Chapter 6

Modelling ATM Case-study

6.1 Introduction

In this chapter, we present another case-study modelled in Event-B to further investigate our research findings of production cell (PC) case-study. This could also help in evaluating the application of existing standard Event-B composition and decomposition techniques for our feature-based reuse approach. We decided to model automated teller machine (ATM) because it provides significant variability as compared to PC. The ATM example is more reuse-oriented in terms of requirements features and suits the traditional feature modelling approach. We can model a product line of ATM systems having different features which can be configured to build variants of ATM. We explore further patterns for composition and to what extent we can reuse existing specifications and their associated proofs to build more products of a product line. This enables us to generalise our feature-oriented reuse framework in the form of a modelling pattern and its application on a different system to that of PC. Based on this case-study, we can then suggest modelling guidelines for future users of Event-B to model distributed systems while improving reusability of specification and their associated proofs. This case-study also generated requirements for future tools and techniques to further enhance reusability of Event-B developments.

6.2 ATM

An ATM provides various services to a bank's customers using their ATM cards issued by the bank. There are some basic services provided by an ATM such as cash withdrawal, viewing account balance and card pin related services. Other services can also be provided by ATMs which vary for different banks and ATM locations, e.g., mobile top up and cash deposit etc. We can build a product line of ATMs to manage variability

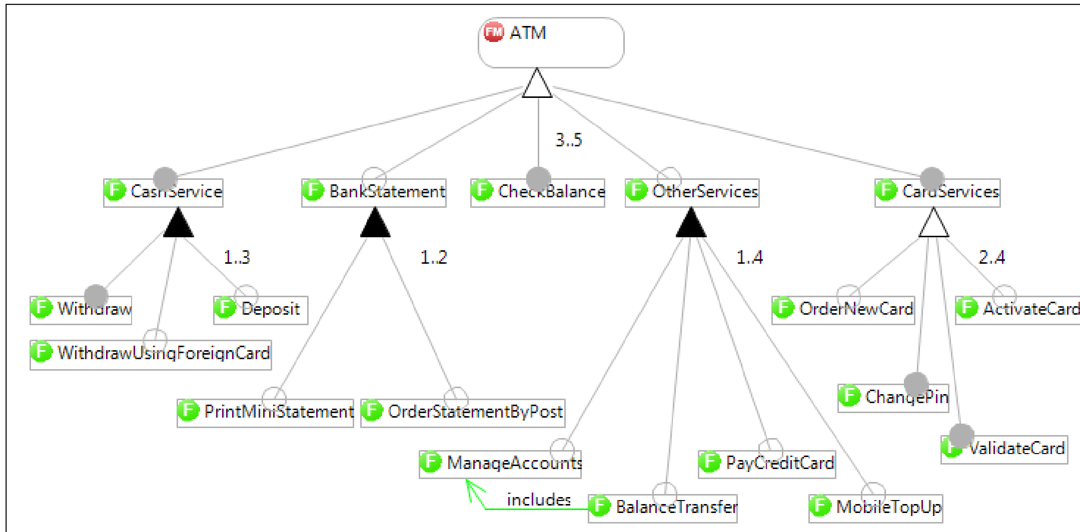


Figure 6.1: ATM feature model

and commonality and benefit from reuse while building several ATM products having different features. A set of available features configured and composed together result in a variant of an ATM product line. Figure 6.1 shows a feature model for the ATM product line and the requirements specification for the ATM case-study is given in Appendix B. Although some ATM requirements have been previously modelled in Event-B [118], we have used a different set of requirements and modelled these in a different way to experiment with our feature-oriented reuse approach.

6.3 Roadmap

In the following section, we show the Event-B development of some ATM features and explore the amount of reuse that can be achieved by using existing (de)composition techniques of Event-B. We also investigate whether existing tools and techniques are capable enough for our proposed feature-oriented modelling in Event-B. Based on this, we can then suggest any requirements for the tools and techniques to be developed in the future to compliment our feature-based reuse framework. By generalising the modelling style used in this case-study, we can also provide a set of guidelines for Event-B users to model feature-oriented systems as expected in product line development.

At first, we modelled two features of ATM product line and by using the two existing decomposition techniques of Event-B, we show how we can avoid reproof efforts through reuse by following a pattern of modelling. Sections 6.4.1 and 6.4.2 discuss the modelling, refinement and decomposition of transfer and deposit features respectively. The composition of various sub-components of the two features (resulting from SED) is presented in Section 6.4.3 to model an ATM product. After modelling and refining these two features to build one ATM product, we modelled another ATM feature - withdraw - to

MACHINE IntegralATM_0
SEES IntegralATM_CO
VARIABLES
bal, cardAcct, validCard
INVARIANTS
inv1 : $bal \in ACCOUNT \rightarrow\!\!\!\!\!+ \mathbb{N}$
inv2 : $cardAcct \in CARD \rightarrow\!\!\!\!\!+ ACCOUNT$
inv3 : $validCard \subseteq CARD$
EVENTS

Event *Transfer* $\widehat{=}$
any
$src_ac, dest_ac, am, c$
where
grd1 : $src_ac \in ACCOUNT$
grd2 : $dest_ac \in ACCOUNT$
grd3 : $c \in validCard$
grd4 : $c \mapsto src_ac \in cardAcct$
grd5 : $am \in \mathbb{N}_1$
grd6 : $src_ac \in dom(bal)$
grd7 : $dest_ac \in dom(bal)$
grd8 : $src_ac \neq dest_ac$
grd9 : $am \leq bal(src_ac)$
then
act1 : $bal := bal \lhd\!\!\!- \{dest_ac \mapsto (bal(dest_ac) + am), src_ac \mapsto (bal(src_ac) - am)\}$
end

Event *Deposit* $\widehat{=}$
any
acc, am, c
where
grd1 : $acc \in ACCOUNT$
grd2 : $acc \in dom(bal)$
grd3 : $am \in \mathbb{N}_1$
grd4 : $c \in validCard$
grd5 : $c \mapsto acc \in cardAcct$
then
act1 : $bal(acc) := bal(acc) + am$
end
END

Figure 6.2: Integral ATM abstract model for two features

build a second ATM product having three features, discussed in Section 6.4.4. We show how we can reuse existing features to build the second product and save user time and effort while making sure that there are no inconsistencies and ambiguities introduced during the application of our suggested modelling pattern, while preserving refinement during the (de)composition. In Section 6.4.5, we evaluate and generalise the suggested modelling pattern. This pattern also supports team-based development where different features of a product line could be modelled in parallel by different teams as far as the restrictions of the pattern are maintained.

6.4 Event-B Modelling of ATM Features

We started with an abstract model of ATM that models its requirements for a set of features. These include: *withdraw, deposit, check balance, balance transfer, change PIN* and *validate card*. So, in the abstract model, we have events for each of these features. Some of these features are mandatory while others are optional, as shown in the ATM feature model (Figure 6.1). Any instance of the ATM feature model must contain the mandatory features. For example, the card validation feature is required for any ATM product. So, we have modelled this as an independent reusable feature to be included in

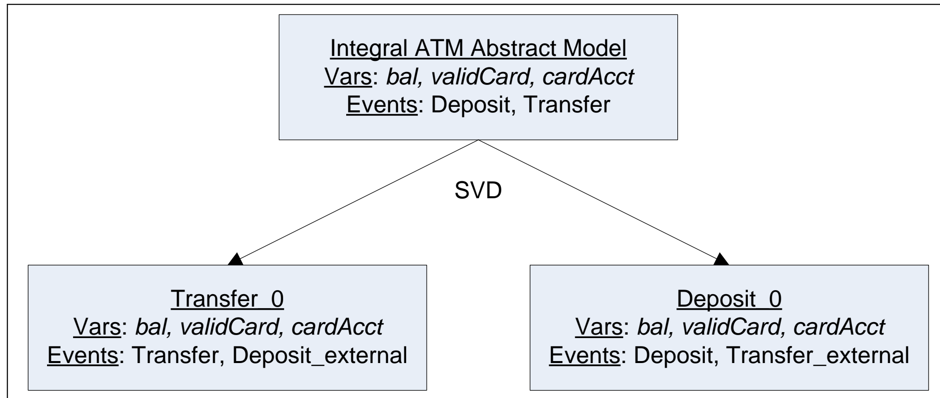


Figure 6.3: ATM integral model decomposed using SVD

any configuration of the ATM product line and this also helps in avoiding redundancy when modelling a subset of features from the feature model.

In order to save time and to use a smaller example, we only consider a subset of these features. So, we have an integral model of the ATM that allows cash deposit and balance transfer between two accounts. This abstract model is shown in Figure 6.2. In terms of user actions, we assume that an ATM card is validated in the card validation feature before any of these two features could be used by the user. We then decomposed this model into deposit and transfer features (Figure 6.3) using shared-variable decomposition (SVD). This acts as a problem decomposition step. This pattern remains valid even if we model and refine the deposit and transfer features separately, as far as the two features could later be shown as a result of shared-variable decomposition of a model. This would require a tool to generate external events in the developments of both features and to validate that the shared-variables are not refined.

As a result of decomposition, the event ***Deposit*** goes into the deposit feature and the ***Transfer*** event goes into the transfer feature - see Figure 6.4. Both features now have shared variables (i.e., `bal, validCard` and `cardAccount`) and external events (i.e., deposit feature has ***Transfer*** as an external event and the transfer feature has ***Deposit*** as an external event). All external events and shared-variables must not be refined as a consequence of SVD. Note that the external events shown in the figure are exact copies of their internal counterparts. This is because there are no local variables present for each of the feature and all the parameters are being used by the shared variables. Normally, external events are abstracted away leaving out details of their local variables which do not appear at this abstract level. Following is the detail for each of these features and their stepwise refinement, both horizontally and vertically.

```
MACHINE Deposit
VARIABLES
    bal        // Shared variable
    validCard  // Shared variable
    cardAcct   // Shared variable
INVARIANTS
    inv4 : bal ∈ ACCOUNT ⇸ ℕ
    inv5 : cardAcct ∈ CARD ⇸ ACCOUNT
    inv6 : validCard ⊆ CARD
EVENTS
Event Deposit ≙
    any
        acc, am, c
    where
        grd1 : acc ∈ ACCOUNT
        grd2 : acc ∈ dom(bal)
        grd3 : am ∈ ℕ1
        grd4 : c ∈ validCard
        grd5 : c ↦ acc ∈ cardAcct
    then
        act1 : bal(acc) := bal(acc) + am
    end
Event Transfer ≙
    External event, DO NOT REFINE
    any
        src_ac, dest_ac, am, c
    where
        grd1 : src_ac ∈ ACCOUNT
        grd2 : dest_ac ∈ ACCOUNT
        grd3 : c ∈ validCard
        grd4 : c ↦ src_ac ∈ cardAcct
        grd5 : am ∈ ℕ1
        grd6 : src_ac ∈ dom(bal)
        grd7 : dest_ac ∈ dom(bal)
        grd8 : src_ac ≠ dest_ac
        grd9 : am < bal(src_ac)
    then
        act1 : bal := bal ⩤ {dest_ac ↦
               (bal(dest_ac) + am), src_ac ↦
               (bal(src_ac) − am)}
    end
END
```

```
MACHINE Transfer
VARIABLES
    bal        // Shared variable
    validCard  // Shared variable
    cardAcct   // Shared variable
INVARIANTS
    inv4 : bal ∈ ACCOUNT ⇸ ℕ
    inv5 : cardAcct ∈ CARD ⇸ ACCOUNT
    inv6 : validCard ⊆ CARD
EVENTS
Event Transfer ≙
    any
        src_ac, dest_ac, am, c
    where
        grd1 : src_ac ∈ ACCOUNT
        grd2 : dest_ac ∈ ACCOUNT
        grd3 : c ∈ validCard
        grd4 : c ↦ src_ac ∈ cardAcct
        grd5 : am ∈ ℕ1
        grd6 : src_ac ∈ dom(bal)
        grd7 : dest_ac ∈ dom(bal)
        grd8 : src_ac ≠ dest_ac
        grd9 : am < bal(src_ac)
    then
        act1 : bal := bal ⩤ {dest_ac ↦ (bal(dest_ac) +
               am), src_ac ↦ (bal(src_ac) − am)}
    end
Event Deposit ≙
    External event, DO NOT REFINE
    any
        acc, am, c
    where
        grd1 : acc ∈ ACCOUNT
        grd2 : acc ∈ dom(bal)
        grd3 : am ∈ ℕ1
        grd4 : c ∈ validCard
        grd5 : c ↦ acc ∈ cardAcct
    then
        act1 : bal(acc) := bal(acc) + am
    end
END
```

Figure 6.4: ATM integral model decomposed using SVD into transfer and deposit features

6.4.1 Refinement of Transfer Feature

The first refinement model of transfer feature refines the ***Transfer*** event for a successful transfer of money (***TransferOk***) and another event (***TransferFails***) is introduced when the transfer fails due to the account balance being less than the transfer amount as shown in Figure 6.5. Other reasons for failure of balance transfer can be introduced in subsequent refinements as required (e.g., failing due to hardware or communication problems, though not considered in the scope of this case-study). The event ***FinishTransfer*** completes the transfer and resets the ATM involved in the transfer for new transaction. As we can not refine the external event ***Despoit*** due to restriction of SVD, it is present at all refinement levels of transfer feature but not shown in the figures due to space limitations. Invariants show the typing of new variables introduced in this refinement, i.e., `transferOkA`, `transferFailA` and `cardInAtm`.

Figure 6.6 shows the events (both new and refined) of transfer feature during all the

INVARIANTS
inv9 : $transferOkA \in \mathbb{P}(ATM)$
in10 : $transferFailA \in \mathbb{P}(ATM)$
inv10 : $cardInAtm \in ATM \rightarrowtail CARD$
inv11 : $transferOkA \cap transferFailA = \varnothing$

Event *TransferOK* $\widehat{=}$
refines *Transfer*
any
src_ac, dest_ac, am, c, a
where
grd1 : $src_ac \in ACCOUNT$
grd2 : $dest_ac \in ACCOUNT$
grd3 : $c \in validCard$
grd4 : $c \mapsto src_ac \in cardAcct$
grd5 : $am \in \mathbb{N}_1$
grd6 : $src_ac \in dom(bal)$
grd7 : $dest_ac \in dom(bal)$
grd8 : $src_ac \neq dest_ac$
grd9 : $am \leq bal(src_ac)$
grd10 : $a \in dom(cardInAtm)$
grd11 : $c = cardInAtm(a)$
grd12 : $a \notin (transferOkA \cup transferFailA)$
then
act1 : $bal := bal \Leftarrow \{dest_ac \mapsto (bal(dest_ac) + am), src_ac \mapsto (bal(src_ac) - am)\}$
act2 : $transferOkA := transferOkA \cup \{a\}$
end

Event *TransferFails* $\widehat{=}$
any
src_ac, dest_ac, am, c, a
where
grd1 : $src_ac \in ACCOUNT$
grd2 : $dest_ac \in ACCOUNT$
grd3 : $c \in validCard$
grd4 : $c \mapsto src_ac \in cardAcct$
grd5 : $am \in \mathbb{N}_1$
grd6 : $src_ac \in dom(bal)$
grd7 : $dest_ac \in dom(bal)$
grd8 : $src_ac \neq dest_ac$
grd9 : $am > bal(src_ac)$
grd10 : $a \in dom(cardInAtm)$
grd11 : $c = cardInAtm(a)$
grd12 : $a \notin (transferOkA \cup transferFailA)$
then
act2 : $transferFailA := transferFailA \cup \{a\}$
end

Event *FinishTransfer* $\widehat{=}$
any
a
where
grd1 : $a \in ATM$
grd2 : $a \in (transferOkA \cup transferFailA)$
then
act1 : $transferOkA := transferOkA \setminus \{a\}$
act2 : $transferFailA := transferFailA \setminus \{a\}$
end

Figure 6.5: Transfer model's invariants and events of first refinement

refinements. The dotted lines show new events which refine *skip* whereas the solid lines show refined events. The event ***TransferFails*** of first refinement has similar structure to ***TransferOk*** event in terms of event refinement and hence not elaborated in the figure.

The second refinement introduces request and response mechanism between ATM and the Bank. Here, ATM sends a balance transfer request to the bank (i.e., ***ReqTransfer*** event), which responds after a successful or failed transfer event takes place (i.e., event ***TransferOk*** or ***TransferFails*** through event ***RespTransferOk*** or ***RespTransferFails*** respectively). The ATM then displays the transfer status (i.e., event ***TransferOkAtm*** or ***TransferFailsAtm***). The Event-B specification of all the events at this refinement level is shown in Figure 6.7. We also introduce a transaction variable **trans** (typing: $trans \in \mathbb{P}(ATM)$) for transferring balance which restricts the transfer request event only if the ATM is not currently in a transaction. An ATM currently in a transfer transaction could be in any of its sub states, as shown in the invariant below. Several new variables are also introduced at this refinement level.

inv18 : $partition(trans, reqTransfer, transferOk, transferFail, respTransferOk, respTransferFail)$

The third level of refinement further refines the request and response mechanisms by decomposing the request event for sending and receiving the request and similarly for the response event. For example, the abstract event ***ReqTransfer*** is decomposed into the refined event ***SendReqTransfer*** (which sends the balance transfer request from the ATM) and the new event ***RecvReqTransfer*** (which receives the balance transfer request at the

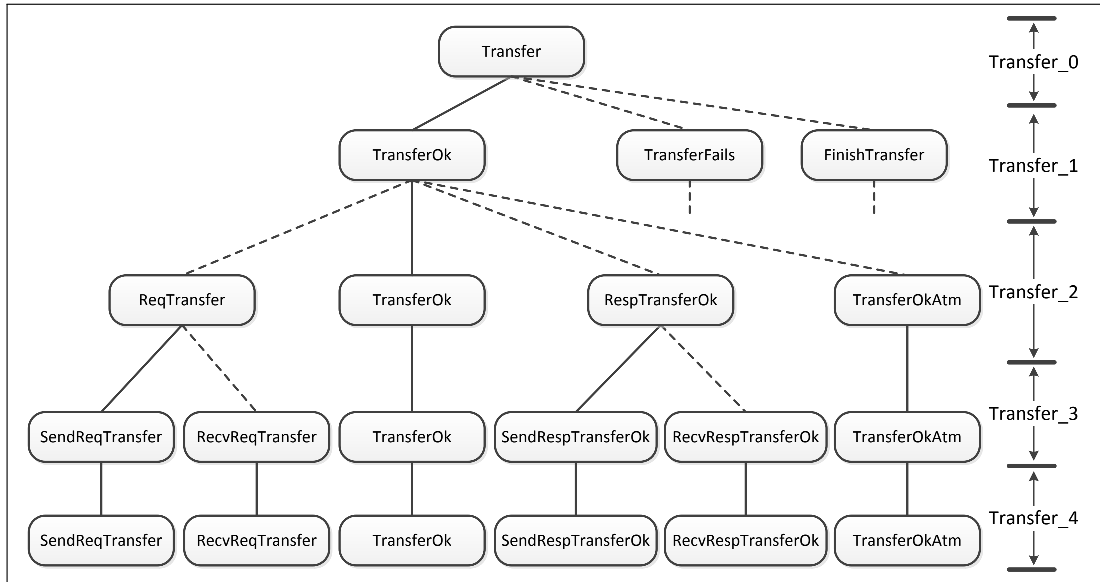


Figure 6.6: Event refinement of transfer feature

Table 6.1: Proof statistics for transfer refinements

Model	Auto	Manual	Total
Transfer_0	7	0	7
Transfer_1	6	0	6
Transfer_2	53	5	58
Transfer_3	58	2	60
Transfer_4	115	8	124

Bank). Similarly, the response abstract event ***RespTransferOk*** is decomposed into the refined event ***SendRespTransferOk*** (which sends the response for a successful transfer from the Bank) and the new event ***RecvRespTransferOk*** (which receives the successful transfer response at the ATM) and so on for the transfer failure events. The abstract request and response variables `reqTransfer, respTransferOk` and `respTransferFail` are also refined as shown below:

`inv7` : $partition(reqTransfer, sendReqTransfer, recvReqTransfer)$

`inv8` : $partition(respTransferOk, sendRespTransferOk, recvRespTransferOk)$

`inv9` : $partition(respTransferFail, sendRespTransferFail, recvRespTransferFail)$

The fourth refinement introduces middleware (MW) between the ATM and the Bank. This refinement also prepares the model to be decomposed using SED into three architectural components of the transfer feature, i.e., ATM, MW and the Bank, where MW is used for communicating between the two. The recomposition of these (ATM, MW, Bank) would refine the feature being decomposed (fourth refinement). Here we do not introduce any further requirements rather the focus is on how the model could be

```
EVENTS
Event ReqTransfer ≙
  any
    src_ac, dest_ac, am, c, a
  where
    grd1 : src_ac ∈ ACCOUNT
    grd2 : dest_ac ∈ ACCOUNT
    grd3 : c ∈ validCard
    grd5 : am ∈ ℕ1
    grd6 : src_ac ≠ dest_ac
    grd9 : c = cardInAtm(a)
    grd10 : a ∉ trans
  then
    act1 : reqTransfer := reqTransfer ∪ {a}
    act2 : trAmount(a) := am
    act3 : srcAcc(a) := src_ac
    act4 : destAcc(a) := dest_ac
    act5 : trans := trans ∪ {a}
  end
Event TransferOK ≙
refines TransferOK
  any
    src_ac, dest_ac, am, c, a
  where
    grd1 : src_ac ∈ ACCOUNT
    grd2 : dest_ac ∈ ACCOUNT
    grd4 : c ↦ src_ac ∈ cardAcct
    grd5 : am ∈ ℕ1
    grd8 : src_ac ≠ dest_ac
    grd9 : am ≤ bal(src_ac)
    grd12 : c = cardInAtm(a)
    grd14 : src_ac = srcAcc(a)
    grd15 : dest_ac = destAcc(a)
    grd16 : am = trAmount(a)
    grd17 : a ∈ reqTransfer
    grd18 : a ∉ (transferOk ∪ transferFail)
  then
    act1 : bal := bal ◁- {dest_ac ↦ (bal(dest_ac) +
           am), src_ac ↦ (bal(src_ac) − am)}
    act5 : transferOk := transferOk ∪ {a}
    act6 : reqTransfer := reqTransfer \ {a}
  end
Event RespTransferOk ≙
refines FinishTransfer
  any
    a
  where
    grd2 : a ∈ transferOk
    grd3 : a ∉ transferFail
  then
    act1 : respTransferOk := respTransferOk ∪ {a}
    act2 : transferOk := transferOk \ {a}
  end
Event TransferFails ≙
refines TransferFails
  any
    src_ac, dest_ac, am, c, a
  where
    grd1 : src_ac ∈ ACCOUNT
    grd2 : dest_ac ∈ ACCOUNT
    grd5 : am ∈ ℕ1
    grd8 : src_ac ≠ dest_ac
    grd9 : am > bal(src_ac)
    grd12 : c = cardInAtm(a)
    grd14 : src_ac = srcAcc(a)
    grd15 : dest_ac = destAcc(a)
    grd16 : am = trAmount(a)
    grd17 : a ∈ reqTransfer
    grd18 : a ∉ (transferOk ∪ transferFail)
  then
    act2 : transferFail := transferFail ∪ {a}
    act3 : reqTransfer := reqTransfer \ {a}
  end
Event RespTransferFail ≙
refines FinishTransfer
  any
    a
  where
    grd2 : a ∈ transferFail
    grd3 : a ∉ transferOk
  then
    act1 : respTransferFail := respTransferFail ∪ {a}
    act2 : transferFail := transferFail \ {a}
  end
Event TransferOKAtm ≙
  any
    a
  where
    grd8 : a ∈ respTransferOk
    grd9 : a ∉ respTransferFail
  then
    act1 : transOkAtm(a) := TRUE
    act5 : respTransferOk := respTransferOk \ {a}
    act6 : trans := trans \ {a}
  end
Event TransferFailsAtm ≙
  any
    a
  where
    grd8 : a ∈ respTransferFail
    grd9 : a ∉ respTransferOk
  then
    act1 : transOkAtm(a) := FALSE
    act5 : respTransferFail := respTransferFail \ {a}
    act6 : trans := trans \ {a}
  end
```

Figure 6.7: Transfer model's events of second refinement

designed towards implementation. Since we would like all features to have the same architectural design, this would enable us to later compose all components of a particular type (i.e., MW or Bank) resulting from the decomposition and refine these further.

In this refinement step, we added new variables to make sure that there are no more shared-variables when we decompose the model into three components and only have shared events. For example, in third refinement, we only had one variable for transfer amount (`trAmount`) whereas we introduced two extra variables (`trAmountB, trAmountM`) to hold transfer amount for each of the three components. This would be used to pass value between the components synchronised through shared events after SED as discussed below. Several new invariants were also added as a result of this. Some of the

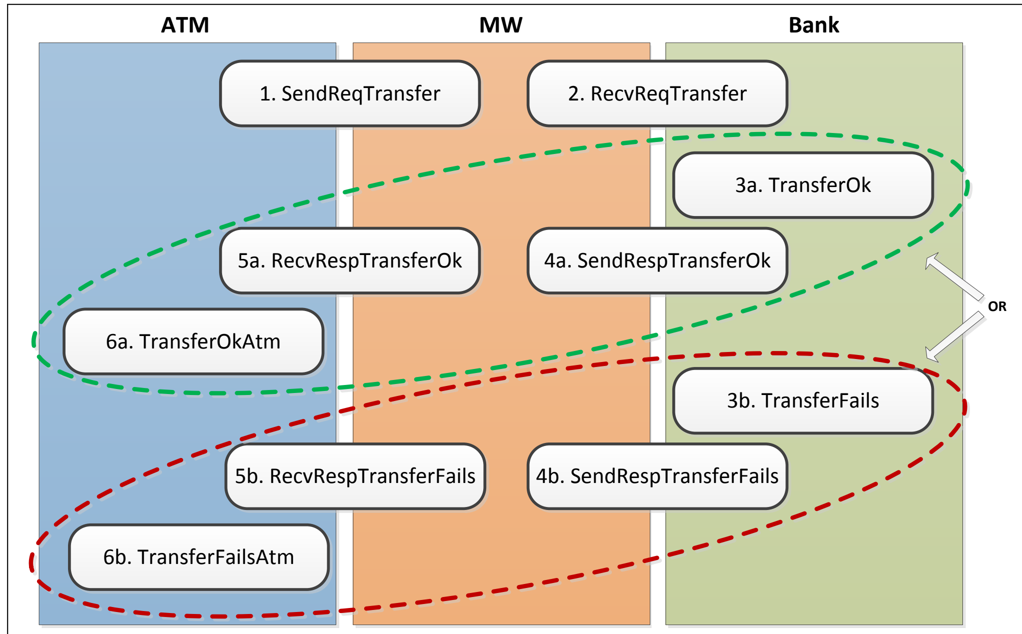


Figure 6.8: Transfer fourth refinement model with events ready for architectural decomposition

invariants at this refinement level are shown below. These invariants allow us to make the sate of three components disjoint by introducing new variables.

inv12 : $\forall a \cdot a \in dom(trAmountM) \Rightarrow a \in dom(trAmount) \land trAmountM(a) = trAmount(a)$

inv13 : $\forall a \cdot a \in recvReqTransfer \land a \in dom(trAmountB) \Rightarrow a \in dom(trAmountM) \land trAmountB(a) = trAmountM(a)$

inv14 : $\forall a \cdot a \in dom(srcAccM) \Rightarrow a \in dom(srcAcc) \land srcAccM(a) = srcAcc(a)$

inv15 : $\forall a \cdot a \in recvReqTransfer \land a \in dom(srcAccB) \Rightarrow a \in dom(srcAccM) \land srcAccB(a) = srcAccM(a)$

inv16 : $\forall a \cdot a \in dom(destAccM) \Rightarrow a \in dom(destAcc) \land destAccM(a) = destAcc(a)$

inv17 : $\forall a \cdot a \in recvReqTransfer \land a \in dom(destAccB) \Rightarrow a \in dom(destAccM) \land destAccB(a) = destAccM(a)$

inv18 : $\forall a \cdot a \in dom(cardInAtmM) \Rightarrow a \in dom(cardInAtm) \land cardInAtmM(a) = cardInAtm(a)$

inv19 : $\forall a \cdot a \in recvReqTransfer \land a \in dom(cardInAtmB) \Rightarrow a \in dom(cardInAtm) \land cardInAtmB(a) = cardInAtm(a)$

inv20 : $\forall a \cdot a \in dom(cardInAtmB) \Rightarrow a \in dom(cardInAtm)$

Figure 6.8 shows the architecture of fourth refinement model including all the events and the component to which these would belong to after decomposition. The events shared between any two components are split into two. The sequence of events is shown by numbering the events in ascending order. Only one of the events from $3a$ or $3b$ would take place for a particular transaction, followed by their corresponding events only (a or b).

The architectural decomposition is achieved using shared-event decomposition (SED). During the decomposition, the shared variable `validCard` is moved to the ATM component and the other shared variables `bal` and `cardAcct` into the Bank component. The external events were also partitioned accordingly, e.g., part of the external event *Deposit* containing variable `validCard` moved to ATM and the rest to the Bank component as shown in Figure 6.9. This splitting of external events is possible since we can leave out such external events (resulting from SVD) when composing models using SVC later. The only restriction of SVD is to not refine these events.

During architectural decomposition, we partitioned variables into different components whereas events were split between any two components. Hence, the components synchronise through these shared-events following the message passing mechanism of SED. For example, in Figure 6.8, the event *SendReqTransfer* is shared between ATM and the MW whereas the event *RecvReqTransfer* is shared between MW and the Bank. Figure 6.10 shows Event-B specification of *SendReqTransfer* event decomposed into two events for ATM and MW components. So, a balance transfer transaction starts when an ATM machine sends a transfer request (e.g., variable `trAmount` holds the transfer amount here) through the MW (e.g., `trAmountM` is passed the transfer amount) which is received by the Bank (e.g., `trAmountB` eventually contains the transfer amount). After processing the transfer request, the Bank then sends a response for a successful or failed transfer through the MW. The ATM finally displays the transfer status accordingly.

This decomposition was performed using the SED tool available as a plug-in for Rodin. We specify names of the components and the variables that each component should contain after the decomposition. This is why there must not be any shared variables among the components resulting from the decomposition. The tool then partitions the events based on the variables. Each of these three components can be further refined but the shared variables and external events (resulting from SVD) must not be refined. It is interesting to note that we have used SED on models that were decomposed using SVD having external events. Table 6.1 shows the proof obligations statistics for each of the transfer refinements and whether the POs were discharged automatically by the Rodin provers or interactively.

6.4.2 Refinement of Deposit Feature

Similar to the transfer feature, we refined and decomposed the deposit feature resulting in three components, i.e., ATM, MW and the Bank. In the first refinement, the abstract event *Deposit* was was refined further to introduce the cash available in an ATM and the card inserted in it as shown below:

Event *Deposit* $\widehat{=}$
// External event, DO NOT REFINE
extends *Deposit*
any
acc, am, c
where
grd1 : $\mathtt{acc} \in \mathtt{ACCOUNT}$
grd2 : $\mathtt{acc} \in \mathtt{dom(bal)}$
grd3 : $\mathtt{am} \in \mathbb{N}_1$
grd4 : $\mathtt{c} \in \mathtt{validCard}$
grd5 : $\mathtt{c} \mapsto \mathtt{acc} \in \mathtt{cardAcct}$
then
act1 : $\mathtt{bal(acc)} := \mathtt{bal(acc)} + \mathtt{am}$
end

Event *Deposit* $\widehat{=}$
// ATM part
// External event, DO NOT REFINE
any
acc, am, c
where
typing_c : $c \in CARD$
typing_am : $am \in \mathbb{Z}$
grd1 : $acc \in ACCOUNT$
grd3 : $am \in \mathbb{N}_1$
grd4 : $c \in validCard$
then
skip
end

Event *Deposit* $\widehat{=}$
// BANK part
// External event, DO NOT REFINE
any
acc, am, c
where
typing_c : $c \in CARD$
typing_am : $am \in \mathbb{Z}$
grd1 : $acc \in ACCOUNT$
grd2 : $acc \in dom(bal)$
grd3 : $am \in \mathbb{N}_1$
grd5 : $c \mapsto acc \in cardAcct$
then
act1 : $bal(acc) := bal(acc) + am$
end

Figure 6.9: Splitting external event ***Deposit*** into two during architectural decomposition using SED

Event *Deposit* $\widehat{=}$
refines *Deposit*
any
$\mathtt{acc}, \mathtt{am}, \mathtt{c}$
a
where
grd1 : $\mathtt{acc} \in \mathtt{ACCOUNT}$
grd2 : $\mathtt{acc} \in \mathtt{dom(bal)}$
grd3 : $\mathtt{am} \in \mathbb{N}_1$
grd4 : $\mathtt{c} \in \mathtt{validCard}$
grd5 : $\mathtt{c} \mapsto \mathtt{acc} \in \mathtt{cardAcct}$
grd6 : $a \in ATM$
grd7 : $a \in dom(atmCash)$
grd8 : $a \in dom(cardInAtm)$
grd9 : $cardInAtm(a) = c$
then
act1 : $\mathtt{bal(acc)} := \mathtt{bal(acc)} + \mathtt{am}$
act2 : $atmCash(a) := atmCash(a) + am$
end

In the second refinement, we introduced the request and response mechanism between the ATM and the Bank. Here, an ATM sends a deposit request to the bank (i.e., ***ReqDeposit*** event), which responds (i.e., ***RespDeposit*** event) after the deposit event takes place at the Bank (i.e., ***DepositB*** event), where the account balance associated with the card is incremented by the deposited amount. Upon successful deposit, the

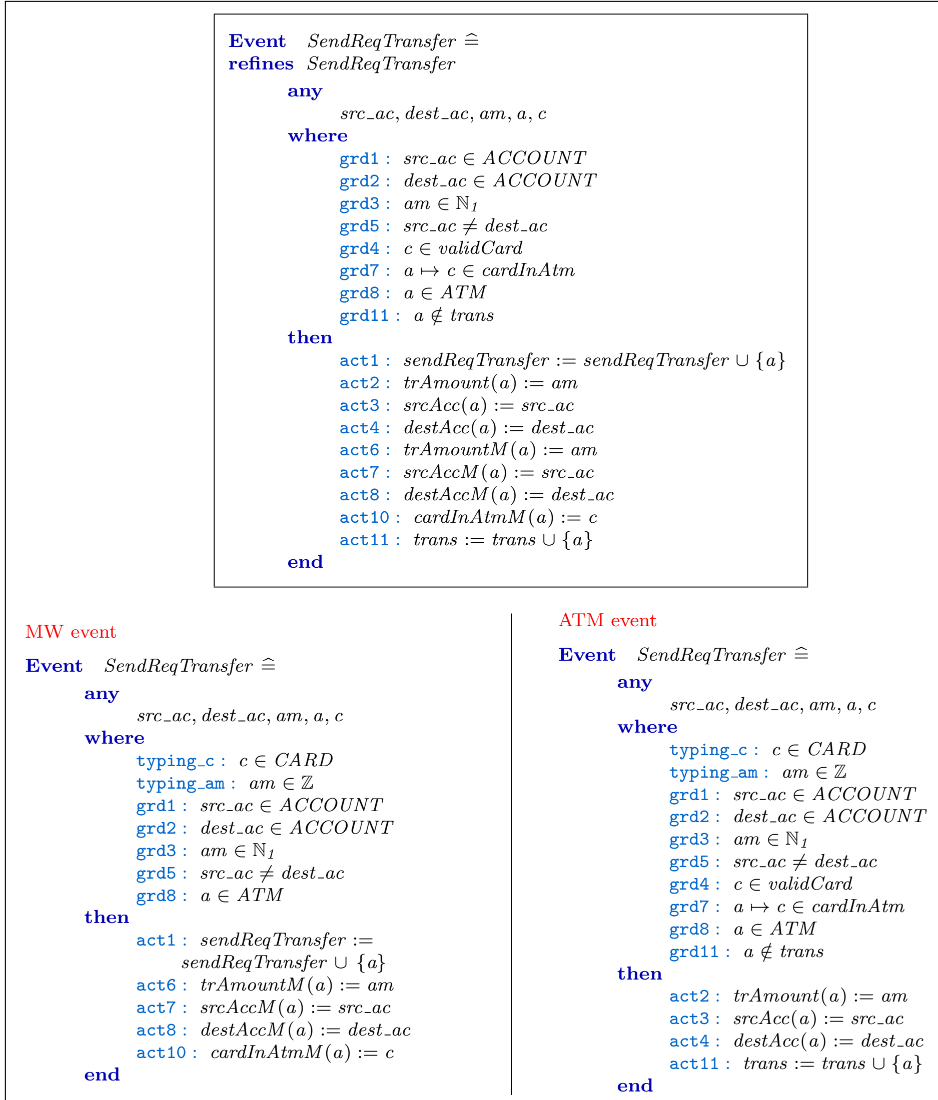
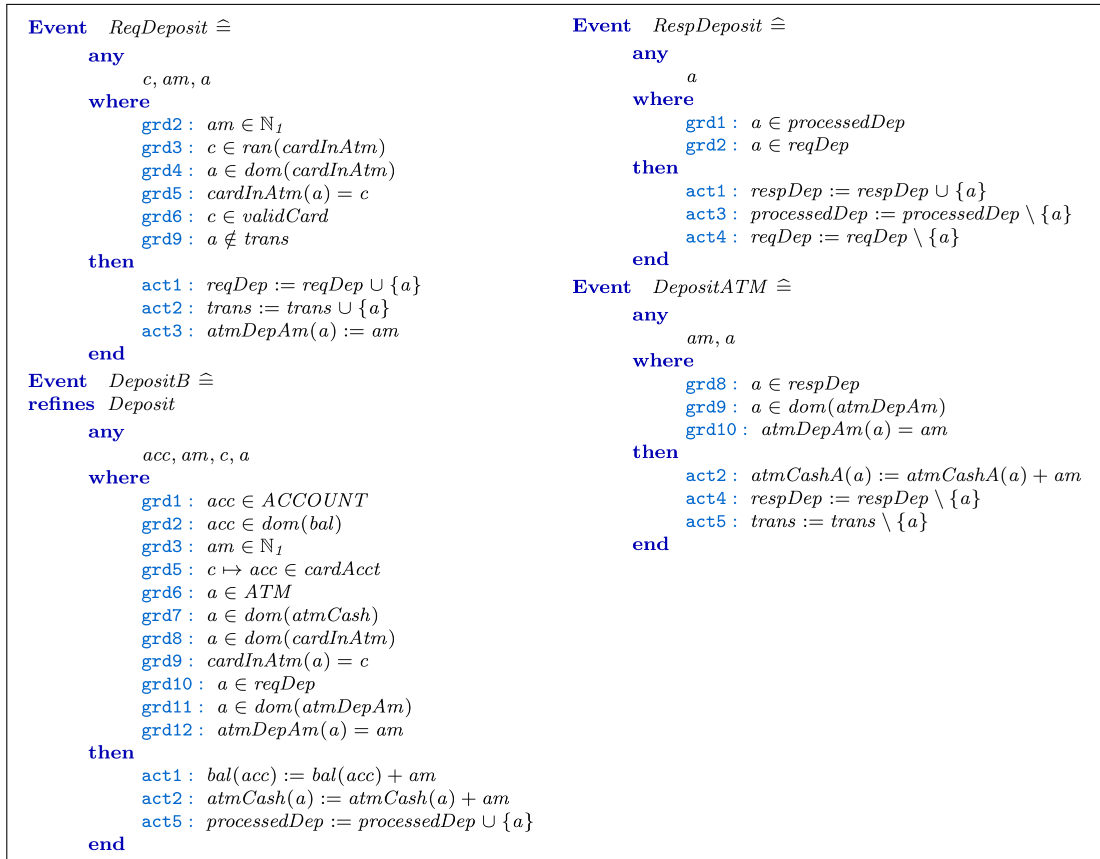


Figure 6.10: An event of transfer feature's fourth refinement model split into two events using SED

ATM displays deposit successful message and also increments the cash available in the ATM (i.e., `DepositATM` event). We also introduced transaction mechanism through the variable `trans` as we did for the Transfer feature's refinement. Figure 6.11 shows the events at this refinement level.

Again, following from the transfer feature refinement, the third level of refinement further refined the request and response mechanisms by partitioning the request event for sending and receiving the request and similarly for the response event. The fourth refinement introduced MW between ATM and the Bank. The proof obligations statistics for this development is given in Table 6.2 and Figure 6.12 shows the events (both new and refined) of deposit feature during all the refinements.

We then decomposed this deposit feature into ATM, MW and the Bank using SED. The shared variables and external events were partitioned in the same way as done in the



```
Event ReqDeposit ≙
    any
        c, am, a
    where
        grd2 : am ∈ ℕ1
        grd3 : c ∈ ran(cardInAtm)
        grd4 : a ∈ dom(cardInAtm)
        grd5 : cardInAtm(a) = c
        grd6 : c ∈ validCard
        grd9 : a ∉ trans
    then
        act1 : reqDep := reqDep ∪ {a}
        act2 : trans := trans ∪ {a}
        act3 : atmDepAm(a) := am
    end
Event DepositB ≙
refines Deposit
    any
        acc, am, c, a
    where
        grd1 : acc ∈ ACCOUNT
        grd2 : acc ∈ dom(bal)
        grd3 : am ∈ ℕ1
        grd5 : c ↦ acc ∈ cardAcct
        grd6 : a ∈ ATM
        grd7 : a ∈ dom(atmCash)
        grd8 : a ∈ dom(cardInAtm)
        grd9 : cardInAtm(a) = c
        grd10 : a ∈ reqDep
        grd11 : a ∈ dom(atmDepAm)
        grd12 : atmDepAm(a) = am
    then
        act1 : bal(acc) := bal(acc) + am
        act2 : atmCash(a) := atmCash(a) + am
        act5 : processedDep := processedDep ∪ {a}
    end
Event RespDeposit ≙
    any
        a
    where
        grd1 : a ∈ processedDep
        grd2 : a ∈ reqDep
    then
        act1 : respDep := respDep ∪ {a}
        act3 : processedDep := processedDep \ {a}
        act4 : reqDep := reqDep \ {a}
    end
Event DepositATM ≙
    any
        am, a
    where
        grd8 : a ∈ respDep
        grd9 : a ∈ dom(atmDepAm)
        grd10 : atmDepAm(a) = am
    then
        act2 : atmCashA(a) := atmCashA(a) + am
        act4 : respDep := respDep \ {a}
        act5 : trans := trans \ {a}
    end
```

Figure 6.11: Deposit feature's events of second refinement

Table 6.2: Proof statistics for deposit refinements

Model	Auto	Manual	Total
Deposit_0	7	0	7
Deposit_1	7	0	7
Deposit_2	27	0	27
Deposit_3	37	0	37
Deposit_4	46	3	49

transfer feature. So, the shared variable `validCard` moved to the ATM component and rest of the shared variables (i.e., `bal` and `cardAcct`) moved to the Bank component. The external events were also partitioned accordingly. The vertical refinement architecture of deposit feature is same as that of the balance transfer feature discussed earlier.

Figure 6.13 shows the architecture of fourth refinement model including all the events and which component these would belong to after decomposition. It also shows the sequence of events. So, for the deposit feature, an ATM sends a deposit request through the MW which is received by the bank. The bank then sends a response after incrementing the account balance through the middleware. The ATM finally displays the deposit successful message and also increments the amount of cash it contains by the deposited

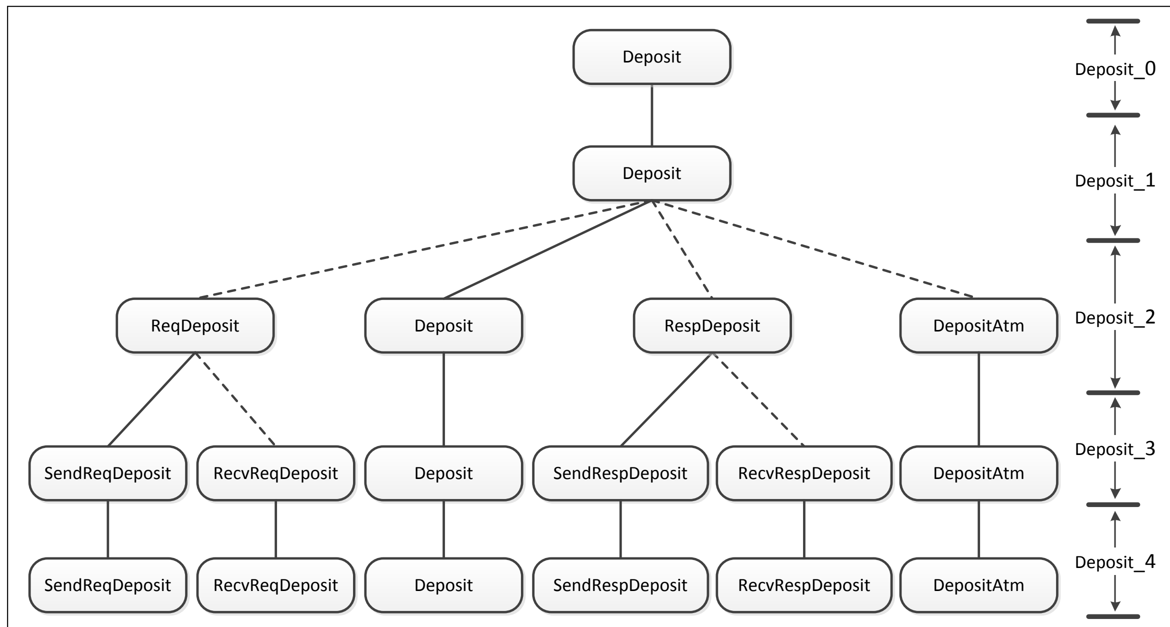


Figure 6.12: Event refinement of deposit feature

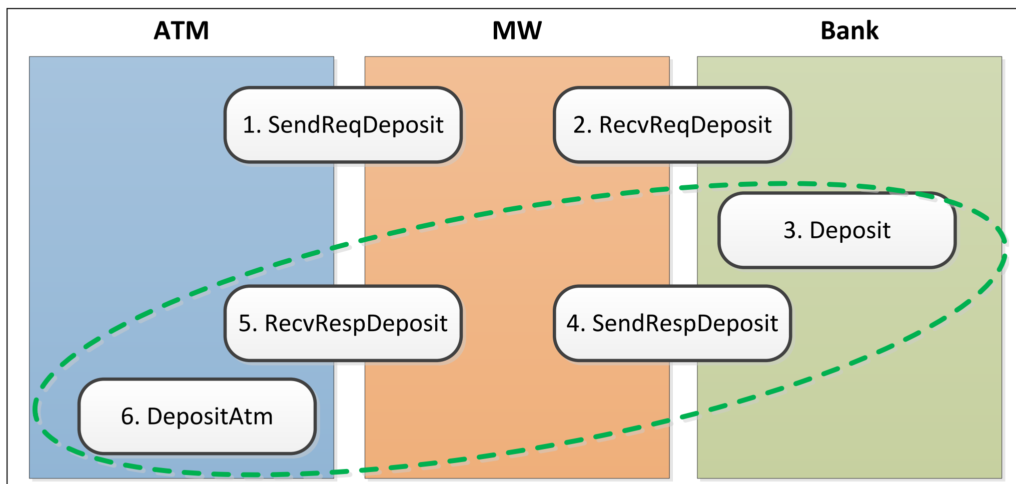


Figure 6.13: Deposit feature's fourth refinement model with events ready for architectural decomposition

amount. These decomposed components synchronise using the shared-events and can be further refined independently.

6.4.3 Composing Sub-components of Two Features

Figure 6.14 shows the development and (de)composition structure for the deposit and transfer features of the ATM. In the figure, asterisk (*) denotes a model with external events, and `bal`, `validCard` and `cardAcct` are the model's shared variables. We needed external events in order to later compose these features using SVC after several refinement steps, to make sure the composition was correct by construction and hence

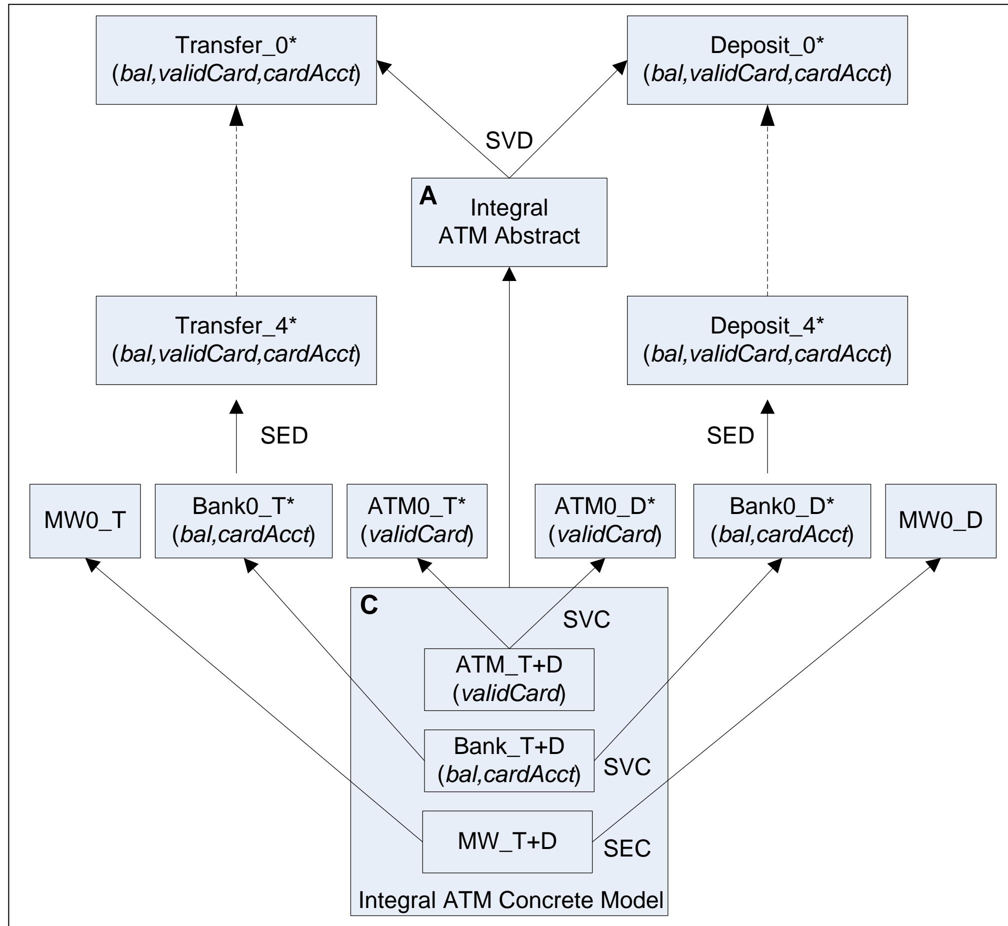


Figure 6.14: Refinement and (de)composition architecture for transfer and deposit features

not need reproving. This means that we had to make sure that the two features were a result of SVD of an integral abstract model, as mentioned earlier. This generated a tooling requirement to automatically generate external events for the features that we would like to compose using SVC and hence saving time to manually do so. Note that in this case study, the shared variables `bal` and `cardAcct` along with their corresponding external events are localised in the Bank component; whereas the shared variable `validCard` and its corresponding external event is localised in the ATM component.

Now that we have the same architectural decomposition (ATM, MW, Bank) for each feature, we would like to compose these models pairwise (i.e., Bank_T+D = Bank_T + Bank_D, etc.) for implementation purposes. In general, the task would of course be more complex, involving more than two features. In our case, where the shared variables `bal` and `cardAcct` are localised into the two architectural Bank components, these can be composed, with the composite Bank refining each Bank component. This is because each Bank's external events are 'cancelled out', or implemented, by the other Bank's actual events. For example, components Bank0_T and ATM0_T of transfer feature have parts of the external event ***Deposit*** as shown below:

Event *Deposit* $\widehat{=}$
// ATM0_T
// External event, DO NOT REFINE
any
acc, am, c
where
typing_c : $c \in CARD$
typing_am : $am \in \mathbb{Z}$
grd1 : $acc \in ACCOUNT$
grd3 : $am \in \mathbb{N}_1$
grd4 : $c \in validCard$
then
skip
end

Event *Deposit* $\widehat{=}$
// BANK0_T
// External event, DO NOT REFINE
any
acc, am, c
where
typing_c : $c \in CARD$
typing_am : $am \in \mathbb{Z}$
grd1 : $acc \in ACCOUNT$
grd2 : $acc \in dom(bal)$
grd3 : $am \in \mathbb{N}_1$
grd5 : $c \mapsto acc \in cardAcct$
then
act1 : $bal(acc) := bal(acc) + am$
end

These external events have been refined by events ***sendReqDeposit*** and ***depositB*** respectively, present in ATM0_D and Bank0_D of deposit feature as shown below:

Event *sendReqDeposit* $\widehat{=}$
any
c, am, a
where
typing_c : $c \in CARD$
typing_am : $am \in \mathbb{Z}$
typing_a : $a \in ATM$
grd2 : $am \in \mathbb{N}_1$
grd3 : $c \in ran(cardInAtm)$
grd4 : $a \in dom(cardInAtm)$
grd5 : $cardInAtm(a) = c$
grd6 : $c \in validCard$
grd9 : $a \notin trans$
then
act2 : $trans := trans \cup \{a\}$
act3 : $atmDepAm(a) := am$
end

Event *depositB* $\widehat{=}$
any
acc, am, a, c
where
typing_c : $c \in CARD$
typing_am : $am \in \mathbb{Z}$
grd1 : $acc \in ACCOUNT$
grd2 : $acc \in dom(bal)$
grd3 : $am \in \mathbb{N}_1$
grd5 : $c \mapsto acc \in cardAcct$
grd6 : $a \in ATM$
grd7 : $a \in dom(atmCash)$
grd8 : $a \in dom(cardInAtmB)$
grd9 : $cardInAtmB(a) = c$
grd11 : $a \in dom(atmDepAmB)$
grd12 : $atmDepAmB(a) = am$
grd13 : $a \in recvReqDep$
then
act1 : $bal(acc) := bal(acc) + am$
act2 : $atmCash(a) := atmCash(a) + am$
act5 : $processedDep := processedDep \cup \{a\}$
end

Figure 6.15 shows the events of Bank_T and Bank_D and the resulting composition Bank_T+D, where external events disappear as a consequence of shared-variable composition. The composition of different components is achieved using our feature composition tool, as there is no tool support for shared-variable composition. Since we had same architecture for both the components, there were conflicting variables and invariants which were resolved by the tool automatically. Table 6.3 shows the number of proof

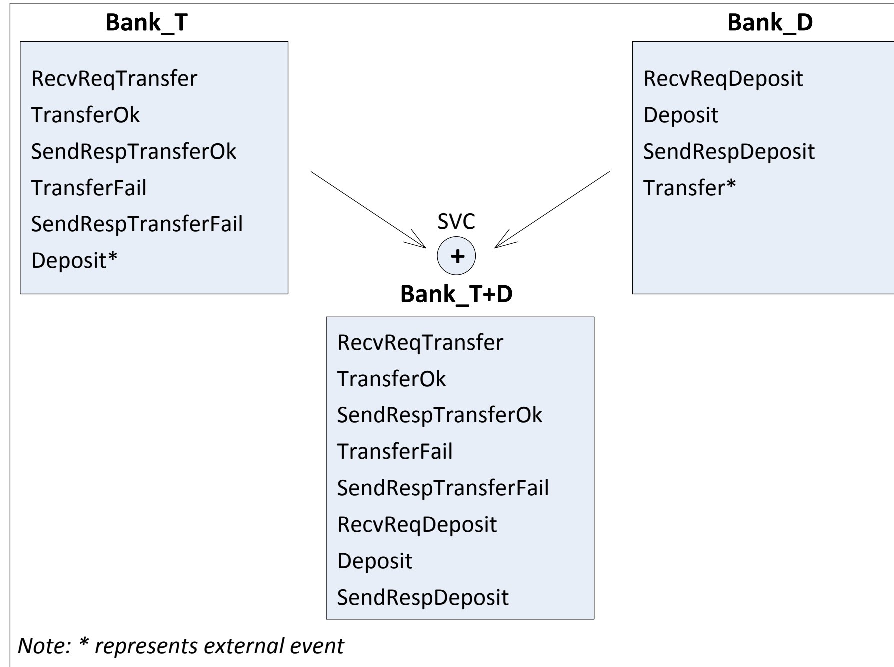


Figure 6.15: Composing bank components of transfer and deposit features

Table 6.3: Proof statistics for deposit and transfer components and their composition

Component	**Transfer** T	**Deposit** D	**Composite** T+D
ATM	11	10	20
MW	17	7	25
Bank	32	18	47
Total	60	35	92

obligations for different components of transfer and deposit features and their composites. This shows that by following the suggested modelling pattern, we can avoid the proof effort for the composite models (i.e., fourth column of the table).

The suggested pattern is correct by construction due to the application of shared-variable and shared-event (de)composition techniques and following their restrictions. In order to prove that the pattern works for this example, we composed the three components (i.e., ATM_T+D, MW_T+D, Bank_T+D) using shared-event composition, though this is not required in practice. The composition resulted in the concrete integral model of ATM (box C in Figure 6.14) which refines the abstract integral model (box A in the figure). This refinement relationship was also proved by refining the integral ATM abstract model up to four refinements and the composition (i.e., ATM concrete model) was proved to refine the fourth refinement of integral ATM model, as shown in Figure 6.16. Here, in order to prove that box C refines box A, we had to model refinements of box B although this is not required in actual development.

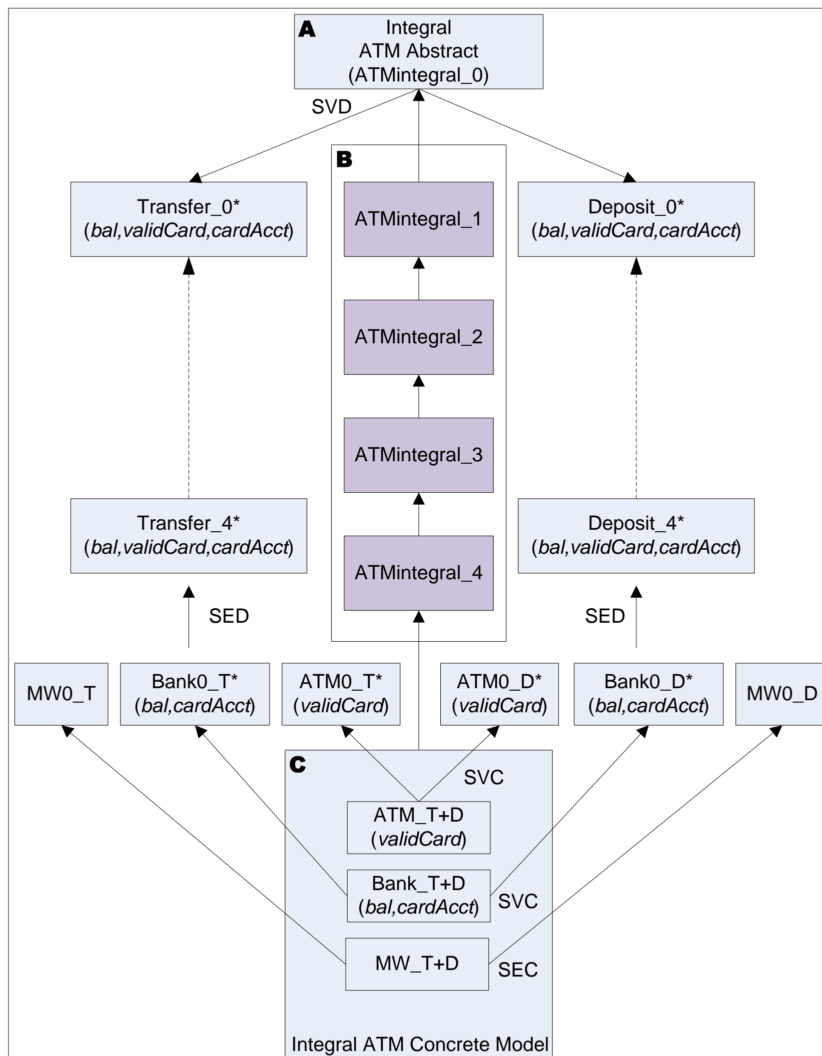


Figure 6.16: Refinement preservation experiment of the suggested pattern

6.4.4 Modelling Second ATM Product By Reusing Existing Features

After modelling an ATM product with two features, we modelled another ATM product having a cash withdraw feature as well (as shown by the dotted box on the right in Figure 6.17). This enabled us to explore how much of the existing models and their proofs could be reused or to what extent we could reduce the overall modelling effort for the second ATM product through reuse after modelling the first one. So, we elaborated the top-level integral model to include the cash withdrawal functionality and decomposed it into three features (i.e., deposit, transfer and withdraw). Figure 6.18 shows the abstract integral Event-B model for the second ATM product.

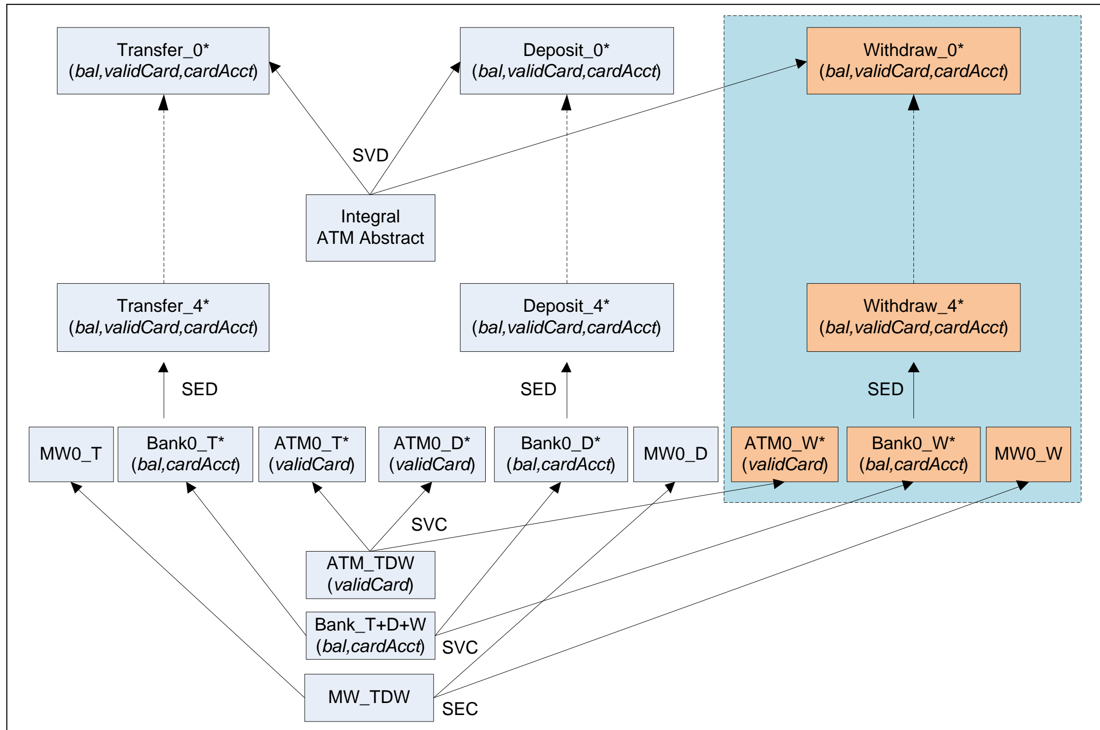


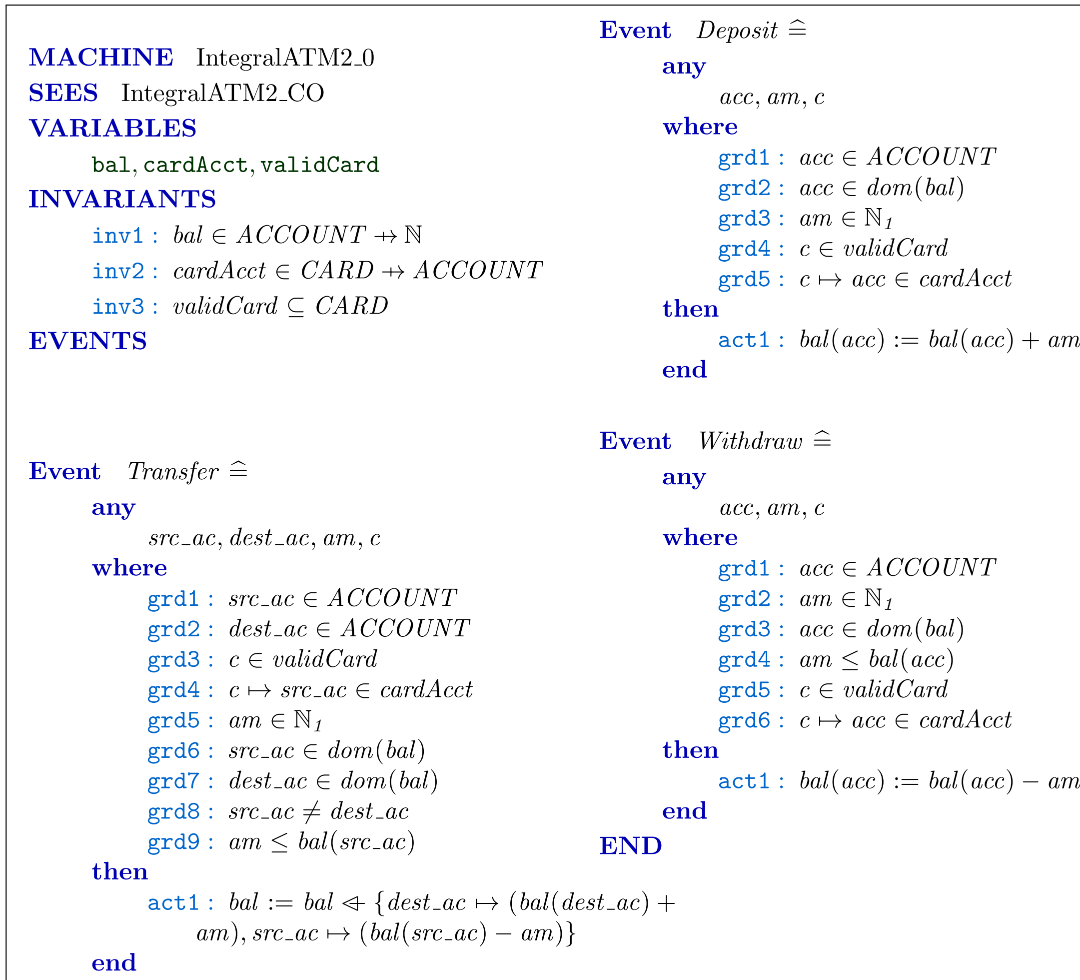
Figure 6.17: Refinement and (de)composition architecture for 2nd ATM product

6.4.4.1 Refinement of Withdraw Feature

We refined the withdraw feature in the same way as the other two features discussed earlier. The first refinement differentiates between the successful and failed withdrawal. The next two refinements introduce the request and response mechanism between Bank and the ATM. The fourth refinement introduces MW so that we can decompose this model into three components (i.e., ATM, MW and Bank), as we did for the other two features. Figure 6.19 shows the events (both new and refined) of withdraw feature during all the refinements. Note that the number of refinement steps for any of the features could be different. This solves the problem of composing non-conformal refinements as discussed in the production cell case-study.

6.4.4.2 Discussion

Provided the new feature is 'non-interfering' - in the sense that in the SVD refinement, the other two features remain unchanged - then all we have to do is refine the withdraw feature only. An interfering feature would require making changes to the existing features which would lead to feature interaction problem mentioned in Chapter 3. If the additional feature only interferes with subset of existing features, then we would only need to re-engineer that subset. This could also be very useful in saving modelling and proof effort when we can reuse a large set of existing features by modifying a small subset of these due to feature interaction of additional feature to model a new product.



MACHINE IntegralATM2_0
SEES IntegralATM2_CO
VARIABLES
bal, cardAcct, validCard
INVARIANTS
inv1 : $bal \in ACCOUNT \rightarrow\!\!\!\!\!\!+ \mathbb{N}$
inv2 : $cardAcct \in CARD \rightarrow\!\!\!\!\!\!+ ACCOUNT$
inv3 : $validCard \subseteq CARD$
EVENTS

Event *Transfer* $\widehat{=}$
any
$src_ac, dest_ac, am, c$
where
grd1 : $src_ac \in ACCOUNT$
grd2 : $dest_ac \in ACCOUNT$
grd3 : $c \in validCard$
grd4 : $c \mapsto src_ac \in cardAcct$
grd5 : $am \in \mathbb{N}_1$
grd6 : $src_ac \in dom(bal)$
grd7 : $dest_ac \in dom(bal)$
grd8 : $src_ac \neq dest_ac$
grd9 : $am \leq bal(src_ac)$
then
act1 : $bal := bal \lhd\!\!\!- \{dest_ac \mapsto (bal(dest_ac) + am), src_ac \mapsto (bal(src_ac) - am)\}$
end

Event *Deposit* $\widehat{=}$
any
acc, am, c
where
grd1 : $acc \in ACCOUNT$
grd2 : $acc \in dom(bal)$
grd3 : $am \in \mathbb{N}_1$
grd4 : $c \in validCard$
grd5 : $c \mapsto acc \in cardAcct$
then
act1 : $bal(acc) := bal(acc) + am$
end

Event *Withdraw* $\widehat{=}$
any
acc, am, c
where
grd1 : $acc \in ACCOUNT$
grd2 : $am \in \mathbb{N}_1$
grd3 : $acc \in dom(bal)$
grd4 : $am \leq bal(acc)$
grd5 : $c \in validCard$
grd6 : $c \mapsto acc \in cardAcct$
then
act1 : $bal(acc) := bal(acc) - am$
end
END

Figure 6.18: Integral ATM abstract model for 2nd ATM product of Figure 6.17

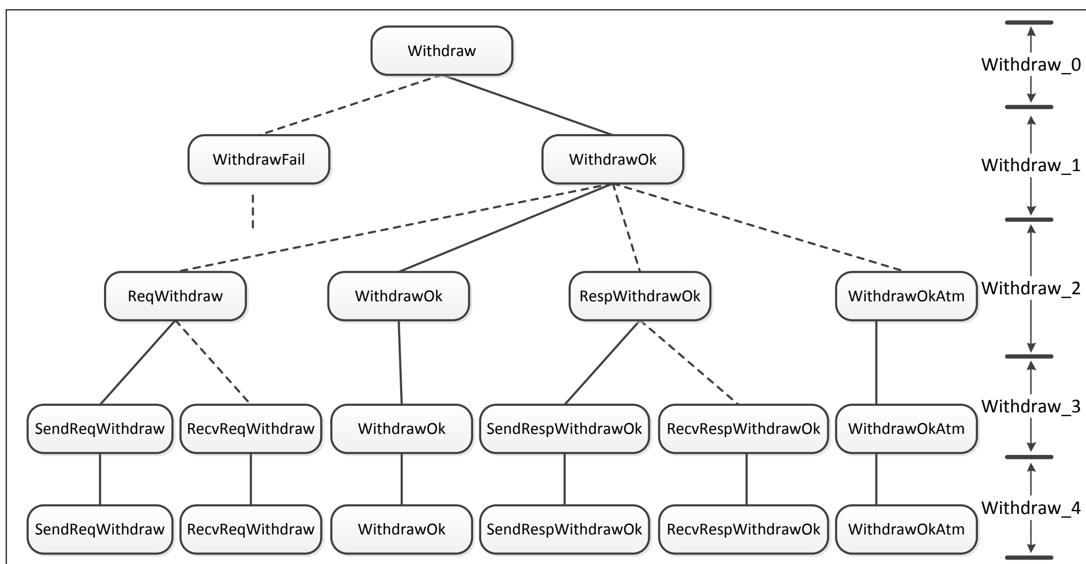


Figure 6.19: Event refinement of withdraw feature

Table 6.4: Proof obligations for withdraw component and the composite models

Component	**Withdraw** W	**Composite** T+D+W
ATM	13	29
MW	9	33
Bank	28	61
Total	50	123

The additional feature must also preserve any invariants related to the shared-variables in the existing features. So, this means that both the deposit and transfer features would now contain external event ***Withdraw*** of the withdraw feature. Since the deposit and transfer components have already been proved, new POs were only generated for the newly added external event acting on the shared variables (i.e., `bal, cardAcct` and `validCard`). Also, these new POs were only generated in the abstract models of deposit and transfer features, no matter how many refinements exist because of the restriction of SVD that external events and shared variables should not be refined. Hence, we only have to discharge these small number of POs when reusing existing models. In this particular example, only three new POs were generated for each of the deposit and transfer abstract models and these were automatically discharged by the Rodin provers. Table 6.4 shows the number of POs for different components of withdraw feature before and after composition with the components of other two features. This shows that we managed to avoid reproving POs for whole developments of deposit and transfer features by following this pattern of reuse (Tables 6.1 and 6.2 show total savings of 375 POs for second ATM product). With the help of proper tool support, we could even avoid discharging POs at abstract models for the components being reused.

6.4.5 Evaluation

We have examined a specific pattern of mixed decomposition-recomposition - SVD followed by SED and then SVC/SEC in a single development. It is possible to do this provided shared variables and their associated external events are not refined and the shared variables are localised in exactly the same component resulting from architectural decomposition of various features. Our case-study example supports this claim where we proved that this reuse pattern preserves the refinement relation provided the restrictions of the pattern stated above are followed.

We can generalise this pattern where the shared variables and their associated external events must be localised in exactly the same component in each of the feature developments. For example, consider two features P and Q with shared-variables x and y and external event(s); which appear to result from SVD of model M as shown in Figure 6.20. After several refinement steps of P, we get Pn where new events and variables (e.g.,

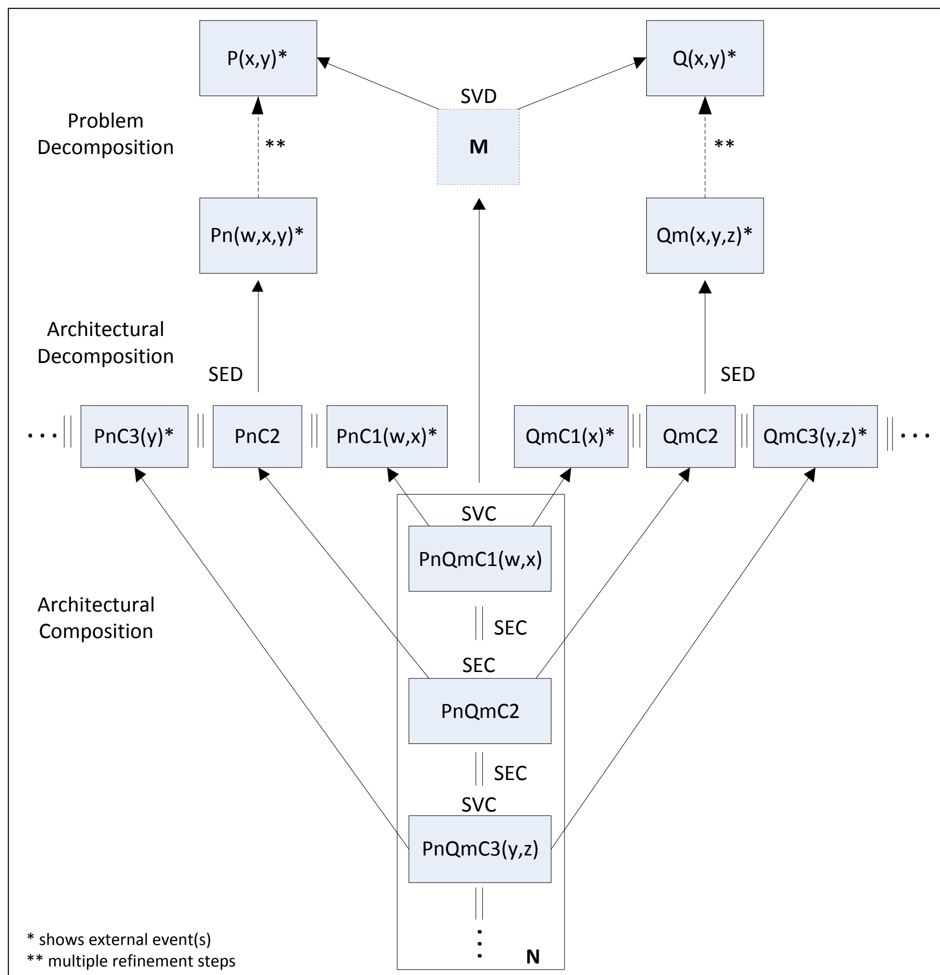


Figure 6.20: Feature-oriented refinement and (de)composition pattern

variable w) are introduced. This model is further decomposed using SED into three components $C1...C3$ where variables w and x are localised into $C1$ and y into $C3$. This pattern requires that the variables x and y must be localised into components $C1$ and $C3$ respectively for the feature Qm (which results from various refinements of Q). This could then scale up to any number of features (as required in product line modelling) without the need for reproving already proved features, and their composition - as suggested earlier - would be correct by construction. These components could be further refined, provided that the restrictions of the two styles of decomposition are observed.

As shown in the figure, we have to use SVC while composing $PnC1(w, x)$ and $QmC1(x)$ as these two components have external events and shared-variable x, and the same for $PnC3(y)$ and $QmC3(y, z)$. Here, we can leave out the external events during SVC since these are cancelled-out by their counter-parts, e.g., external event of PnC1 is cancelled out by an event of QmC1. We can compose $PnC2$ and $QmC2$ using SEC since both components are disjoint, and so on for the rest of the components.

6.5 Conclusion

We have modelled and refined features of ATM product line using Event-B with a view of reusing these features when modelling a second ATM product after the first one. We explored the use of both types of decomposition/composition techniques of Event-B in a single development (i.e., shared-event (SED/SEC) and shared variable (SVD/SVC) (de)composition). We used SVD for problem decomposition earlier in the development where we decomposed the problem into various requirements features. These features, after refining separately, were decomposed again into architectural components using SED. This serves as solution decomposition. These architectural components could then be refined independently as required and could be composed using SVC/SEC for implementation purposes.

This resulted in a modelling pattern which preserves refinement, provided that the restrictions of both styles of decomposition and that of the suggested pattern are observed. These include that the shared-variables and their associated external events are not refined as a consequence of SVD; and are localised in exactly the same type of architectural component during solution decomposition of all the features. Although the pattern is correct by construction due to the application of the two styles of (de)composition (i.e., SVD/SED), this refinement preservation was also verified by manually refining the integral development and proving that the concrete model resulting from this decomposition/recomposition pattern refines the abstract model, which was decomposed using SVD in the beginning.

The first ATM product modelled contained two features, i.e., deposit and transfer. We then modelled another ATM product which contained withdraw feature as well. The second product reused already developed features and hence saved modelling effort. The amount of proof obligations to be discharged for existing features was minimal which could be completely avoided with the help of proper tool support. So, this case-study proved quite useful in terms of suggesting a modelling pattern which employs existing techniques of Event-B. This pattern helps in reusing Event-B developments and their associated proofs as required in product line modelling. Based on our two case-studies, we suggest guidelines for Event-B users in the next chapter, to use feature-oriented reuse approach in Event-B for modelling product lines and also highlight the tooling requirements that need to be implemented to make full use of this reuse approach.