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 modelling 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] — 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 modelling 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] 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

---

1The 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.

---

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

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

4The 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, constraints), 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 constraints in [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 LocalAvailability check for the requested amount of money failed (because $AmountATMUnavailable$ 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, a control state ASM whose component ASMs are detailed in Sect. 3.\footnote{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.}

The requirements [3] foresee also interruptions which can be triggered by a) $Cancel$ commands the user can input ‘any time’ and b) by various $Timeout$ events due to real-time 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

\footnote{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`, `InitializeSession` 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](image)

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.

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

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

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 `Circuits` of card types the ATM accepts.

```plaintext
ValidCard = Readable(insertedCard) and circuit(currCard) ∈ 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) `encodePin` function to the collected Pin `userInput` which is recorded in location `valFor(Pin)` (by the `AskFor(Pin)` submachine `ProcessInputStream(Pin)` defined below).

```plaintext
ValidPin = (pinCode(currCard) = encodePin(valFor(Pin)))
```

![Figure 3: ProcessPin](image)

#### 3.2.1 User Input Requesting Submachine `AskFor`.

Repetedly 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 `RESET_TIMER` with these `param`).
operation the requested amount. To further specify SEND is out of the scope of this case study.

\textbf{CONTACTCENTRALRESOURCE} =
\begin{verbatim}
SEND(encode_un(Atm, Cr, RequestData))
DISPLAY(WaitForCentralResourceContact)
\end{verbatim}

\textbf{where}
\begin{itemize}
\item \textit{Atm} = address(till(self))
\item \textit{Cr} = address(centralResource(currCard))
\item \textit{RequestData} = \textit{opChoiceData(currCard, valFor(OpChoice))}
\item \textit{opChoiceData(card, opn)} =
\begin{itemize}
\item \{(card, opn) \text{ if } opn \in \{\text{Balance, Statement}\}\}
\item \{(card, opn, amount) \text{ if } opn = \text{Withdrawal}\}
\end{itemize}
\end{itemize}

Figure 5: \textit{PROCESSCENTRALRESOURCECONTACT}

\textbf{PROCESSCENTRALRESOURCECONTACT} only triggers the underlying physical process and sets the appropriate \textit{timer} for \textit{ContactCentralResource}. Reasonably the machine can be interrupted but not \textit{Fail} since failed \textit{SENDing} is detected by \textit{PROCESSCENTRALRESOURCERESPONSE}. This explains the definition in Fig. 5.

\textbf{3.5 PROCESSCENTRALRESOURCERESPONSE}

A \textit{ContactResponse} (\textit{ConnectionRefused}) may arrive (from the network or from the central resource) and lead from \textit{WaitFor(ContactCentralResource)} to a \textit{Fail} mode, unless an \textit{INTERRUPT_TRIGGER} occurred.

Possible \textit{CentralResourceResponses} which do not \textit{Fail} but lead to normal \textit{TerminateOp} mode\footnote{To refine \textit{PROCESSCENTRALRESOURCERESPONSE} such that a user in one session can request several operations is left as a model change exercise.} are the following (see Fig. 6 and Fig. 7)\footnote{Closing the connection to the Central Resource serves to minimize the dependence of a till from the connection.}:
\begin{itemize}
\item A response to a \textit{Statement} or \textit{Balance} request or a response to a \textit{Withdrawal} request indicating that the information on the account held at the central resource was not available there (\textit{InfoUnavailable}). In these cases the machine will \textit{TerminateOp} with \textit{Ejecting} the \textit{currCard} and \textit{Displaying} the appropriate information to the user: the actual \textit{balance} or that \textit{InfoUnavailable} 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.
\item A response stating that \textit{currCard} is what the requirements call an \textit{IllegalCard}, i.e. a card that has been
\end{itemize}
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 usually performed 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 we can formulate the arrival of a response message as CRresp having been updated to a value that is different from its defaultVal. Functions type, answer retrieve from a response message its type and content:

$$\text{ResponseFromCR}(\text{param}) = \begin{cases} \text{defaultVal} & \text{if type(CRresp) = param} \\ \text{GrantAmount} = & \text{if type(CRresp) = Withdrawal and answer(CRresp) = Ok} \\ \text{RefusedAmount} = & \text{if type(CRresp) = Withdrawal and (answer(CRresp) = notOk or} \\ & \text{answer(CRresp) = InfoUnavailable)} \end{cases}$$

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

$$\text{TerminateOp(reason, actions)} = \begin{cases} \text{Display(reason)} \\ \text{TerminationActions := actions} \end{cases}$$

Since to Terminate it may be that the till HasMoreTerminationActions to perform, the machine iterates performing each single TerminationAction before it re-enters the
idle mode (in which it is ready to start a new session). 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.

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 $\delta$TIME(Removal).

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.
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(todaty, currCard)
dayOfLastWithdrawal(currCard) := today
RECORDMONEYWITHDRAWALATATM(o) =
    money(Atm) := money(Atm) - o
    // called only after successful money Withdrawal

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 CHECKLOCAVAIL to be AtmAvailable), 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.\(^\text{13}\)

### 3.7 Failure Handling Submachines

In failure mode \textit{Fail}(param) the machine HANDLEFAILURE is called which depends on the \textit{parameter} indicating the kind of Failure (read: non successfully completed ATM session) that happened. For the Fail(InvalidPin) case the requirements request to KEEP(currCard) as ‘Illegal’, in the other Failure cases an ATM typically will EJECT the card.\(^\text{14}\) 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.\(^\text{15}\)

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 \textit{CancelKey} when the ATM IsInCancelRegion or if a \textit{Timeout}(timedOpn) happens, then the machine INTERRUPTTRIGGER activates HANDLEINTERRUPT, e.g. by inserting these events into InterruptEvent.\(^\text{16}\) 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 Timouts. 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)

\(^{13}\)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 referring 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].

\(^{16}\)In general one will have multisets or sequences, but in this case study a set suffices.
HandleInterrupt =
let e = highPriority(InterruptEvent)
Handle(e)
DELETE(e, InterruptEvent)
where
Handle(Cancel) =
if isInCancelRegion(Atm) then
TERMINATESESSION(Cancel)
Handle(timer(timedOperation)) =
if isInTimerRegion(timedOperation) then
TERMINATESESSION(Terminate(timedOperation))
TERMINATEOP(p, EJECT(currCard))
mode := TerminateOp

One can define isInCancelRegion and isInTimerRegions
by referring to the Modes17 in which an interrupt event should
have effect. The following definition expresses that no Cancel
command has any effect outside a user session (when mode =
idle) or when the Atm is performing automatically its fi-
nal 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.

isInCancelRegion(Atm) =
mode \not\in\{idle\} \cup Mode(TERMINATE)
isInTimerRegion(AskFor(param)) =
mode \in \{AskFor(param), WaitFor(param)\}
isInTimerRegion(ContactCentralResource) =
mode = WaitFor(ContactCentralResource)
isInTimerRegion(Removal) =
(mode = WaitFor(Removal))

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

Calendar =
if now = midnight then today := nextDay(today)

4. MODELING THE CENTRAL RESOURCE

Concentrating on what the requirements impose for a cor-
rect 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

17By Mode(M) we denote the set of possible mode values
of M.
18An 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 imple-
ments a decision about to which day to attribute the real withdrawal.
synchronization by the central resource. Thus we stipulate
that there is a set Request into which the AcceptRequests
component can insert messages (say already decoded into a
request format the Central Resource works with) which ar-
rived in the Mailbox of the central resource from a till and
from where asynchronously the HANDLEREQUESTS compo-
nent 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 par-
allelism 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 ‘concur-
rent access to the database from two or more different
tills’ [3], not restricted to interleaving [which is the pre-
ferred way to deal with concurrency in verification tools supporting
e.g. the (Event-)B [1, 2] and TLA+ [14] approach to model-
ing]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 on concurrent
  access to the database from two or more different
tills’ both concerning a same account, so that two si-
multaneously 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.

CentralResource =
AcceptRequests
HANDLEREQUESTS
where AcceptRequests = if MailboxCR \neq \emptyset then
choose R \subseteq MailboxCR and R \neq \emptyset
forall msg \in R 
\{ INSERT(DecodeCR(msg), Request)
DELETE(msg, MailboxCR)

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

We adopt a similar approach for HANDLEREQUESTS, illus-
trating the use of selection functions when modelling with
ASMs. Consider a (possibly dynamic) function selectCR
which each time it is applied to the (dynamically changing)
set Request chooses a Consistent non-empty subset \( R \subseteq
Request \). Then one can HANDLE all requests in \( R \) in paral-
lel. The set is Consistent if it contains no multiple Withdraw
requests concerning a same account. 20

19This does not contradict the fact that the behavioural spec-
fications 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 differ-
ence is that the ASM method allows one to state and prove
properties using mathematics, not only the tool supported
part of it.
20This 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 ≠ ∅ then
let R = selectCR(Request)
//NB. R is assumed to be Consistent
for all r ∈ R { HANDLE(r)
where Consistent(R) =
thereisno r, r′ ∈ R with r ≠ 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 requests and then to harness a set of its instances by the transaction control operator defined in [9].

To retrieve request 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 CRresponse 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, MailboxCR, 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.

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)

21To capture a robustness constraint—namely that keys pressed before the ATM begins to WaitFor(param) yield no input—through INITIALIZEINPUTELABORATION user in-
mode to ReadInputStream and to ProcessInputStream until by a Confirm input it moves to Ready(param) (see Fig. 13).

\[\text{InitializeInputElaboration}(\text{param}) = \]
\[\text{Initialize}(\text{inputStream}) \quad \text{// Start listening to user input}\]
\[\text{Display}(\text{AskFor}(\text{param})) \]
\[\text{if } \text{param} = \text{Pin} \quad \text{then} \]
\[\text{COUNTDown}(\text{attemptsFor}(\text{Pin}))\]

\[\text{Initialize}(\text{Stream}) = (\text{Stream} := [])\]

\[\text{COUNTDown}(\text{attemptsFor}(\text{Pin})) = \]
\[\text{attemptsFor}(\text{Pin}) := \text{attemptsFor}(\text{Pin}) - 1\]

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).\(^2^2\)

### 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 \(\text{randomOrder}(\text{set})\) which randomly yields for any set put is taken only after having called \(\text{AskFor}(\text{param})\) and is stopped when this machine is exited.

\[\text{ReadInputStream} = \]
\[\text{let } \text{PressedKeys} = (\text{key} \mid \text{Pressed}(<\text{key}>)\} \]
\[\text{let } \text{Newinput} = \text{inputval}((\text{randomOrder}((\text{truncate}(<\text{PressedKeys}>)\})) \]
\[\text{// Insert at the left end}\]
\[\text{AddAtTheLeft}(\text{Newinput}, \text{inputStream})\]

\[\text{ProcessInputStream} = \]
\[\text{if } \text{inputStream} \neq [] \quad \text{then}\]
\[\text{let } \text{val} = \text{fstOut}(\text{inputStream}) \]
\[\text{REMOVEATTheRight}(\text{val}, \text{inputStream})\]

### 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 parameter) reflects that the user can change the input any time.

What should happen if the user provides an IllegalFor param input value? 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.
Figure 13: AskFor(param)

\[
\text{UpdateInputBy}(\text{val}, \text{param}) = \\
\quad \text{if } \text{val} \neq \text{Delete} \text{ then AddToInput}(\text{val}, \text{param}) \\
\quad \text{if } \text{val} = \text{Delete} \text{ then RemoveFromInput}(\text{val}, \text{param})
\]

\[
\text{AddToInput}(\text{val}, \text{param}) = \\
\quad \text{userInput} := \text{concatenateAtTheRight}(\text{userInput}, \text{val}) \\
\quad \text{Display}(\text{concatenateAtTheRight}(\text{userInput}, \text{val}), \text{param})
\]

\[
\text{RemoveFromInput}(\text{val}, \text{param}) = \\
\quad \text{userInput} := \text{removeLast}(\text{userInput}) \\
\quad \text{Display}(\text{removeLast}(\text{userInput}), \text{param})
\]

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

\[
\text{Record}(\text{input}, \text{param}) = \\
\quad \text{if } \text{param} \in \{\text{Pin, Amount}\} \text{ then} \\
\quad \quad \text{valFor}(\text{param}) := \text{input} \\
\quad \text{if } \text{param} \in \{\text{Balance, Statement, Withdrawal}\} \text{ then} \\
\quad \quad \text{valFor}(\text{param}) := \text{param}
\]

**Remark.** Since the values ReadInputStream has to AddAtTheLeft and the values ProcessInputStream has to RemoveAtTheRight are different occurrences of the input value 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


Figure 14: ATM (Detailed View with Unfolded Components)


