Modeling for Change via Component-Based Decomposition and ASM Refinement

Egon Börger
Università di Pisa
Dipartimento di Informatica
I-56125 Pisa, Italy
boerger@di.unipi.it

Simone Zenzaro
Università di Pisa
Dipartimento di Informatica
I-56125 Pisa, Italy
zenzaro@di.unipi.it

ABSTRACT

This paper is part of a larger effort to concretely compare different approaches to modelling and implementing software intensive systems, in particular Business Processes (BPs). We illustrate for the Atm case study [3] how to use the Abstract State Machines Method [10] to develop executable models by a) first defining a high-level (easily changeable and reusable) model that can be checked by the domain expert to capture the requirements and b) then refining this model to executable code which the software expert can check to behave correctly with respect to the requirements model.[1]

1. INTRODUCTION

The growing awareness of the importance of high-level modelling for the development of reliable software-intensive systems goes together with a puzzling variety of modelling approaches coming with languages and tool suites for modelling and model validation, verification and implementation. This holds also for the Business Process (BP) domain where up to today despite of intensive efforts (see [23, pg.5] and [7] for further references) no satisfactory standardization helps the BP expert to decide upon which approach to adopt.

In this paper we contribute to the endeavor to concretely compare—using (necessarily small but characteristic) case studies from the literature—some well-known modeling approaches with respect to their conceptual well-foundedness (semantics) and important pragmatic properties (e.g. ease of developing, understanding, changing, reusing, implementing, validating, verifying, documenting models, etc.). We use the Abstract State Machine (ASM) method [10] to capture the requirements of the often-used Atm case study [3] by a high-level (so-called *ground* [6]) model which can be a) inspected for correctness (in its application domain meaning, called *ground model correctness* [6]) and then b) correctly refined to an executable version [24][2]—so that the features of interest can be concretely compared to those of other models and implementations [12, 13].

Through the development of the ground model we illustrate three characteristic properties of the ASM method.

- *Minimality* of ground models: we show that when an element is introduced into the ASM model it is not to comply with some need of the modeling framework but to directly reflect a feature in the requirements. In fact it is crucial that the modeling process is guided by the given application domain problems, not by modelling framework constraints, so that a domain expert can apply the method without further ado.
- *Decomposition* of ASM models: we apply the ASM refinement method [5][3] which allows one to organize complex behaviour by splitting it into (a structure of) components (*horizontal refinement*) which can be inspected in isolation for correctness, respecting Parnas' code inspection guide lines.[4] Guided by the requirements [3] we decompose the Atm ground model into ASM components to separate a) normal, failure (exception) and interrupt (by cancel commands or timeouts) behavior, b) the successive stages of normal Atm behavior and c) concurrency aspects (involving multiple tills). Furthermore we stepwise detail the components (by *vertical refinement*) leading from the high-level architectural ground model view to an executable

[1]The authors gratefully acknowledge support of this work by a fellowship from the Alexander von Humboldt Foundation for a summer term research stay at the universities of Düsseldorf, Bonn-Rhein-Sieg and Kiel, sequel to the *Humboldt Forschungspreis* awarded to the first author in 2007/08.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
S-BPM ONE '15, April 23-24, 2015, Kiel, Germany
Copyright 2015 ACM 978-1-4503-3312-2/15/04 ...$15.00
http://dx.doi.org/10.1145/2723839.2723854.

[2]The ASM method permits refinements also for other engines (e.g. [4]) and by code generation (see for example the ASM2C++ compiler used in [8]).

[3]See the further theoretical underpinning and development based upon a KIV theorem prover formalization in [18, 19, 21, 20, 22].

[4]'the key to inspection of any complex product is a policy of *divide and conquer*, i.e., having the inspector examine small parts of the product in isolation, while making sure that 1) nothing is overlooked and 2) that the correctness of all inspected components implies the correctness of the whole product. The decomposition of the inspection into discrete steps must assure that each step is simple enough that it can be carried out reliably and that one inspection step can be carried out without detailed knowledge of the others.' [17]

model.

- *Completeness* of ground models: to make it checkable by the domain expert that every feature in the requirements that is relevant for the intended system behavior is present in the ground model we define its components by control state ASMs, a class of ASMs with an intuitive graphical (FSM-like flowchart) representation which is easy to grasp for application domain experts and by its rigorously defined behavioural semantics enhances similar (but semantically only loosely defined) UML [15] and BPMN [16] diagrams. Furthermore the simple combination of graphical and textual definitions in control state ASMs allows one to smoothly but precisely integrate data and resource conditions into the control-flow perspective. This allows the BP expert to see the requirements modelled completely (covering control flow, data and resources) and without ambiguity yet close to familiar notations so that model inspection becomes feasible, reducing the risk of misunderstanding between experts and model or software developers.

We avoid a premature introduction of classes and instead concentrate the attention on discovering an appropriate architecture of components with as abstract as possible ('conceptual', application domain focussed) data and operations, a concern shared with the object-oriented world view. Thus we focus on finding out who are the actors of a system and what are the elements which regulate their interaction (shared locations, communication constructs, conditions), a concern shared with the subject oriented approach to BPM (S-BPM [11]). In this way we pave the way for efficient model reuse.

The architectural ATM model in Sect. 2 is detailed in Sect. 3 by models for the components reacting to user input and events and is followed by models for the CENTRALRESOURCE and the communication mechanism (Sect. 4). In Sect. 5 we shortly discuss verification, validation and reuse concerns.

2. MODELING THE ATM

The requirements suggest a natural sequence of actions an Atm performs during a session with a user. We follow it to structure our ATM model using corresponding components M as displayed in Fig. 1. During the execution of M various failures may happen which trigger the ATM to HANDLEFAILURE(M). The *Fail*(M) reasons of every component M depend on security and reliability constraintsin [3]:

- *Fail*(PROCESSCARDINSERTION) if the *insertedCard* is not a *ValidCard*
- *Fail*(PROCESSPIN) if the inserted pin is not a *ValidPin*
- *Fail*(PROCESSOPREQUEST) if during the handling of a user's *Withdrawal* request the *LocalAvail*ability check for the requested amount of money failed (because [5] *AmountATMUnavailabe* or *AmountExceedsDailyLimit*)

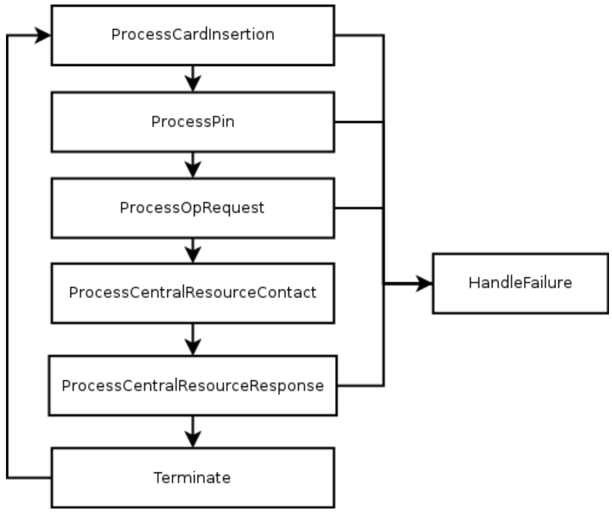

Figure 1: ATM **(Component Structure Macro View) and** HANDLEFAILURE

- *Fail*(PROCESSCENTRALRESOURCERESPONSE) if after ATM had started to *WaitFor*(*ContactCentralResource*) a *ContactResponse*(*ConnectionRefused*) arrives (from the CENTRALRESOURCE or from the network).

We reflect this behaviour by defining the involved normal and failure actions as separate ATM components. This yields the model (structure) in Fig. 1,[6] a control state ASM whose component ASMs are detailed in Sect. 3.[7]

The requirements [3] forsee also interruptions which can be triggered by a) *Cancel* commands the user can input 'any time' and b) by various *Timeout* events due to realtime constraints. We model the effect of such events by an INTERRUPTTRIGGER component (which triggers a component to HANDLEINTERRUPTS) so that the Atm ground model can be defined as parallel composition of two machines:

GROUNDATM =
  if *ThereAreInterrupts*
    then HANDLEINTERRUPT
    else { ATM
          HANDLEFAILURE
  INTERRUPTTRIGGER

3. REQUIREMENTS CAPTURE FOR THE ATM COMPONENTS

3.1 PROCESSCARDINSERTION

[5]We introduce the first *LocalAvail*ability check case for reliability and the second one for security reasons, trying to minimize accesses to the central resource and thereby the possibility that the needed connection can be interrupted. This models 'that the amount withdrawn within the day must be stored on the card or database' [3] by deciding for the 'stored on the card' alternative.

[6]For layout reasons the names of MACHINEs are displayed in the figures (where they appear always in rectangles) in upper camelcase, i.e. as Machine.

[7]The flowchart diagram yields a syntactically correct control state ASM as defined in [10] via an unfolding of the submachines, see Fig. 14. To better visualize the component structure and interaction we hide in Fig. 1 the initial and final control states in the submachines as well as the connections between (more precisely the identification of) the final state of one and the initial state of the next submachine by displaying only the names for the components and the connecting arrows.

A session with (an instance of) the Atm is started when a user physically inserts a card. We describe this by a monitored predicate *CardInserted*, assumed to become true when a card is physically inserted and false when it is physically *Removed*. In the model a session can be started only if the till is in *idle mode*.

To ProcessCardInsertion the machine checks whether the *insertedCard* is a *ValidCard*. If it is, the machine will ReadCard, Initialize Session and *StartPinRequest*, otherwise it moves into *Fail*(*InvalidCard*) mode. This explains the control state ASM definition of ProcessCardInsertion in Fig. 2 in classical flowchart notation: ovals represent modes, rectangles machines (where two or more machines appearing in one rectangle are executed in parallel), rhombs conditions to proceed to the target machine or mode. We highlight *Fail*(*type*) *modes* in which HandleFailure(*type*) (see Sect. 3.7) is started.

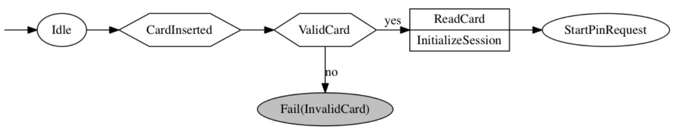

Figure 2: ProcessCardInsertion

To ReadCard means to (try to) retrieve all relevant (static or dynamic) uniquely determined card attribute values from *insertedCard*, e.g.

- *circuit*(*card*) indicates the card type, *pinCode*(*card*), *account(card)*,
- *centralResource*(*card*) holds the current status of the *account(card)*,
- *dailyLimit*(*card*),
- *alreadyWithdrawn*(*day*, *card*) indicates the total amount of money withdrawn this *day* in previous sessions at some tills using *card*, etc.[8]

It is an implementation issue to decide whether these attribute values are locally copied as part of ReadCard. Thus we abstract from single updates by writing *currCard* := *insertedCard* and retrieve the attribute values by applying attribute functions to *currCard*. For robustness reasons we include into the ReadCard component the case that no reading can take place if a card is inserted that is not *Readable* (e.g. corrupted or not an Atm card at all).

Since an Atm session needs some auxiliary locations to store intermediate data we include into the model to InitializeSession which is assumed to update all private ATM locations to their default values, for example by

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

[8]In a ground model ASM we deliberately name the objects and operations of discourse in application domain and not in data representation terms, speaking for example about *cards*, *accounts* and *account*(*card*) directly instead of *cardIds* and *accountIds* assuming *cards* to be uniquely associated with *accountIds*, typical concerns which are dictated by how to represent the objects in code and not by the matter, here that there is a function determining the unique *account* belonging to a *card*.

where *today* is updated at midnight by a Calendar component of the ATM.

The validity check consists in checking whether the *insertedCard* is *Readable* and belongs to one of the *Circuit*s of card types the ATM accepts.

ValidCard =
  Readable(*insertedCard*) **and**
  $circuit(currCard) \in Circuit$

3.2 ProcessPin

To ProcessPin means to first AskFor(*Pin*). If upon reaching *Ready*(*Pin*) *ValidPin* is false but the user *HasMoreAttempts*, pin requests can be repeated until they reach *Fail*(*InvalidPin*) or *ValidPin*—if no InterruptTrigger occurred due to a *Cancel* command or a *Timeout*(*AskFor*(*Pin*)) during the execution of AskFor(*Pin*). This explains the definition of ProcessPin in Fig. 3.

For the *ValidPin* check it is required that the inserted pin is 'encoded by the till and compared with a code stored on the card'[3]. We reflect this by applying an abstract (for concrete circuits refinable) $encode_{Pin}$ function to the collected Pin *userInput* which is recorded in location *valFor*(*Pin*) (by the AskFor(*Pin*) submachine ProcessInputStream(*Pin*) defined below).

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

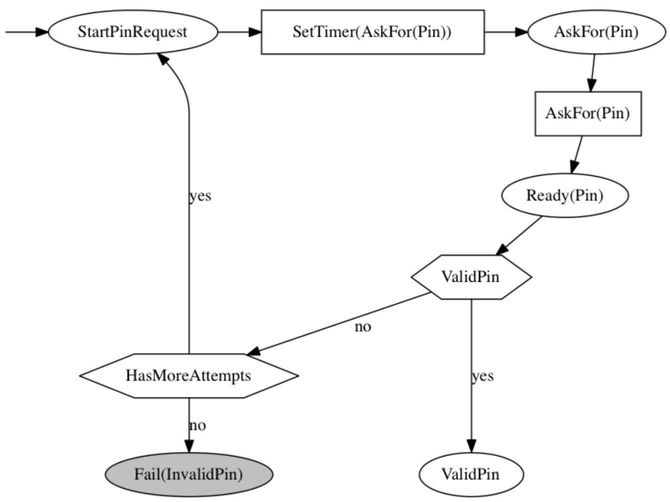

Figure 3: ProcessPin

3.2.1 User Input Requesting Submachine AskFor.

Repeatedly an ATM does AskFor *userInput* for a *Pin*, an operation (*Balance*, *Statement*, *Withdrawal*) or the requested *Amount* of money. Unless an InterruptTrigger occurs it stores this input in a location *valFor*(*param*) and enters mode *Ready*(*param*) (so that it can also ResetTimer with these *param*).

AskFor(*param*) =
  valFor(*param*) := *userInput*
  mode := *Ready*(*param*)
  ResetTimer(AskFor(*param*))

For a refinement which models how to read and process keywise provided input streams see Sect. 5.2.

3.2.2 Timing Mechanism.

To satisfy the requirements for eventual completion within real-time constraints of each started transaction a timing mechanism is needed. To keep the granularity of timing flexible and to abstract from implementation details we introduce for each *timedOpn* a *timer*(*timedOpn*) which comes with an appropriate *deltaTime(timedOpn)* determining the *Timeout*(*timedOpn*) predicate. The monitored location *now* describes the current system time.

SetTimer(*timedOpn*) = (*timer*(*timedOpn*) := *now*)
Timeout(*timedOpn*) =
  $now - timer(timedOpn) > deltaTime(timedOpn)$
ResetTimer(*timedOpn*) = (*timer*(*timedOpn*) := ∞)
 // this update falsifies *Timeout*(*timedOpn*)

3.3 ProcessOpRequest

Once a *ValidPin* is identified ProcessOpRequest must AskFor(*OpChoice*) and SetTimer(*OpChoice*) (the latter for real-time concerns). When after the user's choice it reached mode *Ready*(*OpChoice*) it checks whether there are *RequiredData* for the chosen operation *op*. If not (case $op \in \{Balance, Statement\}$) it enters mode *Ready*(*ContactCR*). Due to [3] there are *RequiredData* only for $op = Withdrawal$, namely the requested *amount* of money the machine must AskFor and CheckLocalAvailability: whether *AmountATMUnavail*able or *AmountExceedsDailyLimit*. This explains the definition in Fig. 4 (next page).

How to seamlessly combine the mere control flow view of CheckLocalAvail with data is illustrated by the ASM refinement in Sect. 5.1.[9]

CheckLocalAvail =
  choose $m \in$ {*Ready*(*ContactCR*),
    Fail(*AmountAtmUnavail*),
    Fail(*AmountExceedsDailyLimit*)} **do**
      mode := *m*
      if *m* = *Ready*(*ContactCR*) **then**
        amount := *valFor*(*Amount*)

3.4 ProcessCentralResourceContact.

We use an abstract Send machine which encodes and Sends messages from a sender (here represented by the address *Atm* of the till which tries to establish the contact) to a receiver (here represented by the *centralResource*(*currcard*)) address with the appropriate *opChoiceData*. The message must contain the information on the *currCard*, the user's operation choice *valFor*(*OpChoice*) and for a *Withdrawal* operation the requested *amount*. To further specify Send is out of the scope of this case study.

ContactCentralResource =
  Send($encode_{till}$(*Atm*, *Cr*, *RequestData*))
  Display(*WaitingForCentralResourceContact*)
where
  Atm = *address*(*till*(**self**))
  Cr = *address*(*centralResource*(*currCard*))
  RequestData =
    opChoiceData(*currCard*, *valFor*(*OpChoice*))
  opChoiceData(*card*, *opn*) =

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

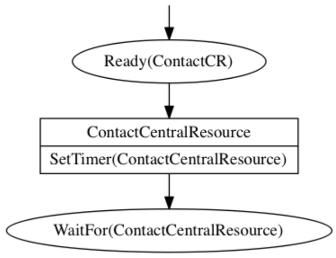

Figure 5: ProcessCentralResourceContact

ProcessCentralResourceContact only triggers the underlying physical process and sets the appropriate *timer* for *ContactCentralResource*. Reasonably the machine can be interrupted but not *Fail* since failed Sending is detected by ProcessCentralResourceResponse. This explains the definition in Fig. 5.

3.5 ProcessCentralResourceResponse

A *ContactResponse*(*ConnectionRefused*) may arrive (from the network or from the central resource) and lead from *WaitFor*(*ContactCentralResource*) to a *Fail* mode, unless an InterruptTrigger occured.

Possible *CentralResourceRes*ponses which do not *Fail* but lead to normal *TerminateOp* mode[10] are the following (see Fig. 6 and Fig. 7)[11]:

- A response to a *Statement* or *Balance* request or a response to a *Withdrawal* request indicating that the information on the account held at the central resource was not available there (*InfoUnavailable*). In these cases the machine will TerminateOp with Ejecting the *currCard* and Displaying the appropriate information to the user: the actual *balance* or that *InfoUvailable* or the confirmation that the requested statement will be sent by post or that the requested amount cannot be allowed any more for the account.
- A response stating that *currCard* is what the requirements call an *IllegalCard*, i.e. a card that has been

[9] Into the submachine CheckLocalAvail one could introduce yet another *timer*(*CheckLocalAvail*) but we abstain from this because this machine executes automatically and presumably fast.

[10] To refine ProcessCentralResourceResponse such that a user in one session can request several operations is left as a model change exercise.

[11] Closing the connection to the Central Resource serves to minimize the dependence of a till from the connection.

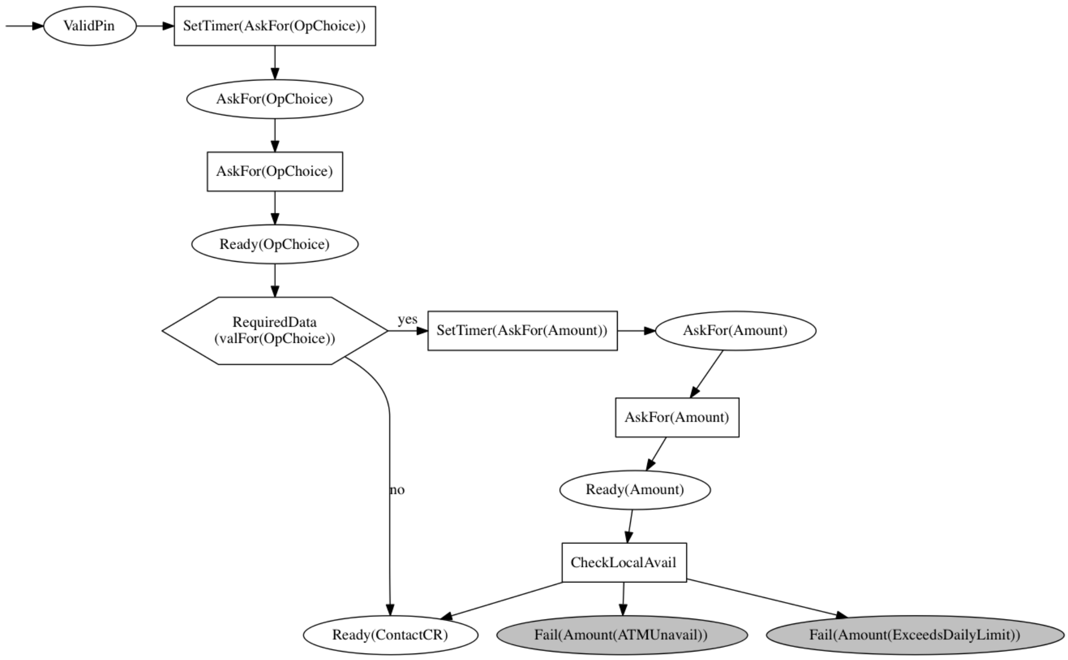

Figure 4: ProcessOpRequest

blocked (e.g. by the owner who reported that it has been lost/stolen or by the bank because of irregular card owner behaviour). In this case the requirements request to TerminateOp with Keeping the *currCard* and maybe informing the user about this.

- A response that the requested *amount* of money is granted. In this case the requirements request to TerminateOp with Ejecting the money *amount*, an action tills usually perform only after the *currCard* has been Ejected by the till and been *Removed* by the user.

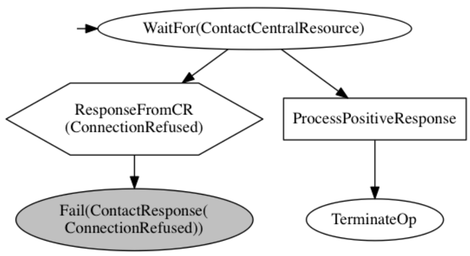

Figure 6: ProcessCentralResourceResponse

3.5.1 Atm *Communication Predicates.*

We define here the conditions which guard the action to be taken upon arrival of a response from the Central Resource. We model the arrival of such a response message as update of a monitored Atm (say mailbox) location *CRresp* by the underlying message passing system. Assuming that initially (e.g. in InitializeSession) this location is set to a message *defaultVal*ue we can formulate the arrival of a response message as *CRresp* having been updated to a value that is different from its *defaultVal*ue. Functions *type*, *answer* retrieve from a response message its type and content:

$$\begin{aligned}
&\mathit{ResponseFromCR}(\mathit{param}) = \\
&\quad (\mathit{CRresp} \neq \mathit{defaultVal}\ \textbf{and}\ \mathit{type}(\mathit{CRresp}) = \mathit{param})
\end{aligned}$$

$$\begin{aligned}
&\mathit{GrantedAmount} = \\
&\quad \mathit{type}(\mathit{CRresp}) = \mathit{Withdrawal}\ \textbf{and}\ \mathit{answer}(\mathit{CRresp}) = \\
&\quad \mathit{Ok}
\end{aligned}$$

$$\begin{aligned}
&\mathit{RefusedAmount} = \\
&\quad \mathit{type}(\mathit{CRresp}) = \mathit{Withdrawal}\ \textbf{and} \\
&\qquad (\mathit{answer}(\mathit{CRresp}) = \mathit{notOk}\ \textbf{or} \\
&\qquad \mathit{answer}(\mathit{CRresp}) = \mathit{InfoUnavailable})
\end{aligned}$$

In TerminateOp$(\mathit{CRresp}, \mathit{Eject}(\mathit{currCard}))$ the *CRresp* parameter serves for $\mathit{type}(\mathit{CRresp}) \in \{\mathit{Balance}, \mathit{Withdrawal}\}$ to permit Display(CRresp) (see the definition below) to retrieve the information on the $\mathit{answer}(\mathit{CRresp})$.

3.6 Terminate and TerminateOp

Atm enters *TerminateOp* mode to a) Eject or Keep the *currCard* (depending on the given *reason* to terminate) and b) for a successful *Withdrawal* to also Eject the requested *amount* of money. To model this we define TerminateOp as parametrized by a) the *reason* why to *TerminateOp*—about which the user is informed by Displaying a screen (or a voice communication)—and b) the sequence of *actions* to be executed for that *reason*.

$$\begin{aligned}
&\textsc{TerminateOp}(\mathit{reason}, \mathit{actions}) = \\
&\quad \textsc{Display}(\mathit{reason}) \\
&\quad \mathit{TerminationActions} := \mathit{actions}
\end{aligned}$$

Since to Terminate it may be that the till *HasMoreTerminationActions* to perform, the machine iterates performing each single TerminationAction before it re-enters the

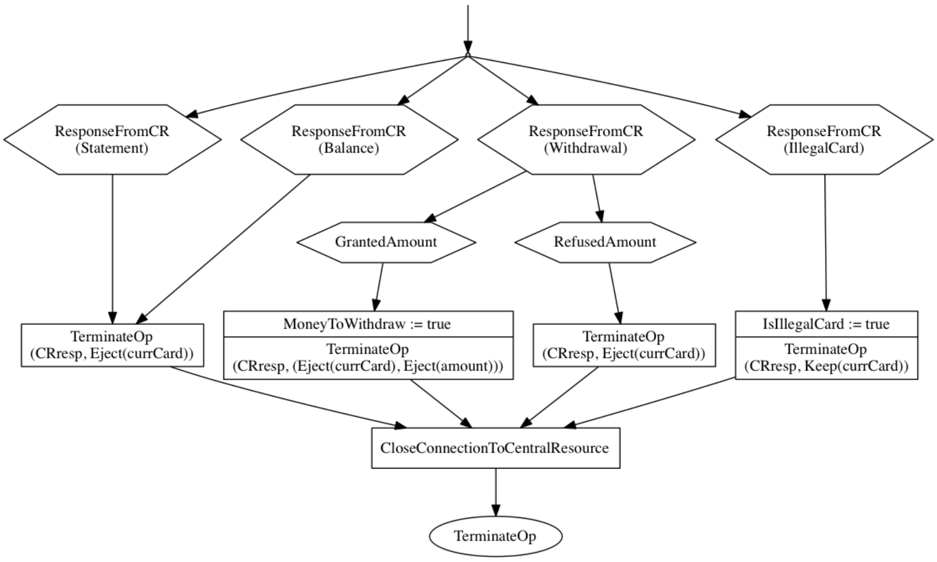

Figure 7: PROCESSPOSITIVERESPONSE

idle mode (in which it is ready to start a new session).[12]

This explains the definition of TERMINATE in Fig. 8. During this phase the user cannot provide any more any input, a robustness condition for the till.

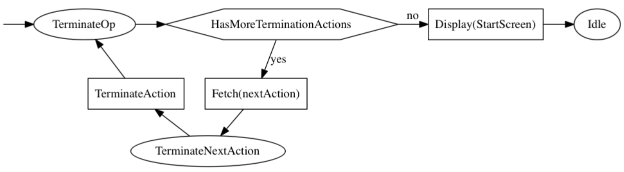

Figure 8: TERMINATE

In TERMINATEACTION the actions can be *EjectActions* (i.e. EJECT(*card*) or EJECT(*money*)) but—to satisfy the reliability requirement 'to minimise the possibility of the use of stolen cards' [3]—also *KeepActions* (namely KEEP(*card*) or KEEP(*money*), see Fig. 9) in case the user did not withdraw the ejected card/money within some time interval, say of length *deltaTime*(*Removal*).

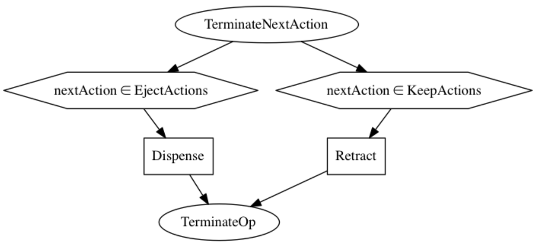

Figure 9: TERMINATEACTION

To DISPENSE in case of a *GrantedAmount* response from the central resource, after the *currCard* has been EJECTed there is still *MoneyToWithdraw* so that, to satisfy the *dailyLimit* requirement, the ATM is defined in Fig. 10 to:

- RECORDMONEYWITHDRAWALONCARD for *today* before the *currCard* is EJECTed,
- RECORDMONEYWITHDRAWALATATM after the successful removal of the money *amount* requested by the user.

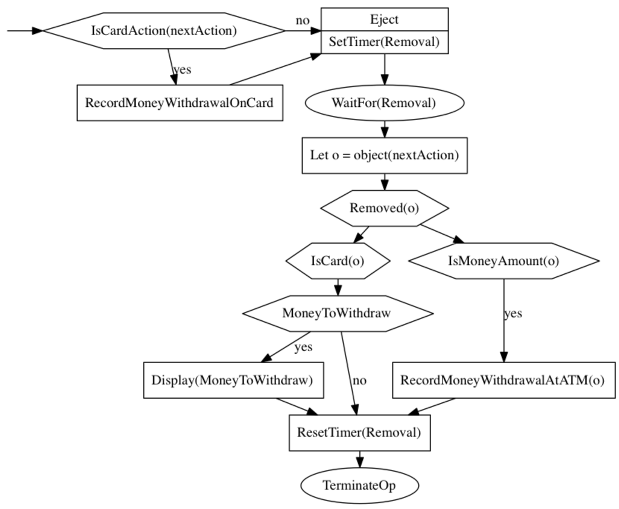

Figure 10: DISPENSE(o)

[12] To CLEARSESSION upon moving back to *idle* mode belongs to garbage collection. INITIALIZESESSIONFOR(*insertedCard*) prevents old session values to move into a new session.

RETRACT =
  REMOVE
    // physically remove card or money from slot
  LOGMISSEDWITHDRAWAL
  $mode := TerminateOp$
RECORDMONEYWITHDRAWALONCARD =
  if $MoneyToWithdraw$ **then**
    // true only if GrantedAmount
    $alreadyWithdrawn(today, currCard) :=$
      $amount + alreadyWithdrawn(today, currCard)$
    $dayOfLastWithdrawal(currCard) := today$

RECORDMONEYWITHDRAWALATATM(o) =
  $money(Atm) := money(Atm) - o$
  // called only after successful money Witdrawal

To make the interface between the logical ASM ground model and its physical environment explicit we leave REMOVE abstract and interpret it when concerning a card as triggering the physical action to insert the ejected *card* into a stock of retained cards.

Remark. RECORDMONEYWITHDRAWALONCARD is done at the latest possible moment, namely just before *currCard* is physically EJECTed. Nevertheless the risk remains that should the ATM fail to also EJECT the requested and granted *amount* of money (although *amount* has been checked before in CHECKLOCALAVAIL to be *AtmAvail*able), then the *amount* has been added anyway to *alreadyWithdrawn* for parameters ($today, currCard$) and thus affects checking the *WithinDailyLimit* condition for further withdrawal attempts made later *today* by the user of the *currCard*. To prepare the ground for a later correction of such a mismatch LOGMISSEDWITHDRAWAL is added to REMOVE.[13]

3.7 Failure Handling Submachines

In failure mode $Fail(param)$ the machine HANDLEFAILURE is called which depends on the *param*eter indicating the kind of *Fail*ure (read: non successfully completed ATM session) that happened. For the $Fail(InvalidPin)$ case the requirements request to KEEP*(currCard)* as "Illegal", in the other *Fail*ure cases an Atm typically will EJECT the card.[14] Other options are possible, e.g. one could offer in case of $Fail(AmountExceedsDailyLimit)$ more $attemptsFor(Amount)$ with lower amounts by inserting a loop as illustrated for *attemptsFor(Pin)*, but in view of the case study character of this work we leave this as an exercise.[15]

[13] In a real ATM more logging takes place, easily included into our model, but since the case study requirements do not mention logging we do not consider it further.

[14] In particular we interpreted the meaning of "Illegal card" [3] as refering to cards the central resource declares as illegal, excluding unreadable cards and cards not belonging to the till's circuits—which in the model are ejected.

[15] For the same reason we leave it as an exercise to define a refinement that reflects the particular 'change any time' case that 'Customers can change ... any time ... the amount they want to withdraw' mentioned in [3].

HANDLEFAILURE($param$) =
  if $mode = Fail(param)$ **then**
    if $param = InvalidPin$ **then**
      TERMINATEOP($InvalidPin$, KEEP($currCard$))
    else
      TERMINATEOP($param$, EJECT($currCard$))
    CLOSECONNECTIONTOCENTRALRESOURCE
    $mode := TerminateOp$
where
  CLOSECONNECTIONTOCENTRALRESOURCE =
    RESETTIMER($ContactCentralResource$)
    DISCONNECTATMFROMCR
    //trigger to physically disconnect

In the same way one can define other HANDLEFAILURE components, e.g. allowing the user to continue with some operation at the till or to include triggering repair services in case of physical defects of the till, etc., depending on the requirements.

3.8 The Interrupt Components

If the user has *Pressed* the *CancelKey* when the ATM *IsInCancelRegion* or if a $Timeout(timedOpn)$ happens, then the machine INTERRUPTTRIGGER activates HANDLEINTERRUPT, e.g. by inserting these events into *InterruptEvent*.[16] HANDLEINTERRUPT chooses a *highPriority* interrupt event e to HANDLE(e); when defining the *highPriority* function one could for example declare *Cancel* commands to be of higher priority than *Timeout*s. For *Cancel* events e occurring when $IsInCancelRegion$(ATM) and for *Timeout* events e concerning a *timedOpn* occurring when *IsInTimerRegion(timedOpn)* HANDLE(e) means to TERMINATESESSION with EJECTing the card.

Remark. For running ATM scenarios in CoreASM with an input providing user, in [24] 'any time' is interpreted as 'any user input time'.

INTERRUPTTRIGGER =
  if $Pressed(CancelKey)$
    and $IsInCancelRegion$(ATM) **then**
      INSERT($Cancel, InterruptEvent$)
  forall $timedOpn \in \{AskFor(param),$
    $ContactCentralResource,$
    $Removal\}$ **do**
      if $Timeout(timedOpn)$ **and**
        $IsInTimerRegion(timedOpn)$ **then**
          INSERT($timer(timedOpn), InterruptEvent$)
          RESETTIMER($timedOpn$)

[16] In general one will have multisets or sequences, but in this case study a set suffices.

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

One can define *IsInCancelRegion* and *IsInTimerRegion*s by refering to the *Modes*[17] in which an interrupt event should have effect. The following definition expresses that no *Cancel* command has any effect outside a user session (when $\mathit{mode} = \mathit{idle}$) or when the Atm is performing automatically its final stage to TERMINATE the session. Also a *Timeout* has an effect for a *timedOperation* only if it *IsInTimerRegion*. HANDLEINTERRUPT discards interrupt events which happen outside the region where they are defined to have an effect.

$$
\begin{array}{l}
\mathit{IsInCancelRegion}(\textsc{Atm}) = \\
\quad \mathit{mode} \notin \{\mathit{idle}\} \cup \mathit{Mode}(\textsc{Terminate}) \\[1em]
\mathit{IsInTimerRegion}(\mathit{AskFor}(\mathit{param})) = \\
\quad \mathit{mode} \in \{\mathit{AskFor}(\mathit{param}), \mathit{WaitFor}(\mathit{param})\} \\[1em]
\mathit{IsInTimerRegion}(\mathit{ContactCentralResource}) = \\
\quad \mathit{mode} = \mathit{WaitFor}(\mathit{ContactCentralResource}) \\[1em]
\mathit{IsInTimerRegion}(\mathit{Removal}) = \\
\quad (\mathit{mode} = \mathit{WaitFor}(\mathit{Removal}))
\end{array}
$$

3.9 The ATM CALENDAR Component

For a correct handling of the requested *dailyLimit* of cards a CALENDAR component (which in turn requires a CLOCK component) is needed which updates *today* every midnight, using a calendar function to compute the *nextDay*:[18]

$$
\begin{array}{l}
\textsc{Calendar} = \\
\quad \textbf{if } \mathit{now} = \mathit{midnight} \textbf{ then } \mathit{today} := \mathit{nextDay}(\mathit{today})
\end{array}
$$

4. MODELING THE CENTRALRESOURCE

Concentrating on what the requirements impose for a correct communication between tills and a central resource we only need to model two modules to ACCEPTREQUESTS and to HANDLEREQUESTS. We make no assumption on their synchronization by the central resource. Thus we stipulate that there is a set *Request* into which the ACCEPTREQUESTS component can insert messages (say already *decode*d into a request format the Central Resource works with) which arrived in the *Mailbox* of the central resource from a till and from where asynchronously the HANDLEREQUESTS component can fetch requests to handle them.

Since requests can be assumed to have a unique identity it is consistent to permit in the model simultaneous access to *Request* by the two components, even for multiple messages or (under certain constraints, see below) multiple requests at a time. This is easily modelled exploiting the ASM parallelism and leaves the greatest possible freedom to schedule message accepting and handling by the central resource any way which is reasonable to implement the required 'concurrent access to the database from two or more different tills' [3], not restricted to interleaving (which is the preferred way to deal with concurrency in verification tools supporting e.g. the (Event)-B [1, 2] and TLA$^+$ [14] approach to modeling)[19]. In particular it permits to separate the two distinct features of the concurrency concern stated in [3], namely to

- guarantee exclusive access to an account upon 'concurrent access to the database from two or more different tills' both concerning a same account, so that two simultaneously present requests cannot violate the other requirement that only 'any amount up to the total in the account may be withdrawn',
- allow for flexible priority resp. scheduling policies to implement concurrent database accesses for different accounts.

$$
\begin{array}{l}
\textsc{CentralResource} = \\
\quad \textsc{AcceptRequests} \\
\quad \textsc{HandleRequests} \\
\textbf{where } \textsc{AcceptRequests} = \textbf{if } \mathit{Mailbox}_{CR} \neq \emptyset \textbf{ then} \\
\quad \textbf{choose } R \subseteq \mathit{Mailbox}_{CR} \textbf{ and } R \neq \emptyset \\
\quad\quad \textbf{forall } \mathit{msg} \in R \left\{ \begin{array}{l} \textsc{Insert}(\mathit{decode}_{CR}(\mathit{msg}), \mathit{Request}) \\ \textsc{Delete}(\mathit{msg}, \mathit{Mailbox}_{CR}) \end{array} \right.
\end{array}
$$

The freedom of choosing a mailbox subset allows one to exploit for further refinements of **choose** any form of parallelism the central resource offers.

We adopt a similar approach for HANDLEREQUESTS, illustrating the use of selection functions when modelling with ASMs. Consider a (possibly dynamic) function select_{CR} which each time it is applied to the (dynamically changing) set *Request* chooses a *Consistent* non-empty subset $R \subseteq \mathit{Request}$. Then one can HANDLE all requests in R in parallel. The set is *Consistent* if it contains no multiple Withdraw requests concerning a same account. [20]

[17] By $\mathit{Mode}(M)$ we denote the set of possible *mode* values of M.

[18] An issue to be addressed for a real Atm is what should happen to attempts to withdraw money around midnight. We leave it as an exercise to develop a refinement (in particular of RECORDMONEYWITHDRAWALONCARD) which implements a decision about to which day to attribute the real withdrawal.

[19] This does not contradict the fact that the behavioural specifications in the ASM, (Event-) B and TLA$^+$ approaches (namely by control state ASMs, (Event-) B machines resp. TLA$^+$ state machines) are often rather similar. The difference is that the ASM method allows one to state and prove properties using mathematics, not only the tool supported part of it.

[20] This definition guarantees exclusive Withdrawal access per account and leaves any other combination of accesses to the same account to the database parallelism. For simultaneous Withdrawal and Balance (or Statement) access to the same account by two users it provides the account total *be-*

HANDLEREQUESTS = **if** *Request* $\neq \emptyset$ **then**
let $R = select_{CR}(Request)$
//NB. R is assumed to be *Consistent*
$$\textbf{forall } r \in R \begin{cases} \text{HANDLE}(r) \\ \text{DELETE}(r, Request) \end{cases}$$
where $Consistent(R) =$
thereisno $r, r' \in R$ **with** $r \neq r'$
and $account(r) = account(r')$
and $op(r) = op(r') = Withdrawal$

Remark on concurrency. Another approach to guarantee transactional behaviour of the Central Resource in the presence of multiple HANDLEREQUESTS instances is to define HANDLEREQUESTS(*request*) for single *request*s and then to harness a set of its instances by the transaction control operator defined in [9].

To retrieve *req*uest resp. to encode response data HANDLE(*req*) uses appropriate functions like *sender*(*req*), *card*(*req*), *op*(*req*), *account*(*req*), etc. which we deliberately leave abstract to be further refinable for concrete databases. HANDLE(*req*) triggers the CentralResource to SEND a *CRresp*onse of type *op*(*req*) to the *sender*(*req*), where *op*(*req*) is one of *Withdrawal*, *Statement* or *Balance*.

For *Balance* requests 'Information on accounts is held in a central database and may be unavailable' and in this case viewing the *account* *balance*(*acc*) 'may not be possible' [3]. It seems reasonable to apply the same principle to Amount requests. This explains the definition of HANDLE(*req*) in Fig. 11 (next page).

5. MODEL VERIFICATION, VALIDATION, REFINEMENTS

5.0.1 Ground Model Correctness.

All the till properties required in [3] hold for the ATM ground model, in fact they drove its definition. In particular the model is defined to 'minimize the possibility of the use of stolen card to gain access to an account' by KEEPing "IllegalCard"s (including cards that are recognized as blocked) and cards or money which after having been EJECTed are not *Removed* in due time.

The model by its abstract communication concepts (namely SEND, $Mailbox_{CR}$, *ResponseFromCR* and *CRresp*) not only leaves space for a great variety of different communication protocols, but also permits to refine the features for the connection between tills and the central resource to any reasonable (feasible) implementation, e.g. using 'a data line between each till and the central database' as suggested in [3] and considering that the connection used for the communication can be interrupted (by a *Timeout* or a *Cancel* command from the user or by being simply *Refused* by the network or the database).

The model satisfies the property that 'once a user has initiated a transaction, the transaction is completed at least eventually and preferably within some real time constraint'. It follows for every run from the modelled *Timeout* features.

fore the withdrawal. If one wants to get the account total *after* the simultaneous Withdrawal to be sent back to the information requesting user one can refine the consistency condition accordingly and refine the HANDLE(*req*) machine defined below by a new clause for simultaneous Withdrawal and Balance (or Statement) requests for the same account.

To support ground model inspection by validation through testing (running scenarios) the GROUNDATM has been refined in [24] to a CoreASM executable version. As to be expected through experiments with the CoreASM executable version we found various flaws, incoherences and places for improvement of GROUNDATM. Due to the mathematical foundation of ASMs which we need not explain here—the interested reader is referred to [10]—the *implementation correctness* obtainable by ASM refinements can be shown using mathematical (including machine supported) methods, accompanying traditional code inspection, testing and property verification methods. For references see [10, Ch.9.4.3].

5.0.2 Reuse and changes during maintenance by refinements.

The concern to keep the ground model components and definitions as abstract as possible, driven by the desire to reflect the requirements without adding anything concerning implementation issues, yields automatically a model which can be easily changed to accommodate changing requirements in three ways: a) by defining some abstract model elements in a specific way (which is supported by the ASM refinement method [5]) or b) by adding new elements to capture new features via conservative (purely incremental) ASM refinements or c) by changing the definitions for given model elements to capture non-incremental requirements changes. We mentioned some simple examples in the preceding sections.

5.1 Refinement of the CHECKLOCALAVAIL Component

CHECKLOCALAVAIL of Sect. 3.3 is data refined by the definition in Fig. 12. The refinement provides the exact reasons (of local availability) for which the machine may become *Ready* to contact the central resource.

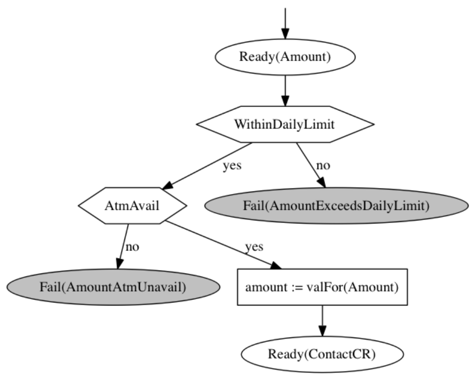

Figure 12: CHECKLOCALAVAIL

5.2 Refinement of the ASKFOR Component

We refine ASKFOR to stepwise read and process input key values unless interrupted. The refined machine starts to INITIALIZEINPUTELABORATION, in particular to DISPLAY the request to the user,[21] and then enters *WaitFor*(*param*)

[21] To capture a robustness constraint—namely that keys pressed before the ATM begins to *WaitFor*(*param*) yield no input—through INITIALIZEINPUTELABORATION user in-

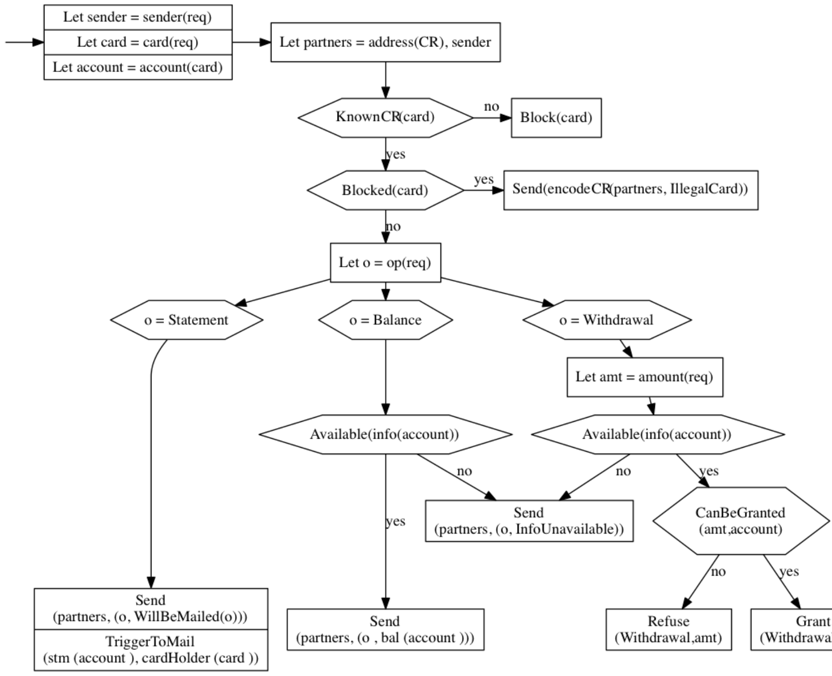

Figure 11: HANDLE(*req*)

mode to READINPUTSTREAM and to PROCESSINPUTSTREAM until by a *Confirm* input it moves to *Ready*(*param*) (see Fig. 13).

INITIALIZEINPUTELABORATION(*param*) =
  INITIALIZE(*inputStream*) // Start listening to user input
  INITIALIZE(*userInput*) // Start processing user input
  DISPLAY(*AskFor*(*param*))
  if *param* = *Pin* **then**
    COUNTDOWN(*attemptsFor*(*Pin*))

INITIALIZE(*Stream*) = (*Stream* := [])

COUNTDOWN(*attemptsFor*(*Pin*)) =
  attemptsFor(*Pin*) := *attemptsFor*(*Pin*) − 1

The COUNTDOWN of *attemptsFor*(*Pin*) serves for multiple *attemptsFor(Pin)* insertion, assuming that *attemptsFor*(*Pin*) is initialized to an Atm specific positive value (e.g. in INITIALIZESESSION).[22]

5.2.1 Submachine to READINPUTSTREAM from User.

What shall happen if a user hits simultaneously multiple keys? Typically the hardware transforms this into a randomly ordered *inputStream*. We model this by using a function *randomOrder* which randomly yields for any *set* a sequence *randomOrder*(*set*) that is ADDed to the current *inputStream*. Furthermore typically the hardware before applying *randomOrder* to a *set* will *truncate*(*set*) in a device dependent manner to a subset. A function *inputVal* yields elementwise for every pressed key in a given sequence its input value.

READINPUTSTREAM =
  let *PressedKeys* = {*key* | *Pressed*(*key*)}
  let *Newinput* = *inputval*(*randomOrder*(
    truncate(*PressedKeys*)))
      // Insert at the left end
      ADDATTHELEFT(*Newinput*, *inputStream*)

PROCESSINPUTSTREAM =
  if *inputStream* ≠ [] **then**
    let *val* = *fstOut*(*inputStream*)
    REMOVEATTHERIGHT(*val*, *inputStream*)

5.2.2 Submachine to PROCESSINPUTSTREAM(*param*).

It does UPDATEINPUTBY the values that are *LegalFor param* (e.g. (alpha-) numerical values or keys with predefined values), one by one (say from right to left) from *inputStream* to (subsequently to be elaborated) *userInput* until a *Confirming* value is encountered triggering a normal *Ready*(*param*) exit (unless an INTERRUPTTRIGGER occurs). The *Delete* key (which the requirements impose to be considered as *LegalFor* every *param*eter) reflects that the user can change the input any time.

What should happen if the user provides an $IllegalFor_{param}$ input *val*ue? How to HANDLEILLEGALINPUT(*val*, *param*) is a functional behaviour issue not considered in [3]. Here we decided to ignore such input, but to inform the user about it by a DISPLAY.

put is taken only after having called ASKFOR(*param*) and is stopped when this machine is exited.

[22] Some banks use a *dailyPinRequestLimit*(*card*) to prevent repeated successive sessions attempting to find out the correct pin. It is a simple model change exercise to refine PROCESSPIN correspondingly. Be aware of the resulting feature interaction with *dailyLimit* in case a user on one day makes numerous *Withdrawal*s without exceeding the *dailyLimit* and without inserting an invalid pin.

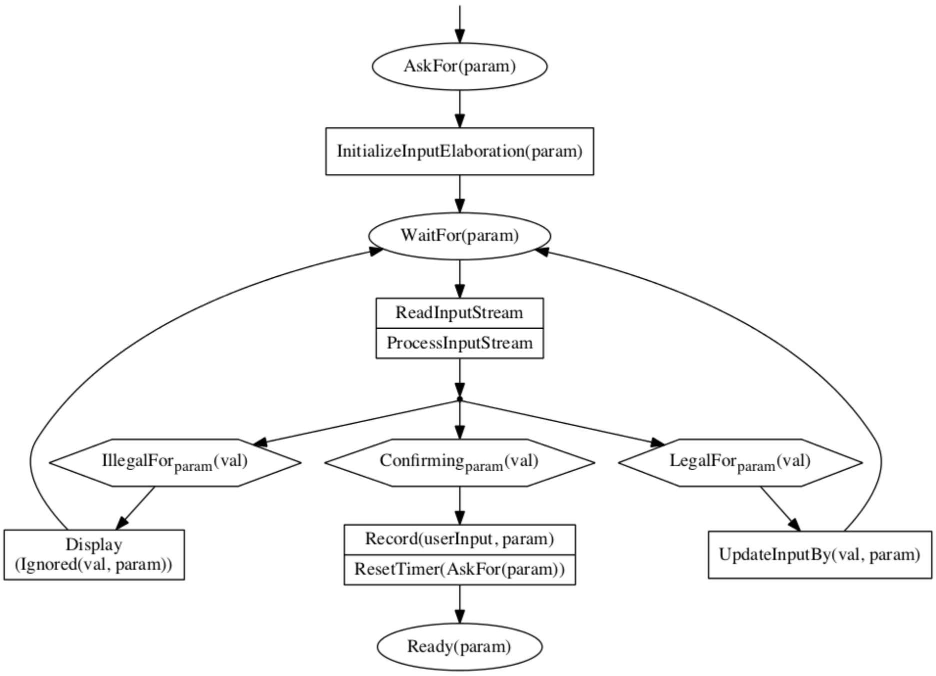

Figure 13: $\textsc{AskFor}(\mathit{param})$

$$
\begin{array}{l}
\textsc{UpdateInputBy}(\mathit{val}, \mathit{param}) = \\
\quad \textbf{if } \mathit{val} \neq \mathit{Delete} \textbf{ then } \textsc{AddToInput}(\mathit{val}, \mathit{param}) \\
\quad \textbf{if } \mathit{val} = \mathit{Delete} \textbf{ then} \\
\quad\quad \textsc{RemoveFromInput}(\mathit{val}, \mathit{param})
\end{array}
$$

$$
\begin{array}{l}
\textsc{AddToInput}(\mathit{val}, \mathit{param}) = \\
\quad \mathit{userInput} := \mathit{concatenateAtTheRight}(\mathit{userInput}, \mathit{val}) \\
\quad \textsc{Display}(\mathit{concatenateAtTheRight}(\mathit{userInput}, \mathit{val}), \mathit{param})
\end{array}
$$

$$
\begin{array}{l}
\textsc{RemoveFromInput}(\mathit{val}, \mathit{param}) = \\
\quad \mathit{userInput} := \mathit{removeLast}(\mathit{userInput}) \\
\quad \textsc{Display}(\mathit{removeLast}(\mathit{userInput}), \mathit{param})
\end{array}
$$

$$
\mathit{Confirming}_{\mathit{param}}(\mathit{val}) \text{ if and only if} \\
\begin{cases}
\mathit{param} \in \{\mathit{Pin}, \mathit{Amount}\} \textbf{ and } \mathit{val} = \mathit{Confirm} \\
\mathit{param} = \mathit{OpChoice} \textbf{ and} \\
\quad \mathit{val} \in \{\mathit{Balance}, \mathit{Statement}, \mathit{Withdrawal}\}
\end{cases}
$$

$$
\begin{array}{l}
\textsc{Record}(\mathit{input}, \mathit{param}) = \\
\quad \textbf{if } \mathit{param} \in \{\mathit{Pin}, \mathit{Amount}\} \textbf{ then} \\
\quad\quad \mathit{valFor}(\mathit{param}) := \mathit{input} \\
\quad \textbf{if } \mathit{param} \in \{\mathit{Balance}, \mathit{Statement}, \mathit{Withdrawal}\} \textbf{ then} \\
\quad\quad \mathit{valFor}(\mathit{param}) := \mathit{param}
\end{array}
$$

Remark. Since the values ReadInputStream has to AddAtTheLeft and the values ProcessInputStream has to RemoveAtTheRight are different occurrences of the *inputval*ue of pressed keys the two operations can consistently be executed simultaneously at the right resp. left end of *inputStream*.

To appear in the Proceedings of S-BPM One 2015 to be published by ACM.

6. REFERENCES

[1] J.-R. Abrial. *The B-Book*. Cambridge University Press, Cambridge, 1996.

[2] J.-R. Abrial. *Modeling in Event-B*. Cambridge University Press, 2010.

[3] AnonymousAuthor. Automatic teller machine or till: Case study. Formulated 1999 as *Modelling in two formalisms: The FM'99 ATM modelling challenge* for the FM99 Conference and as *A Cash-point Service Example* in the IFAD document V6.3.0a, reused 2013 for the Dagstuhl Seminar on *Integration of Tools for Rigorous Software Construction and Analysis* to which A. Fleischmann added in 2014 the change and cancel requirements.

[4] The ASM Metamodel website. `http://asmeta.sourceforge.net`.

[5] E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.

[6] E. Börger. Construction and analysis of ground models and their refinements as a foundation for validating computer based systems. *Formal Aspects of Computing*, 19:225–241, 2007.

[7] E. Börger. Approaches to modeling business processes. A critical analysis of BPMN, Workflow Patterns and YAWL. *J.SSM*, pages 1–14, 2011.

[8] E. Börger, P. Päppinghaus, and J. Schmid. Report on a practical application of ASMs in software design. In *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 361–366. Springer, 2000.

[9] E. Börger and K.-D. Schewe. Specifying transaction control to serialize concurrent program executions. In *Proc. ABZ 2014*, volume 8477 of *LNCS*, pages 142–157. Springer, 2014.

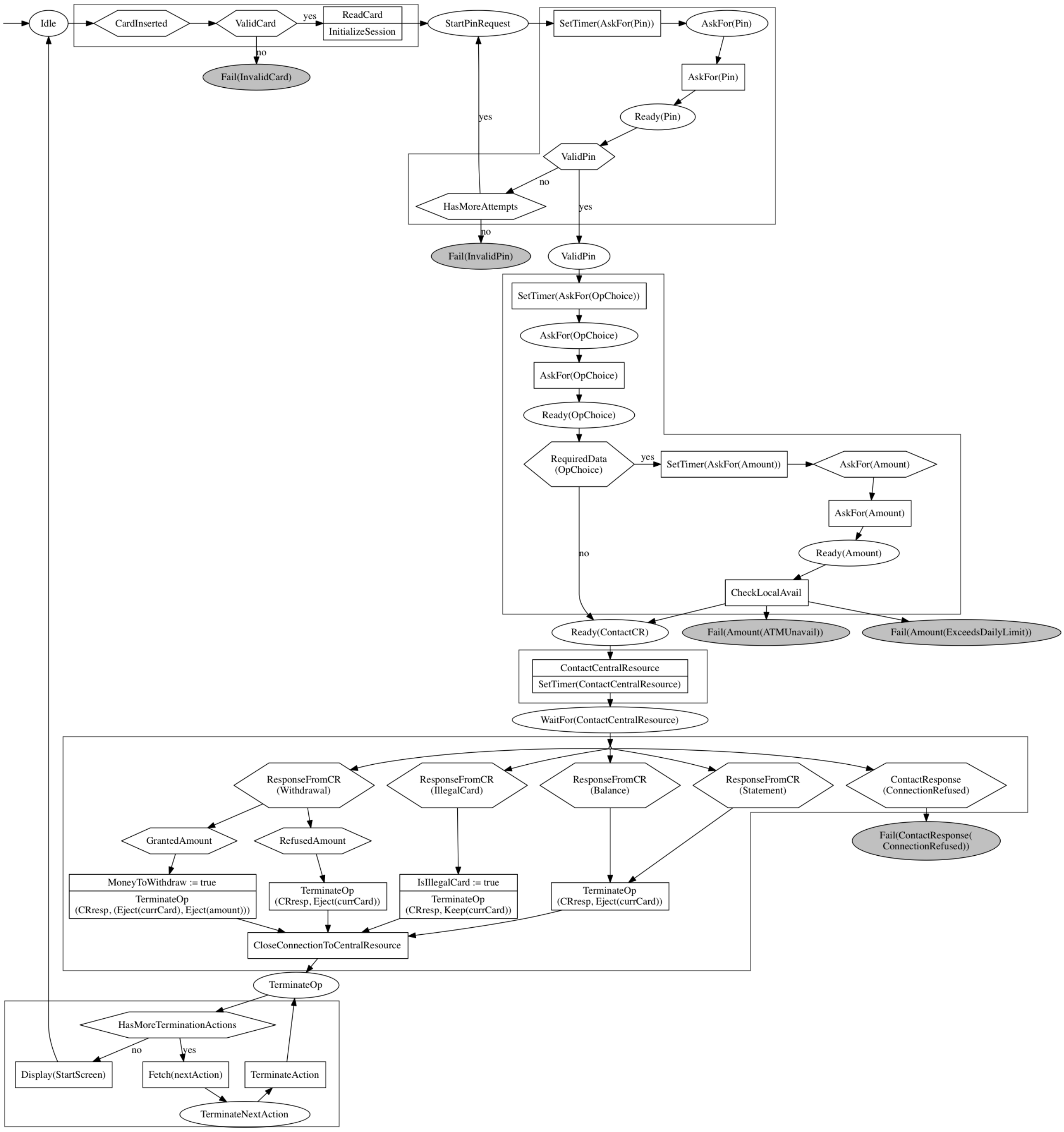

Figure 14: ATM **(Detailed View with Unfolded Components)**

[10] E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.

[11] A. Fleischmann, W. Schmidt, C. Stary, S. Obermeier, and E. Börger. *Subject-Oriented Business Process Management*. Springer Open Access Book, 2012.

[12] A. Hense. A CSPm model for the automated teller machine case study. In J. Ehlers and B. Thalheim, editors, *Proc. S-BPM ONE 2015 (Special Session on Comparative Case Studies)*. ACM, ACM Digital Library, 2015. Docu of executable available at http://www.bis.inf.fh-brs.de/bpmcasestudies/.

[13] A. Hense and R. Malz. Automation of the automated teller machine case study with YAWL. In J. Ehlers and B. Thalheim, editors, *Proc. S-BPM ONE 2015 (Special Session on Comparative Case Studies)*. ACM, ACM Digital Library, 2015. Docu of YAWL executable available at http://www.bis.inf.fh-brs.de/bpmcasestudies/.

[14] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2003. Available at `http://lamport.org`.

[15] OMG. UML 2.0 superstructure specification. http://www.omg.org/cgi-bin/doc?formal/05-07-04.

[16] OMG. Business Process Model and Notation (BPMN). http://www.omg.org/spec/BPMN/2.0, 2011. formal/2011-01-03.

[17] D. Parnas and M. Lawford. The role of inspection in software quality assurance. *IEEE Transactions on Software engineering*, 29(8):674–676, 2003.

[18] G. Schellhorn. Verification of ASM refinements using generalized forward simulation. *J.Universal Computer Science*, 7(11):952–979, 2001.

[19] G. Schellhorn. ASM refinement and generalizations of forward simulation in data refinement: A comparison. *Theoretical Computer Science*, 336(2-3):403–436, 2005.

[20] G. Schellhorn. ASM refinement preserving invariants. *J.UCS*, 14(12), 2008.

[21] G. Schellhorn. Completeness of ASM refinement. *Electr. Notes TCS*, 214, 2008.

[22] G. Schellhorn. Completeness of fair ASM refinement. *SCP*, 76(9):756–773, 2011.

[23] A. ter Hofstede, W. van der Aalst, M. Adams, and N. Russell, editors. *Modern Business Process Automation*. Springer, 2010.

[24] S. Zenzaro. A CoreASM refinement implementing the ATM ground model. http://www.bis.inf.fh-brs.de/bpmcasestudies/, October 2014.