particular when named rules are called
- **Prove** that the system satisfies the following conditions:
  - a sold out product is never made part of an invoice, the system will automatically replace it by its substitute (if there exists one)
  - friend clients (nobody else) get a discount (the same for all), no invoice is ever made for dubious clients
  - (discounted) total of an invoice  $\leq$  maximum invoice allowance
  - no invoice lists an ordered product more than once

# Invoice-System built from Submachines

- Invoice-System split into submachines, each with its own operations, specifically related to its main domain of clients, invoices, products



# Client-Machine : Signature

- CLIENT : main (dynamic) domain, partitioned by category
- category : CLIENT  $\rightarrow$  CATEGORY = {normal, friend, dubious}
- allowance : CLIENT  $\rightarrow$  N (suppressing the currency)
- address : CLIENT  $\rightarrow$  ADDRESS , etc. for other attributes
- discount : CATEGORY  $\rightarrow$  { n/100 | n  $\in$  [0, ..., 100] }  
governed by category satisfying discount(friend) = f-disc ,  
discount(c) = 1 else
- The machine provides an extendable set of operations, e.g.
  - c  $\leftarrow$  create-client(Category, Allow, Address) (c, params optional)
  - modify-attribute(Client, Val) optional lists attribute, Val
  - inspect-attribute(Client)
  - inspect-archive(Client) providing archival info on Clients's past attributes, invoices, ...

## Client-Machine : Defining create-rules

- **c** ← **create-client (Category, Allow,...)** ≡  
If ok (Category, Allow, ...) then (type check the params!)  
let Client = new (CLIENT) in (providing fresh object in domain)  
category(Client) := Category  
allowance(Client) := Allow  
... (set default values for attributes ∉ list of params!)  
c := Client

NB. let x = new(Domain) in Rule abbreviates

let x = new in ( Domain(x) := true, Rule )

- **Reporting parameter type errors:** add the else-branches  
If not ok (Attribute) then report type-error (Attribute)
- **Reflecting resource bounds:** refine the above defined Rule to  
If |CLIENT| < max-CLIENT then Rule else report “CLIENT is full”  
adding |CLIENT| := |CLIENT| +1 to then-branch of Rule

## Client-Machine : Defining modify-rules

- **modify-attribute** (Client, AttributeVal)  $\equiv$   
If ok (AttributeVal) then attribute(Client) := AttributeVal

**Structured variant** (encapsulating a flat set of ops modify-attribute belonging to Client-Machine): call a machine **modify** (Client) that provides a set of ops modify-attribute (AttributeVal)

For report of parameter type errors, add else-branch:

If not ok (AttributeVal) then report type-error (AttributeVal)

For archival purposes, one may enrich modify-rules by storing old values via history updates in the then-branch, e.g.

attribute-history(Client) :=  
attribute-history(Client) #attribute(Client)

## Client-Machine : Defining inspect-rules

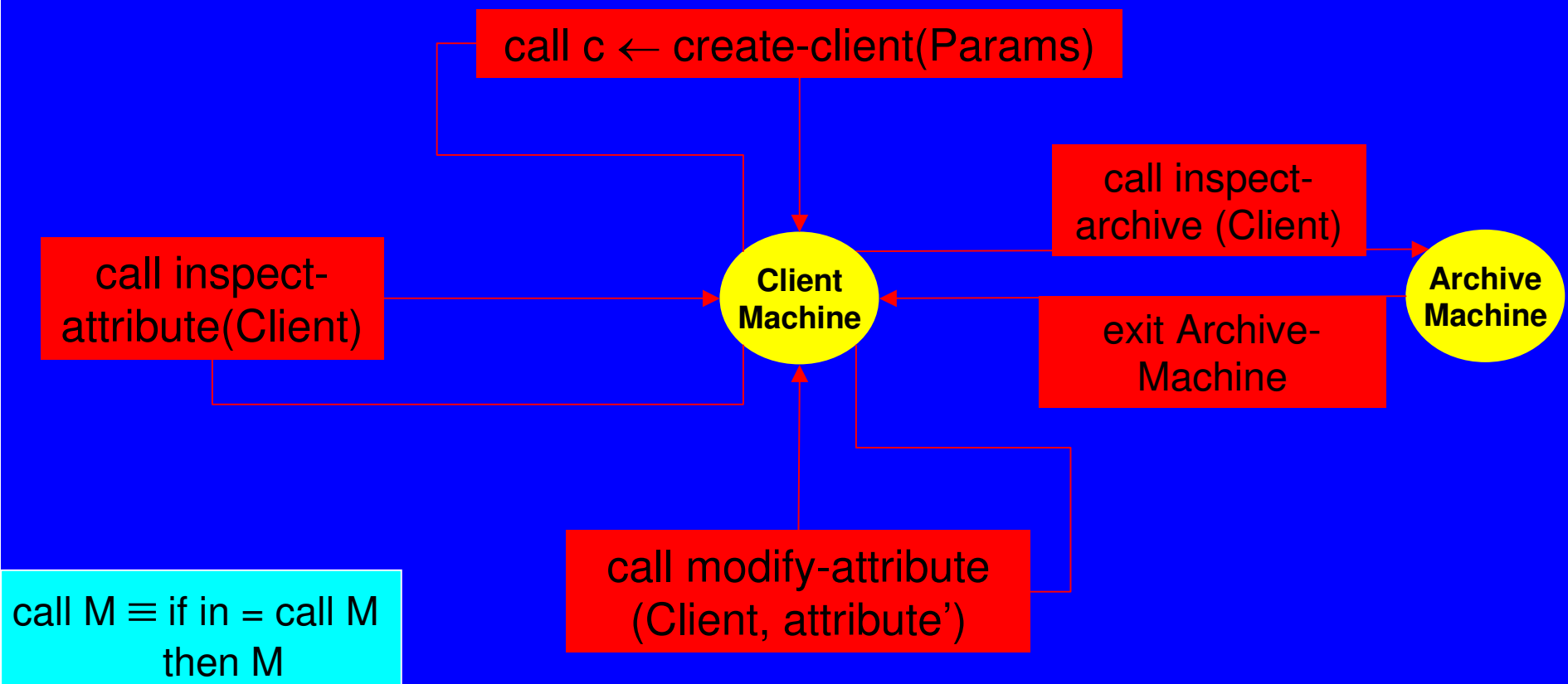
**inspect-attribute (Client)  $\equiv$**

If **inspection-permitted (attribute, Client)** (check access rights)  
then **display := attribute (Client)** (use abstract **display loc**)

- **inspect-archive (Client)** defined to enter a submachine with new locations supposed to store info on Clients's past attributes, invoices,..., and operations to retrieve this info
- Easy extendability: following the above scheme one can add further rules, e.g. for payment control, etc.

# Client-Machine : User Input Driven Execution of Rules

- The abstract one-user Client-Machine looks like a 1-control-state (flat) ASM executing at each moment at most one of its named rules.
- inspect-archive(Client) looks like a call of another machine, offering (maybe returning results of) information retrieval operations.



# Product-Machine : Signature

- **PRODUCT** : dynamic domain with bound **max-PRODUCT** : **N**
- **price** : **PRODUCT**  $\rightarrow$  **N** (static list-price fct)
- **status** : **PRODUCT**  $\rightarrow$  **STATUS** = {avail, sold-out}
- **substitute** : **PRODUCT**  $\rightarrow$  **PRODUCT**  $\cap$  { **p** | **status** (**p**)=avail }  
(NB: value may be undef if there is no available substitute)
- The machine provides an extendable set of operations, e.g.
  - **p**  $\leftarrow$  **create-product** (**Price**, **Status**, **Substitute**)  
(**p**, params optional)
  - **modify-attribute**(**Product**, **Val**) optional lists **attribute**, **Val**
  - **inspect-attribute**(**Product**)

## Product-Machine : rules for correct substitute handling

Goal: guarantee that only available products serve as substitute

- **make-unavailable (Prod)**  $\equiv$  If  $\text{Prod} \in \text{PRODUCT}$  then  
     $\text{status}(\text{Prod}) := \text{sold-out}$   
    delete Prod as substitute

( $\equiv$  forall  $p \in \text{PRODUCT}$  do if  $\text{substitute}(p) = \text{Prod}$  then  $\text{substitute}(p) := \text{undef}$  )

- **assign-substitute (Prod, Prod')**  $\equiv$   
    If  $\text{Prod}, \text{Prod}' \in \text{PRODUCT}$  then  
        If  $\text{status}(\text{Prod}') = \text{available}$   
            then  $\text{substitute}(\text{Prod}) := \text{Prod}'$   
            else report "Prod' is unavailable & cannot serve as substitute"

Extension by a rule to call **make-unavailable** for Prod when in the real business it has been sold out, and of a strategy to replace Prod as a substitute: choosing a Prod' to call **assign-substitute** (-,Prod'). This necessitates to refine  $\text{status}(\text{Prod})$  to keep track, in the Invoice-Machine, of the quantity of available items Prod.

# Invoice-Machine : Signature

- **INVOICE** main dynamic domain with bound **max-INVOICE : N**
  - Invoice-Machine uses (fcts/domains of) **Client/Product-Machine**
  - **client : INVOICE → CLIENT** (“invoices are issued to clients”)
  - **discount: INVOICE → { n/100 | n ∈ [0,...,100]}**
  - **allowance: INVOICE → N** (overloading of fct names)
  - **total : INVOICE → N** constraint: total (i) ≤ allowance (i) for all i
  - **POSITION**: auxiliary domain, to locate the items of (the “objects” occurring in) invoices, with bound **max-POSITION : N** and item attributes:
    - **article: POSITION → PRODUCT** (article occurring in a position)
    - **quantity, unit-cost : POSITION → N**
    - **inv: POSITION → INVOICE** (yields THE Invoice where Pos belongs to)
    - **pos: INVOICE → PowerSet (POSITION)** (set of ordered items in
- NB.  $\text{pos}(i) = \text{inv}^{-1}(\{i\})$  invoice)

# Invoice-Machine : Creating/Deleting Invoices

- **$i \leftarrow \text{create-invoice}(\text{Client}) \equiv$**

If  $\text{Client} \in \text{CLIENT}$  &  $\text{category}(\text{Client}) \neq \text{dubious}$  then  
if  $|\text{INVOICE}| < \text{max-INVOICE}$

then let  $\text{Inv} = \text{new}(\text{INVOICE})$  in

$\text{client}(\text{Inv}) := \text{Client}$

$\text{discount}(\text{Inv}) := \text{discount}(\text{category}(\text{Client}))$       taken upon  $\text{Inv}$  creation

$\text{allowance}(\text{Inv}) := \text{allowance}(\text{Client})$                       taken upon  $\text{Inv}$  creation

$\text{total}(\text{Inv}) := 0$

$i := \text{Inv}$

$|\text{INVOICE}| := |\text{INVOICE}| + 1$

else report “INVOICE is full”

- **$\text{remove-invoice}(\text{Inv}) \equiv$**  If  $\text{Inv} \in \text{INVOICE}$

    then  $\text{INVOICE}(\text{Inv}) := \text{false}$ ,  $|\text{INVOICE}| := |\text{INVOICE}| - 1$

    else report “you are trying to remove something different from an invoice”

One may add to (the then-branch of) this rule some **garbage collection updates**:  $\text{client/discount/allowance/total}(\text{Inv}) := \text{undef}$

## Invoice-Machine : inserting new items into invoices

- **insert-product (Inv, Prod)  $\equiv$**

If  $Inv \in \text{INVOICE}$  &  $Prod \in \text{PRODUCT}$  then

If  $\text{status}(Prod) = \text{avail}$

then If  $Prod$  does not occur in  $Inv$  (for all  $p \in \text{pos}(Inv) : \text{article}(p) \neq Prod$ )

then if  $|\text{POSITION}| < \text{max-POSITION}$

then let  $Pos = \text{new}(\text{pos}(Inv))$  in

$\text{article}(Pos) := Prod$

$\text{quantity}(Pos) := 0$

$\text{unit-cost}(Pos) := \text{price}(Prod)$  (taken upon  $Pos$  creation)

$\text{inv}(Pos) := Inv$  (each  $Pos$  must belong to exactly 1  $Inv$ )

$|\text{POSITION}| := |\text{POSITION}| + 1$

else report “there is no position left in this invoice”

else report “ $Prod$  has already been inserted into this invoice”

(each product is allowed to appear at most once in an  $Inv$ )

else If  $\text{substitute}(Prod) \neq \text{undef}$  &  $\text{status}(\text{substitute}(Prod)) = \text{avail}$

then  $\text{insert-product}(Inv, \text{substitute}(Prod))$  (recursive rule!)

else report “ $Prod$  is unavailable and without available substitute”

## Invoice-Machine : Searching for Products in Invoices

- **Pos**  $\leftarrow$  **the-position (Inv, Prod)**  $\equiv$

(find out whether and where a product occurs)

If  $Inv \in \text{INVOICE}$  &  $Prod \in \text{PRODUCT}$  then

If Prod does occur in Inv

(i.e. for some  $p \in \text{pos}(Inv) : \text{article}(p) = \text{Prod}$ )

then  $\text{Pos} := \iota p (p \in \text{pos}(Inv) : \text{article}(p) = \text{Prod})$

(Hilbert's  $\iota$ -operator describing "the" unique object satisfying Q)

else report "Prod does not appear in Inv"

# Incrementing Quantity of Prods in Invoice Positions

- **increment-product-in-position (Pos, Quantity)  $\equiv$**

If  $Pos \in \text{POSITION}$  &  $Quantity \in \mathbb{N}$  then

If  $\text{status}(\text{article}(Pos)) = \text{avail}$

then let  $\Delta = \text{Quantity} \times \text{unit-cost}(Pos) \times \text{discount}(\text{inv}(Pos))$

if  $\text{total}(\text{inv}(Pos)) + \Delta \leq \text{allowance}(\text{inv}(Pos))$  (check inv-total limit)

then  $\text{quantity}(Pos) := \text{quantity}(Pos) + \text{Quantity}$

$\text{total}(\text{inv}(Pos)) := \text{total}(\text{inv}(Pos)) + \Delta$

else report “Adding Quantity to Pos would exceed the allowance”

else If  $\text{substitute}(\text{article}(Pos)) \neq \text{undef}$  &

$\text{status}(\text{substitute}(\text{article}(Pos))) = \text{avail}$

then ??? report “Prod has been replaced by its substitute”

else report “Prod is unavailable and without available substitute”

??? One could refine the rule by a parameter *article*, & replace ??? by **increment-product-in-position(substitute(article(Pos)), Pos, Quantity)**

But still some questions remain: should *article*(Pos), which had been initialized by **create-position(Inv, Prod)**, be updated to its substitute? Similarly for *unit-cost*(Pos), or is  $\text{price}(\text{Prod}) = \text{price}(\text{substitute}(\text{Prod}))$  assumed?

# History Ex1: keeping track of stock-availability

To keep track of the quantity of available items it suffices to refine rule **increment-product-in-position** (**Pos**, **Quantity**) by

- replacing the guard **status (...) = avail** by the more detailed condition **Quantity ≤ stock-quantity (...)**
- add in the inner then-branch the update  
**stock-quantity (...) := stock-quantity (...) – Quantity**

A rule with monitored guard may reflect stock increases:

If **supply (n of Prod)** then

**stock-quantity (Prod) := stock-quantity (Prod) + n**

Similar updates apply at other **creating/modifying** rules.

NB. Using **stock-quantity**, status becomes a derived fct, say

**status(Prod)=sold-out iff stock-quantity(Prod) < min**

One also gets a rule to call **make-unavailable** when stock gets low:

If **stock-quantity (Prod) < min (Prod)** then **make-unavailable (Prod)**

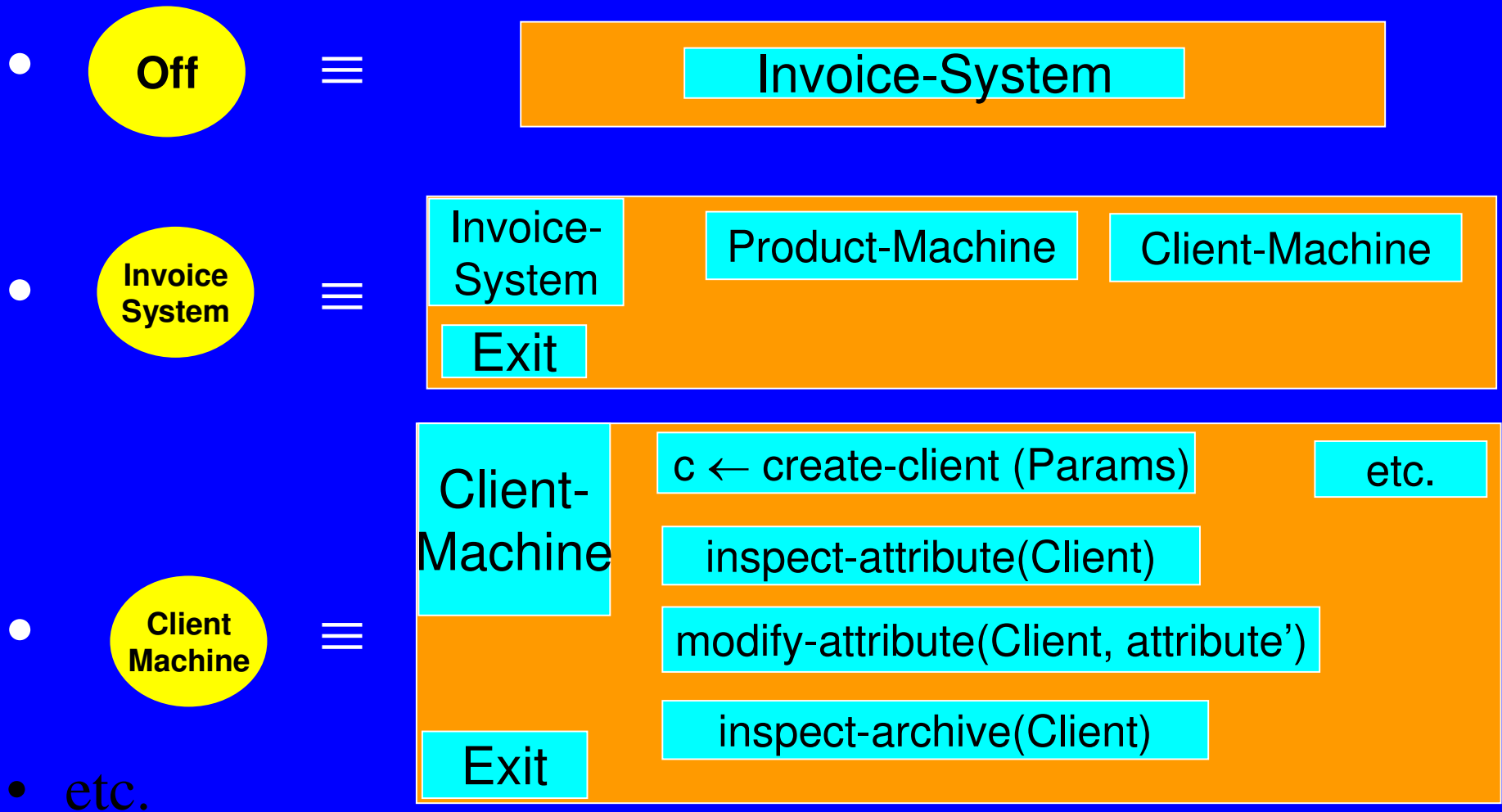
# Invoice System: Proving the Required Properties

- Report if upon inserting a new product (or its substitute) or upon incrementing its quantity in an Invoice:
  - the product is sold out and there is no available substitute
  - the maximum invoice allowance (of the discounted total) would be exceeded by adding the product (or its substitute):  
this has been realized in insert-product (Inv, Prod) and in increment-product-in-position (Pos,Quantity)
- Report parameter type errors in named rules: this has been shown in many rules
- Prove the following properties for the system:
  - a sold out product is never made part of an invoice, the system will automatically replace it by its substitute (if there exists one)  
Follows from the definition of rules insert-product (Inv, Prod) and increment-product-in-position (Pos,Quantity)

## Invoice System: Proving the Required Properties (Cont'd)

- **friend clients (nobody else) get a discount (the same for all)**  
statically guaranteed by **discount** condition in Client Machine:  $\text{discount}(\text{friend}) = \text{f-disc}$  ,  $\text{discount}(c) = 1$  else.  
For the run-time it suffices to prove that whenever **category** of a client is changed, then also its **discount** is changed correspondingly.
- **no invoice is ever made for dubious clients**  
guaranteed by the guard **category (Client)  $\neq$  dubious** in  $i \leftarrow \text{create-invoice}(\text{Client})$
- **(discounted) total(invoice)  $\leq$  maximum invoice allowance**  
guaranteed by guard of the only rule which can change **total(invoice)**, namely **increment-product-in-position (Pos, Quantity)**
- **no invoice lists an ordered product more than once**  
guaranteed by the guard **Prod does not occur in Inv** in the rule **insert-product (Inv, Prod)**

# GUI refinement of control states & input by screens & clicking options



Pure Data Refinement

ctl = c refined as  
currscreen = screen (c)

in = call M refined by button M clicked

in = exit M refined by button Exit in M clicked

# Refining attributes and their updates: par refinement

- Parallel refinement of one rule by multiple rules:

e.g. replacing `address := Address` by the following updates with Address refined to (First Name, Family Name, Street, Town, County, Post-Code, Country):

`first-name := First Name`

`family-name := Family Name`

`street := Street`

`town := Town`

`county := County`

`post-code := Post-Code`

`country := Country`

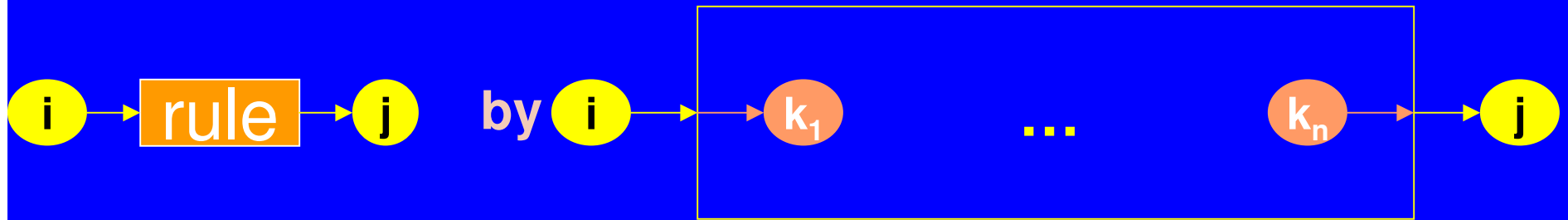
NB. All updates are

executed in parallel

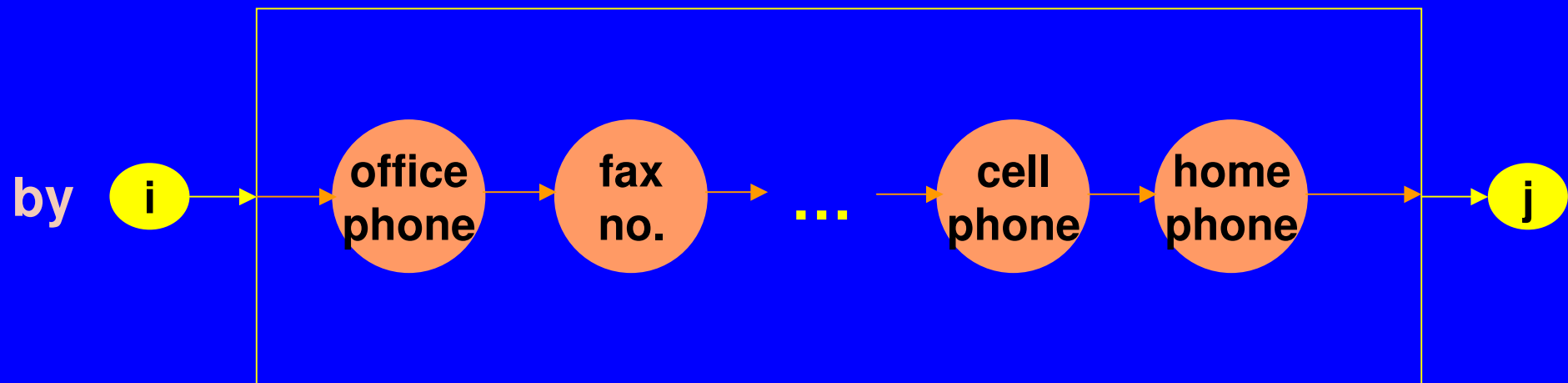
NB. refinement is 1 to 1

step & obviously correct

# Refining attributes and their updates: seq refinement replacing rule by a sequential machine



1 to n step  
refinement



refining the data structure phone &  
sequentializing its updates piecemeal

# Restrictions of one-user one-operation per time

- Parameterizing **ctl**, **in** by agents (“self”) provides multiple instances of Invoice-System Machine.
- But how to distribute the use of CLIENT, PRODUCT, INVOICE and of their functions to multiple users?
  - Pbl of global fcts in component machines
    - e.g. max-CLIENT, stock-quantity, substitute,...
  - Pbl of shared fcts in simultaneous execs of rules, e.g.
    - modify-attribute or create-object rules with same params
    - modify & remove rules for related items (e.g. for an invoice)
    - increment-product-in-position & supply for the same product
  - Pbl of how far the parallelism should go
    - different agents working on same invoice?
  - Pbl of security conds: should allowance of clients remain per invoice when multiple invoices can be issued simuly for same CI?

## Exercise on Electronic Commerce Models

- Extend the above ASM model for the Invoice System for describing the state changes involved in electronic commerce negotiations, concerning the traded products, the negotiators, their orders, the laws accepted as basis for the particular negotiation, etc. See
  - B. Fordham and S. Abiteboul and Y. Yesha: Evolving Databases: An Application to Electronic Commerce. Proceedings of the International Database Engineering and Applications Symposium (IDEAS), August 1997, Montreal.

- The Formalization Problem
  - Fundamental Questions to be Asked
- Modeling Use Cases
  - ATM (Cash Machine)
  - Password Change
  - Alarm Clock
- Incremental Design (Refinement)
  - Invoice System
- Managing Teams of Agents
  - Telephone Exchange

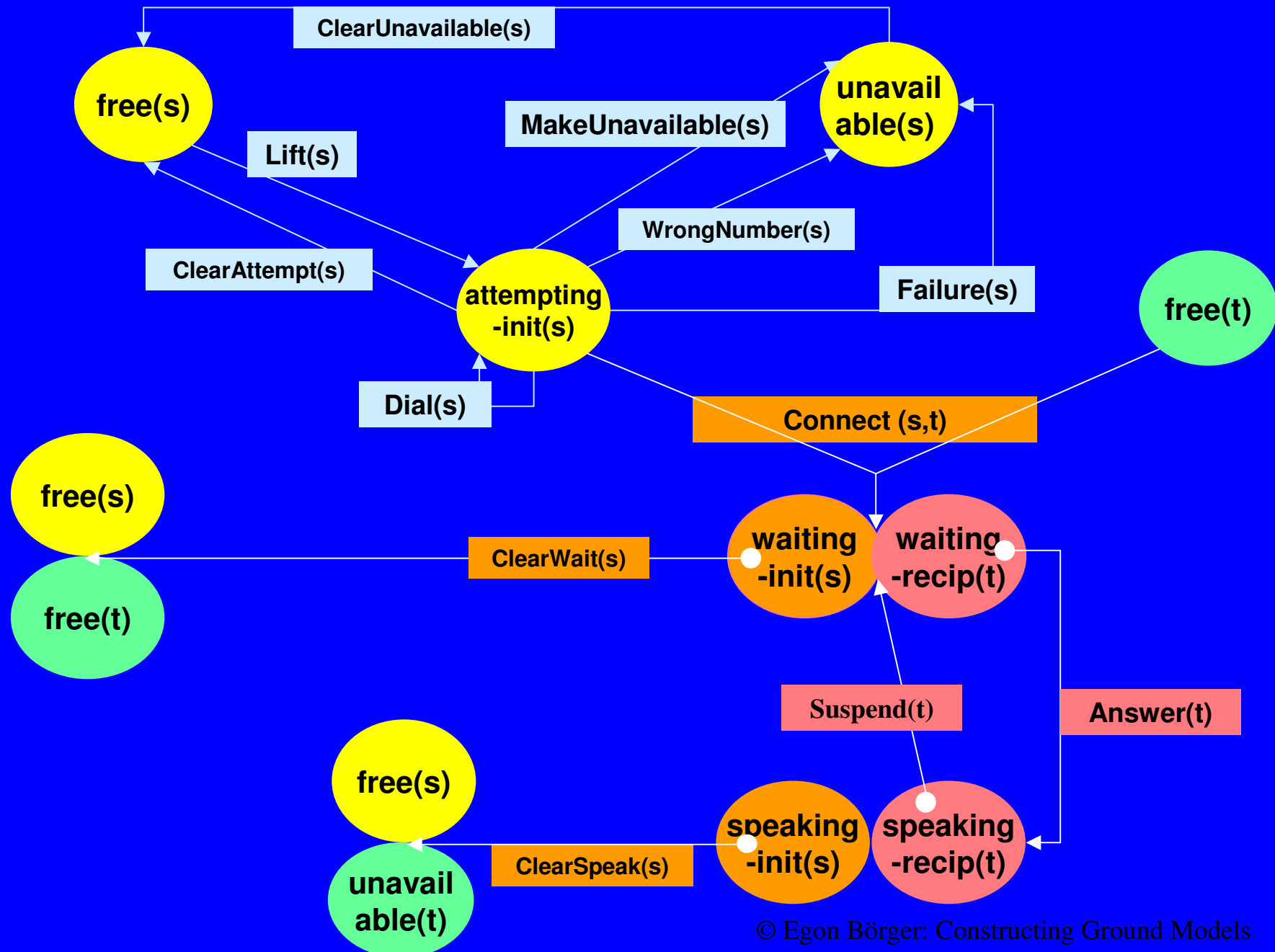
# Telephone Exchange: Status of Subscribers

- The system controls a network through which pairs among a set of **subscribers** establish & clear phone conversations
- Subscribers at each moment are in one **status** out of:
  - **free** (neither engaged in, nor attempting, any phone conversation)
  - **unavailable** due to a timeout or an unsuccessful attempt to call
  - initiator status :
    - **attempting-init** attempting to call somebody by dialing his number
    - **waiting-init** waiting for somebody to answer, after having been connected
    - **speaking-init** speaking to the called subscriber
  - recipient status :
    - **waiting-recv** being connected: phone ringing or conversation suspended
    - **speaking-recv** speaking to the initiator of the phone conversation

# Telephone Exchange: System & Subscriber Operations

- Free Subscribers can **Lift** their handset, thus becoming attempting initiators.
- In every initiator state, initiators (& only them) can **Clear** their call, thus becoming again free.
- Initiators in attempting status can **Dial**, trying to complete a subscriber's number.
- The system can **MakeUnavailable** an initiator, due to a timeout, or due to an unsuccessful attempt to call because of **WrongNumber** or **Failure** (called subscriber not free).
- The system can **Connect** an initiator and a free recipient, making them both waiting (for the recipient to answer).
- A recipient can **Answer** and **Suspend** a phone call.

# Telephone Exchange Use Cases: Defining the Succession of Operations



## Defining Single Agent Operations for Telephone Exchange

- Abbreviations for the graphical control state notation:
  - single agent control states (**unary ctl**) :  
 $\text{ctl}(\mathbf{s}) = \text{Stat}(\mathbf{s}) \equiv \text{ctl}(\mathbf{s}) = \text{Stat}$  same with := instead of =
  - team control states (**binary ctl**, manipulated componentwise):  
 $\text{ctl}(\mathbf{s}, \mathbf{t}) = \text{Stat}_1(\mathbf{s}) \text{Stat}_2(\mathbf{t}) \equiv$   
 $\text{ctl}(\mathbf{s}) = \text{Stat}_1 \ \& \ \text{ctl}(\mathbf{t}) = \text{Stat}_2$  same with := instead of =
- **Lift** ( $\mathbf{s}$ )  $\equiv$  if in ( $\mathbf{s}$ ) = Lift then DialedSofar ( $\mathbf{s}$ ) := [ ]  
LastOpnTime( $\mathbf{s}$ ) := currtime
- **ClearAttempt** ( $\mathbf{s}$ ) = **ClearUnavailable** ( $\mathbf{s}$ )  $\equiv$   
if in( $\mathbf{s}$ )  $\in$  {ClearAttempt, ClearUnavailable} then skip
- **MakeUnavailable** ( $\mathbf{s}$ )  $\equiv$  (internal transition, without input)  
if currtime – LastOpnTime ( $\mathbf{s}$ ) > max-pause then skip

## Defining Single Agent Operations for Telephone Exchange

- **Dial (s)**  $\equiv$  if CompletableNr (DialedSofar(s))  
then DialedSofar (s) := append (in (s), DialedSofar ( s ))  
LastOpnTime (s) := currtime
- **WrongNumber (s)**  $\equiv$   
(internal transition, without input)  
if IncorrectNr (DialedSofar (s)) then skip
- **Failure (s)**  $\equiv$  (internal transition, without input)  
if CorrectNr (DialedSofar(s))  
& status (subscriber (DialedSofar (s)))  $\neq$  free  
then skip

Assumption: CompletableNr, IncorrectNr, CorrectNr  
are pairwise disjoint

## Defining Team Operations for Telephone Exchange

- **Connect (s,t)**  $\equiv$  (team building operation)

If t = subscriber (DialedSofar (s)) then

if CorrectNr (DialedSofar(s)) & status (t) = free

then caller (t) := s

- **Answer (t)**  $\equiv$  if in (t) = Answer then skip

(pure ctl state transition of the team)

NB. The effect is that status (t) changes to speaking-recv  
and status (caller (t) ) changes to speaking-init

- **Suspend (t)**  $\equiv$  if in (t) = Suspend then skip

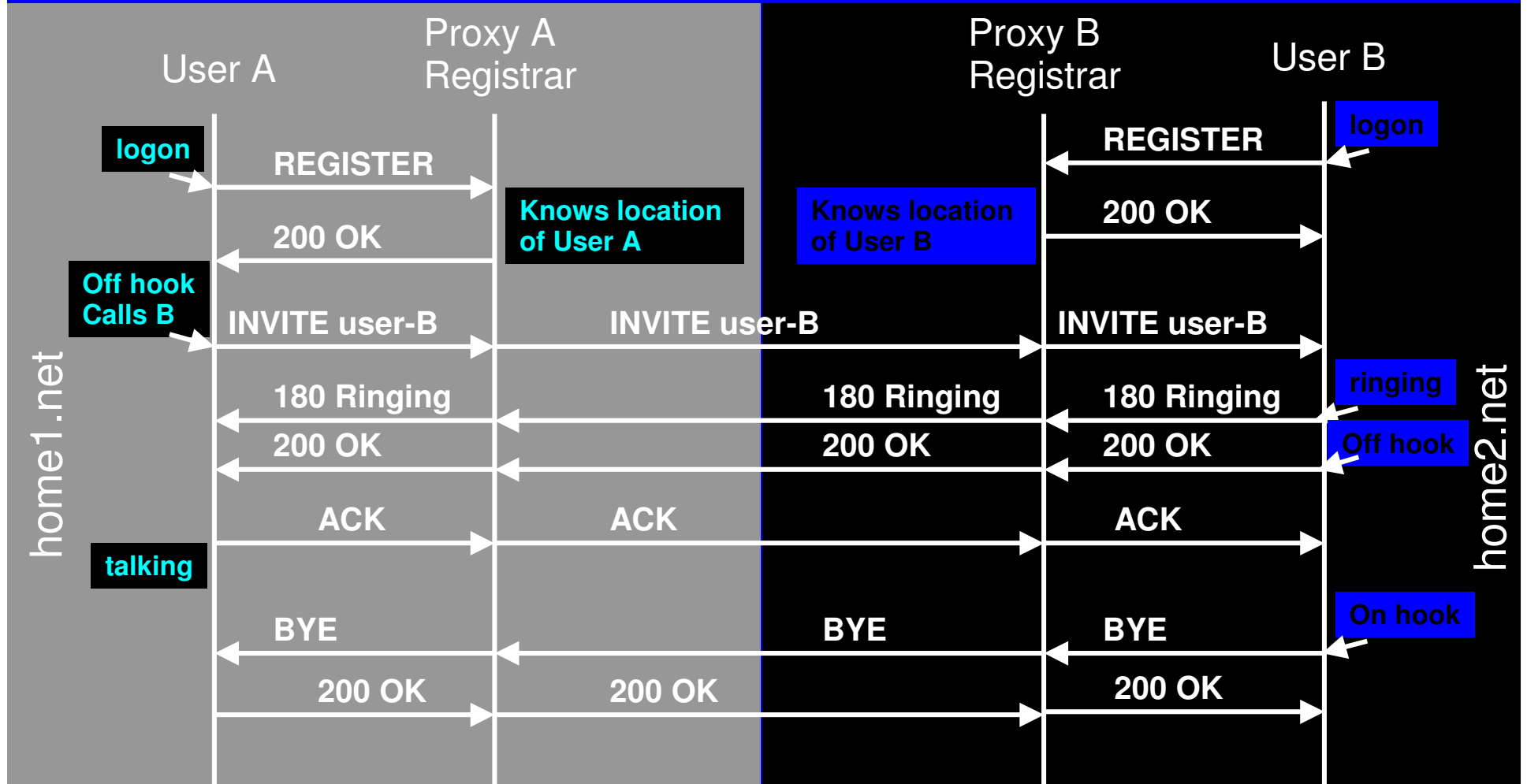
(team ctl state trans)

- **ClearSpeak (s) = ClearWait (s)**  $\equiv$

if in (s)  $\in$  {ClearSpeak, ClearWait} then

Let caller (t) = s in caller (t) := undef (uncoupling the team)

# Session Initiation Protocol (SIP) Flow Example (MSC-like description)



## References

E. Börger, R. Stärk: **Abstract State Machines**

**A Method for High-Level System Design and Analysis**

Springer-Verlag 2003, see <http://www.di.unipi.it/~AsmBook>

J.-R. Abrial: **The B-Book**

(Cambridge University Press 1996)

I. Sommerville: **Software Engineering**

(Addison-Wesley 1992 etc.)

M. Allemand, C. Attiogbe, H. Habrias: **Comparing  
Systems Specification Techniques.**

Proc., IRIN Nantes, March 1998. ISBN 2-906082-  
29-5