Egon Börger (Pisa) & Alexander Raschke (Ulm)

Modeling an Automatic Teller Machine

Illustrating Componentwise and Stepwise ASM Definition

Università di Pisa, Dipartimento di Informatica boerger@di.unipi.it
Universität Ulm, Abteilung Informatik alexander.raschke@uni-ulm.de

See Ch. 2.4 of Modeling Companion[1]

[1] Unless stated otherwise, some figures are ©2015 ACM and are used with licence 4271320674209.

ATM requirements (1)

PlantReq. There are many tills which can access a central resource containing the detailed records of customers' bank accounts.

TillAccessReq. A till is used by inserting a card and typing in a PIN which is encoded by the till and compared with a code stored on the card.

FunctionalReq. After successfully identifying themselves to the system, customers may try to:

1. view the balance of their accounts,
2. make a withdrawal of cash,
3. ask for a statement of their account to be sent by post.

Information on accounts is held in a central database and may be unavailable. If the database is available, any amount up to the total in the account may be withdrawn, subject to a fixed daily limit on withdrawals.

ATM requirements (2)

CardReq. The fixed daily limit on withdrawals means that the amount withdrawn within the day must be stored on the card. ”Illegal” cards are kept by the till.

InterruptReq. A till or the Central Resource can be interrupted and the connection between them can fail.

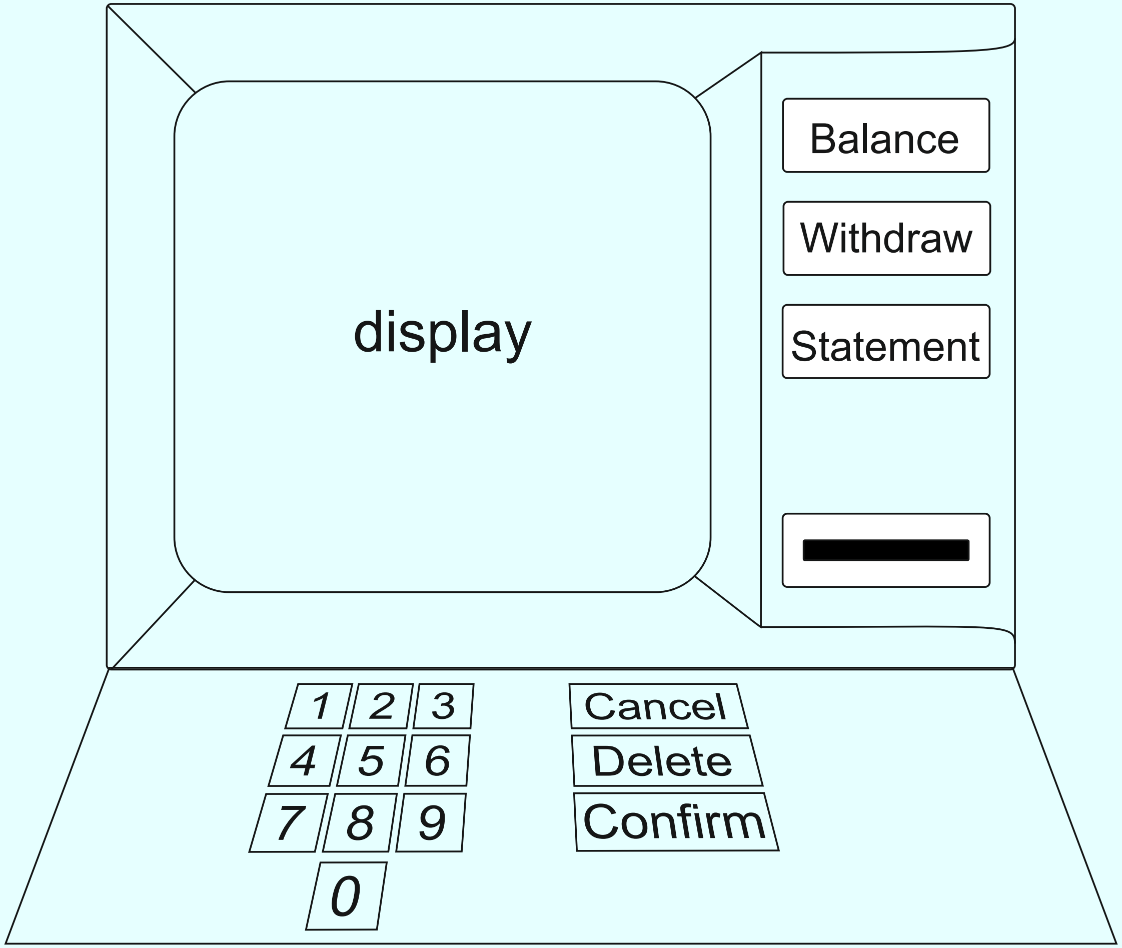
CustomerInterruptReq. Customers can change or cancel their request any time, e.g. stop the usage, change amount they want to withdraw.

ConcurrencyReq. Concurrent access to the database from two or more different tills is allowed, in particular concurrent attempts from two card holders who are authorised to use the same account.

TransactionalReq. Once a user has initiated a transaction, the transaction is completed at least eventually, and preferably within some real time constraint.

ReliabilityReq. Minimise the possibility of the use of stolen cards to gain access to an account.

ATM interface structure

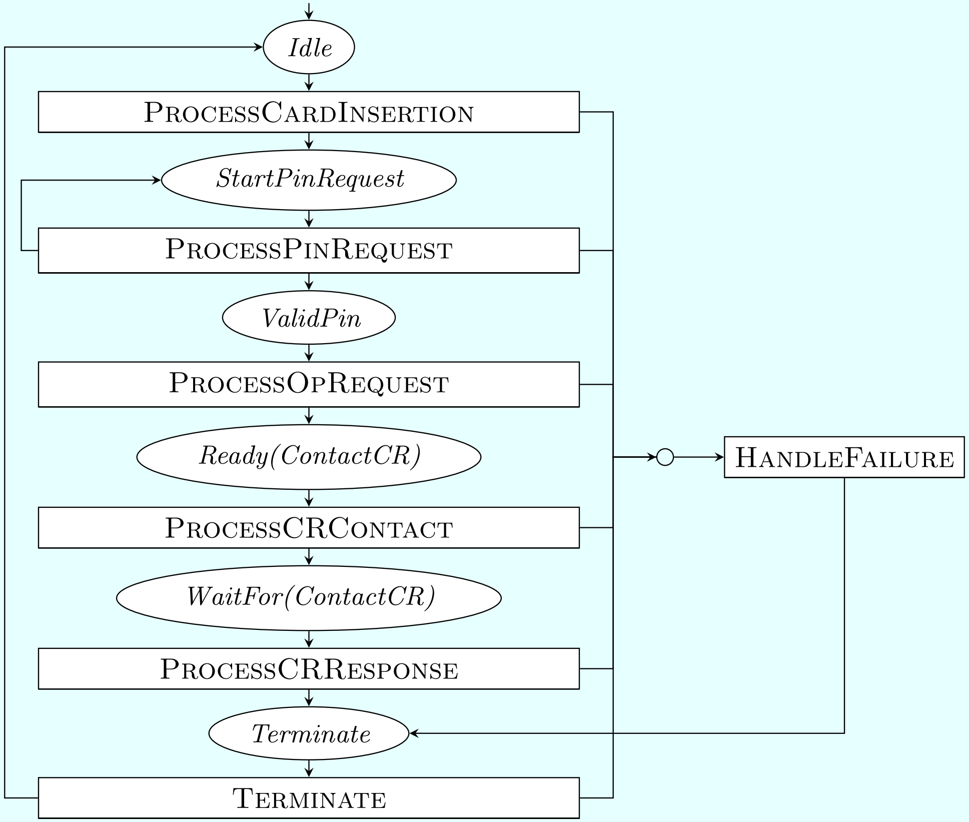


[2]

[2] Figure © 2018 Springer-Verlag Germany, reused with permission.

ATM controller architecture (Component Structure View)

TillAccessReq and *FunctionalReq* request a sequence of actions reflected by sequential composition of ground model out of action components:



Component failure triggers exit to HANDLEFAILURE component.

ATM ground model

In parallel to the 'normal' session execution by Atm, at any moment InterruptTriggers may occur and must be handled

- by *CustomerInterruptReq* and *InterruptReq*

```
GroundAtm =
  if ThereAreInterrupts
    then  HandleInterrupt
    else
      Atm
      HandleFailure
  InterruptTrigger
```

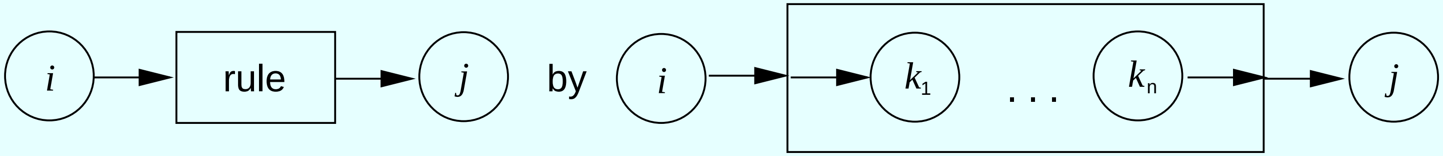
We now procedurally refine each component of GroundAtm

- to capture/complete one by one the corresponding *AtmReq*uirements

Procedural $(1, n+1)$ control state ASM refinements

Procedural ASM refinement follows Knuth's advice (1974):

> . . . we rapidly loose our ability to understand larger and larger flowcharts; some intermediate levels of abstraction are necessary.
>
> . . . we should give meaningful *names for the larger constructs in* our program that correspond to *meaningul levels of abstraction*, and we should define those levels of abstraction in one place, and merely use their names (instead of including the detailed code) when they are used to build larger concepts.



In general multiple entries/exits and arbitrary—even run-time determined—step relations (m, n) are allowed.

Card insertion: questions about requirements

$\mathit{CardInserted}$: monitored location

- becoming true/false upon physical card insertion/removal

A number of *questions about the requirements*:

- ATM presumably assumed to be used any time only by one user
  - i.e. new session can be started only when till is in $\mathit{mode} = \mathit{idle}$
- card validity check presumably assumed
  - $\textsc{InitializeSession}$ and $\mathit{StartPinRequest}$ only if can $\textsc{ReadCard}$
    - upon $\mathit{Fail}(\mathit{InvalidCard})$ move to $\textsc{HandleFailure}$
- meaning of $\mathit{ValidCard}$?

$$\mathit{ValidCard} =$$
$$\mathit{Readable}(\mathit{insertedCard})\ \textbf{and}\ \mathit{circuit}(\mathit{currCard}) \in \mathit{Circuit}$$

Card insertion: meaning of $\textsc{ReadCard}$

Domain experts must decide which are the attributes the reader can retrieve from a card, so that the till can manage the session:

- $circuit(card)$ describing the card type, $pinCode(card)$, $account(card)$
- $centralResource(card)$ where the $account(card)$ is managed
- $dailyLimit(card)$
- $alreadyWithdrawn(day, card)$ indicating the total amount of money withdrawn this day (as a date) in previous sessions at some tills using $card$
- $dayOfLastWithdrawal(card)$

Abstract from how $\textsc{ReadCard}$ records card attributes:

- $currCard := insertedCard$ with derived function $attribute(currCard) = attributeValReadFrom(insertedCard)$

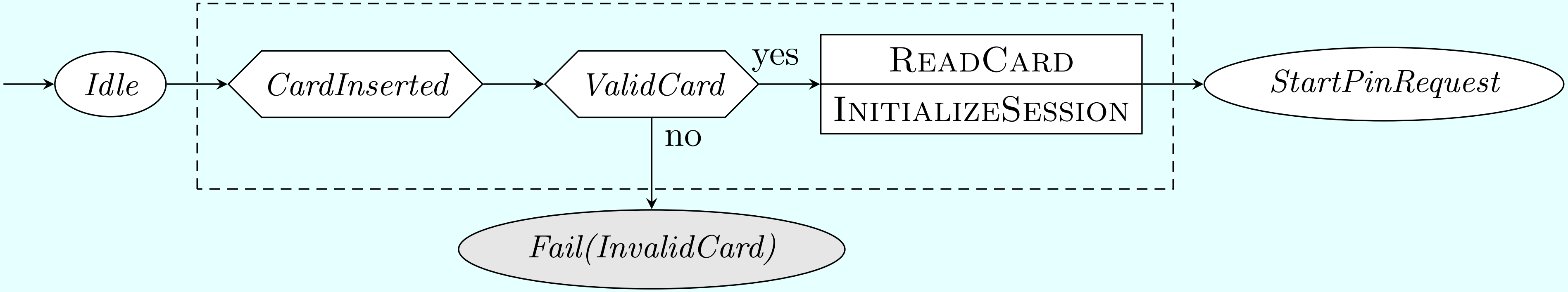
To-be-updated card attributes

Due to *CardReq*, the location $alreadyWithdrawn(today, card)$ must be updated, say in a component $\textsc{InitializeSession}$:

```
if $dayOfLastWithdrawal(card) < today$ then
    $alreadyWithdrawn(today, card) := 0$
```

Therefore $today$, which is monitored for $\textsc{ProcessCardInsertion}$, must be assumed to be updated at midnight by a $\textsc{Calendar}$ component of the ATM.

Component ProcessCardInsertion



Formally this procedural refinement is of type $(1, 1)$

- because reading the guard goes together with writing the updates, whatever is performed in passing from one to the next control state counts as one step

Component $\textsc{ProcessPin}$: modeling for change

- 'Typing in a PIN' in TillAccessReq describes one kind of *user/machine interaction*:

  – till will $\textsc{AskFor}(Pin)$ and check whether input is a $ValidPin$

  We make $\textsc{AskFor}$ *reusable as parameterized component*

  – parameter specifies the type of interaction object
- We separate a detailed component definition— involving processing keywise provided input streams—from its interface behavior spec

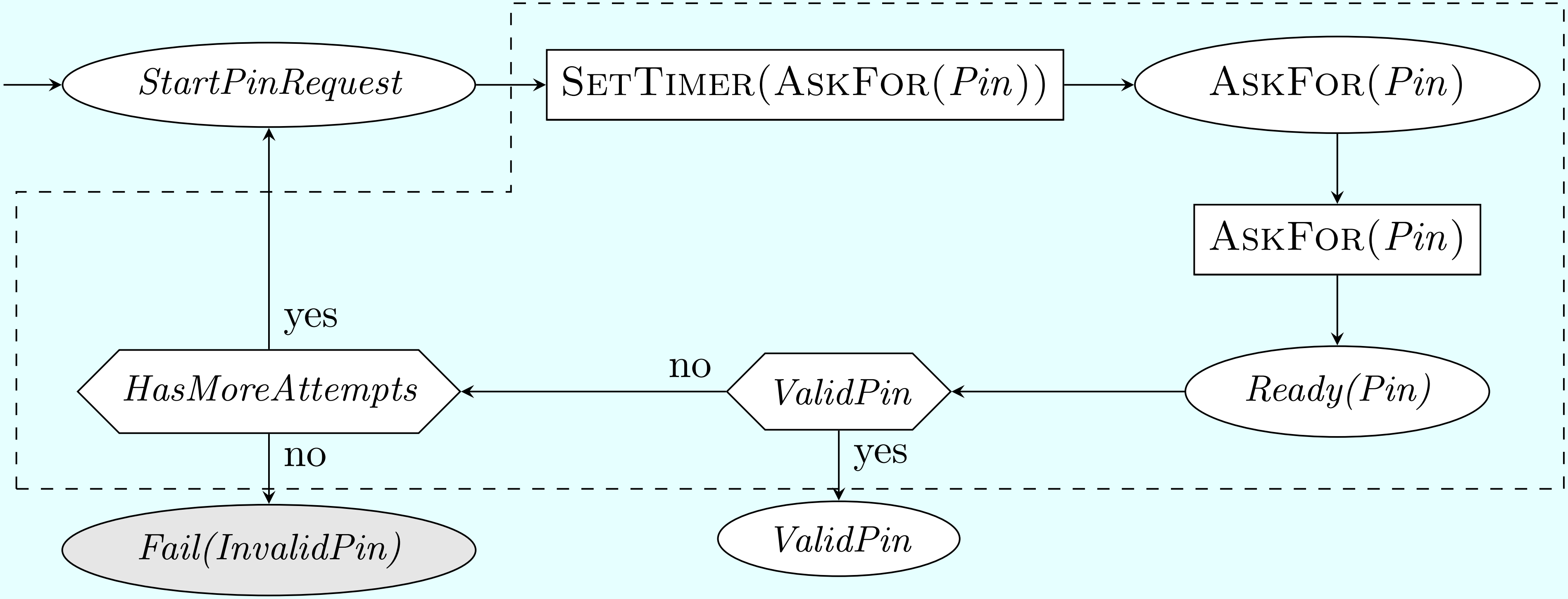
  – elaborate and store the monitored $userInput$ value asked for in a location, say $valFor(param)$, and enter mode $Ready(param)$

```
$\textsc{AskFor}(param) =$
  $valFor(param) := userInput$
  $mode := Ready(param)$
  $\textsc{ResetTimer}(\textsc{AskFor}(param))$
```

Definition of ProcessPin

Consider *additional req*: user $HasMoreAttempts$ to input pin

- until $ValidPin$ or $Fail(InvalidPin)$ or timeout interrupt



$$ValidPin = (pinCode(currCard) = encode_{Pin}(valFor(Pin)))$$

where by TillAccessReq $pinCode$ extracts the 'code stored on the card' and $encode_{Pin}$ performs the 'encoding by the till'

Parameterized ASMs are called by name

Declaration of parameterized ASM $N(x_1, \dots, x_n) = M$

- where M is an ASM whose free variables occur in $x_1, \dots, x_n$

permits to call $N(exp_1, \dots, exp_n)$ (by name) whereby

- body M of the machine declaration is executed with the variables x_i substituted by the call parameters exp_i (not by their values)
  - call parameters are evaluated only in the state in which the body is executed
- executing a *submachine call is treated as one atomic step*
  - M may contain recursive calls of N

  which yields a defined result only if the execution of the machine body yields a defined result

NB. *call by value* is definable by

$$N(exp_1, \dots, exp_n) = \textbf{let } (x_1 = exp_1, \dots, x_n = exp_n) \textbf{ in } M$$

Component $\textsc{ProcessOpRequest}$

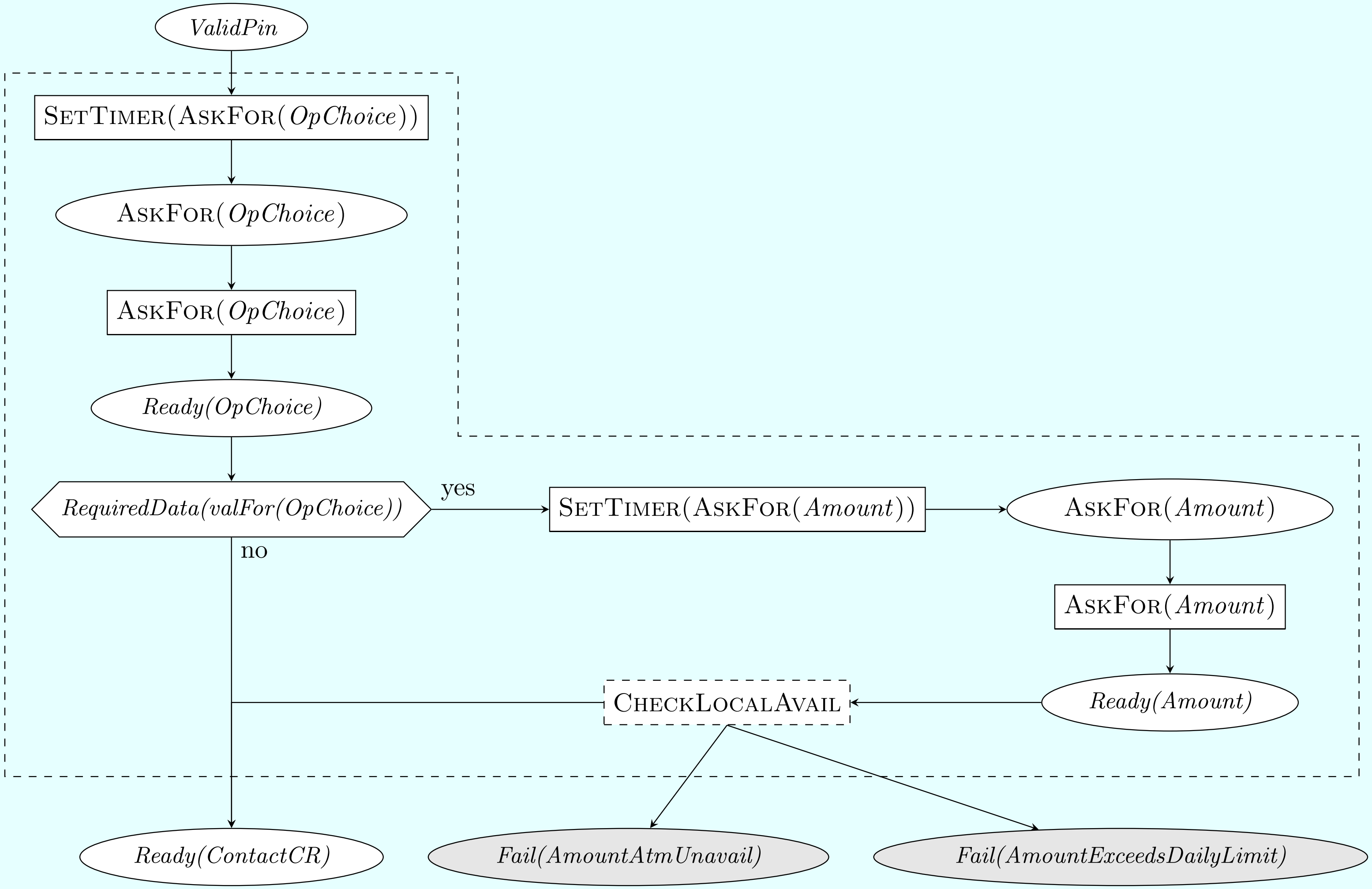
By FunctionalReq users have $OpChoice$ to ask for their account balance or an account statement or a cash withdrawal

- $\textsc{AskFor}(OpChoice)$ captures this choice
- if $op = Withdrawal$ the till acquires further $RequiredData$ via $\textsc{AskFor}(Amount)$

FunctionalReq also requests to $\textsc{CheckLocalAvail}$*ability* of the requested money

- whether $AmountExceedsDailyLimit$
- to which (for the sake of illustration) we add an $AmountATMUnavail$ability check

ProcessOpRequest component of Atm



Describing nondeterminism by ASM choose construct

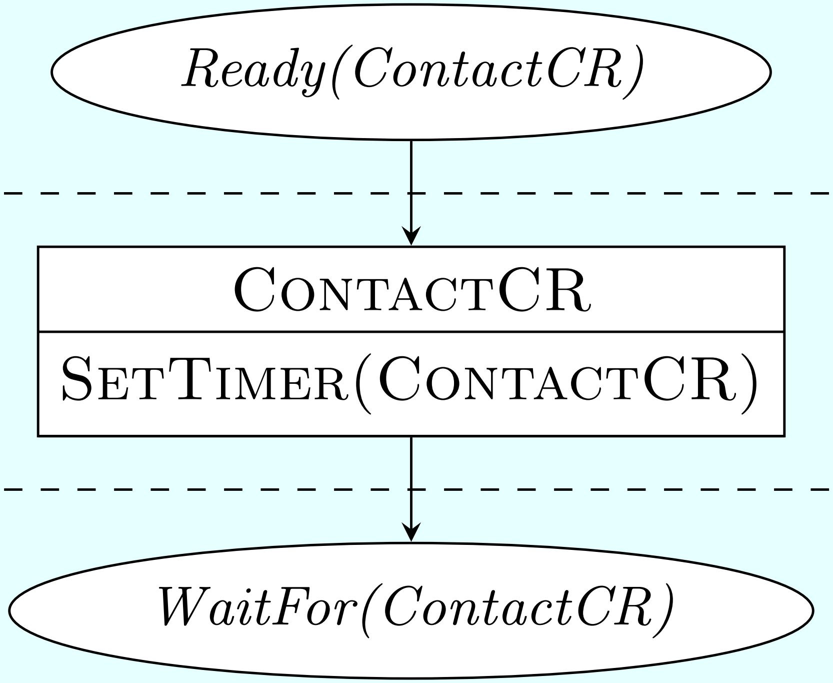
For CheckLocalAvail we show how to *specify an interface behavior*:

- to which next control state the component may proceed depending on the underlying data: normal exit $Ready(ContactCR)$ or a $Fail$ure exit

```
CheckLocalAvail =
  choose $m \in NxtCtlState$ // abstract from data determining $m$
    $mode := m$
    if $m = Ready(ContactCR)$ then
      $amount := valFor(Amount)$
where $NxtCtlState =$
  $\{Ready(ContactCR),$
    $Fail(AmountAtmUnavail),$
    $Fail(AmountExceedsDailyLimit)\}$
```

Component PROCESSCRCONTACT

- triggers a request that is sent to Central Resource (CR)
- makes the till $WaitFor(ContactCR)$ until a response is received
  - by *InterruptReq* unless a timeout or contact failure happen



ContactCR

$$\textsc{Send}(encode_{till}(Atm, CR, RequestData))$$

$$\textsc{Display}(WaitingForCentralResourceContact)$$

where

$$Atm = address(till(\textbf{self}))$$

$$CR = address(centralResource(currCard))$$

$$RequestData = opChoiceData(currCard, valFor(OpChoice))$$

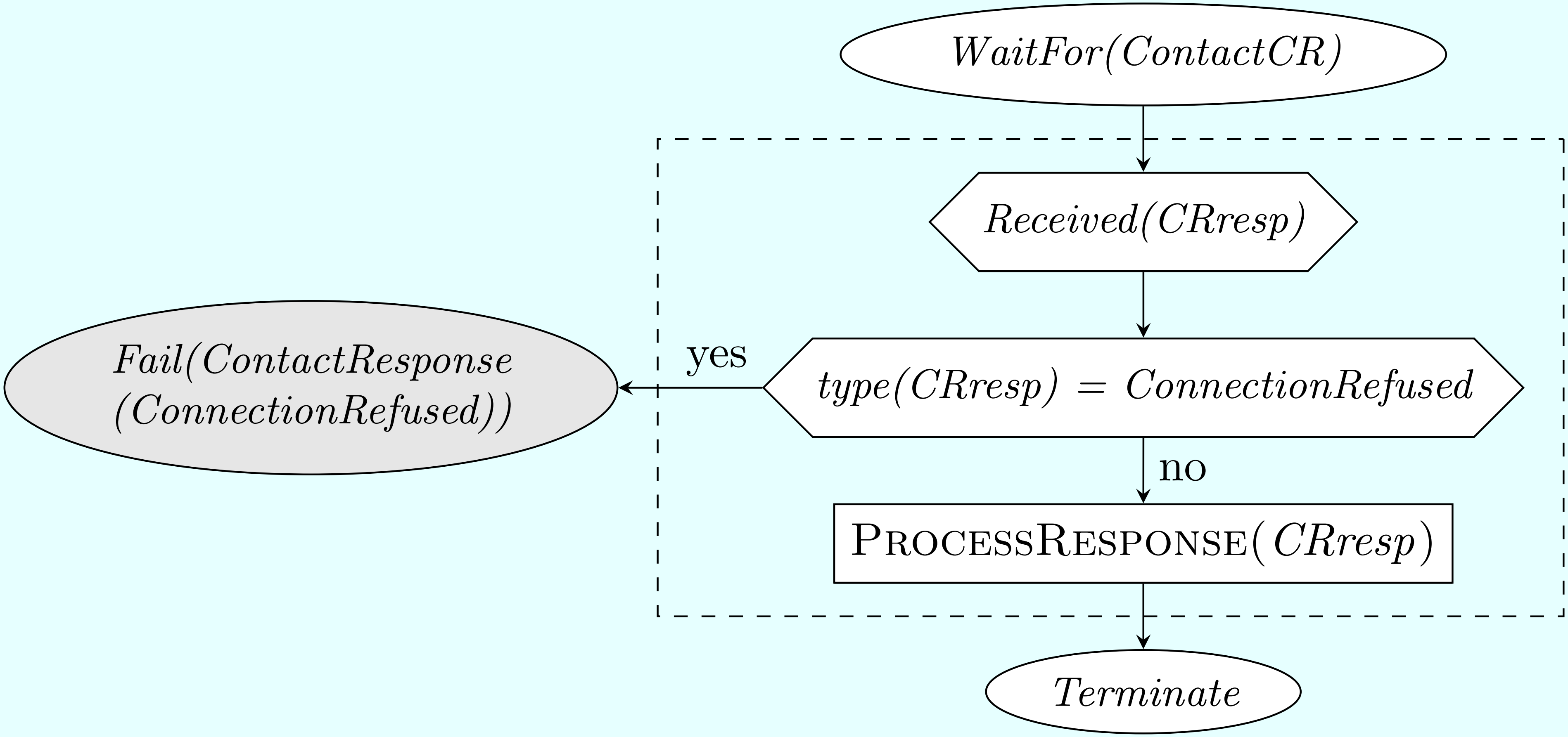
$$opChoiceData(card, op) =$$

$$\begin{cases} (card, op) & \textbf{if } op \in \{Balance, Statement\} \\ (card, op, amount) & \textbf{if } op = Withdrawal \end{cases}$$

Interface to $\textsc{ProcessCRResponse}$ component:

- monitored location $CRresp$ where response messages from the Central Resource (CR) are received

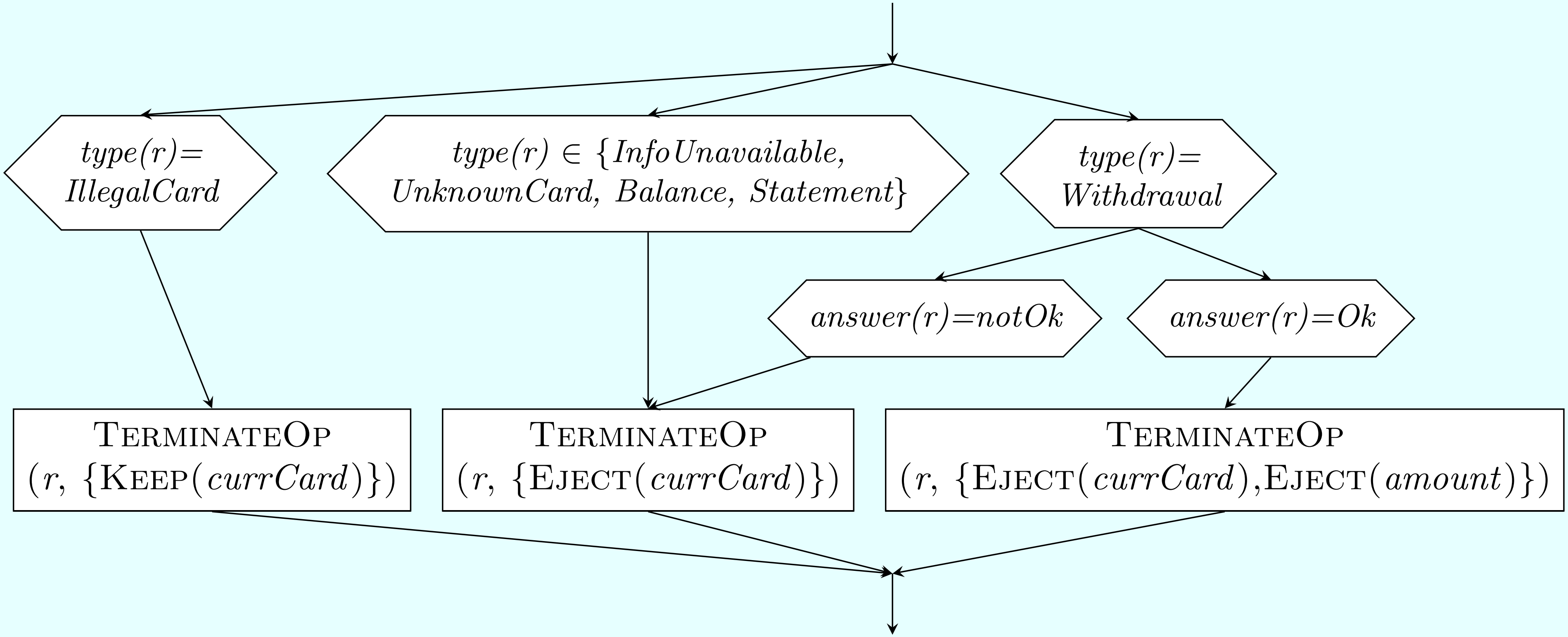
ProcessCRResponse



WaitFor(ContactCR)
Received(CRresp)
type(CRresp) = ConnectionRefused
yes
Fail(ContactResponse (ConnectionRefused))
no
ProcessResponse(*CRresp*)
Terminate

- by *InterruptReq* a $ConnectionRefused$ response may arrive
- other $CRresp$onses lead the till to normally ProcessResponse

ProcessResponse(r)



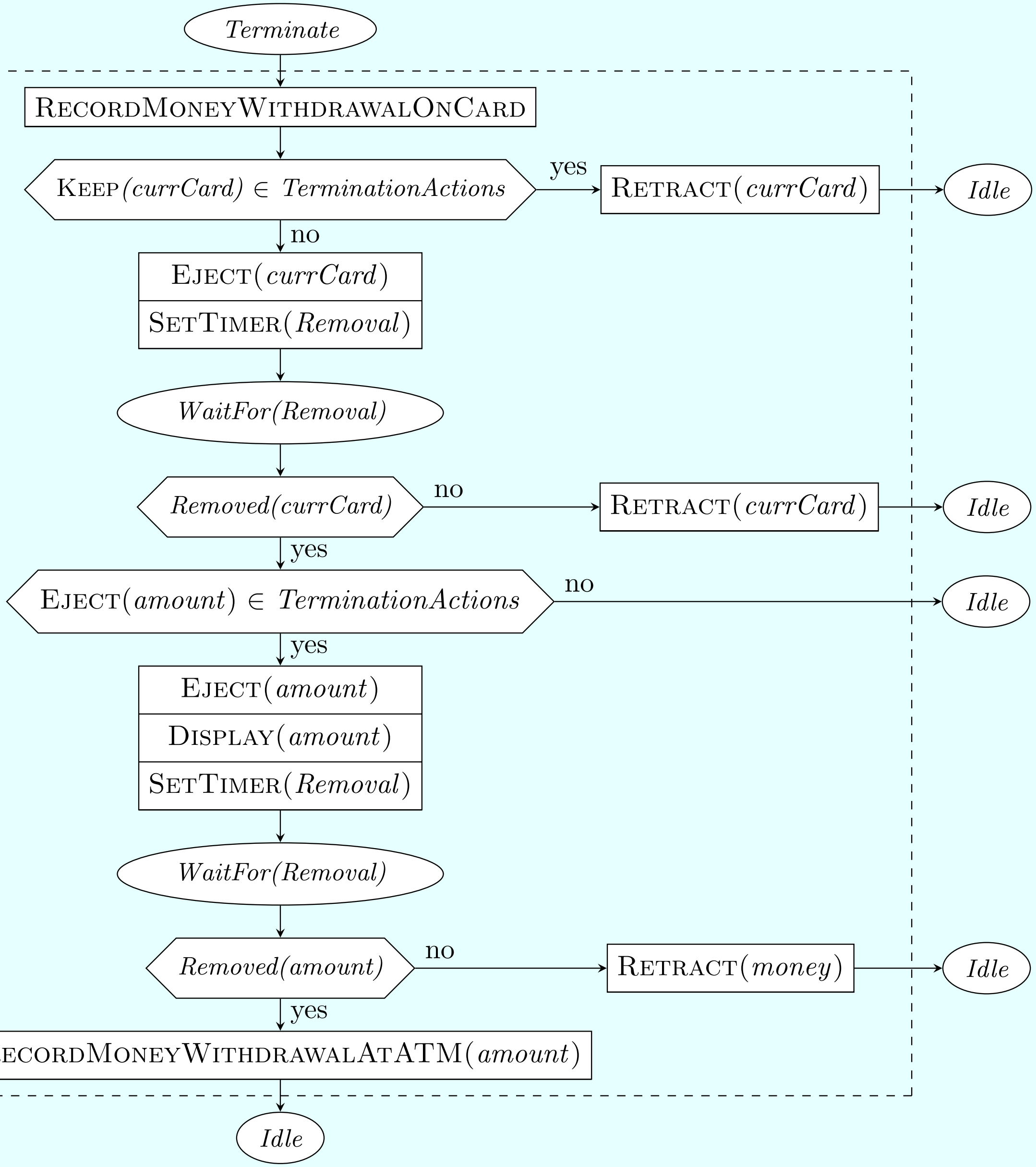
type(r)= IllegalCard
type(r) ∈ {InfoUnavailable, UnknownCard, Balance, Statement}
type(r)= Withdrawal
answer(r)=notOk
answer(r)=Ok
TerminateOp (r, {Keep(currCard)})
TerminateOp (r, {Eject(currCard)})
TerminateOp (r, {Eject(currCard),Eject(amount)})

TerminateOp$(reason, actions)$ =

  Display$(reason)$ -- explain action to the user

  $TerminationActions := actions$ -- executed by Terminate

Component Terminate



Terminate **macros**

```
$\textsc{Retract}(o) =$
  $\textsc{Remove}(o)$                    -- physically remove card or money from slot
  $\textsc{LogMissedWithdrawal}(o)$                                    -- if applicable
$\textsc{RecordMoneyWithdrawalOnCard} =$
  if $MoneyWithdrawalToRecord$ then
    $alreadyWithdrawn(today, currCard) :=$
      $amount + alreadyWithdrawn(today, currCard)$
    $dayOfLastWithdrawal(currCard) := today$
    $MoneyWithdrawalToRecord := false$
$\textsc{RecordMoneyWithdrawalAtAtm}(o) =$
  $money(Atm) := money(Atm) - o$
```

HandleFailure: an example

Modular (case-by-case) definition via parameterization of $Fail(param)$ values of $mode$:

```
HandleFailure =                    -- called when $mode = Fail(param)$
  if $mode = Fail(InvalidPin)$ then
    TerminateOp$(InvalidPin, \{$Keep$(currCard)\})$
  else  TerminateOp$(mode, \{$Eject$(currCard)\})$
  CloseConnectionToCentralResource
```

- Eject unreadable cards, cards of not accepted circuits, etc.
- NB. cards the CR declares as $IllegalCard$ are kept by the corresponding TerminateOPeration

Interrupts with interrupt region

Modular (case-by-case) definition via

- parameterization
- separately definable concept of *interrupt region* where interrupt events should have an effect

Exl: interrupt is triggered (inserted into $InterruptEvent$):

- when user has $Pressed$ the $CancelKey$ and $\textsc{Atm}$ $IsInCancelRegion$
- automatically upon a $Timeout(timedOp)$ event when the $\textsc{Atm}$ $IsInTimerRegion$ for the $timedOp$

$\textsc{InterruptTrigger} =$

$\quad \textsc{InterruptBy}(Cancel)$

$\quad \textsc{InterruptBy}(Time)$

$\quad \ldots$

Cancel and timeout interrupts

```
$\textsc{InterruptBy}(\mathit{Cancel}) =$
  if $\mathit{Pressed}(\mathit{CancelKey})$ and $\mathit{IsInCancelRegion}(\textsc{Atm})$
    then $\textsc{Insert}(\mathit{Cancel}, \mathit{InterruptEvent})$
$\textsc{InterruptBy}(\mathit{Time}) =$
  forall $\mathit{timedOp} \in \{\mathit{AskFor}(\mathit{param}), \mathit{ContactCR}, \mathit{Removal}\}$
  if $\mathit{Timeout}(\mathit{timedOp})$ and $\mathit{IsInTimerRegion}(\mathit{timedOp})$ then
    $\textsc{Insert}(\mathit{timer}(\mathit{timedOp}), \mathit{InterruptEvent})$
    $\textsc{ResetTimer}(\mathit{timedOp})$
```

HandleInterrupt with priority scheme

```
$\textsc{HandleInterrupt} =$
  let $e = \mathit{highPriority}(\mathit{InterruptEvent})$
    $\textsc{Handle}(e) \quad \textsc{Delete}(e, \mathit{InterruptEvent})$
where
$\textsc{Handle}(\mathit{Cancel}) =$
  if $\mathit{IsInCancelRegion}(\textsc{Atm})$ then $\textsc{TerminateSession}(\mathit{Cancel})$
$\textsc{Handle}(\mathit{timer}(\mathit{timedOperation})) =$
  if $\mathit{IsInTimerRegion}(\mathit{timedOperation})$ then
    $\textsc{TerminateSession}(\mathit{Timeout}(\mathit{timedOperation}))$
$\textsc{TerminateSession}(p) =$
  $\textsc{DisconnectAtmFromCR}$
  $\textsc{TerminateOp}(p, \textsc{Eject}(\mathit{currCard}))$
  $\mathit{mode} := \mathit{Terminate}$
```

Defining interrupt regions

In control state ASMs interrupt regions are definable by $mode$ intervals.

- Exl: no $Cancel$ command has any effect outside a user session (when $mode = idle$) or when the ATM is performing automatically its final stage to $\textsc{Terminate}$ the session
  $IsInCancelRegion(\textsc{Atm}) = mode \notin \{Idle, Terminate\})$

Analogously for timer regions:

$$IsInTimerRegion(AskFor(param)) = \\ \quad mode \in \{AskFor(param), WaitFor(param)\}$$

$$IsInTimerRegion(ContactCR) = \\ \quad mode = WaitFor(ContactCR)$$

$$IsInTimerRegion(RemovalCard) = \\ \quad (mode \in \\ \qquad \{WaitFor(RemovalCard), WaitFor(RemovalMoney)\})$$

CentralResource

- works asynchronously together with multiple ATMs
- to satisfy the ConcurrencyReq, our spec permits any processing order for independent requests
  - separate priority and scheduling concerns from per-account-exclusive access guarantee in FunctionalReq

```
$\textsc{CentralResource} =$
  one of $(\textsc{AcceptRequests}, \textsc{HandleRequests})$
where $\textsc{AcceptRequests} =$
  if $Mailbox_{CR} \neq \emptyset$ then                    -- if some msgs arrived
    choose $R \subseteq Mailbox_{CR}$ with $R \neq \emptyset$              -- select some
      forall $msg \in R$    -- move them from mailbox into internal record
        $\textsc{Insert}(decode_{CR}(msg), Request)$
        $\textsc{Delete}(msg, Mailbox_{CR})$
```

HandleRequests **component of** CentralResource

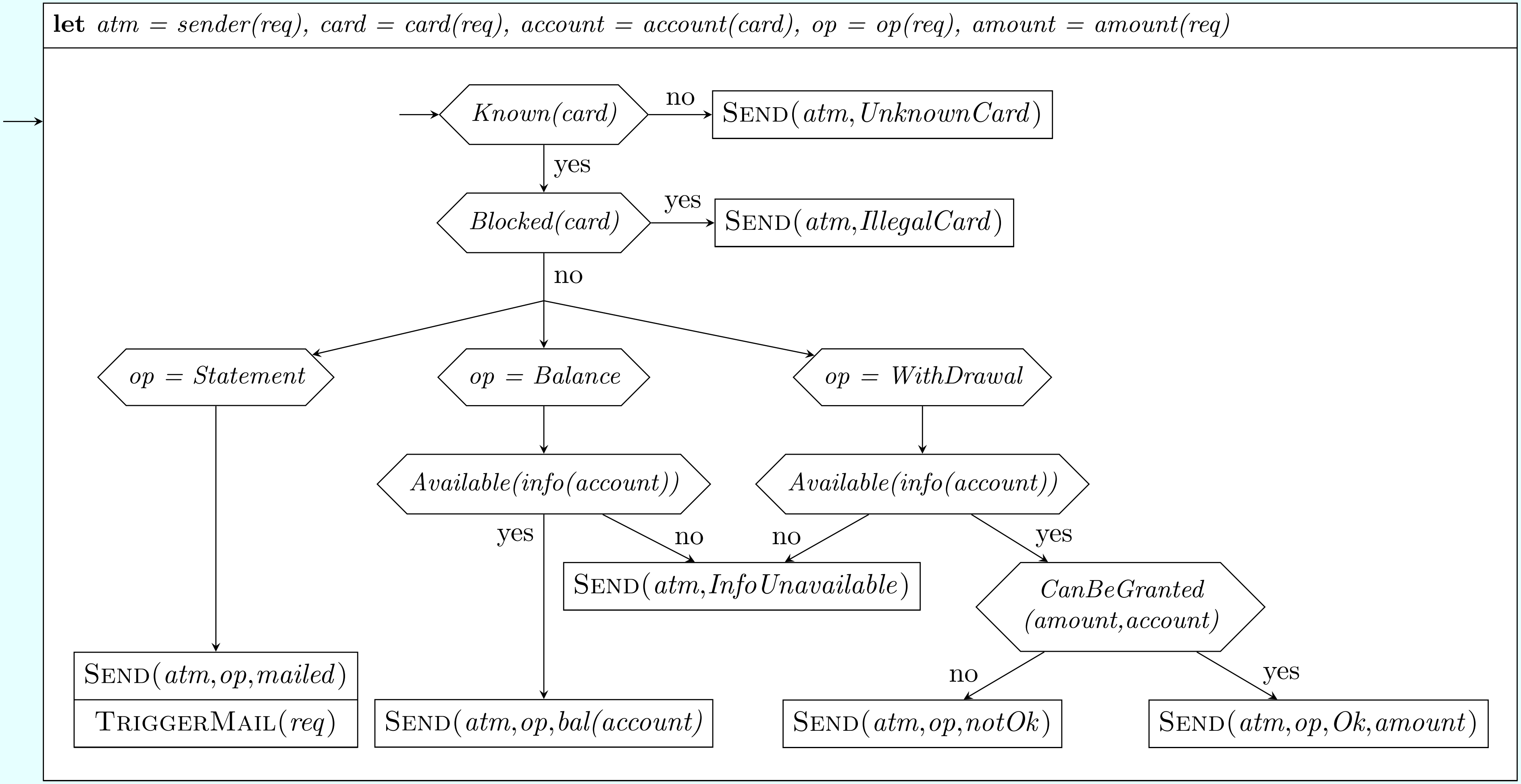
Let $select_{CR}$ be any policy for selecting a $Consistent$ set of requests for a parallel handling.

```
$\textsc{HandleRequests} =$
  if $Request \neq \emptyset$ then                -- if there are requests
    let $R = select_{CR}(Request)$                -- select a $Consistent$ subset
    forall $r \in R$                              -- $\textsc{Handle}$ all of them
      $\textsc{Handle}(r)$
      $\textsc{Delete}(r, Request)$
```

```
$Consistent(R)$ iff                               -- no two withdrawals from one account
  thereisno $r, r' \in R$ with $r \neq r'$ and
    $account(r) = account(r')$ and $op(r) = op(r') = Withdrawal$
```

$\textsc{Handle}(req)$ component of $\textsc{Atm}$

How CR elaborates a correct $CRresponse$ of type $op(req)$:



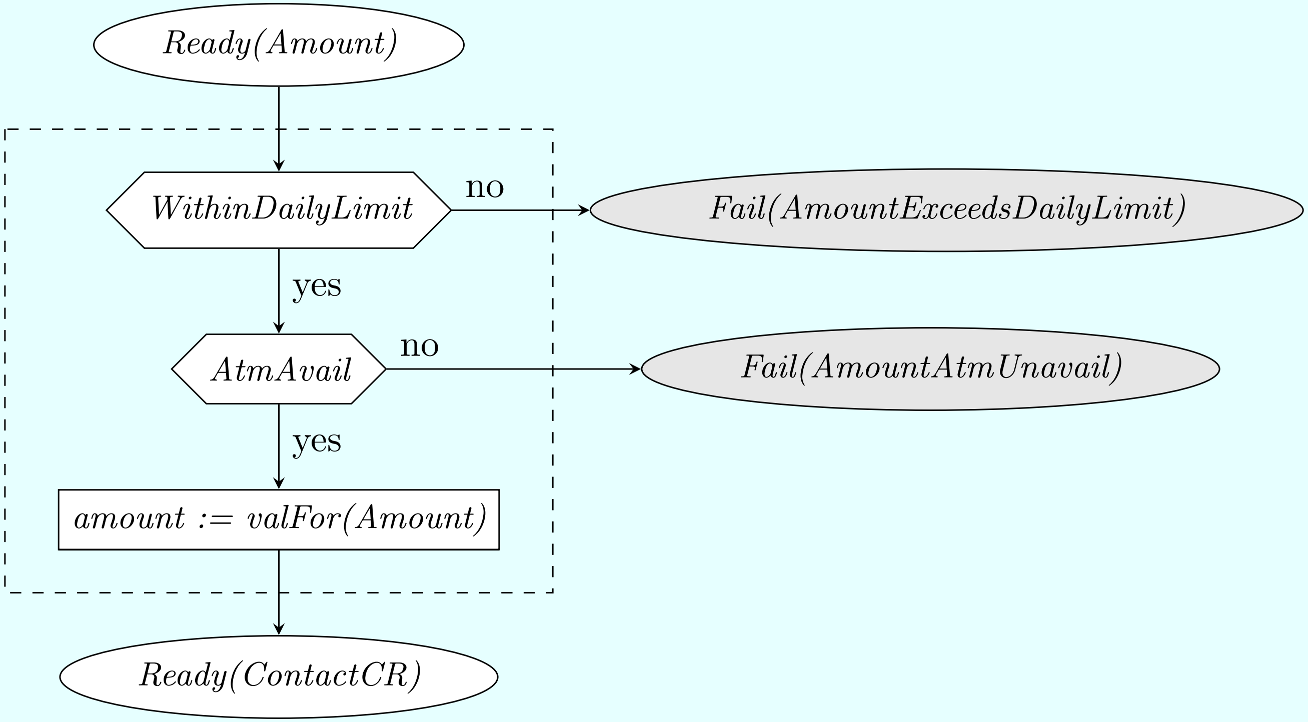
Integrate data into ctl flow: by data refinement (type $(1,1)$)

$\textsc{CheckLocalAvail} = \textbf{choose } m \in \mathit{NxtCtlState}$

$\quad \mathit{mode} := m$

$\quad \textbf{if } m = \mathit{Ready}(\mathit{ContactCR}) \textbf{ then } \mathit{amount} := \mathit{valFor}(\mathit{Amount})$

Refinement computing how mode update depends on data:



Integrate data into ctl flow: by procedural refinement

Exl: combined data and operation refinement of $\textsc{AskedFor} =$

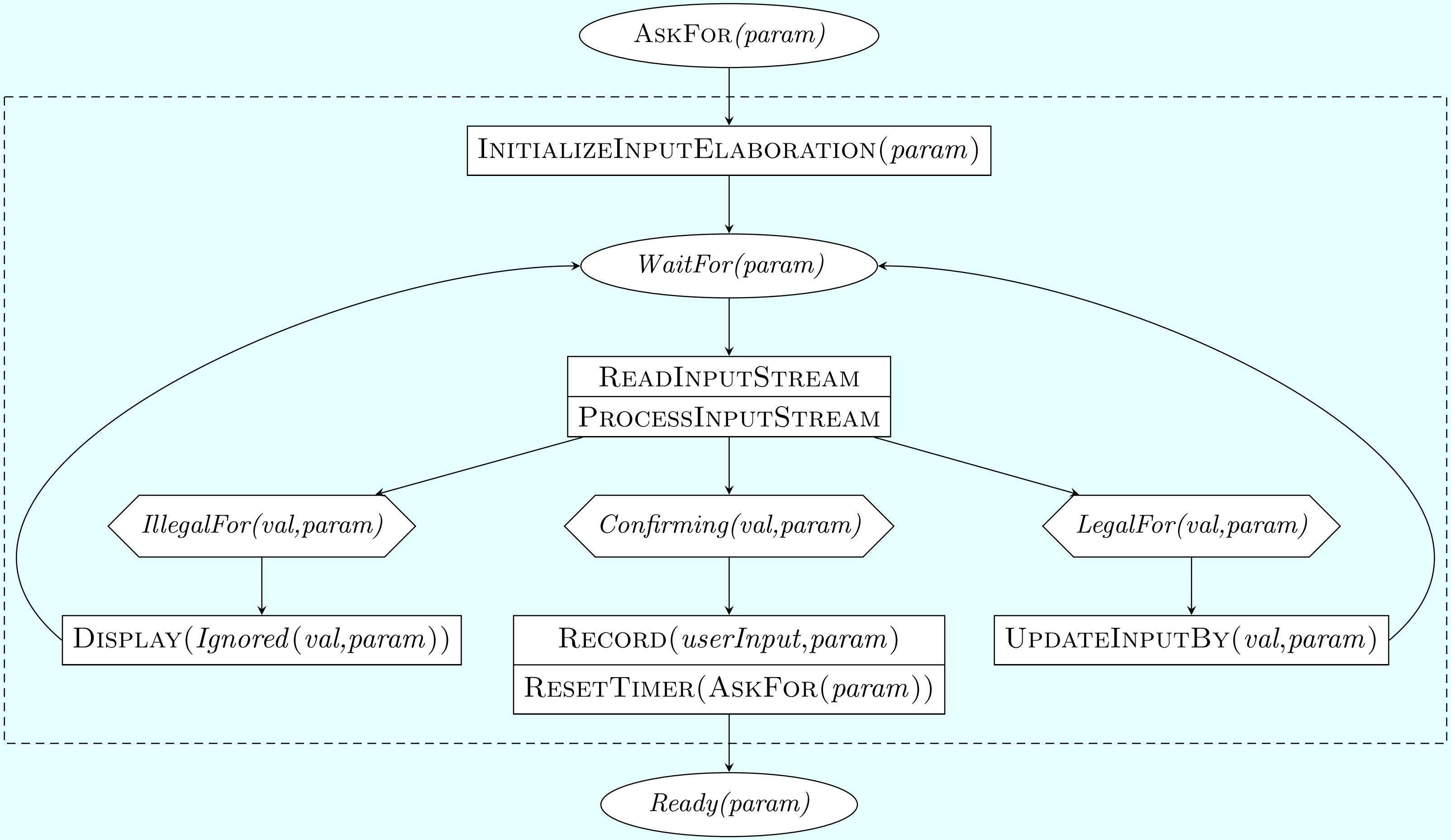
$$valFor(param) := userInput \quad mode := Ready(param)$$
$$\textsc{ResetTimer}(\textsc{AskFor}(param)))$$

Idea: implement successive reading and processing of single input key values inserted by the user as follows:

- start to $\textsc{InitializeInputElaboration}$
  - $\textsc{Display}$ request to user
  - guarantee robustness: keys pressed before start of $WaitFor(param)$ should yield no input
- iterate $\textsc{ReadInputStream}$ and $\textsc{ProcessInputStream}$
- upon a $Confirm$ key input move to $Ready(param)$

Procedural refinement of $\textsc{AskedFor}$ component

$$\begin{aligned}\textsc{AskedFor} = {} & \mathit{valFor}(\mathit{param}) := \mathit{userInput},\\ & \mathit{mode} := \mathit{Ready}(\mathit{param}), \textsc{ResetTimer}(\textsc{AskFor}(\mathit{param}))\end{aligned}$$



Initialization component of refined $\textsc{AskedFor}$

```
$\textsc{InitializeInputElaboration}(param) =$
    $\textsc{Initialize}(inputStream)$    -- Start listening to user input
    $\textsc{Initialize}(userInput)$    -- Start processing user input
    $\textsc{Display}(AskFor(param))$    -- Ask user for $param$
    **if** $param = Pin$ **then** $\textsc{CountDown}(attemptsFor(Pin))$
```

The auxiliary macros are defined as follows:

$\textsc{Initialize}(Stream) = (Stream := [\,])$

$\textsc{Initialize}(userInput) = (userInput := [\,])$

$\textsc{CountDown}(attemptsFor(Pin)) =$

$\quad attemptsFor(Pin) := attemptsFor(Pin) - 1$

NB. An initialization of $attemptsFor(Pin)$ belongs to (for example) $\textsc{InitializeSession}$.

$\textsc{Read/ProcessInputStream}$ component

- what if a user hits simultaneously a set of multiple keys?
  - hardware transforms the set into a randomly ordered $inputStream$
  - before applying $randomOrder$ to a set, the hardware will $truncate(set)$ in a device dependent manner to a subset
- $inputVal$ yields input value sequence for key sequence

```
$\textsc{ReadInputStream} =$
  let $PressedKeys = \{key \mid Pressed(key)\}$
  let $Newinput =$
    $inputval(randomOrder(truncate(PressedKeys)))$
      $\textsc{AddAtTheLeft}(Newinput, inputStream)$
$\textsc{ProcessInputStream} =$
  if $inputStream \neq []$ then
    let $val = fstOut(inputStream)$                -- say rightmost element
    $\textsc{RemoveAtTheRight}(val, inputStream)$
```

UpdateInputBy **component**

- writes the *inputStream* values that are *LegalFor param* into *userInput*
- since user can change the input any time (CustomerInterruptReq), *Delete* key is *LegalFor* every *param*

```
$\textsc{UpdateInputBy}(val, param) =$
    if $val \neq Delete$ then $\textsc{AddToInput}(val, param)$
    if $val = Delete$ then $\textsc{RemoveFromInput}(param)$
$\textsc{AddToInput}(val, param) =$
    $userInput := concatenateAtTheRight(userInput, val)$
    $\textsc{Display}(concatenateAtTheRight(userInput, val), param)$
$\textsc{RemoveFromInput}(param) =$
    $userInput := removeLast(userInput)$
    $\textsc{Display}(removeLast(userInput), param)$
```

Last AskFor step when the user confirms the input

- the input is recorded in interface location $valFor(param)$
- due to in-time termination the timer is reset
- $mode$ switches to $Read(param)$

$Confirming(val, param)$ if and only if

$$\begin{cases} param \in \{Pin, Amount\} \textbf{ and } val = Confirm \\ param = OpChoice \textbf{ and } val \in \{Balance, Statement, Withdrawal \end{cases}$$

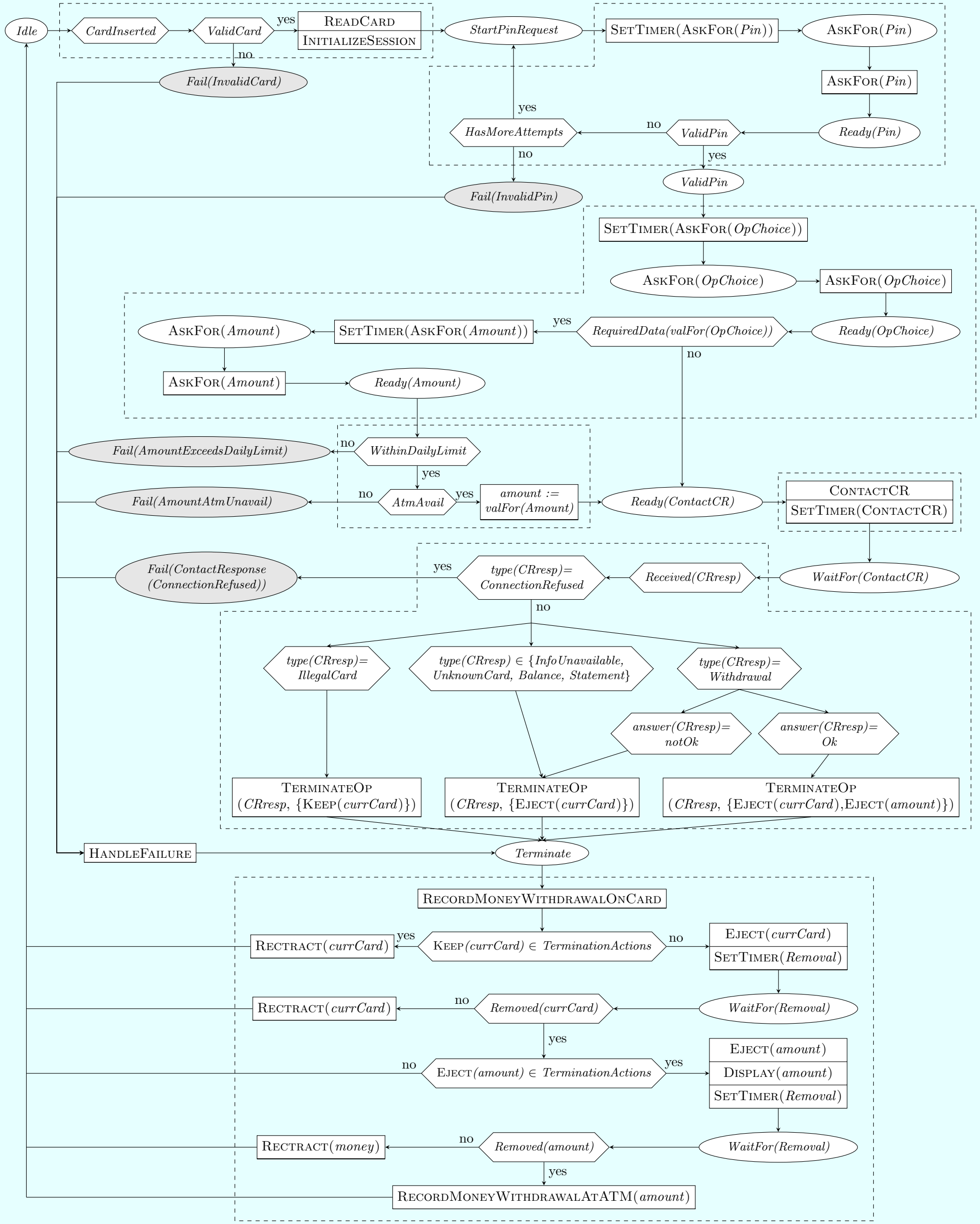
$\textsc{Record}(input, param) =$

$$valFor(param) := input \ \textbf{ if } param \in \{Pin, Amount\}$$

$$valFor(param) := param \ \textbf{ if}$$

$$\quad param \in \{Balance, Statement, Withdrawal\}$$

ATM Unfolded Refined View



References

- Requirements from: Automatic Teller Machine or Till: Case Study
  - *The FM'99 ATM modelling challenge*
- E. Börger and S. Zenzaro: Business Process Modeling for Reuse and Change via Component-Based Decomposition and ASM Refinement.
  - Proc. S-BPM One 2015. ACM Digital Library, ISBN 978-1-4503-3312-2.
- S. Zenzaro: A CoreASM refinement implementing the ATM ground model.
  - available at http://modelingbook.informatik.uni-ulm.de (October 2014)
- E. Börger and A. Raschke: Modeling Companion for Software Practitioners. Springer 2018 `http://modelingbook.informatik.uni-ulm.de`

Copyright Notice