Under consideration for publication in Formal Aspects of Computing

An Incremental Development of the Mondex System in Event-B

Michael Butler and Divakar Yadav

School of Electronics and Computer Science
University of Southampton

Abstract. A development of the Mondex system was undertaken using Event-B and its associated proof tools. An incremental approach was used whereby the refinement between the abstract specification of the system and its detailed design was verified through a series of refinements. The consequence of this incremental approach was that we achieved a very high degree of automatic proof. The essential features of our development are outlined. We also present some modelling and proof guidelines that we found helped us gain a deep understanding of the system and achieve the high degree of automatic proof.

Keywords: Event-B; system design; refinement; mechanical proof; methodological guidelines

1. Introduction

We undertook an Event-B development of the Mondex system using the B4free [5] and Click'n'Prove [2] tools for support. B4free is a proof obligation generator and proof tool for B. Click'n'Prove provides a sophisticated front-end to B4free that increases productivity considerably when doing interactive proofs. The 'classical' B-Method developed by Abrial [1] is geared towards the development of correct non-concurrent programs. Event-B [3] is an evolution of B that is geared towards understanding and reasoning about systems (as opposed to programs) including reactive and concurrent systems. Event-B is influenced by the action system approach of Back and others [4].

A major goal for us in tackling the Mondex system with Event-B was to achieve a high degree of automatic proof. Thus a key feature of our development was the use of many levels of refinement in order to factor the proof effort into small, easily manageable steps. Our complete development consists of an abstract specification and a further nine levels of refinement. Through careful use of small refinement steps and appropriate intermediate abstractions, we were able to achieve an impressive degree of automatic proof using B4free. The full development resulted in 679 proof obligations. Of these, 662 were proved completely automatically by B4free. That is over 97% were proved automatically. The remaining 17 were proved interactively using B4free. This refinement approach together with the B4free tool supports an incremental style

Correspondence and offprint requests to: Michael Butler, School of Electronics and Computer Science, University of Southampton, Highfield, Southampton SO17 1BJ, UK email mjb@ecs.soton.ac.uk

of system development. Through successive refinements we incrementally add more features to the models and we incrementally achieve a proof that our abstract model is refined by the final detailed system model.

When presenting a completed refinement chain it is natural to present it from top to bottom. But it is important to emphasis that our refinement chain was not constructed in a top-down manner. We started with the highest level specification and then produced a model approximating the lowest level in the completed chain. However in attempting to prove refinement between these models it was clear that the abstraction gap was too large would have required a complex gluing invariant. Instead we decided that some intermediate abstraction was required. Having made proof progress with that intermediate model, it was observed that further intermediate models were required to ease the completion of the chain. Furthermore, as our understanding of the system deepened through this modelling and reasoning, situations arose whereby it was necessary to modify models higher up the chain. Any modifications to the refinement chain had an impact on the existing proofs. Luckily, the high degree of automatic proof, together with the proof re-run facilities of tool, made re-establishing the correctness of the chain relatively painless.

The development took approximately two weeks of total effort spread over several months. Most of the effort was in constructing models at different levels of abstraction and in constructing appropriate gluing invariants. We did not attempt to use the invariants from the existing Z development. Instead we made very heavy use of the B4free interactive prover to guide the construction of gluing invariants. That is, by attempting an interactive proof of an unproved proof obligation and using the built-in simplifiers, one can usually identify an invariant property that would help to discharge the unproved proof obligation. By adding that invariant to the model in a form that closely matches the proof obligation, one usually finds that the proof obligation is subsequently discharged automatically. Most of the interactive proof effort was used to discover invariants rather than to discharge proof obligations.

Rather than trying to translate the existing Z development [10] into Event-B, we essentially treated the Z development as a requirements document. Nonetheless our abstract model captures the same security property as the abstract Z specification (no money is created) and our most refined model captures all the behaviour of the protocol, including error and error recovery behaviour.

Our abstract specification includes events modelling atomic transfer of value between purses, loss of value from a purse, recovery of lost value and checking of purse balances. The security property is represented by not allowing money to be created only transferred or recovered. The nature of the refinement that we verified using B4free is safety refinement, that is, any behaviour (trace of events) of a refined model must be a behaviour of the abstract model. Thus, since a behaviour which results in the creation of money is prevented in the abstract model, it is also prevented in a correctly refined model. Note that we have not verified that liveness is preserved by our refinement, that is we have not verified the absence of divergence and we have not verified that our refinements preserve the enabledness of abstract events. Indeed some behaviours that are possible in the abstract model are not possible in the most detailed refinement. For example, in the abstract model it is possible for a purse to be involved in several transactions simultaneously whereas this is not possible in our detailed refinement (similar to the Z development).

Our refinements break the atomicity of the transfer into several steps following the definition of the Mondex protocol. At any stage a purse can abort a transaction. Where appropriate, aborted transactions are logged in a purse archive. If the logs of a transaction from both purses can be reconciled, then the lost value can be recovered. Our refinement includes explicit message transfer between purses. Our model of messaging allows for replay attempts by an intruder. Through refinement we verify that replay attacks are prevented by the unique sequence number each purse gives to a transaction.

We proceed by describing the abstract specifications in Section 2. In Section 3 we describe the first refinement step in which the atomicity of transfer is broken. In Section 4 we outline the main features of the remainder of the chain. The remaining sections cover specific issues such as discovering invariants, discovering modelling errors and treating other features of the models. We conclude by outlining some general modelling guidelines that we found useful in this work.

2. Abstract Specification of Transfer, Loss and Recovery

Our abstract B model of the purse system introduces a type $PURSE$ to distinguish purses. The specification consists of just three state variables, $purse$, $abal$ and $lost$, typed as follows:

$purse \subseteq PURSE$

$TransferOK =$
ANY $p1, p2, a$ WHERE
$p1 \in purse$
$p2 \in purse$
$p1 \neq p2$
$abal(p1) \geq a$
THEN
$abal(p1) := abal(p1) - a \ \|$
$abal(p2) := abal(p2) + a$
END

$TransferFail =$
ANY $p1, a$ WHERE
$p1 \in purse$
$abal(p1) \geq a$
THEN
$abal(p1) := abal(p1) - a \ \|$
$lost(p1) := lost(p1) + a$
END

$Recover =$
ANY $p1, a$ WHERE
$p1 \in purse$
$lost(p1) \geq a$
THEN
$lost(p1) := lost(p1) - a \ \|$
$abal(p1) := abal(p1) + a$
END

Fig. 1. Abstract value changing events

$$abal \in purse \rightarrow \mathbb{N}$$
$$lost \in purse \rightarrow \mathbb{N}$$

The variable *purse* represents the set of purses that currently exist in the system. It is a variable since purses can be created and destroyed. The monetary balance on each purse is represented by the variable *abal*, a total function from *purse* to naturals. The third variable *lost* is a very abstract representation of money that has been lost to a purse as a result of some previous failed transactions. It is included at the abstract level in order to be able to model the recovery mechanism whereby money lost to a purse in a transaction may be recovered at a later stage.

Three significant events of the abstract model are given in Fig. 1. *TransferOK* represents the successful transfer of an amount a from purse $p1$ to purse $p2$. The guard of *TransferOK* states that provided $p1$ and $p2$ are different valid purses and there are sufficient funds, then the amount is transferred in an atomic step. The *TransferFail* event models the loss of money as a result of transaction failure. The cause of transaction failure is made more specific in the refined layers. The *TransferOK* and *TransferFail* events should be viewed together are modelling the possible outcome of a transaction which either succeeds or fails. The third event in Fig. 1, *Recover*, is an abstract model of the mechanism whereby money previously lost to $p1$ is recovered by $p1$.

3. Breaking the Atomicity of Transfer

In the abstract model, we saw that value is transferred between two purses in a single atomic step. This provides for a clear and simple specification of the essence of the protocol. However, in the real protocol, the transfer of value is not atomic. Instead, the money is first deducted from the source account and then at a separate later stage may be added to the target account. The system consist of many purses engaged in value transfer so multiple parallel protocol runs will be interleaved and many events may occur in between deduction and addition of value.

The main stages of the protocol are outlined diagrammatically in Fig. 2. A protocol runs starts when two purses come together in a terminal device. The terminal device will send a *start* message to both purses. After that, the target purse sends a request message *req* to the source account. On receipt of the *req* message, the source purse $p1$ deducts the amount from its balance and sends a *val* message to the target account. The money is now in transit. At some stage later, the *val* message may be received by the target purse $p2$ which proceeds to add the amount to its balance and send an acknowledgement to $p1$. Receipt of a *val* message by the target purse represents successful completion of a balance transfer. For purse $p2$ the protocol run ends after it sends the acknowledgement while for purse $p1$ the protocol run ends after it receives the acknowledgement. A protocol run may fail when it is aborted by either side. An abort might be caused by a timeout or by a customer removing their purse from a terminal device. The diagram in Fig. 2 also includes the states that a purse may be in during a protocol run. For example, the source purse is in the state *epr* (*expecting request*) in between receipt of a *start* message and receipt of a *req* message. The other significant states are *epv* (*expecting value*) and (*expecting acknwledgement*).

Our view was that the abstraction gap between our abstract specification and concrete model involving all the details of the protocol events and states is too large for the refinement to be proved easily. Rather than attempting to prove such a refinement, we instead introduced an intermediate transaction abstraction.

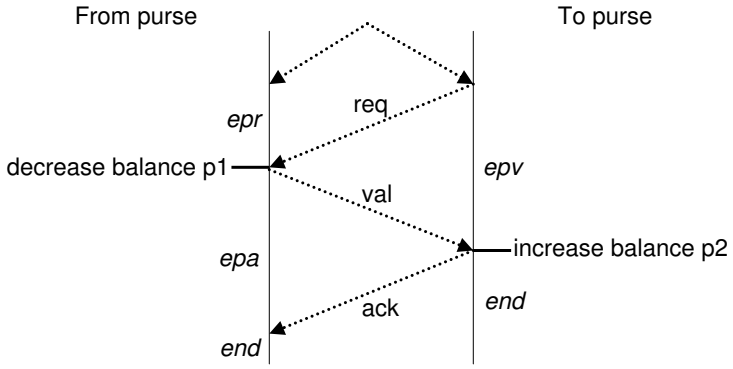

Fig. 2. Overview of the Mondex protocol

We will see that this intermediate abstraction provides sufficient structure for us to break the atomicity of the money transfer without overwhelming our reasoning with too much detail. A transaction consists of a source and target purse as well as an amount so we introduce a record type $TRANS$ modelling transactions as follows:

$$\begin{aligned} TRANS :: & \; from \in PURSE; \\ & \; to \in PURSE; \\ & \; amount \in \mathbb{N} \end{aligned}$$

Note that record declaration are not part of B but [6] shows how record declarations like the above may be represented in B in a straightforward way. The fields of a record declaration represent projection functions mapping transactions to values of an appropriate type, e.g., this declaration specifies that $from$ is a function from $TRANS$ to $PURSE$.

Once it has been created, a transaction can be in one of following four states: *Idle*, *Pending*, *Recover* or *Ended*. A transaction is *idle* on creation while a transaction is *pending* in between deduction from the source and addition to the target. From the pending state, a transaction may go to the *ended* state on successful completion or to the *recover* state because of an abort. Note that these states are not explicit states in the Mondex protocol. These are states we introduced as part of our intermediate abstraction. A variable $trans \subseteq TRANSACTION$ is introduced to model the set of transaction that have been created. We also need to model the states a transaction may be in. One possible approach is to introduce a variable representing a function mapping current transactions ($trans$) to an enumerated set of states:

$$status \in trans \rightarrow \{ \; Idle, \; Pending, \; Recover, \; Ended \; \}$$

However, as we discuss later, we found it more effective (in terms of ease of proof) to represent the purse states as disjoint sets of transactions as follows:

$$\begin{aligned} & idle \subseteq trans \\ & pending \subseteq trans \\ & recover \subseteq trans \\ & ended \subseteq trans \\ & disjoint(\; idle, \; pending, \; recover, \; ended \;) \end{aligned}$$

The event modelling the creation of a transaction is modelled in Fig. 3, putting the new transaction in the idle state. The guard ensures that the value selected for the transaction t is fresh. i.e., not contained in $trans$, and has appropriate field values, e.g., $from(t) = p1$.

The concrete variable *cbal* introduced in the first refinement replaces the abstract variable *abal*. These

$StartTrans =$
ANY $p1, p2, a, t$ WHERE
$p1 \in purse$
$p2 \in purse$
$p1 \neq p2$
$t \in TRANS \setminus trans$
$from(t) = p1$
$to(t) = p2$
$am(t) = a$
THEN
$trans := trans \cup \{t\}$
$idle := idle \cup \{t\}$
END

Fig. 3. Starting a Transaction

$Deduct =$
ANY $p1, a, t$ WHERE
$p1 \in purse$
$t \in idle$
$from(t) = p1$
$am(t) = a$
$cbal(p1) \geq a$
THEN
$pending := pending \cup \{t\}$
$idle := idle \setminus \{t\}$
$pendF := pendF \cup \{p1 \mapsto t\}$
$cbal(p1) := cbal(p1) - a$
END

$Increase =$
$/* refines\ TransferOK */$
ANY $p2, a, t$ WHERE
$p2 \in purse$
$t \in pending$
$to(t) = p2$
$am(t) = a$
THEN
$ended := ended \cup \{t\}$
$pending := pending \setminus \{t\}$
$pendF := pendF \setminus \{p1 \mapsto t\}$
$cbal(p2) := cbal(p2) + a$
END

$TransferFail =$
$/* refines\ TransferFail */$
ANY $p1, a, t$ WHERE
$p1 \in purse$
$t \in pending$
$from(t) = p1$
$am(t) = a$
THEN
$recover := recover \cup \{t\}$
$pending := pending \setminus \{t\}$
$lostF := lostF \cup \{p1 \mapsto t\}$
$pendF := pendF \setminus \{p1 \mapsto t\}$
END

Fig. 4. Starting a Transaction and Deduct events

two variables will not always be equal because of the splitting of the atomicity of value transfer between the *Deduct* event and the *Increase* event shown in Fig. 4. The *Increase* event is a refinement of the abstract *TransferOk* event and represents successful transfer of value. The role of the *pendF* variable will be explained later. Since the concrete balance is decreased in the separate *Deduct* event, the abstract balance of a purse p will be equal to its refined balance plus the sum of the amounts of all values in transit from p. This relationship between abstract and concrete balance is made explicit in the gluing invariant for the refinement. The refined *TransferFail* event changes a *pending* transaction to the *recover* state. Notice that the abstract *lost* variable is not used in the refinement. Instead lost value is embodied in the amounts of all transactions that are in the *recover* state. The relationship between this and the abstract *lost* variable will be made clear below.

Given that $from$ is a function from transactions to purses and *pending* is a set of pending transactions, the domain restriction expression $pending \lhd from$ (of type $pending \rightarrow PURSE$) is a function mapping pending transactions to their source purse. This means that $(pending \lhd from)^{-1}[\ \{p\}\ ]$ represents the set of pending transactions for which p is the source purse. In order to express the summation of transaction amounts, we introduce a constant function *sum* that takes a finite set of transactions and returns the sum of the amounts of each of the transactions in the set. The *sum* function is defined by the following axioms:

$$sum \in \mathbb{F}(TRANS) \rightarrow \mathbb{N}$$
$$t \notin s \Rightarrow sum(s \cup \{t\}) = sum(s) + t$$
$$t \in s \Rightarrow sum(s \setminus \{t\}) = sum(s) - t$$

We can now define the gluing invariant linking the abstract and concrete balances:

$$abal(p) = cbal(p) + sum(\ (pending \lhd from)^{-1}[\ \{p\}\ ]\ )$$

Our experience with this invariant was that the complexity of the expression over which we sum impedes the

mechanical proof. We chose to introduce a redundant variable $pendF$ corresponding to $(pending \lhd from)^{-1}$. This allows the gluing invariant to be simplified to the following:

$abal(p) = cbal(p) + sum(\ pendF[\ \{p\}\ ]\ )$

The redundant variable is removed in a subsequent refinement step using the gluing invariant

$pendF = (pending \lhd from)^{-1}$

The gluing invariant linking the abstract $lost$ variable and the recoverable transactions is defined as follows:

$lost(p) = sum(\ (recover \lhd from)^{-1}[\ \{p\}\ ]\ )$

As with $pendF$, we introduced a redundant variable $lostF$ which is removed in a subsequent refinement step using the gluing invariant

$lostF = (recover \lhd from)^{-1}$

4. Overview of the Full Refinement Chain

So far we have described our abstract model of the Mondex system and the first refinement. Rather than presenting the other eight refinement stages in similar detail, we will just present a sufficient overview of the remaining refinement stages in order to help the reader understand the rationale for each refinement stage.

Level 1 As we have seen our first model captures atomic transfer of money, transaction failure and recovery of lost value. It also models creation of new purses and balance query on a purse. We will look at balance query in more detail later in Section 6.

Level 2 This level has been outlined in the previous section. An abstract notion of transaction is introduced with end-to-end state (*Idle*, *Pending*, *Recover* or *Ended*). Each transaction has a balance and a $from$ and a to purse. The money transfer process is split into two separate events, one which decreases the source purse and the other which increases the target balance. Freshness of a new transaction t is specified by ensuring t is not contained in the set of existing transactions $trans$.

Level 3 This is a very simple refinement in which the redundant variables $pendF$ and $lostF$ are removed using the invariants described previously.

Level 4 This is a relatively complex refinement in which the the end-to-end state of transactions (*Idle*, *Pending*, ...) is replaced by dual states in which both parties to a transaction each have their own local protocol state. Some of these local states were illustrated in Fig. 2. For example the source purse $p1$ of a transaction t is in the epr (expecting request) state in between receiving a $start$ message from a terminal and receiving a req message from the target purse. Formally this is modelled by the condition $t \in epr$ where $from(t) = p1$. Level 4 also models abort states for purses in a transaction. For example, $t \in abortepv$ models the situation in which the target side of transaction t aborted having previously been in the state epv.

Fig. 5 shows three of the events of Level 4. The $StartTrans$ event models the initiation of a new transaction. The '...' indicates that the guard of $StartTrans$ is the same as in the previous level (Fig. 3). Instead of adding the new transaction to the abstract $idle$ set, it is added to both $idleF$ and $idleT$. This models both source and target sides being initialised to idle and allows the purses to leave the idle state asynchronously via separate events.

The $StartTo$ event models the transition in the target purse to the epv (expecting value) state. The target side of the transaction should be in the idle state ($t \in idleT$ in the guard of $StartTo$) allowing the target side to change to the epv state (the actions of $StartTo$ move t from the $idleT$ set to the epv set).

The $Deduct$ event models the transition in the source purse when it receives a $request$ message. There are three conditions that should hold for $Deduct$ to be fired. The source side of the transaction should be in the idle state ($t \in epr$), there should be sufficient funds in the source side and the target side of the transaction should already have made the $StartTo$ transition. The target side can abort from the epv state so it will be in the epv state or the $abortepv$ state after making the $StartTo$ transition ($t \in (epv \ \cup \ abortepv)$). These

$StartTrans =$
ANY $p1, p2, a, t$ WHERE
  ...
THEN
  $trans := trans \cup \{t\}$
  $idleT := idleT \cup \{t\}$
  $idleF := idleF \cup \{t\}$
END

$StartTo =$
ANY $p2, t$ WHERE
  $p2 \in purse$
  $t \in idleT$
  $p2 = to(t)$
THEN
  $epv := epv \cup \{t\}$
  $idleT := idleT \setminus \{t\}$
END

$Deduct =$
ANY $p1, a, t$ WHERE
  $p1 \in purse$
  $t \in epr$
  $t \in (epv \cup abortepv)$
  $p1 = from(t)$
  $a = am(t)$
  $a \leq cbal(p1)$
THEN
  $epa := epa \cup \{t\}$
  $epr := epr \setminus \{t\}$
  $cbal(p1) := cbal(p1) - a$
END

Fig. 5. *StartTrans*, *StartTo* and *Deduct* events for Level 4

$StartTrans =$
ANY $p1, p2, a, t$ WHERE
  ...
THEN
  ...
  $startfromM := startfromM \cup \{t\}$
  $startToM := startToM \cup \{t\}$
END

$StartTo =$
ANY $p2, t$ WHERE
  ...
  $t \in startFromM$
THEN
  ...
  $reqM := reqM \cup \{t\}$
END

$Deduct =$
ANY $p1, a, t$ WHERE
  $p1 \in purse$
  $t \in epr$
  $t \in reqM$
  $p1 = from(t)$
  $a = am(t)$
  $a \leq cbal(p1)$
THEN
  ...
  $valM := valM \cup \{t\}$
END

Fig. 6. *StartTrans*, *StartTo* and *Deduct* events for Level 5

conditions allow the source side to deduct the transaction amount from its balance and to change to the *epa* (expecting acknowledgement) state.

The relationship between the abstract end-to-end states and the more concrete dual states of a transaction will be elaborated in Section 5. As a taster, one of the gluing invariants linking these representations is the following:

$$(epa \cup abortepa) \cap epv \quad \subseteq \quad pending$$

This can be read as specifying that when the source purse of a transaction is in the *epa* or *abortepa* state and the target purse is in the *epv* state then the abstract end-to-end state of the transaction is *pending*.

Level 5 In the previous Level 4, events that change the state of $p1$, e.g., *Decrease*, were allowed to access the state of $p2$ directly and vice versa. In practice this is not possible of course. Instead the purses send messages to each other as outlined in Fig. 2 to indicate changes to their state. Level 5 introduces explicit messaging between purses corresponding to the messages illustrated in Fig. 2. Now, instead of directly accessing the state of the other purse, one purse gains information about the other purse when it receives a message. For example, when the source purse of a transaction receives a *req* message it knows that the target purse is either in the *epv* state or the *abortepv* state for that transaction. Variables are introduced to represent messages that have been sent as part of a transaction. At this level, rather than having a single 'ether' containing all the different kinds of message that can be sent (*req*, *val*,...), we have separate variables for the different kinds of messages. For example the variable $reqM \subseteq trans$ represents *req* messages that have been sent: $t \in reqM$ means that purse $from(t)$ has sent a *req* message to purse $to(t)$ for transaction t.

Fig. 6 shows the Level 5 refinements of the Level 4 events of Fig. 5. In Fig. 6 it can be seen that the Level 5 *StartTrans* event adds t to both the $startfromM$ and $starttoM$ sets. This is an abstract representation of the sending of start messages to both sides by the terminal equipment. The *StartTo* event adds t to the $reqM$ set. This is an abstract representation of the sending of a request message by the target side. In Fig. 5, the *Deduct* event, which occurs on the source side, sees the state of the target side directly ($t \in (epv \cup abortepv)$).

$StartTrans =$
ANY $p1, p2, a, t$ WHERE
$p1 \in purse \setminus active$
$p2 \in purse \setminus active$
$t \in TRANS \setminus trans$
$p1 = from(t)$
$p2 = to(t)$
$a = am(t)$
THEN
$trans := trans \cup \{t\}$
$active := active \cup \{p1\} \cup \{p2\}$
$idleFP := idleFP \cup \{p1\}$
$idleTP := idleTP \cup \{p2\}$
$currentF(p1) := t$
$currentT(p2) := t$
$startfromM := startfromM \cup \{t\}$
$startToM := startToM \cup \{t\}$
END

$StartFrom =$
ANY $p1, t$ WHERE
$p1 \in idleFP$
$t \in startfromM$
$t = currentF(p1)$
$p1 = from(t)$
THEN
$eprP := eprP \cup \{p1\}$
$idleFP := idleFP \setminus \{p1\}$
END

Fig. 7. *StartTrans* and *StartFrom* events for Level 6

In the refinement of *Deduct* in Fig. 6, the source side instead infers the state of the target side through the receipt of a message for the transaction ($t \in reqM$). Since the guard of an event cannot be weakened when we refine it, we use the following invariant to verify this refinement:

$$epr \cap reqM \quad \subseteq \quad epv \cup abortepv$$

That is, if the source side of t is in the *epr* state and a request message has been sent for t, then the target side is either in the *epv* state or the *abortepv* state. The Level 5 *Deduct* event also adds t to the $valM$ set modelling the sending of a *val* message to the target.

Level 6 Up to Level 5 our models allow a purse to be involved in multiple simultaneous transactions. With the purse usage model in which two purses come together to exchange value by being placed in the same physical terminal, the purses are physically constrained to being involved in at most one transaction. This constraint is introduced in Level 5 by introducing for each purse a single current transaction and an archive of aborted transactions. So a purse either has a current transaction for which it is the source $currentF \in purse \nrightarrow trans$ or for which it is the target $currentT \in purse \nrightarrow trans$. Instead of representing the state of each transaction ($idleF$, *epr*, *epa*...) as in previous levels, we only need to represent the state of the current transaction for each purse. Thus, for example, the variable $idleF \subseteq trans$ (representing the set of transactions whose source side is idle) is replaced by the variable $idleFP \subseteq purse$ (representing the set of purses acting as a source and in the idle state for their current transaction). These variables are linked by the following gluing invariant:

$$currentF[idleFP] \quad \subseteq \quad idleF$$

That is, if a purse is in the $idleFP$ state at Level 6, then the current transaction of that purse is in the $idleF$ state at Level 5. If a transaction aborts, then both sides will asynchronously place a log of that aborted transaction in their respective archives.

Fig. 7 shows the *StartTrans* and *StartFrom* events of Level 6. The guard of *StartTrans* means that neither purse can be active, i.e., involved in a transaction. Both purses are made active and are added to $idleFP$ and $idleTP$ sets respectively. The current transaction of both purses is set to the the new transaction t and $Startfrom$ and $StartTo$ messages are sent. In the *StartFrom* event, a source purse $p1$ that is idle ($p1 \in idleFP$) and receives a $Startfrom$ message corresponding to its current transaction ($t \in currentF(p1)$ and $p1 = from(t)$) can progress to the *epr* state.

It is interesting to note that it is not quite the case that a purse is only ever involved in one transaction. For example, once a target purse $P2$ sends an acknowledgement to source purse $P1$ it is finished with the transaction. $P2$ could then start a new transaction with a purse $P3$ before $P1$ receives the acknowledgement (probably a rare occurrence). So $P1$ would still be involved in a transaction with $P2$ though $P2$ is also

involved in a transaction with $P3$. Luckily this does not cause us any difficulty since up to Level 6 we allowed for a purse being involved in any number of transactions simultaneously.

Level 7 Up to Level 6, the variable *trans* represents a history of all the transactions ever created in the system and was used to define the freshness of new transactions. In practice no such central history exists so global freshness needs to be ensured by the two purses involved in any new transaction. This is achieved by each purse providing a sequence number that is locally fresh for that purse. The combination of both locally fresh sequence numbers and the identity of the purses is sufficient to ensure global freshness. In Level 7 the global history *trans* is removed and instead each purse is given a local history of the sequence numbers that it has used. The variables $used \in purse \leftrightarrow \mathbb{N}$ maps each purse to the set of sequence numbers it has used. The transaction record type is extended with two fields of type $\mathbb{N}$, the source and target sequence numbers ($Fseqno$ and $Tseqno$ respectively). The following gluing invariant specifies that for any transaction t in the abstract history variable *trans*, the source sequence number of that transaction will be in the used set of the source purse:

$$t \in trans \quad \Rightarrow \quad (\ from(t) \mapsto Fseqno(t)\ ) \in used$$

There is a similar invariant for target sequence numbers.

Fig. 8 shows the $StartTrans$ and $StartFrom$ events of Level 7. In the guard of $StartTrans$, the more abstract freshness condition $t \notin trans$ is replaced by the condition that the sequence numbers $n1$ and $n2$ where not previously used by their respective purses ($p1 \mapsto n1 \notin used$ and $p2 \mapsto n2 \notin used$). In the previous level (Fig. 7), the current transaction of the purses was set by the $StartTrans$ event ($currentF(p1) := t$ and $currentT(p2) := t$). In the Mondex protocol, the source purse does not receive the full details of the transaction until it receives the $StartFrom$ message. Before then the source purse knows only its sequence number but not the target sequence number nor the amount. To reflect this, the Level 7 refinement sets $currentF(p1)$ in the $StartFrom$ event rather than the $StartTrans$ event. Because the Level 6 $StartFrom$ does not set $currentF(p1)$, $currentF$ is replaced by $currentF2$ with the following gluing invariant:

$$p \in (active \setminus idleFP) \quad \Rightarrow \quad currentF(p) = currentF2(p)$$

The source purse accepts a transaction provided the source sequence number of the transaction corresponds to the purse's current sequence number ($Fseqno(t) = currentSeqNo(p1)$ in the guard of $StartFrom$). The setting of the current transaction on the target side to moved to the $StartTo$ event in a similar way.

The reader may have observed that a single sequence number coming from say the source purse would be sufficient to ensure uniqueness of the transaction. In fact the sequence numbers serve a further purpose: they act as challenge values that prevent message replay attacks. For this to be effective, both purses need to provide challenge values. We treat replays in a very simple way. A replay attack involves resending messages that were previously sent. In our model messages are never removed from the message buffers. This means that a message replay only ever adds messages to the buffer that are already in the buffer. A replay is thus a skip and has no effects at the abstract level.

Level 8 In Level 8, instead of maintaining a history of used sequence numbers, each purse stores just a single sequence counter ($currentSeqNo(p)$) which is increased by an arbitrary amount each time it is involved in starting a new transaction. The gluing invariant is as follows:

$$p \mapsto n \in used \quad \Rightarrow \quad n \leq currentSeqNo(p)$$

Fig. 9 shows the $StartTrans$ event of Level 8. It can be seen that the Level 7 freshness condition $p1 \mapsto n1 \notin used$ is replaced by $n1 > currentSeqNo(p1)$ in Level 8.

Level 9 In Level 6, the state of the current transaction of each purse was represented by the disjoint variables $eprP$, $epaP$, etc. In Level 9, this disjoint sets model of transaction states is replaced by 'status' functions mapping each purse to an enumerated set of states. If p is involved in a transaction as the source purse, then $statusF(p)$ represents the transaction state of that purse $\{IDLE, EPR, EPA\}$. The relevant gluing invariant is as follows:

$$eprP \quad = \quad statusF^{-1}[\ \{EPR\}\ ]$$

There are similar invariants for the other disjoint state sets.

$StartTrans =$
ANY $p1, p2, a, t, n1, n2$ WHERE
$p1 \in purse \setminus active$
$p2 \in purse \setminus active$
$t \in TRANS$
$p1 = from(t)$
$p2 = to(t)$
$a = am(t)$
$n1 = Fseqno(t)$
$n2 = Tseqno(t)$
$p1 \mapsto n1 \notin used$
$p2 \mapsto n2 \notin used$
THEN
$active := active \cup \{p1\} \cup \{p2\}$
$idleFP := idleFP \cup \{p1\}$
$idleTP := idleTP \cup \{p2\}$
$startfromM := startfromM \cup \{t\}$
$startToM := startToM \cup \{t\}$
$used := used \cup \{p1 \mapsto n1\} \cup \{p2 \mapsto n2\}$
$currentSeqNo(p1) := n1$
$currentSeqNo(p2) := n2$
END

$StartFrom =$
ANY $p1, t$ WHERE
$p1 \in idleFP$
$t \in startfromM$
$Fseqno(t) = currentSeqNo(p1)$
$p1 = from(t)$
THEN
$eprP := eprP \cup \{p1\}$
$idleFP := idleFP \setminus \{p1\}$
$currentF2(p1) := t$
END

Fig. 8. *StartTrans* and *StartFrom* events for Level 7

$StartTrans =$
ANY $p1, p2, a, t, n1, n2$ WHERE
...
$n1 = Fseqno(t)$
$n2 = Tseqno(t)$
$n1 > currentSeqNo(p1)$
$n2 > currentSeqNo(p2)$
THEN
...
$startToM := startToM \cup \{t\}$
$currentSeqNo(p1) := n1$
$currentSeqNo(p2) := n2$
END

Fig. 9. Part of *StartTrans* event for Level 8

Level 10 In Level 10, the separate variables for the different kinds of messages introduced in Level 5 are merged into a single 'ether' variable representing the messages that have been sent. A message type $MESS$ is introduced, and several disjoint subsets of the message type are defined as constants. For example $REQ\_MESS \subseteq MESS$ represents the type of request messages. The type $MESS$ is a record with a transaction field $trn \in MESS \rightarrow TRANS$. Given that $ether \subseteq MESS$ is the variable representing all messages ever sent, the set of request messages sent is represented by the expression $ether \cap REQ\_MESS$. In Level 10, the abstract variable $reqM$ is represented by extracting the transaction field of all request messages sent as described by the following gluing invariant:

$$reqM \quad = \quad trn[\ ether \cap REQ\_MESS\ ]$$

Similar invariants are used for the other message types.

Fig. 10 shows the *StartFrom* and *Deduct* events of Level 10. It can been seen that the state of a source purse is represented by the *statusF* function (e.g., $statusF(p1) = IDLEF$ in *StartFrom*). The use of the message type of the *ether* variable can also be seen. In the *StartFrom* event, source purse $p1$ accepts and message $m \in ether$ of type SF_MESS and extracts the transaction details t from the message

$StartFrom =$
ANY $p1, t, m$ WHERE
$statusF(p1) = IDLEF$
$m \in ether$
$m \in SF\_MESS$
$t = trn(m)$
$Fseqno(t) = currentSeqNo(p1)$
$p1 = from(t)$
THEN
$statusF(p1) := epr$
$currentF(p1) := t$
END

$Deduct =$
ANY $p1, a, t, m1, m2$ WHERE
$statusF(p1) = EPR$
$m1 \in ether$
$m1 \in REQ\_MESS$
$t = trn(m1)$
$p1 = from(t)$
$a = am(t)$
$a \leq cbal(p1)$
$m2 \in VAL\_MESS$
$t = trn(m2)$
THEN
$statusF(p1) := EPA$
$ether := ether \cup \{m2\}$
$cbal(p1) := cbal(p1) - a$
END

Fig. 10. *StartTrans* and *StartFrom* events for Level 10

Level	POs	iPOs
L1	24	0
L2	91	15
L3	14	0
L4	143	0
L5	57	0
L6	183	0
L7	25	0
L8	23	2
L9	73	0
L10	46	0
Total	679	17

Fig. 11. Statistics from the mechanical proof effort

($t = trn(m)$). In the *Deduct* event, source purse $p1$ accepts a REQ_MESS message $m1$ from the ether and adds a VAL_MESS message $m2$ to the ether.

This completes our overview of the full refinement chain.

5. Proof and Invariant Discovery

All the proof obligations for all ten levels were generated and proved using the B4free prover. The statistics from the mechanical proof effort for all of the refinement levels are outlined in Fig. 11. In the table, the POs column represents the total number of proof obligations generated for each level. The iPOs column represents the number of those proof obligations that had to be proved *interactively*. Those proof obligations that were not proved interactively were proved completely automatically by the prover. So, all the proofs were completely automatic for all levels except at Levels 2 and 8. It is worth commenting on the interactive proofs that were required at Level 2. All these proofs involved either the *sum* function that sums the amounts of a finite set of transactions or finiteness constraints on sets of transactions. The main interactive steps involved instantiating the axioms for *sum* described in Section 3. The automatic prover did not manage to instantiate these axioms appropriately and instead the instantiation had to be done manually through the interactive prover.

We now outline the way in which the invariant used to prove the refinement between Levels 3 and 4. Recall that Level 3 models transaction states by an abstract end-to-end state (*pending*, *recover*,...) while Level 4 replaces this by dual local states for both the source and target purses of a transaction. Initially the model at Level 4 was constructed with just a basic typing invariant and no gluing invariant. The first pass of automatic proof resulted in several proof obligations that could not be proved automatically. Some of these

Deduct$(PO1)$	***TransferOK***$(PO2)$	***TransferFail***$(PO3)$
$t \in trans$	$t \in trans$	$t \in trans$
$t \in epr$	$t \in (epa \cup abortepa)$	$t \in abortepa$
$t \in (epv \cup abortepv)$	$t \in epv$	$t \in epv$
$\Rightarrow$	$\Rightarrow$	$\Rightarrow$
$t \in idle$	$t \in pending$	$t \in pending$

Fig. 12. Proof Obligations- I

$epr \subseteq idle$
$epv \cap (epa \cup abortepa) \subseteq pending$
$epa \cap (epv \cup abortepv) \subseteq pending$
$epa \cap abortepa = \phi$
$epv \cap abortepv = \phi$
$abortepa \cap abortepv \subseteq recover$

Fig. 13. Invariants-I

$idleF \subseteq idle$
$idleT \cap (epa \cup abortepa) = \phi$
$idleT \cap (epv \cup abortepv) = \phi$
$pending \subseteq (epa \cup abortepa)$
$endT \cap pending = \phi$
$(epv \cup abortepv) \cap endT = \phi$

Fig. 14. Invariants-II

$epr \cap idleF = \phi$
$idleT \cap endT = \phi$
$endT \subseteq ended$
$abortepa \cap epr = \phi$
$epa \cap epr = \phi$
$idleF \cap epa = \phi$

Fig. 15. Invariants-III

are shown in Fig. 12. Obligation *PO1* arises from the need to show that when the refined *Deduct* operation is enabled, then the abstract *Deduct* event is also enabled. Similarly for the other obligations.

If we were to add *PO1*, universally quantified over variable t, as an invariant then *PO1* is discharged automatically. However, in this case, our intuition tells us that it is sufficient to have $t \in epr$ at the concrete level in order to have $t \in idle$ at the abstract level. So we try the simpler invariant

$$\forall t \cdot (\ t \in epr \ \Rightarrow \ t \in idle \ )$$

Furthermore, this invariant can be represented in a point-free way using subset ordering:

$$epr \ \subseteq \ idle$$

We chose this point-free form as our experience with the automatic prover is that is usually very good at dealing with quantifier-free goals. The full set of invariants introduced in this first round of proof is shown in Fig. 13. All of these were 'discovered' by examining the non-proved proof obligations as described above.

When the invariants given as *Invariants-I* are added to the model, new proof obligations associated with these new invariants are generated. In order to discharge these additional proof obligations we added another set of invariants to our model given as *Invariants-II* (Fig. 14). When *Invariants-II* are added to the model, further proof obligations are generated. These proof obligations are discharged by adding another set of invariant given as *Invariants-III* (Fig. 15).

After three iterations of invariant strengthening we arrived at set of invariants that are sufficient to discharge all the proof obligations for this refinement stage automatically. An appropriate gluing invariant is key to proving the correctness of a refinement step. In this section we outlined how we used the non-proved proof obligations and the interactive prover to guide us in constructing a gluing invariant. As well as easing the burden of inventing the gluing invariant, this approach also has the consequence that the form of the gluing invariant we use closely matches the form of the proof obligations thereby making the mechanical proofs much easier and in many cases completely automatic.

6. Other Issues: Errors, Recovery and Balance Check

In this section we outline how we addressed some additional features of the protocol including modelling aborts, recovery and balance checking.

6.1. Errors

Fig. 16 shows abort events for when a source purse is in the *epa* state. Both $AbortEPA1$ and $AbortEPA2$ are from Level 4 of the refinement chain. Event $AbortEPA1$ represents the occurrence of an abort from state *epa* when the target side has already aborted from the *epv* state. Event $AbortEPA2$ represents the occurrence of an abort from state *epa* when the target side has not aborted from the *epv* state. The reason

$AbortEPA1 =$
/ ∗ refines $TransferFail$ ∗ /
ANY t WHERE
$t \in epa$
$t \in abortepv$
THEN
$abortepa := abortepa \cup \{t\}$
$epa := epa \setminus \{t\}$
END

$AbortEPA2 =$
/ ∗ refines $skip$ ∗ /
ANY t WHERE
$t \in epa$
$t \notin abortepv$
THEN
$abortepa := abortepa \cup \{t\}$
$epa := epa \setminus \{t\}$
END

$AbortEPA =$
/ ∗ refines $AbortEPA1$, $AbortEPA2$ ∗ /
ANY t WHERE
$t \in epa$
THEN
$abortepa := abortepa \cup \{t\}$
$epv := epa \setminus \{t\}$
END

Fig. 16. Level 4 and 5 $AbortEPV$ events

we make the distinction between these two cases is that the different cases refine different abstract events. In the first case ($t \in abortepv$), aborting from the *epa* state is a refinement of the abstract $TransferFail$ event. At Level 4, we regard a transaction as having failed when both sides have aborted. Abort of just one side is not sufficient since a transaction can succeed even if the source side aborts from the *epa* state. Execution of $AbortEPS1$ results in both sides being in aborted states and thus it refines the abstract $TransferFail$ event. Execution of $AbortEPA2$ results only in one side being in an aborted state and thus refines *skip*. Note that the bodies of the events for both cases are identical.

In the real system, since an abort is local to a purse, we cannot distinguish between cases that depend on the state of another purse. This means we should merge the two cases of $AbortEPA$ into a single event that is independent of the state of a target purse. In Event-B we can do this through a refinement: a refined event can refine several abstract events [7]. In Fig. 16, $AbortEPA$ is a Level 5 event that merges both of the Level 4 events and is independent of *abortepv*. The conditions under which this merge is valid is that both abstract events must have identical bodies and the guard of the refined event must imply the disjunction of the guards of the abstract events. These conditions indeed hold between the events of Fig. 16.

This treatment of abort events highlights the importance of the ability to reason globally that Event-B refinement supports. At Level 4 we were able to distinguish between the two cases of $AbortEPA$ by taking a global view, i.e., including representations of the state of the target purse in events of the source purse. The two cases refine two different abstract events. In a subsequent refinement we remove the distinction by merging the cases and removing the dependence on the state of the target purse. The distinction is being made at design time, through modelling and refinement proofs, but is not being made at run time.

6.2. Recovery

Fig. 17 shows the events modelling recovery of lost money at four different levels of abstraction. As explained in Section 1, the Level 1 event is very simple: an amount a can be recovered by purse $p1$ provided at least that amount is in $lost(p1)$. Transactions were introduced at Level 2 with the abstract end-to-end states including *pending* and *recover*. The Level 2 recover event is allowed if a transaction is in the *recover* state. The gluing invariant required to prove this refinement (linking the abstract *lost* variable with recoverable transaction) was shown at the end of Section 3. At Level 4 we distinguished between source and target purse states. Now a transaction is recoverable when the source side has aborted from the *epa* state and the target side has aborted from the *epv* state, hence the guard of the Level 4 recover event. This refinement requires the following gluing invariant:

$$abortepa \cap abortepv \subseteq recover$$

In Level 10, we do not maintain information about all transactions. Instead a purse will log only those transactions that have aborted from the *epa* state (in $archF$) or the *epv* state (in $archT$). The Level 10 recover event is allowed when both purses in a transaction have a log for that transaction in their archive. The gluing invariants that are used to prove the refinement are:

$$dom(archF) \subseteq abortepa$$
$$dom(archT) \subseteq abortepv$$

$(L1)$ *Recover* =
ANY $p1, a$ WHERE
  $p1 \in purse$
  $lost(p1) \geq a$
THEN
  $lost(p1) := lost(p1) - a \parallel$
  $abal(p1) := abal(p1) + a$
END

$(L2)$ *Recover* =
ANY $p1, a, t$ WHERE
  $p1 \in purse$
  $t \in recover$
  $from(t) = p1$
  $am(t) = a$
THEN
  $end := end \cup \{t\}$
  $recover := recover \setminus \{t\}$
  $cbal(p1) := cbal(p1) + a$
END

$(L4)$ *Recover* =
ANY $p1, a, t$ WHERE
  $p1 \in purse$
  $t \in abortepa$
  $t \in abortepv$
  $from(t) = p1$
  $am(t) = a$
THEN
  $endF := endF \cup \{t\}$
  $endT := endT \cup \{t\}$
  $abortepa := abortepas \setminus \{t\}$
  $abortepv := abortepvs \setminus \{t\}$
  $cbal(p1) := cbal(p1) + a$
END

$(L10)$ *Recover* =
ANY $p1, p2, a, t$ WHERE
  $p1 \in purse$
  $p2 \in purse$
  $t \mapsto p1 \in archF$
  $t \mapsto p2 \in archT$
  $from(t) = p1$
  $to(t) = p2$
  $am(t) = a$
THEN
  $archF := archF \setminus \{t \mapsto p1\} \parallel$
  $archT := archT \setminus \{t \mapsto p2\} \parallel$
  $cbal(p1) := cbal(p1) + a$
END

Fig. 17. Recover events

6.3. Balance Checking

We included a balance check feature in our development. Recall that we have an abstract and concrete balance related by :

$$abal(p) = cbal(p) + sum(pendF[\{p\}])$$

The abstract and concrete balances are equal when $pendF[\{p\}]$ is empty. In the implementation, the balance query for purse p returns $cbal(p)$. We then considered how to model the balance query in terms of the abstract balance. The approach we adopted was to introduce an exact balance check and an approximate balance check giving an under-approximation of the balance. These events are modelled in Fig. 18. In Fig. 18 we do not distinguish the conditions under which an exact or approximate balance is returned as we don't have enough information available to make the distinction at the abstract level. Fig. 19 shows Level 10 concrete versions of the balance check events where disjoint conditions under which these are distinguishable are made explicit. The *GetBalanceExact* of Fig. 19 shows that when a purse is not in an *EPA* state and its *archF* is empty, then balance check on the purse returns exactly the same balance as the abstract model would. In other cases balance check may be an under approximation of the balance. In the real system, no distinction is made between exact and inexact balance checks and thus the two events could be merged in a subsequent refinement to the *GetBalance* event of Fig. 19. As with the treatement of aborting transactions in Section 6.1, we can distinguish the cases at design time, though we don't distinguish them at run time. Interestingly in this case we could also distinguish them at run time since the distinguishing condition is based on state that is local to a purse. A purse could indicate whether there may be money that could be recovered later. It is less clear that this would be an attractive feature.

$GetBalanceExact =$
ANY p, a WHERE
  $p \in purse$
  $a! = abal(p)$
THEN
  $skip$
END

$GetBalanceApprox =$
ANY p, a WHERE
  $p \in purse$
  $a! \leq abal(p)$
THEN
  $skip$
END

Fig. 18. Abstract balance query events

$GetBalanceExact =$
ANY p, a WHERE
  $p \in purse$
  $statusF(p) \neq EPA$
  $p \notin ran(archF)$
  $a! = cbal(p)$
THEN
  $skip$
END

$GetBalanceApprox =$
ANY p, a WHERE
  $p \in purse$
  $(\ statusF(p) = EPA \quad \vee$
    $p \in ran(archF)\ )$
  $a! = cbal(p)$
THEN
  $skip$
END

$GetBalance =$
ANY p, a WHERE
  $p \in purse$
  $a! = cbal(p)$
THEN
  $skip$
END

Fig. 19. Concrete balance query events

7. Invalid Modelling, Modification and Reproof

After we had achieved our first fully proved complete refinement chain, we discovered that we had made an invalid modelling assumption at Level 7 when the sequence numbers were introduced. In our model we assumed that the sequence numbers were exchanged between both purses as part of the *StartTrans* event. In the real protocol one purse only becomes aware of the the sequence number generated by the other purse when it receives a *start* message and *start* messages are sent after the *StartTrans* event. A consequence of our invalid modelling assumption was that re-play attacks were prevented without requiring both purses to provide a sequence number. Now sequence numbers were only required to ensure uniqueness of the transaction and, as pointed out above, a single sequence number is sufficient for that. Under the invalid modelling assumption, our refinement proof only required one sequence number.

We observed that only one sequence number was required for the proof without understanding why this was. At that point the dual role of the sequence numbers in ensuring uniqueness and in preventing replays was not fully clear to us. Luckily Gerhard Schellhorn pointed out an attack that that would arise were there only one sequence number. This attack didn't arise in our models because of the invalid assumption we made, but it did lead us to realise the modelling error we had made. This highlights the importance of informally reviewing all modelling assumptions during formal development to ensure their validity. There is a danger of focusing solely on the correctness of proofs as a measure of the validity of models.

Once we realised our error we modified the refinement chain to so that we correctly modelled the protocol. We did not need to change the chain up to Level 5. Only Levels 6 to 10 needed modification. The modifications required were relatively small and the only significant new invariants required were at Level 6. It took less than half a day to make the modifications and re-establish a fully proved refinement chain. We believe the relative ease with which this change was achieved was because of our use of multiple refinement levels with small abstraction gaps. This meant that the impact of the change was reasonably well localised and that therefore the re-proof effort was relatively small. The high degree of automatic proof that our approach entailed also meant that most of the re-proof required was achieved automatically in any case.

As with the original Z development, our Event-B development ignores details of the set up phase of the protocol. Schellhorn et al [9] describe the set up phase in more detail. This involves the terminal asking each purse for its current sequence number. On producing the final revision of this paper we realised that we had made one final error in our modelling. In our model, each purse increases its sequence number in the *StartTrans* event. In fact in the protocol, a purse only increases its sequence number on receipt of a *StartTo* or *StartFrom* message. This means that if an interaction aborts before this stage, a purse will not increase its sequence number. Our developmetn could be adjusted to deal with this by making the *StartTo* and *StartFrom* events both be refinements of *StartTrans*.

8. Modelling Guidelines

Our main guideline is that incremental development is a good way of achieving automatic proof. The cost of an incremental approach over a big-step approach is that we need to construct more models. But if we manage to achieve a higher degree of automatic proof as a result then the overall cost can be reduced. We believe that we achieved a large reduction in overall effort in our Mondex development than we would otherwise have achieved had we taken a big-step approach. It might seem that having a large refinement chain makes it very difficult to make any changes. In fact our experience was the opposite. By having such a high degree of automatic proof, when making changes to the chain, the cost of reproving the correctness of the chain was small. This was highlighted by the need to fix our modelling error as outlined in Section 7. Of course this ease of modification depends on how big any change is and very large changes might not have such a low reproof cost. In general however, an incremental approach means that we are more likely to be keeping on top of any complexity and that any changes required will be relatively small. We now outline some additional more specific guidelines.

Refining Atomicity : Our approach to refining atomicity is simple but effective. Breaking atomicity involves replacing an abstract atomic event by appropriate sequencing of several concrete events. The technique we used is to match the abstract atomic event to the concrete events whose occurrence represents the achievement of the abstract events. The other events are treated as refinement of skip. For example, the abstract Level 1 *TransferOk* event is matched with the Level 2 *Increase* event, i.e., *Increase* is treated as a refinement of *TransferOk*. This is because the transfer always succeeds once the target side receives a *val* and increases its balance. Often the abstract event is matched to a completion of a transaction or protocol run though this is not necessary. For example, in our case further events can occur after the *Increase* event, i.e., the source side can receive an *ack* or can abort.

A further aspect to note is that a transaction typically has more than one outcome, e.g., success or failure. We deal this by having separate abstract atomic events for the different cases, e.g, *TransferOk* or *TransferFail*. In some cases, an aborting final outcome will be a refinement of skip at the abstract level.

Ensure your formal models are valid: The modelling error outlined in Section 7 highlighted the importance of reviewing all modelling assumptions made in constructing the refinement chain. Our experience from other case studies is that this is facilitated by having good requirements and design documents. The (informal) process of reconciling requirements and design documents with the formal models helps in the discovery of invalid modelling assumptions.

As abstract as possible: When introducing more computational structure in a refinement step, it is good to keep the state representation as abstract as possible. That is, the refined state representation should be sufficiently detailed to allow for the new computational structure to be represented, but no more detailed than that. Splitting the atomicity of the money transfer is an example of what we mean by introducing more computational structure. To do this we introduced the representation of a transaction. Rather than doing this by using the detailed dual transaction states of the parties to a transaction, we instead used the simple end-to-end transaction states. This representation was sufficient to allow us to split the atomicity of the transfer but was much simpler than the more detailed dual state representation.

Redundancy can be useful: At some levels of the chain we found it useful to employ redundant variables. For example the *pendF* variable used in the first refinement (Section 3) was redundant in that its value could be extracted from other variables of the model. However, making it an explicit variable allowed us to simplify the gluing invariant and hence ease the proof. Redundant variables can be removed in subsequent refinement steps and often the proof is completely automatic provided the appropriate extraction invariant is provided.

Let the proof obligations guide you: In Section 5 we outlined how we used the proof obligations and the prover to construct a sufficient invariant to prove the refinement. By using invariants that are close to the form of the proof obligations, the proof effort is usually eased. However, this must not be done blindly. One needs to convince oneself informally that a newly discovered invariant really is expected to be an invariant. In some cases it will become clear from inspecting a proof obligation that it can never be proved and that the models or existing invariants need to be 'fixed'.

Keep separate structures separate: In Sections 3 and 4 we explained how we used disjoint sets to model transaction states. For example, at Level 4 the variable $epr \subseteq trans$ was introduced to represent the set of transactions whose source purse is expecting a request. An alternative representation of the transaction states (like that introduced in Level 9 for purse state) would be a function from transaction to a set of enumerate values:

$$cstatusF \in trans \rightarrow \{IdleF, Epr, AbortEpr, ...\}$$

Now, recall from Section 5 that we needed an invariant specifying that if a transaction at Level 4 is in the *epr* request, then at Level 3 the transaction must be in the idle state. Using the state function representation, this could be expressed in one of the following forms:

$$\forall t \cdot (\ t \in trans \wedge cstatusF(t) = Epr \ \Rightarrow \ astatus(t) = Idle\ ) \tag{1}$$

$$cstatusF^{-1}[\{Epr\}] \ \subseteq \ astatus^{-1}[\{Idle\}] \tag{2}$$

With the disjoint sets representation that we used, the gluing invariant is expressed more simply as follows

$$epr \ \subseteq \ idle \tag{3}$$

Clearly, (3) is simpler than either (1) or (2) and therefore the proof effort is eased. But the disjoint set representation has a further significant property. Consider the $AbortEPA$ event that changes the state of a transaction from the *epa* state to the *abortepa* state. With the disjoint set representation this event will be independent of the *epr* variable and so no proof obligation will arise from invariant (3) for that event. With the state function representation, the $AbortEPA$ event will change the $cstatusF$ variable and therefore will give rise to a proof obligation for invariants (1) and (2). So what is going on here? The single state function is effectively bundling transactions with different states into single function. But to specify the gluing invariant we need to separate these out using some non-trivial expression. By keeping them as separate variables we don't need to use some non-trivial expression and furthermore concepts that are separate are kept separate and less proof obligations result. Once we no longer need to keep them separate for the proof effort, we can bundle the separate variables into a single variables using an appropriate gluing invariant and, as with eliminating redundant variables, the proof is usually automatic. Note that we used a similar technique when the different kinds of messages were modelled as separate variables at Level 5 and then merged into a single 'ether' at Level 10.

9. Concluding

We used the original Z development [10] as a requirements document rather than trying to faithfully reproduce it and we did not use invariants of that development. Unlike the original Z development, we did treat balance recovery and balance checks. The original Z development used backwards data refinement rather than the more usual forwards data refinement (see, for example, [8]). Backwards refinement is typically required when nondeterminism in the abstract model is moved to a later point at the concrete level. In the case of the Z development, the abstract completion events were refined by the start of the transaction at the concrete level. The abstract choice between a balance transfer succeeding or failing is made to correspond to the start of the transaction at the concrete level. At the concrete level success or failure is not known until the completion of a transaction. This arrangement requires the use of backwards refinement. In our development the abstract completion events are mapped to the completion events of the transaction at the concrete level, i.e., nondeterminism is not moved around between levels. This means that forward refinement can be used (and the B provers only support forward refinement).

We have outlined how an incremental refinement approach to the Mondex system allowed us to achieve a very high degree of automatic proof. In addition to the incremental approach, the use of the guidelines outlined in Section 8 also contributed to the high degree of automatic proof. The approach we have taken is not specific to Event-B. We believe a similar approach could be taken using other state-based notations such as ASM, TLA or Z. The powerful support provided by the B4free tool was essential to achieving what we believe was a very successful development. B4free was used to generate the hundreds of proof obligations and to discharge those obligations automatically and interactively. Another key role of the tool was in helping us to discover appropriate gluing invariants to prove the refinements. Without this level of automated support, making the changes to the refinement chain that we did make would have been far too tedious. In summary

some key lessons are that incremental development with small refinement steps, appropriate abstractions at each level and powerful tool support are all invaluable in this kind of formal development

References

[1] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] Jean-Raymond Abrial and Dominique Cansell. Click'n'Prove: Interactive Proofs within Set Theory. In *Theorem Proving in Higher Order Logics*, volume 2758 of *LNCS*, pages 1–24, 2003.

[3] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition and instantiation of discrete models. *Fundamentae Informatica*, 2006. To appear.

[4] Ralph Back. Refinement calculus, part II: Parallel and reactive programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer-Verlag, 1990.

[5] Clearsy. B4free tool homepage. www.b4free.com.

[6] Neil Evans and Michael Butler. A proposal for records in Event-B. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 221–235. Springer, 2006.

[7] Stefan Hallerstede. Justifications for the Event-B modelling notation. In Jacques Julliand and Olga Kouchnarenko, editors, *B*, volume 4355 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 2007.

[8] Jifeng He, C. A. R. Hoare, and Jeff W. Sanders. Data refinement refined. In Bernard Robinet and Reinhard Wilhelm, editors, *ESOP*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer, 1986.

[9] Gerhard Schellhorn, Holger Grandy, Dominik Haneberg, Nina Moebius, and Wolfgang Reif. A systematic verification approach for Mondex electronic purses using ASMs. In *Dagstuhl Seminar on Rigorous Methods for Software Construction and Analysis*, volume (to appear) of *LNCS*, 2007.

[10] Susan Stepney, David Cooper, and Jim Woodcock. An electronic purse specification, refinement, and proof. Technical Report PRG-126, Oxford University Computing Laboratory, 2000. www-users.cs.york.ac.uk/ susan/bib/ss/z/monog.htm.