AN ELECTRONIC PURSE
Specification, Refinement, and Proof

by

Susan Stepney
David Cooper
Jim Woodcock

Oxford University Computing Laboratory
Programming Research Group

AN ELECTRONIC PURSE
Specification, Refinement, and Proof

by

Susan Stepney
David Cooper
Jim Woodcock

Technical Monograph PRG-126
ISBN 0–902928–41–4

July 2000

Oxford University Computing Laboratory
Programming Research Group
Wolfson Building
Parks Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Software Engineering Centre
Wolfson Building
Parks Road
Oxford OX1 3QD
England

email: stepneys@logica.com
cooperd@praxis-cs.co.uk
Jim.Woodcock@comlab.ox.ac.uk

Contents

Acknowledgments

The work described in this monograph took place as part of a development funded by the NatWest Development Team (now *platform seven*).

Part of the refinement work was carried out by Eoin MacDonnell.

Chapter 1

Introduction

1.1 The application

This case study is a reduced version of a real development by the NatWest Development Team (now *platform seven*) of a Smartcard product for electronic commerce. This development was deeply security critical: it was vital to ensure that these cards would not contain any bugs in implementation or design that would allow them to be subverted once in the field.

The system consists of a number of *electronic purses* that carry financial value, each hosted on a Smartcard. The purses interact with each other via a communications device to exchange value. Once released into the field, each purse is on its own: it has to ensure the security of all its transactions without recourse to a central controller. All security measures have to be implemented on the card, with no real-time external audit logging or monitoring.

1.1.1 Models

We develop two key models in this case study. The first is an *abstract* model, describing the world of purses and the exchange of value through atomic transactions, expressing the security properties that the cards must preserve. The second is a *concrete* model, reflecting the design of the purses which exchange value using a message protocol. Both models are described in the Z notation [Spivey 1992b] [Woodcock & Davies 1996] [Barden *et al.* 1994], and we prove that the concrete model is a *refinement* of the abstract.

Abstract model

The abstract model is small, simple, and easy to understand. The key operation

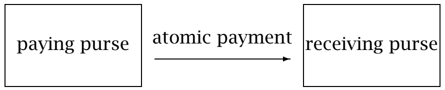

Figure 1.1: An atomic transaction in the abstract model

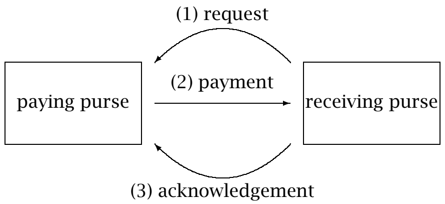

Figure 1.2: Part of the *n*-step protocol used to implement the atomic transaction in the concrete model.

transfers a chosen amount of value from one purse to another; the operation is modelled as an atomic action that simultaneously decrements the value in the paying purse and increments the value in the receiving purse (figure 1.1). Two key system security properties are maintained by this and other operations:

- no value may be created in the system; and
- all value is accounted in the system (no value is lost).

The simplicity of the abstract model allows these properties to be expressed in a way that is easily understood by the client.

Concrete model

The concrete model is rather more complicated, reflecting the details of the real system design. The key changes from the abstract are:

- transactions are no longer atomic, but instead follow an *n*-step protocol (figure 1.2);
- the communications medium is insecure and unreliable;
- transaction logging is added to handle lost messages; and
- there are no global properties—each purse has to be implemented in isolation.

The basic protocol is:

1. the communications device ascertains the transaction to perform;
2. the receiving purse requests the transfer of an amount from the paying purse;
3. the paying purse sends that amount to the receiving purse; and
4. the receiving purse sends an acknowledgement of receipt to the paying purse.

The protocol, although simple in principle, is complicated by several facts: the protocol can be stopped at any point by removing the power from a card; the communications medium could lose a message; and a wire tapper could record a message and play it back to the same or different card later. In the face of all these possible actions, the protocol must implement the atomic transfer of value correctly, as specified in the abstract model.

1.1.2 Proofs

All the security properties of the abstract model are *functional*, and so are preserved by refinement.

The purpose of performing the proof is to give a very high assurance that the chosen design (the protocol) does, indeed, behave just like the abstract, atomic transfers. We choose to do rigorous proofs by hand: our experience is that current proof tools are not yet appropriate for a task of this size. We did, however, type-check the statements of the proof obligations and many of the proof steps using a combination of *f*UZZ [Spivey 1992a] and Formaliser [Flynn *et al.* 1990] [Stepney]. As part of the development process, all proofs were also independently checked by external evaluators.

1.2 Overview of model and proof structure

The specification and security proof have the following structure (summarised in figure 1.3):

- Security Properties, SPs:
  - The *Security Properties* are defined in terms of constraints on secure operations; they are *formalised* in terms of the appropriate model concepts (see later).

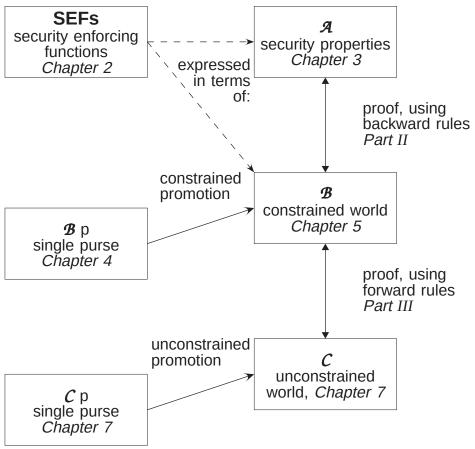

Figure 1.3: Overview of document organisation, with model and proof structure

  - In some cases, where it may not be evident that a model captures a particular constraint, the desired property is recast as a *theorem* and proved.

- Abstract model, $\mathcal{A}$: We define an *abstract model* (Chapter 3), which forms the *Formal Security Policy Model*; it consists of a global model in terms of a simple *state and operations*:
  - the state is a world of (abstract) purses; and
  - the operations are defined on this state.
- Between model, $\mathcal{B}$: Next we build a *'between' levels model*. This is the first refinement towards the implementation of purses consisting of local state information only. This model, $\mathcal{B}$, is structured as a promoted state-and-operations model:
  - The state of a single (concrete) purse, and the corresponding single-purse operations, are defined (Chapter 4).
  - The purses and operations are *promoted* to a global state and operations (Chapter 5). Constraints are put on this promotion to enable the correctness proofs to be performed.

- Concrete model, $\mathcal{C}$: Our final model is the *concrete level model*, which forms the *Formal Architectural Design*. This model, $\mathcal{C}$, is structured as a promoted state-and-operations model, very similar to $\mathcal{B}$, except it has no constraints on the promotion:
  - The state of a single (concrete) purse, and the corresponding single-purse operations, are defined (Chapter 7).
  - The purses and operations are *promoted* to a global state and operations, with *no* constraints (Chapter 7).
- Security proof $\mathcal{A}$–$\mathcal{B}$: The *security policy* is proved to hold for $\mathcal{B}$ by proving that $\mathcal{B}$ is a *refinement* of $\mathcal{A}$. This forms the first part of *Explanation of Consistency*.
  - The retrieve relation *Rab*, relating the $\mathcal{B}$ and $\mathcal{A}$ worlds, is defined (Chapter 10).
  - The *security policy* is shown to hold for $\mathcal{B}$ by proof that $\mathcal{B}$ refines $\mathcal{A}$, using the 'backward' proof rules (Part II). This proof comprises the bulk of the proof work.
- Security proof $\mathcal{B}$–$\mathcal{C}$: The *security policy* is proved to hold for $\mathcal{C}$ by proving that $\mathcal{C}$ is a *refinement* of $\mathcal{B}$ (and hence of $\mathcal{A}$, by transitivity of refinement). This forms the remaining part of *Explanation of Consistency*.
  - The retrieve relation *Rbc*, relating the $\mathcal{C}$ and $\mathcal{B}$ worlds, is defined (Chapter 26).
  - The *security policy* is shown to hold for $\mathcal{C}$ by proof that $\mathcal{C}$ refines $\mathcal{B}$, using the 'forward' proof rules (Part III). These two levels are relatively close, so this proof is relatively straightforward.

The mathematical operators and schemas defined in this document are included in the index at the end of the document.

1.3 Rationale for model structure

As noted above, this case study has been adapted from a larger, real development. In order to produce a case study of a size appropriate for public presentation, much of the real functionality has had to be removed. Some of the structure of the larger specification has remained present in the smaller one, although it might not have been used had the smaller specification been written from scratch. This omitted functionality, whilst important from a business perspective, is peripheral to the central security requirements.

1.4 Rationale for proof structure

Imagine two specifications $\mathcal{A}$ and C, which describe executable machines. Imagine that, on every step, each machine consumes an input and produces an output. Finally, imagine that every execution of C, viewed solely in terms of inputs and outputs, could equally well have been an execution of $\mathcal{A}$. In this sense, $\mathcal{A}$ can *simulate* any behaviour of C. If this is the case, then we say that C is a *refinement* of $\mathcal{A}$.

This is exactly what we want to prove in our case study: that the concrete model is a refinement of the abstract one.

Refinement is an ordering between specifications that captures an intuitive notion of when a concrete specification implements an abstract one. This allows us to postpone implementation detail in writing our top-level specification, focussing only on essential properties. The cost of this abstraction is the need to refine the specification, reifying data structures and algorithms; refinement is a formal technique for ensuring that essential properties are present in a more concrete specification.

Nondeterminism is used in an abstract specification to describe alternative acceptable behaviours; in choosing a concrete refinement of an abstract specification, some of these nondeterministic choices may be resolved. Since we view $\mathcal{A}$ and C only terms of inputs and outputs, nondeterminism present in $\mathcal{A}$ may be resolved at a different point in C.

Our abstract model, chosen to represent the difference between secure and insecure transactions very clearly, has nondeterminism in a different place from the implementation. In fact, it has it in a place that precludes proof using the forward rules of [Spivey 1992b, section 5.6]. For this reason we use the backward rules to prove against the abstract model.

At the concrete level, we must describe the purse behaviour in a way that closely mirrors the actual design. An important (and obvious) property of the design is that the purses are *independent*, that is, each purse acts on the basis of its own, local knowledge, and we have no control over the communications medium between purses. This can be expressed cleanly in Z by building a model of an individual purse in isolation, and then *promoting* [Barden *et al.* 1994, chapter 19] this model to a world with many purses. To express the fact that we have no global control over the purses nor over the communications medium, we must use an unconstrained promotion. This we do in the C model.

Why do we not, then, do a single backward proof step from the $\mathcal{A}$ model to the C model?

For technical reasons, the backward proof rules need the more concrete specification to be tightly constrained in its state space. The form of the proofs forces the description of the state space to include explicit predicates excluding all but valid states. However, these predicates are not expressible locally to purses, and hence cannot be included in specification derived by unconstrained promotion. That is, we cannot express the predicates needed for the proof in the C model.

We therefore introduce an intermediate model, the $\mathcal{B}$ model, which is a *constrained* promotion, and hence can contain the predicates needed for the backward proofs. We then prove a refinement from $\mathcal{A}$ to $\mathcal{B}$ using the backward rules. But now the constrained promotion $\mathcal{B}$ is very close to the unconstrained promotion C, and in particular the nondeterminism is resolved in the same place in both models, allowing the forward rules to be used. This we do in our proof of refinement from $\mathcal{B}$ to C.

1.5 Status

The specification and theorems have been parsed and type-checked using *f*UZZ [Spivey 1992a]. There is no use of the `%%unchecked` parser directive in the specification, in the statement of theorems, or in the statement of most of the intermediate goals; however, some reasoning steps have hidden declarations to make them type-check and some do not conform to *f*UZZ's syntax at all.

Part I

Models

Chapter 2

Security Properties

2.1 Introduction

This chapter gathers together the Security Properties (SPs) definitions, for reference. The SPs are formalised in terms of the abstract and concrete models, making use of definitions in Chapters 3 and 4. (The index can be used to find the definitions of these terms.) The full meaning and effect of a SP can be seen only in the context of the model that includes it.

2.2 Abstract model SPs

The following SPs are expressed in terms of the abstract model $\mathcal{A}$, defined in chapter 3.

2.2.1 No value creation

Security Property 1. *No value may be created in the system: the sum of all the purses' balances does not increase.*[1]

$$\begin{array}{|l}
\textit{NoValueCreation} \\
\hline
\Delta \textit{AbWorld} \\
\hline
\textit{totalAbBalance}\ \textit{abAuthPurse}' \leq \textit{totalAbBalance}\ \textit{abAuthPurse} \\
\hline
\end{array}$$

[1] Proved to hold for the model, section 2.4. *NoValueCreation* requires that the sum of the before balances is greater or equal to the sum of the after balances. The abstract model enforces a stronger condition: that transfers change only the purses involved in the transfer and only by the amount stated in the transfer.

2.2.2 All value accounted

Security Property 2.1. *All value must be accounted for in the system: the sum of all purses' balances and lost components does not change.*[2]

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AllValueAccounted}} \\
\Delta \textit{AbWorld} \\
\hline
\textit{totalAbBalance}\ \textit{abAuthPurse}' + \textit{totalLost}\ \textit{abAuthPurse}' = \\
\quad \textit{totalAbBalance}\ \textit{abAuthPurse} + \textit{totalLost}\ \textit{abAuthPurse} \\
\hline
\end{array}$$

2.2.3 Authentic purses

Security Property 3. *A transfer can occur only between authentic purses.*[3]

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Authentic}} \\
\textit{AbWorld} \\
\textit{name}? : \textit{NAME} \\
\hline
\textit{name}? \in \operatorname{dom} \textit{abAuthPurse} \\
\hline
\end{array}$$

2.2.4 Sufficient funds

Security Property 4. *A transfer can occur only if there are sufficient funds in the from-purse.* [4]

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{SufficientFundsProperty}} \\
\textit{AbWorld} \\
\textit{TransferDetails}? \\
\hline
\textit{value}? \leq (\textit{abAuthPurse}\ \textit{from}?).\textit{balance} \\
\hline
\end{array}$$

2.3 Concrete model SPs

The following SPs are expressed in terms of the between (and concrete) model $\mathcal{B}$, defined in chapter 4.

[2] Proved to hold for the model, section 2.4. The concrete level SP 2.2 uses logging to support this SP.

[3] Used in the definition of: *AbTransferOkay* and *AbTransferLost*, section 3.3.3.

[4] Used in the definition of: *AbTransferOkay* and *AbTransferLost*, section 3.3.3. Used in the proof of: SP1, section 2.4.1, section 2.4.3; SP2, section 2.4.2, section 2.4.4. Note that the model also ensures that the *balance* and *value*? are non-negative.

2.3.1 Exception logging

Security Property 2.2. *If a purse aborts a transfer at a point where value could be lost, then the purse logs the details.*[5]

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{LogIfNecessary}} \\
\Delta \textit{ConPurse} \\
\hline
\textit{exLog}' = \textit{exLog} \cup (\mathsf{if}\ \textit{status} \in \{\textit{epv}, \textit{epa}\} \mathsf{then} \{\textit{pdAuth}\} \mathsf{else} \varnothing) \\
\hline
\end{array}$$

The only times the log need be updated are if the purse is in *epv* (having sent the *req* message) or in *epa* (having sent the *val* but not yet received the *ack*). In all other cases the transfer has not yet got far enough for the purse to be worried that the transfer has failed, or has got far enough that the purse is happy that the transfer has succeeded.

2.4 SPs and the models

All the SPs hold in the appropriate models.

In most cases, this is obviously true, by construction: the SPs appear as explicit predicates in the relevant definitions. However, *NoValueCreation* and *AllValueAccounted* are not explicitly included in the operation that changes the relevant components: *AbTransfer*. In this section, we demonstrate that the abstract model indeed satisfies these SPs. That is:

$$\textit{AbTransferOkay} \vdash \textit{NoValueCreation} \land \textit{AllValueAccounted}$$

$$\textit{AbTransferLost} \vdash \textit{NoValueCreation} \land \textit{AllValueAccounted}$$

$$\textit{AbIgnore} \vdash \textit{NoValueCreation} \land \textit{AllValueAccounted}$$

In the proofs below, we use the *TD* form of the definitions, by [*cut*]ting in the appropriate *TransferDetails*.

2.4.1 Transfer okay, no value creation

$$\textit{AbTransferOkayTD} \vdash \textit{NoValueCreation}$$

[5] Used in the definition of: *AbortPurse*, section 4.8.2.

Proof:

$$
\begin{aligned}
& \mathit{totalAbBalance}\ \mathit{abAuthPurse}' \\
& \quad = \mathit{totalAbBalance}(\{\mathit{from}?, \mathit{to}?\} \ntriangleleft \mathit{abAuthPurse}') \\
& \qquad + (\mathit{abAuthPurse}'\ \mathit{from}?).\mathit{balance} \\
& \qquad + (\mathit{abAuthPurse}'\ \mathit{to}?).\mathit{balance} && [\mathit{totalAbBalance}] \\[1ex]
& \quad = \mathit{totalAbBalance}(\{\mathit{from}?, \mathit{to}?\} \ntriangleleft \mathit{abAuthPurse}) \\
& \qquad + ((\mathit{abAuthPurse}\ \mathit{from}?).\mathit{balance} - \mathit{value}?) \\
& \qquad + ((\mathit{abAuthPurse}\ \mathit{to}?).\mathit{balance} + \mathit{value}?) && [\mathit{AbTransferOkay}] \\[1ex]
& \quad = \mathit{totalAbBalance}\ \mathit{abAuthPurse} \\
& \quad \leq \mathit{totalAbBalance}\ \mathit{abAuthPurse}
\end{aligned}
$$

■ 2.4.1

2.4.2 Transfer okay, all value accounted

$\mathit{AbTransferOkayTD} \vdash \mathit{AllValueAccounted}$

Proof:

$$
\begin{aligned}
& \mathit{totalAbBalance}\ \mathit{abAuthPurse}' + \mathit{totalLost}\ \mathit{abAuthPurse}' \\
& \quad = \mathit{totalAbBalance}(\{\mathit{from}?, \mathit{to}?\} \ntriangleleft \mathit{abAuthPurse}') \\
& \qquad\quad + (\mathit{abAuthPurse}'\ \mathit{from}?).\mathit{balance} \\
& \qquad\quad + (\mathit{abAuthPurse}'\ \mathit{to}?).\mathit{balance} && [\mathit{totalAbBalance}] \\
& \qquad + \mathit{totalLost}(\{\mathit{from}?, \mathit{to}?\} \ntriangleleft \mathit{abAuthPurse}') \\
& \qquad\quad + (\mathit{abAuthPurse}'\ \mathit{from}?).\mathit{lost} \\
& \qquad\quad + (\mathit{abAuthPurse}'\ \mathit{to}?).\mathit{lost} && [\mathit{totalLost}] \\[1ex]
& \quad = \mathit{totalAbBalance}(\{\mathit{from}?, \mathit{to}?\} \ntriangleleft \mathit{abAuthPurse}) \\
& \qquad\quad + ((\mathit{abAuthPurse}\ \mathit{from}?).\mathit{balance} - \mathit{value}?) \\
& \qquad\quad + ((\mathit{abAuthPurse}\ \mathit{to}?).\mathit{balance} + \mathit{value}?) \\
& \qquad + \mathit{totalLost}(\{\mathit{from}?, \mathit{to}?\} \ntriangleleft \mathit{abAuthPurse}) \\
& \qquad\quad + (\mathit{abAuthPurse}\ \mathit{from}?).\mathit{lost} \\
& \qquad\quad + (\mathit{abAuthPurse}\ \mathit{to}?).\mathit{lost} && [\mathit{AbTransferOkay}] \\
& \quad = \mathit{totalAbBalance}\ \mathit{abAuthPurse} + \mathit{totalLost}\ \mathit{abAuthPurse}
\end{aligned}
$$

■ 2.4.2

2.4.3 Transfer lost, no value creation

$\mathit{AbTransferLostTD} \vdash \mathit{NoValueCreation}$

Proof:

$$
\begin{aligned}
& \mathit{totalAbBalance}\ \mathit{abAuthPurse}' \\
& \quad = \mathit{totalAbBalance}(\{\mathit{from}?, \mathit{to}?\} \ntriangleleft \mathit{abAuthPurse}') \\
& \qquad + (\mathit{abAuthPurse}'\ \mathit{from}?).\mathit{balance} \\
& \qquad + (\mathit{abAuthPurse}'\ \mathit{to}?).\mathit{balance} && [\mathit{totalAbBalance}] \\[1ex]
& \quad = \mathit{totalAbBalance}(\{\mathit{from}?, \mathit{to}?\} \ntriangleleft \mathit{abAuthPurse}) \\
& \qquad + ((\mathit{abAuthPurse}\ \mathit{from}?).\mathit{balance} - \mathit{value}?) \\
& \qquad + (\mathit{abAuthPurse}\ \mathit{to}?).\mathit{balance} && [\mathit{AbTransferLost}] \\[1ex]
& \quad = \mathit{totalAbBalance}\ \mathit{abAuthPurse} - \mathit{value}? && [\mathit{totalAbBalance}] \\
& \quad \leq \mathit{totalAbBalance}\ \mathit{abAuthPurse}
\end{aligned}
$$

■ 2.4.3

2.4.4 Transfer lost, all value accounted

$\mathit{AbTransferLostTD} \vdash \mathit{AllValueAccounted}$

Proof:

$$
\begin{aligned}
& \mathit{totalAbBalance}\ \mathit{abAuthPurse}' + \mathit{totalLost}\ \mathit{abAuthPurse}' \\
& \quad = \mathit{totalAbBalance}(\{\mathit{from}?, \mathit{to}?\} \ntriangleleft \mathit{abAuthPurse}') \\
& \qquad\quad + (\mathit{abAuthPurse}'\ \mathit{from}?).\mathit{balance} \\
& \qquad\quad + (\mathit{abAuthPurse}'\ \mathit{to}?).\mathit{balance} && [\mathit{totalAbBalance}] \\
& \qquad + \mathit{totalLost}(\{\mathit{from}?, \mathit{to}?\} \ntriangleleft \mathit{abAuthPurse}') \\
& \qquad\quad + (\mathit{abAuthPurse}'\ \mathit{from}?).\mathit{lost} \\
& \qquad\quad + (\mathit{abAuthPurse}'\ \mathit{to}?).\mathit{lost} && [\mathit{totalLost}] \\[1ex]
& \quad = \mathit{totalAbBalance}(\{\mathit{from}?, \mathit{to}?\} \ntriangleleft \mathit{abAuthPurse}) \\
& \qquad\quad + ((\mathit{abAuthPurse}\ \mathit{from}?).\mathit{balance} - \mathit{value}?) \\
& \qquad\quad + (\mathit{abAuthPurse}\ \mathit{to}?).\mathit{balance} \\
& \qquad + \mathit{totalLost}(\{\mathit{from}?, \mathit{to}?\} \ntriangleleft \mathit{abAuthPurse}) \\
& \qquad\quad + ((\mathit{abAuthPurse}\ \mathit{from}?).\mathit{lost} + \mathit{value}?) \\
& \qquad\quad + (\mathit{abAuthPurse}\ \mathit{to}?).\mathit{lost} && [\mathit{AbTransferLost}] \\
& \quad = \mathit{totalAbBalance}\ \mathit{abAuthPurse} + \mathit{totalLost}\ \mathit{abAuthPurse}
\end{aligned}
$$

■ 2.4.4

2.4.5 Transfer ignore

$AbIgnore \vdash NoValueCreation \land AllValueAccounted$

Proof:
Follows directly from the definition of *AbIgnore*, which changes none of the relevant values.

■ 2.4.5
■ 2.4
■ 2

Chapter 3

Abstract model: security policy

3.1 Introduction

The abstract model specification has the following parts:

- State: the abstract world of purses
- Operations: secure changes from one abstract state to another
- Initialisation: the abstract world starts off secure
- Finalisation: a way of observing part of the abstract world to determine that it is secure

3.2 The abstract state

3.2.1 A purse

An abstract *AbPurse* consists of a *balance*, the value stored in the purse; and a *lost* component, the total value lost during unsuccessful transfers. (The unsuccessful, but still secure, transfer is defined in section 3.3.3.)

$AbPurse \mathrel{\widehat=} [\ balance, lost : \mathbb{N}\ ]$

3.2.2 Transfer details

Each purse has a distinct, unique name.

$[NAME]$

The details of a particular transfer include the names of the *from* and *to* purses and the value to be transferred.

$$
\begin{array}{|l}
\textit{TransferDetails} \\
\hline
\textit{from}, \textit{to} : \textit{NAME} \\
\textit{value} : \mathbb{N} \\
\hline
\end{array}
$$

Although it is not permitted to perform a transfer between a purse and itself, the constraint $\textit{from} \neq \textit{to}$ is checked during *AbTransfer*, rather than put in *TransferDetails*, since it is permitted to *request* a transfer with $\textit{from} = \textit{to}$.

Transactions involving zero value are allowed.

3.2.3 Abstract world

The abstract world model contains a mapping from purse names to abstract purses. The domain of this function corresponds to authentic purses, those that may engage in transfers[1]. We allow only a finite number of authentic purses, to ensure a well-defined total value in the system.

$$\textit{AbWorld} \mathrel{\widehat{=}} [\, \textit{abAuthPurse} : \textit{NAME} \twoheadrightarrow \textit{AbPurse} \,]$$

3.3 Secure operations

Having defined our abstract world, *AbWorld*, we now define operations on the world that respect the relevant SPs. We call these *secure operations*. They comprise:

- *AbIgnore*: securely do nothing
- *AbTransfer*: securely transfer balance between purses, or securely 'lose' the balance

3.3.1 Abstract inputs and outputs

We are to prove that the implementation is a refinement of the abstract security policy specification. This is made simpler if every operation has an input and an output, and if all operations' inputs and outputs are of the same type.

So we define the inputs and outputs (some being 'dummy' values) using a free type construct:

$$
\begin{array}{l}
\textit{AIN} ::= \textit{aNullIn} \\
\qquad\quad \mid \ \textit{transfer}\langle\!\langle \textit{TransferDetails} \rangle\!\rangle
\end{array}
$$

[1] SP 3, 'Authentic purses', section 2.2.3.

$$\textit{AOUT} ::= \textit{aNullOut}$$

Every abstract operation has the following properties:

$$
\begin{array}{|l}
\textit{AbOp} \\
\hline
\Delta \textit{AbWorld} \\
a? : \textit{AIN};\ a! : \textit{AOUT} \\
\hline
a! = \textit{aNullOut} \\
\hline
\end{array}
$$

The output is always *aNullOut* (that is, we are not interested in the abstract output).

3.3.2 Abstract ignore

Any operation has the option of securely doing nothing.

$$
\begin{array}{|l}
\textit{AbIgnore} \\
\hline
\textit{AbOp} \\
\hline
\textit{abAuthPurse}' = \textit{abAuthPurse} \\
\hline
\end{array}
$$

3.3.3 Transfer

The transfer operation changes only the balance and lost component of the relevant purses.

$$\textit{AbPurseTransfer} \mathrel{\widehat{=}} \textit{AbPurse} \setminus (\textit{balance}, \textit{lost})$$

The secure transfer operations change at most the *from* and *to* purse states: all other purse states are unchanged.

$$
\begin{array}{|l}
\textit{AbWorldSecureOp} \\
\hline
\textit{AbOp} \\
\textit{TransferDetails}? \\
\hline
a? \in \operatorname{ran} \textit{transfer} \\
\theta \textit{TransferDetails}? = \textit{transfer}^{\sim} a? \\
\{\textit{from}?, \textit{to}?\} \mathbin{\lhd\!\!\!-} \textit{abAuthPurse}' = \{\textit{from}?, \textit{to}?\} \mathbin{\lhd\!\!\!-} \textit{abAuthPurse} \\
\hline
\end{array}
$$

A transfer can securely succeed between two purses if they are distinct, both purses are authentic[2], and the *from* purse has sufficient funds[3].

$$
\begin{array}{|l}
\textit{AbTransferOkayTD} \\
\hline
\textit{AbWorldSecureOp} \\
\hline
\textit{Authentic}[\,\textit{from}?/\textit{name}?\,] \\
\textit{Authentic}[\,\textit{to}?/\textit{name}?\,] \\
\textit{SufficientFundsProperty} \\[4pt]
\textit{to}? \neq \textit{from}? \\[4pt]
\textit{abAuthPurse}'\ \textit{from}? = (\,\mu\, \Delta \textit{AbPurse} \mid \\
\qquad\quad \theta \textit{AbPurse} = \textit{abAuthPurse}\ \textit{from}? \\
\qquad\quad \land\ \textit{balance}' = \textit{balance} - \textit{value}? \\
\qquad\quad \land\ \textit{lost}' = \textit{lost} \\
\qquad\quad \land\ \Xi \textit{AbPurseTransfer} \\
\quad \bullet\ \theta \textit{AbPurse}'\,) \\[4pt]
\textit{abAuthPurse}'\ \textit{to}? = (\,\mu\, \Delta \textit{AbPurse} \mid \\
\qquad\quad \theta \textit{AbPurse} = \textit{abAuthPurse}\ \textit{to}? \\
\qquad\quad \land\ \textit{balance}' = \textit{balance} + \textit{value}? \\
\qquad\quad \land\ \textit{lost}' = \textit{lost} \\
\qquad\quad \land\ \Xi \textit{AbPurseTransfer} \\
\quad \bullet\ \theta \textit{AbPurse}'\,) \\
\hline
\end{array}
$$

The operation transfers *value*? from the *from* purse to the *to* purse[4]. All the other components of the *from*? and *to*? purses are unchanged, and all other purses are unchanged.

The model is more constrained than required by the SPs, and hence it represents a sufficient, but not necessary, behaviour to conform to the SPs.

Hiding the auxiliary inputs gives the *Okay* operation as:

$$\textit{AbTransferOkay} \mathrel{\widehat=} \textit{AbTransferOkayTD} \setminus (\textit{to}?, \textit{from}?, \textit{value}?)$$

A transfer can securely lose value between two purses if they are distinct, both purses are authentic[5], and the *from* purse has sufficient funds[6].

[2] SP 3, 'Authentic purses', section 2.2.3.
[3] SP 4, 'Sufficient funds', section 2.2.4.
[4] SP 1, 'No value created', section 2.2.1.
[5] SP 3, 'Authentic purses', section 2.2.3.
[6] SP 4, 'Sufficient funds', section 2.2.4.

$$
\begin{array}{|l}
\textit{AbTransferLostTD} \\
\hline
\textit{AbWorldSecureOp} \\
\hline
\textit{Authentic}[\,\textit{from}?/\textit{name}?\,] \\
\textit{Authentic}[\,\textit{to}?/\textit{name}?\,] \\
\textit{SufficientFundsProperty} \\[4pt]
\textit{to}? \neq \textit{from}? \\[4pt]
\textit{abAuthPurse}'\ \textit{from}? \in \{\, \Delta \textit{AbPurse} \mid \\
\qquad\quad \theta \textit{AbPurse} = \textit{abAuthPurse}\ \textit{from}? \\
\qquad\quad \land\ \textit{balance}' = \textit{balance} - \textit{value}? \\
\qquad\quad \land\ \textit{lost}' = \textit{lost} + \textit{value}? \\
\qquad\quad \land\ \Xi \textit{AbPurseTransfer} \\
\quad \bullet\ \theta \textit{AbPurse}'\,\} \\[4pt]
\textit{abAuthPurse}'\ \textit{to}? = \textit{abAuthPurse}\ \textit{to}? \\
\hline
\end{array}
$$

The operation removes *value*? from the *from* purse's balance,[7] and adds it to the *from* purse's *lost* component.[8] All the other components of the *from*? purse are unchanged, The *to* purse and all other purses are unchanged.

Hiding the auxiliary inputs gives the *Okay* operation as:

$$\textit{AbTransferLost} \mathrel{\widehat=} \textit{AbTransferLostTD} \setminus (\textit{to}?, \textit{from}?, \textit{value}?)$$

The full transfer operation can also securely do nothing, *AbIgnore*. The full transfer operation is

$$\textit{AbTransfer} \mathrel{\widehat=} \textit{AbTransferOkay} \lor \textit{AbTransferLost} \lor \textit{AbIgnore}$$

3.4 Abstract initial state

One conventional definition of the initial state of a system is as being empty; operations are used to add elements to the state until the desired configuration is reached. However, we do not wish to add new abstract purses to the domain of *abAuthPurse*, so we cannot start with a system containing no authentic purses. So we set up an arbitrary initial state, which satisfies the predicate of *AbWorld*′.

$$\textit{AbInitState} \mathrel{\widehat=} \textit{AbWorld}'$$

[7] SP 1, 'No value created', section 2.2.1.
[8] SP 2, 'All value accounted', section 2.2.2.

So we say that *AbInitState* has some particular value, we just do not say what that particular value *is*. The particular value chosen is irrelevant to the security of the system; any starting state would be secure.

Initialisation also defines the mapping from global (that is, observable) inputs to abstract (that is, modelled) inputs. This is just the identity relation in the $\mathcal{A}$ model:

$$AbInitIn \mathrel{\widehat=} [\, a?, g? : AIN \mid a? = g? \,]$$

3.5 Abstract finalisation

We must 'observe' each security relevant component of the world, in order to determine that the security properties do indeed hold. Observation is usually performed by enquiry operations, and any part of the state not visible through some enquiry operation is deemed unimportant. However, in our case there are no abstract enquiry operations to observe state components, but there are security properties related to them, and so they *are* important. We use *finalisation* to observe them.

Finalisation takes an abstract state, and 'projects out' the portion of it we wish to observe, into a global state. Here we choose to observe the entire abstract state.

The global state is the same as the abstract state:

$$GlobalWorld \mathrel{\widehat=} [\, gAuthPurse : NAME \twoheadrightarrow AbPurse \,]$$

Finalisation gives the global state corresponding to an abstract state. These are mostly the identity relations in the $\mathcal{A}$ model:

$$\begin{array}{|l}
\textit{AbFinState} \\
\hline
AbWorld \\
GlobalWorld \\
\hline
gAuthPurse = abAuthPurse \\
\hline
\end{array}$$

Finalisation also defines the mapping from abstract outputs to global (that is, observable) outputs.

$$AbFinOut \mathrel{\widehat=} [\, a!, g! : AOUT \mid a! = g! \,]$$

Chapter 4

Between model, single purse operations

4.1 Overview

This chapter covers the purse-level operations, which are: abort, the start operations, the transfer operations *req*, *val* and *ack*, read log, and clear log.

For the sake of simplicity, we assume that concrete and abstract *NAME*s are drawn from the same sets.

In this section we refer to 'concrete' rather than 'between' purse, because, as we see later, there is no difference between the two structurally. The only difference between the $\mathcal{B}$ and C worlds is fewer global constraints in the latter.

4.2 Status

A concrete purse has a *status*, which records its progress through a transaction.

$$STATUS ::= eaFrom \mid eaTo \mid epr \mid epv \mid epa$$

The statuses are: *eaFrom* 'expecting any payer', *eaTo* 'expecting any payee', *epr* 'expecting payment req', *epv* 'expecting payment val', and *epa* 'expecting payment ack'.

4.3 Message Details

The abstract level describes the operations that transfer value. Purses are sent instructions via messages, and we present the structure of compound messages in this section.

The abstract level describes a transfer of value from one purse to another. We implement this at the concrete level by a protocol consisting of messages.

- A single transfer involves many messages. So we need a way to distinguish messages: we use a tag for *req*, *val* or *ack*.
- We have no control over the concrete messages, and cannot forbid the duplication of messages. So we need a way to distinguish separate transactions: we use sequence numbers that are increased between transactions. (The transaction sequence number is implemented as a sufficiently large number. Provided that the initial sequence number is quite small, and each increment is small, we need not worry about overflow, since the purse will physically wear out first.

4.3.1 Start message counterparty details

The counterparty details of a payment, which are transmitted with a *start* message, identify the other purse, the *value* to be transferred, and the other purse's transaction sequence number.

```
┌─ CounterPartyDetails ─────────────────────
│ name : NAME
│ value : ℕ
│ nextSeqNo : ℕ
└───────────────────────────────────────────
```

4.3.2 Payment log message details

Purses store current payment details, and exception log records that hold sufficient information about failed or problematic transactions to reconstruct the value lost in the transfer[1]. The payment log details identify the different *from* and *to* purses and the *value* to be transferred (as in the abstract *TransferDetails*) and also the purses' transaction sequence numbers. The combination of purse name and sequence number *uniquely* identifies the transaction.

```
┌─ PayDetails ──────────────────────────────
│ TransferDetails
│
│ fromSeqNo, toSeqNo : ℕ
├──────────
│ from ≠ to
└───────────────────────────────────────────
```

[1] Concrete SP 2.2, 'Exception logging', section 2.3.1.

We can put the constraint about distinct purses in the *PayDetails*, because this check is made in *ValidStartTo*/*From*, before the details are set up.

4.4 Clear Exception Log Validation

CLEAR is the set of clear codes for purse exception logs.

[*CLEAR*]

A clear code is provided by an external source (section 5.7.1) in order to clear a purse's exception log (section 4.10.2).

image is a function to calculate the clear code for a given non-empty set of exception records.

$$image : \mathbb{P}_1\ PayDetails \rightarrowtail CLEAR$$

image takes a set of exception logs, and produces another value used to validate a log clear command. For each set of *PayDetails*, there is a unique clear code.

The *BetweenWorld* model is designed so that no logs are ever lost. Indeed, we must prove that no logs are lost in the refinement of each operation — this is an implicit part of the refinement correctness proofs. The *BetweenWorld* mechanism to ensure that no logs are lost relies on two assumptions:

- clear codes are only ever generated from sets of *PayDetails* that are stored in the *archive* (a secure store of log records introduced later)
- clear codes unambiguously identify sets of *PayDetails*

The second of these assumptions is captured formally by the *injective* function *image*[2].

[2] In practice, *image* is not injective on general sets of *PayDetails*, but it *is* injective when restricted to the sets actually encountered.

4.5 Messages

There are various kinds of messages:

$$
\begin{aligned}
MESSAGE ::=\ & startFrom\langle\langle CounterPartyDetails\rangle\rangle \\
\mid\ & startTo\langle\langle CounterPartyDetails\rangle\rangle \\
\mid\ & readExceptionLog \\
\mid\ & req\langle\langle PayDetails\rangle\rangle \\
\mid\ & val\langle\langle PayDetails\rangle\rangle \\
\mid\ & ack\langle\langle PayDetails\rangle\rangle \\
\mid\ & exceptionLogResult\langle\langle NAME \times PayDetails\rangle\rangle \\
\mid\ & exceptionLogClear\langle\langle NAME \times CLEAR\rangle\rangle \\
\mid\ & \perp
\end{aligned}
$$

The first group of messages may be unprotected. Their forgeability is modelled by having them all present in the initial message ether (see section 6.1).

The second group of messages are all that need to be cryptographically protected. Their unforgeability is modelled by having them added to the message ether only by specified operations.

$\perp$, 'forged', is a message emitted by operations that ignore the (irrelevant) input message, or emitted by non-authentic purses. It is also the input message to the *Ignore*, *Increase* and *Abort* operations. $\perp$ is implemented as an unprotected status message, as an error message, as a 'forged' message, or as 'silence'. As far as the model is concerned, we choose not to distinguish these messages from each other, only from the other distinguished ones. (See also section 5.8.)

A complete payment transaction is made up of a *startFrom*, *startTo*, *req*, *val*, and *ack* message.

4.6 A concrete purse

A concrete purse has a current balance, an exception log for recording failed or problematic transfers, a name, a transaction sequence number to be used for the next transaction, the payment details of the current transaction, and a status indicating the purse's position in the current transaction.

ConPurse

$$
\begin{aligned}
& balance : \mathbb{N} \\
& exLog : \mathbb{P}\, PayDetails \\
& name : NAME \\
& nextSeqNo : \mathbb{N} \\
& pdAuth : PayDetails \\
& status : STATUS \\
\hline
& \forall pd : exLog \bullet name \in \{pd.from, pd.to\} \\
& status = epr \Rightarrow name = pdAuth.from \\
& \qquad \wedge\ pdAuth.value \leq balance \\
& \qquad \wedge\ pdAuth.fromSeqNo < nextSeqNo \\
& status = epv \Rightarrow pdAuth.toSeqNo < nextSeqNo \\
& status = epa \Rightarrow pdAuth.fromSeqNo < nextSeqNo
\end{aligned}
$$

The name is included in the purse's state so that the purse itself can check it is the correct purse for this transaction.

The predicate on the purse state records the following constraints:

P–1 $\forall pd : exLog \bullet name \in \{pd.from, pd.to\}$
All log details in the exception log refer to this purse, as the *from* or the *to* party[3].

P–2 $status = epr \Rightarrow$
$name = pdAuth.from$
$\wedge\ pdAuth.value \leq balance$
$\wedge\ pdAuth.fromSeqNo < nextSeqNo$
If the purse is expecting a payment request, then:

(a) it is the *from* purse of the current transaction[4].
(b) it has sufficient funds for the request [5] (this condition is required because there is no check for sufficient funds on receipt of the request)
(c) its next sequence number is greater than the current transaction's sequence number[6]

P–3 $status = epv \Rightarrow pdAuth.toSeqNo < nextSeqNo$

[3] Used in: *AuxWorld* does not add constraints, section 5.2.1.
[4] Used in: *CReq*, B–9, section 29.4.
[5] Used in: *Req*, case 1, *SufficientFundsProperty*, section 18.7.2; *Req*, case 2, *SufficientFundsProperty*, section 18.8.2; *Req*, case 3, *SufficientFundsProperty*, section 18.9.2.
[6] Used in: *CReq*, B–3, section 29.4.

If the purse is expecting a payment value, then its next sequence number is greater than the current transaction's sequence number[7]

P–4 $status = epa \Rightarrow pdAuth.fromSeqNo < nextSeqNo$
If the purse is expecting a payment acknowledgement, then its next sequence number is greater than the current transaction's sequence number[8]

4.7 Single Purse operations

4.7.1 Overview

The concrete purse specification is structured around the various purse-level operations:

- invisible operations
  - *IncreasePurse*
  - *AbortPurse*
- value transfer operations
  - *StartFromPurse*
  - *StartToPurse*
  - *ReqPurse*
  - *ValPurse*
  - *AckPurse*
- exception logging operations
  - *ReadExceptionLogPurse*
  - *ClearExceptionLogPurse*

4.8 Invisible operations

Several concrete operations have a common effect on the state visible in the model (they affect only implementation state not visible in the model).

[7]Used in: *CAbort*, B–6, section 28.5.
[8]Used in: *CAbort*, B–5, section 28.5.

4.8.1 Increase Purse

The *IncreasePurseOkay* operation is used to model actual purse operations that do not have any effect on the state visible in this model, except for increasing the sequence number.

In a simple increase transaction, only the purse's sequence number may change. All other components remain unchanged.

$$ConPurseIncrease \mathrel{\widehat{=}} ConPurse \setminus (nextSeqNo)$$

IncreasePurseOkay

$\Delta ConPurse$
$m?, m! : MESSAGE$

$\Xi ConPurseIncrease$
$nextSeqNo' \geq nextSeqNo$
$m! = \perp$

4.8.2 Abort Purse

The *AbortPurseOkay* operation is used to model actual purse operations that do not have any effect on the state visible in this model, but that abort and log incomplete transactions.

In a simple abort transaction, only the purse's sequence number, exception log, *pdAuth* and status may change. All other components remain unchanged.

$$ConPurseAbort \mathrel{\widehat{=}} ConPurse \setminus (nextSeqNo, exLog, pdAuth, status)$$

AbortPurseOkay places the purse in status *eaFrom* (where the *pdAuth* component is undefined), logging any incomplete transactions if necessary[9]. No other component of the purse is altered, except for *nextSeqNo*, which may increase arbitrarily.

[9]Concrete SP 2.2, 'Exception logging', section 2.3.1.

$$
\begin{array}{|l}
\textit{AbortPurseOkay} \\
\hline
\Delta ConPurse \\
m?, m! : MESSAGE \\
\hline
\Xi ConPurseAbort \\
LogIfNecessary \\[4pt]
status' = eaFrom \\
nextSeqNo' \geq nextSeqNo \\
\hline
\end{array}
$$

We do not, at this stage, put any restrictions on the output message $m!$. Later, we either compose *AbortPurseOkay* with another operation, using the latter's $m!$, or we promote *AbortPurseOkay* to the world level, where we define $m! = \perp$.

4.9 Value transfer operations

The *StartTo* and *StartFrom* operations, when starting from *eaFrom*, change only the sequence number, the stored *pdAuth*, and the status of a purse.

$$ConPurseStart \mathrel{\widehat=} ConPurse \setminus (nextSeqNo, pdAuth, status)$$

The *Req* operation change only the balance and the status of a purse.

$$ConPurseReq \mathrel{\widehat=} ConPurse \setminus (balance, status)$$

The *Val* operation change only the balance and the status of a purse.

$$ConPurseVal \mathrel{\widehat=} ConPurse \setminus (balance, status)$$

The *Ack* operation changes only the status of a purse, and allows the *pdAuth* to change arbitrarily.

$$ConPurseAck \mathrel{\widehat=} ConPurse \setminus (status, pdAuth)$$

4.9.1 StartFromPurse

A *startFrom* message is valid only if it refers to a different purse from the receiver, and mentions a value for which the *from* purse has sufficient funds.

$$
\begin{array}{|l}
\textit{ValidStartFrom} \\
\hline
ConPurse \\
m? : MESSAGE \\[4pt]
cpd : CounterPartyDetails \\
\hline
m? \in \operatorname{ran} startFrom \\[4pt]
cpd = startFrom^{\sim}\, m? \\[4pt]
cpd.name \neq name \\
cpd.value \leq balance \\
\hline
\end{array}
$$

To perform the *StartFromPurseEafromOkay* operation, a purse must receive a valid *startFrom* message, and be in *eaFrom*.

$$
\begin{array}{|l}
\textit{StartFromPurseEafromOkay} \\
\hline
\Delta ConPurse \\
m?, m! : MESSAGE \\[4pt]
cpd : CounterPartyDetails \\
\hline
ValidStartFrom \\
status = eaFrom \\[4pt]
\Xi ConPurseStart \\[4pt]
nextSeqNo' > nextSeqNo \\[4pt]
pdAuth' = (\mu\, PayDetails \mid \\
\qquad from = name \\
\qquad \land to = cpd.name \\
\qquad \land value = cpd.value \\
\qquad \land fromSeqNo = nextSeqNo \\
\qquad \land toSeqNo = cpd.nextSeqNo\,) \\[4pt]
status' = epr \\[4pt]
m! = \perp \\
\hline
\end{array}
$$

The *StartFromPurseEafromOkay* operation stores the payment details consisting of the counterparty details and its own name and sequence number (for later validation), moves to the *epr* state, increases its sequence number, and sends an unprotected status message.

The *StartFromPurseOkay* operation first aborts (logging the pending payment if necessary, and moving to *eaFrom*), then performs the *StartFromPurse-*

EafromOkay operation.

$$
\begin{array}{l}
\mathit{StartFromPurseOkay} \mathrel{\widehat{=}} \\
\quad \mathit{AbortPurseOkay} \mathbin{;} \mathit{StartFromPurseEafromOkay} \setminus (\mathit{cpd})
\end{array}
$$

4.9.2 StartToPurse

A *startTo* message is valid only if it refers to a different purse from the receiver.

$$
\begin{array}{|l}
\hline
\mathit{ValidStartTo} \\
\mathit{ConPurse} \\
m? : \mathit{MESSAGE} \\
\mathit{cpd} : \mathit{CounterPartyDetails} \\
\hline
m? \in \operatorname{ran} \mathit{startTo} \\
\mathit{cpd} = \mathit{startTo}^{\sim}\, m? \\
\mathit{cpd}.\mathit{name} \neq \mathit{name} \\
\hline
\end{array}
$$

To perform the *StartToPurseEafromOkay* operation, a purse must receive a valid *startTo* message, and be in *eaFrom*.

$$
\begin{array}{|l}
\hline
\mathit{StartToPurseEafromOkay} \\
\Delta \mathit{ConPurse} \\
m?, m! : \mathit{MESSAGE} \\
\mathit{cpd} : \mathit{CounterPartyDetails} \\
\hline
\mathit{ValidStartTo} \\
\mathit{status} = \mathit{eaFrom} \\
\Xi \mathit{ConPurseStart} \\
\mathit{nextSeqNo}' > \mathit{nextSeqNo} \\
\mathit{pdAuth}' = (\, \mu\, \mathit{PayDetails} \mid \\
\qquad \mathit{to} = \mathit{name} \\
\qquad \land \mathit{from} = \mathit{cpd}.\mathit{name} \\
\qquad \land \mathit{value} = \mathit{cpd}.\mathit{value} \\
\qquad \land \mathit{toSeqNo} = \mathit{nextSeqNo} \\
\qquad \land \mathit{fromSeqNo} = \mathit{cpd}.\mathit{nextSeqNo}\,) \\
\mathit{status}' = \mathit{epv} \\
m! = \mathit{req}\; \mathit{pdAuth}' \\
\hline
\end{array}
$$

The *StartToPurseOkay* operation logs the pending payment, if necessary; it stores the payment details, consisting of the counterparty details and its own name and sequence number, for later validation; it moves to the *epr* state; it increases its sequence number; and it sends a *req* message containing the stored payment details.

The *StartToPurseOkay* operation first aborts (logging the pending payment if necessary, and moving to *eaFrom*), then performs the *StartToPurseEafromOkay* operation.

$$
\begin{array}{l}
\mathit{StartToPurseOkay} \mathrel{\widehat{=}} \\
\quad \mathit{AbortPurseOkay} \mathbin{;} \mathit{StartToPurseEafromOkay} \setminus (\mathit{cpd})
\end{array}
$$

4.9.3 ReqPurse

An authentic request message is a *req* message containing the correct stored payment details (which were stored on receipt of the *startFrom* message).

$$
\begin{array}{|l}
\hline
\mathit{AuthenticReqMessage} \\
\mathit{ConPurse} \\
m? : \mathit{MESSAGE} \\
\hline
m? = \mathit{req}\; \mathit{pdAuth} \\
\hline
\end{array}
$$

To perform the *ReqPurseOkay* operation, a purse must receive a *req* message with the payment details, and be in the *epr* state,

$$
\begin{array}{|l}
\hline
\mathit{ReqPurseOkay} \\
\Delta \mathit{ConPurse} \\
m?, m! : \mathit{MESSAGE} \\
\hline
\mathit{AuthenticReqMessage} \\
\mathit{status} = \mathit{epr} \\
\Xi \mathit{ConPurseReq} \\
\mathit{balance}' = \mathit{balance} - \mathit{pdAuth}.\mathit{value} \\
\mathit{status}' = \mathit{epa} \\
m! = \mathit{val}\; \mathit{pdAuth} \\
\hline
\end{array}
$$

The purse decrements its balance, moves to the *epa* state, and sends a *val* message containing the stored payment details.

4.9.4 ValPurse

An authentic value message is a *val* message containing the correct stored payment details (which were stored on receipt of the *startTo* message).

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AuthenticValMessage}} \\
\hline
\textit{ConPurse} \\
m? : \textit{MESSAGE} \\
\hline
m? = \textit{val}\ \textit{pdAuth} \\
\hline
\end{array}
$$

To perform the *ValPurseOkay* operation, a purse must receive a *val* message with the payment details, and be in the *epv* state,

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ValPurseOkay}} \\
\hline
\Delta \textit{ConPurse} \\
m?, m! : \textit{MESSAGE} \\
\hline
\textit{AuthenticValMessage} \\
\textit{status} = \textit{epv} \\
\Xi \textit{ConPurseVal} \\
\textit{balance}' = \textit{balance} + \textit{pdAuth.value} \\
\textit{status}' = \textit{eaTo} \\
m! = \textit{ack}\ \textit{pdAuth} \\
\hline
\end{array}
$$

The purse increments its balance, moves to the *eaTo* state, and sends an *ack* message containing the stored payment details.

4.9.5 AckPurse

An authentic acknowledge message is an *ack* message containing the correct stored payment details (which were stored on receipt of the *startFrom* message).

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AuthenticAckMessage}} \\
\hline
\textit{ConPurse} \\
m? : \textit{MESSAGE} \\
\hline
m? = \textit{ack}\ \textit{pdAuth} \\
\hline
\end{array}
$$

To perform the *AckPurseOkay* operation, a purse must receive an *ack* message with the payment details, and be in the *epa* state.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AckPurseOkay}} \\
\hline
\Delta \textit{ConPurse} \\
m?, m! : \textit{MESSAGE} \\
\hline
\textit{AuthenticAckMessage} \\
\textit{status} = \textit{epa} \\
\Xi \textit{ConPurseAck} \\
\textit{status}' = \textit{eaFrom} \\
m! = \perp \\
\hline
\end{array}
$$

The purse moves to the *eaFrom* state, and sends an unprotected status message.

4.10 Exception logging operations

4.10.1 ReadExceptionLogPurse

To perform the *ReadExceptionLogPurseEafromOkay* operation, a purse must receive a *readExceptionLog* message and be in the *eaFrom* state.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ReadExceptionLogPurseEafromOkay}} \\
\hline
\Xi \textit{ConPurse} \\
m?, m! : \textit{MESSAGE} \\
\hline
m? = \textit{readExceptionLog} \\
\textit{status} = \textit{eaFrom} \\
m! \in \{\perp\} \cup \{\, \textit{ld} : \textit{exLog}' \bullet \textit{exceptionLogResult}(\textit{name}, \textit{ld})\,\} \\
\hline
\end{array}
$$

The operation sends an unprotected status message (modelling 'record not available') or a protected *exceptionLogResult* message containing one of the exception logs tagged with its name[10].

The *ReadExceptionLogPurseOkay* operation first aborts (logging any pending payment, and moving to *eaFrom*), and then performs the *ReadExceptionLogPurseEafromOkay* operation.

$$
\begin{array}{l}
\textit{ReadExceptionLogPurseOkay} \mathrel{\widehat{=}} \\
\quad \textit{AbortPurseOkay} \mathbin{⨾} \textit{ReadExceptionLogPurseEafromOkay}
\end{array}
$$

[10] This gives a non-deterministic response, because we do not model exception log record numbers.

4.10.2 ClearExceptionLogPurse

During a clear log transaction the purse's exception log may change, but no other component can change.

$$ConPurseClear \mathrel{\widehat{=}} ConPurse \setminus (exLog)$$

To perform the *ClearExceptionLogPurseOkay* operation, a purse must have a non-empty exception log and receive a valid *exceptionLogClear* message. If the purse receives a valid *exceptionLogClear* message, has no transaction in progress and has an empty exception log, then the purse ignores the message.

First we define how the purse clears its log in *eaFrom*:

$$
\begin{array}{|l}
\multicolumn{1}{l}{ClearExceptionLogPurseEafromOkay} \\
\hline
\Delta ConPurse \\
m?, m! : MESSAGE \\
\hline
exLog \neq \emptyset \\
m? = exceptionLogClear(name, image\ exLog) \\
status = eaFrom \\
\Xi ConPurseClear \\
exLog' = \emptyset \\
m! = \perp \\
\hline
\end{array}
$$

The purse clears its exception log, and sends an unprotected status message.

The *image* ensures that log messages have at least been read and moved to the archive (see *AuthoriseExLogClear*, section 5.7.1). Procedural mechanisms must ensure that archive information is not lost[11].

There is a four stage protocol for reading and clearing exception logs: reading a log to the ether, copying a log from the ether to the archive, authorising a purse exception log clear based on what's in the archive, and clearing a purse's exception log having received authorisation. We note that as a result of this protocol, if *ClearExceptionLogPurseOkay* aborts and logs an uncompleted transaction, then the purse's exception log will not be cleared. The reason for this is as follows. The purse gets to *eaFrom* by aborting any uncompleted transaction. If this would create a new exception record, the clear transaction could not occur, because the (imaged) exception log in the message would not match the actual exception log in the purse.

[11] Concrete SP 2.2, 'Exception logging', section 2.3.1.

The full clear exception log operation for a purse is thus defined to abort an uncompleted transaction first, and then clear the log if appropriate.

$$
\begin{array}{l}
ClearExceptionLogPurseOkay \\
\quad \mathrel{\widehat{=}} AbortPurseOkay \mathbin{⨾} ClearExceptionLogPurseEafromOkay
\end{array}
$$

Chapter 5

Between model, promoted world

5.1 The world

The individual purse operations are *promoted* to the 'world of purses'. This world contains the purses, a public *ether* containing all previous messages sent, and a private *archive*, which is a secure store of exception logs, each exception log tagged with the purse that recorded it. Information cannot be deleted from the archive, so that the store of exception logs is persistent. This is to be implemented by mechanisms outside the target of evaluation.

$$
\begin{array}{|l}
Logbook : \mathbb{P}(NAME \leftrightarrow PayDetails) \\
\hline
Logbook = \mathbb{P}(\{PayDetails \bullet from \mapsto \theta PayDetails\} \\
\qquad \cup \{PayDetails \bullet to \mapsto \theta PayDetails\})
\end{array}
$$

A *Logbook* is a set of log details, each tagged with a name, where that name is either that of the *to* purse or that of the *from* purse in the log details.

In addition, the *archive*'s tagged log details

$$
\begin{array}{|l}
\textit{ConWorld} \\
\hline
conAuthPurse : NAME \twoheadrightarrow ConPurse \\
ether : \mathbb{P}\, MESSAGE \\
archive : Logbook \\
\hline
\forall n : \mathrm{dom}\, conAuthPurse \bullet (conAuthPurse\ n).name = n \\
\forall nld : archive \bullet first\ nld \in \mathrm{dom}\, conAuthPurse \\
\hline
\end{array}
$$

The *archive* is a *Logbook*. In addition, the *archive*'s tagged log details are tagged only with authentic purse names.

to \ from		*epr*	*epa*	(diff trans, incl *eaFrom*) no log	(diff trans, incl *eaFrom*) log
epv		0	?	0	?
eaTo		×	0	0	0
(diff trans, incl *eaFrom*)	no log	0	0	0	0
	log	0	1	0	1

Figure 5.1: The amount lost on the current transaction for each possible state of the purses. '0' means the value has definitely not been lost; '1' means the value has definitely been lost; '?' means the value may be lost; '×' means that this state is impossible.

5.2 Auxiliary definitions

We define some auxiliary components, for ease of proof later. These components are described in detail after the schema. The set *definitelyLost* captures those transactions that have proceeded far enough that we know they cannot succeed. The set *maybeLost* captures those transactions that have proceeded far enough that they will lose money if something goes wrong, but that could equally well continue to successful completion. In the other transactions, either the transaction has not proceeded far enough to lose anything, or has proceeded so far that the value has definitely been received.

The way in which the concrete state of the purses relates to the amount of value 'lost' in the transaction can be represented by the table shown in figure 5.1, where the amount lost on the current transaction is shown for each possible state of the purses, including purses that have moved on to a different transaction, with or without logging this one.

AuxWorld

$$
\begin{array}{l}
\mathit{ConWorld} \\[4pt]
\mathit{allLogs} : \mathit{NAME} \leftrightarrow \mathit{PayDetails} \\
\mathit{authenticFrom}, \mathit{authenticTo} : \mathbb{P}\,\mathit{PayDetails} \\
\mathit{fromLogged}, \mathit{toLogged} : \mathbb{P}\,\mathit{PayDetails} \\
\mathit{toInEpv}, \mathit{toInEapayee}, \mathit{fromInEpr}, \mathit{fromInEpa} : \mathbb{F}\,\mathit{PayDetails} \\
\mathit{definitelyLost} : \mathbb{P}\,\mathit{PayDetails} \\
\mathit{maybeLost} : \mathbb{F}\,\mathit{PayDetails}
\end{array}
$$

$$
\begin{array}{l}
\mathit{allLogs} = \mathit{archive} \\
\qquad \cup \{\, n : \mathrm{dom}\,\mathit{conAuthPurse};\ \mathit{pd} : \mathit{PayDetails} \mid \\
\qquad\quad \mathit{pd} \in (\mathit{conAuthPurse}\ n).\mathit{exLog} \,\} \\[8pt]
\mathit{authenticFrom} \\
\quad = \{\, \mathit{pd} : \mathit{PayDetails} \mid \mathit{pd}.\mathit{from} \in \mathrm{dom}\,\mathit{conAuthPurse} \,\} \\
\mathit{authenticTo} \\
\quad = \{\, \mathit{pd} : \mathit{PayDetails} \mid \mathit{pd}.\mathit{to} \in \mathrm{dom}\,\mathit{conAuthPurse} \,\} \\[8pt]
\mathit{fromLogged} = \{\, \mathit{pd} : \mathit{authenticFrom} \mid \mathit{pd}.\mathit{from} \mapsto \mathit{pd} \in \mathit{allLogs} \,\} \\
\mathit{toLogged} = \{\, \mathit{pd} : \mathit{authenticTo} \mid \mathit{pd}.\mathit{to} \mapsto \mathit{pd} \in \mathit{allLogs} \,\} \\[8pt]
\mathit{toInEpv} = \{\, \mathit{pd} : \mathit{authenticTo} \mid \\
\qquad (\mathit{conAuthPurse}\ \mathit{pd}.\mathit{to}).\mathit{status} = \mathit{epv} \\
\qquad \wedge (\mathit{conAuthPurse}\ \mathit{pd}.\mathit{to}).\mathit{pdAuth} = \mathit{pd} \,\} \\
\mathit{toInEapayee} = \{\, \mathit{pd} : \mathit{authenticTo} \mid \\
\qquad (\mathit{conAuthPurse}\ \mathit{pd}.\mathit{to}).\mathit{status} = \mathit{eaTo} \\
\qquad \wedge (\mathit{conAuthPurse}\ \mathit{pd}.\mathit{to}).\mathit{pdAuth} = \mathit{pd} \,\} \\
\mathit{fromInEpr} = \{\, \mathit{pd} : \mathit{authenticFrom} \mid \\
\qquad (\mathit{conAuthPurse}\ \mathit{pd}.\mathit{from}).\mathit{status} = \mathit{epr} \\
\qquad \wedge (\mathit{conAuthPurse}\ \mathit{pd}.\mathit{from}).\mathit{pdAuth} = \mathit{pd} \,\} \\
\mathit{fromInEpa} = \{\, \mathit{pd} : \mathit{authenticFrom} \mid \\
\qquad (\mathit{conAuthPurse}\ \mathit{pd}.\mathit{from}).\mathit{status} = \mathit{epa} \\
\qquad \wedge (\mathit{conAuthPurse}\ \mathit{pd}.\mathit{from}).\mathit{pdAuth} = \mathit{pd} \,\} \\[8pt]
\mathit{definitelyLost} = \mathit{toLogged} \cap (\mathit{fromLogged} \cup \mathit{fromInEpa}) \\
\mathit{maybeLost} = (\mathit{fromInEpa} \cup \mathit{fromLogged}) \cap \mathit{toInEpv}
\end{array}
$$

These auxiliary definitions put no further constraints on the state, but simply define the derived components. Hence they do not need to be implemented. They are defined merely for ease of use later. We prove that this is so in section 5.2.1 below.

The auxiliary components represent the following:

- *allLogs*: All the exception logs; all those logs in the archive, and those still uncleared in purses.
- *authenticFrom*, *authenticTo*: All possible payment details referring to authentic *from* purses, and authentic *to* purses.
- *fromLogged*: All those payment details logged by a *from* purse.
- *toLogged*: All those details logged by a *to* purse.
- *toInEpv*: All those details for which the *to* purse is authentic, and is currently in *epv* with those details stored. This is a finite set, because *conAuthPurse* is a finite function.
- *toInEapayee*: All those details for which the *to* purse is authentic, and is currently in *eaTo* with those details stored.
- *fromInEpr*: All those details for which the *from* purse is authentic, and is currently in *epr* with those details stored.
- *fromInEpa*: All those details for which the *from* purse is authentic, and is currently in *epa* with those details stored.
- *definitelyLost*: All those details for which we know now that the value has been lost. The *val* message was definitely sent and definitely not received, so ultimately both purses will log the transaction. The authentic *to* purse has logged, which it would not have done had it sent the *ack*, and the authentic *from* purse has sent the *val* and not received the *ack*, and so never will. See figure 5.2
- *maybeLost*: All those details that refer to value that may yet be lost or may yet be transferred successfully from this purse, but which have already definitely *left* the purse. This occurs when the authentic *from* purse has sent the *val* and not received the *ack* and the authentic *to* purse is in *epv*, waiting for the *val* that it may or may not get. See figure 5.2 It is a finite set, because *toInEpv* is a finite set.

We have the identity

$$
\begin{array}{l}
AuxWorld \\
\vdash \\
definitelyLost \cup maybeLost = \\
\quad (fromInEpa \cup fromLogged) \cap (toInEpv \cup toLogged)
\end{array}
$$

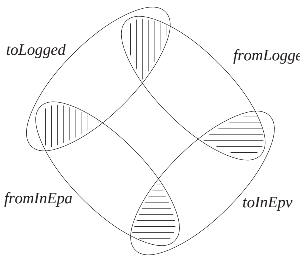

Figure 5.2: The sets *definitelyLost* (vertical hatching) and *maybeLost* (horizontal hatching) as subsets of the other auxiliary definitions.

The later proofs of operations that change purse status (the two start, three protocol and log enquiry operations) are based on how the relevant *pd* moves in and out of the sets *maybeLost* and *definitelyLost*. (These sets are disjoint in the *BetweenWorld*, because of the *BetweenWorld* constraints on log sequence numbers; see lemma 'lost', section C.13.)

5.2.1 *AuxWorld* does not add constraints

AuxWorld introduces some new variables, but does not add any further constraints on *ConWorld*. We define the schema that represents just the new variables introduced by *AuxWorld*.

$$NewVariables \mathrel{\widehat{=}} \exists\, ConWorld \bullet AuxWorld$$

We prove that no further constraints are added by proving the following statement.

$$ConWorld \vdash \exists_1\, NewVariables \bullet AuxWorld$$

Proof:
First we prove existence. We normalise the schemas, drawing out any predicates hidden in the declarations for the new variables. Only one predicate appears, limiting *allLogs* to be a valid *Logbook*.

$$ConWorld \vdash \exists_1\, NewVariables \bullet AuxWorld \land allLogs \in Logbook$$

Rewrite all the equations for the new variables so that each new variable in *AuxWorld* is defined only in terms of variables of *ConWorld*. We then use the one point rule to remove the existential quantification. This leaves just the normalised predicate in addition to *ConWorld*.

$$
\begin{array}{l}
\mathit{ConWorld} \\
\vdash \\
\mathit{ConWorld} \\
\land\ \mathit{archive} \cup \{\, n : \mathrm{dom}\, \mathit{conAuthPurse};\ \mathit{pd} : \mathit{PayDetails} \mid \\
\qquad\qquad \mathit{pd} \in (\mathit{conAuthPurse}\, n).\mathit{exLog} \,\} \\
\qquad \in \mathit{Logbook}
\end{array}
$$

From the definition of *archive*, *archive* is in *Logbook*. From constraint P–1 in *ConPurse*, the set of named exception logs is also in *Logbook*. This discharges the existence proof.

To prove uniqueness, we need only note that the equations defining the new variables are all equality to an expression, and by the transitivity of equality, all possible values are equal.

■ 5.2.1

5.3 Constraints on the ether

We put some further constraints on the state to forbid 'future messages' and 'future logs', and to record the progress of the protocol.

$$
\begin{array}{l}
\textit{BetweenWorld} \\
\hline
\mathit{AuxWorld} \\
\hline
\forall\, \mathit{pd} : \mathit{PayDetails} \mid \mathit{req}\, \mathit{pd} \in \mathit{ether} \bullet \mathit{pd} \in \mathit{authenticTo} \\
\\
\forall\, \mathit{pd} : \mathit{PayDetails} \mid \mathit{req}\, \mathit{pd} \in \mathit{ether} \bullet \\
\quad \mathit{pd}.\mathit{toSeqNo} < (\mathit{conAuthPurse}\, \mathit{pd}.\mathit{to}).\mathit{nextSeqNo} \\
\forall\, \mathit{pd} : \mathit{PayDetails} \mid \mathit{val}\, \mathit{pd} \in \mathit{ether} \bullet \\
\quad \mathit{pd}.\mathit{toSeqNo} < (\mathit{conAuthPurse}\, \mathit{pd}.\mathit{to}).\mathit{nextSeqNo} \\
\quad \land\ \mathit{pd}.\mathit{fromSeqNo} < (\mathit{conAuthPurse}\, \mathit{pd}.\mathit{from}).\mathit{nextSeqNo} \\
\forall\, \mathit{pd} : \mathit{PayDetails} \mid \mathit{ack}\, \mathit{pd} \in \mathit{ether} \bullet \\
\quad \mathit{pd}.\mathit{toSeqNo} < (\mathit{conAuthPurse}\, \mathit{pd}.\mathit{to}).\mathit{nextSeqNo} \\
\quad \land\ \mathit{pd}.\mathit{fromSeqNo} < (\mathit{conAuthPurse}\, \mathit{pd}.\mathit{from}).\mathit{nextSeqNo}
\end{array}
$$

$$
\begin{array}{l}
\forall\, \mathit{pd} : \mathit{fromLogged} \bullet \\
\quad \mathit{pd}.\mathit{fromSeqNo} < (\mathit{conAuthPurse}\, \mathit{pd}.\mathit{from}).\mathit{nextSeqNo} \\
\forall\, \mathit{pd} : \mathit{toLogged} \bullet \mathit{pd}.\mathit{toSeqNo} < (\mathit{conAuthPurse}\, \mathit{pd}.\mathit{to}).\mathit{nextSeqNo} \\
\forall\, \mathit{pd} : \mathit{fromLogged} \mid \\
\quad (\mathit{conAuthPurse}\, \mathit{pd}.\mathit{from}).\mathit{status} \in \{\mathit{epr}, \mathit{epa}\} \bullet \\
\quad \mathit{pd}.\mathit{fromSeqNo} \\
\qquad < (\mathit{conAuthPurse}\, \mathit{pd}.\mathit{from}).\mathit{pdAuth}.\mathit{fromSeqNo} \\
\forall\, \mathit{pd} : \mathit{toLogged} \mid (\mathit{conAuthPurse}\, \mathit{pd}.\mathit{to}).\mathit{status} \in \{\mathit{epv}, \mathit{eaTo}\} \bullet \\
\quad \mathit{pd}.\mathit{toSeqNo} < (\mathit{conAuthPurse}\, \mathit{pd}.\mathit{to}).\mathit{pdAuth}.\mathit{toSeqNo} \\
\\
\forall\, \mathit{pd} : \mathit{fromInEpr} \bullet \mathsf{disjoint}\, \langle \{\mathit{val}\, \mathit{pd}, \mathit{ack}\, \mathit{pd}\}, \mathit{ether} \rangle \\
\forall\, \mathit{pd} : \mathit{PayDetails} \bullet \\
\quad (\mathit{req}\, \mathit{pd} \in \mathit{ether} \land \mathit{ack}\, \mathit{pd} \notin \mathit{ether}) \\
\qquad \Leftrightarrow (\mathit{pd} \in \mathit{toInEpv} \cup \mathit{toLogged}) \\
\forall\, \mathit{pd} : \mathit{PayDetails} \mid \mathit{val}\, \mathit{pd} \in \mathit{ether} \land \mathit{pd} \in \mathit{toInEpv} \bullet \\
\quad \mathit{pd} \in \mathit{fromInEpa} \cup \mathit{fromLogged} \\
\forall\, \mathit{pd} : \mathit{fromInEpa} \cup \mathit{fromLogged} \bullet \mathit{req}\, \mathit{pd} \in \mathit{ether} \\
\\
\mathit{toLogged} \in \mathbb{F}\, \mathit{PayDetails} \\
\\
\forall\, \mathit{pd} : \mathit{exceptionLogResult}^{\sim} (\!| \mathit{ether} |\!) \bullet \mathit{pd} \in \mathit{allLogs} \\
\forall\, \mathit{pds} : \mathbb{P}_1\, \mathit{PayDetails};\ \mathit{name} : \mathit{NAME} \mid \\
\qquad \mathit{exceptionLogClear}(\mathit{name}, \mathit{image}\, \mathit{pds}) \in \mathit{ether} \bullet \\
\quad \{\mathit{name}\} \times \mathit{pds} \subseteq \mathit{archive} \\
\\
\forall\, \mathit{pd} : \mathit{fromLogged} \cup \mathit{toLogged} \bullet \mathit{req}\, \mathit{pd} \in \mathit{ether}
\end{array}
$$

These constraints express the following conditions (numbered for future reference in the refinement proofs):

B–1 All *req* messages in the *ether* refer to authentic *to* purses [1].

B–2 There are no 'future' *req* messages [2]: all *req* messages in the *ether* hold a *to* purse sequence number less than that purse's next sequence num-

[1] Used in *Req*, case 4, section 18.10.

[2] Used in: *StartTo*, location of *pdThis*, section 17.3; *CStartTo*, B–16, section 29.3; *CReq*, B–3, section 29.4.

ber. (It puts no constraint on the *from* purse's sequence number, because the *from* purse mentioned in a *req* message need not have started the transaction yet, and need not even be authentic.)

B–3 There are no 'future' *val* messages [3]: all *val* messages in the *ether* hold a *to* purse sequence number less than that purse's next sequence number and a *from* purse sequence number less than that purse's next sequence number.

B–4 There are no 'future' *ack* messages [4]: all *ack* messages in the *ether* hold a *to* purse sequence number less than that purse's next sequence number and a *from* purse sequence number less than that purse's next sequence number.

B–5 There are no 'future' *from* logs based on the *nextSeqNo* of the *from* purse [5].

B–6 There are no 'future' *to* logs based on the *nextSeqNo* of the *to* purse [6].

B–7 There are no 'future' *from* logs based on the *pdAuth.fromSeqNo* of a purse in *epr* or *epa* [7]: all *from* logs refer only to past *from* transactions. So all *from* logs referring to a purse that is currently in a transaction as a *from* purse (that is, in *epr* or *epa*), hold a *from* sequence number strictly less than that purse's stored current transaction sequence number.

B–8 There are no 'future' *to* logs based on the *pdAuth.toSeqNo* of a purse in *epv* or *eaTo* [8]: all *to* logs refer only to past *to* transactions. So all *to* logs referring to a purse that is currently in a transaction as a *to* purse (in *epv*), hold a *to* sequence number strictly less than that purse's stored current transaction sequence number.

B–9 If the *from* purse is in *epr* then there is no *val* message [9] or *ack* message[10] in the *ether*.

B–10 There is a *req* message but no *ack* message in the *ether* precisely when the *to* purse is in *epv* or has logged the transaction [11].

[3]Used in: *CStartFrom*, B–9, section 29.2; *CStartTo*, B–11, section 29.3. *CVal*, B–4, section 29.5.
[4]Used in: *CStartFrom*, B–9, section 29.2; *CStartTo*, B–10, section 29.3.
[5]Used in: *CStartFrom*, B–7, section 29.2.
[6]Used in: *CStartTo*, B–8, 29.3. 29.3
[7]Used in: *StartFrom*, location of *pdThis*, section 16.3; *CReq*, B–7, section 29.4; lemma 'notLoggedAndIn', section C.12.
[8]Used in: *CVal*, B–8, section 29.5; lemma 'notLoggedAndIn', section C.12.
[9]Used in: *CVal*, B–9, section 29.5.
[10]Used in *Req*, case 4, section 18.10.
[11]Used in: *StartTo*, location of *pdThis*, section 17.3; *Req*, case 4, section 18.10; *Ack*, behaviour of *definitelyLost*, section 20.6.5; *Ack*, behaviour of *maybeLost*, section 20.6.6; *CAbort*, B–10, section 28.5; *CAbort*, B–16, section 28.5; *CAck*, B–11, section 29.6.

B–11 If the *to* purse is in *epv* and there is a *val* message in the *ether*, then either the *from* purse is in *epa* or has logged the transaction [12].

B–12 If the *from* purse is in *epa* or has logged the transaction, then there is a *req* in the *ether* [13].

B–13 The set *toLogged* is finite. This is sufficient to ensure that *definitelyLost* is finite [14].

B–14 Log result messages are logged. The log details of any *exceptionLogResult* message in the ether is either archived or in a purse transaction exception log [15].

B–15 Exception log clear messages refer only to archived logs [16].

B–16 For each *PayDetails* in the logs there is a corresponding *PayDetails* in a *req* message in the ether [17].

That the actual implementation does indeed satisfy this predicate needs to be proved, by a further, small, refinement, that *ConWorld* and the operations refine *BetweenWorld* and the operations (see Part III).

5.4 Framing schema

A framing schema is used to promote the purse operations.

[12]Used in: *Val*, behaviour of *maybeLost*, section 19.6.7.
[13]Used in *StartTo*, location of *pdThis*, section 17.3; *CAbort*, B–12, section 28.5; *CAbort*, B–16, section 28.5.
[14]Used in: various *Rab* schemas, section 10.1
[15]Used in: *Archive*, section 24.2; *CArchive*, section 29.10.
[16]Used in: *ExceptionLogClear*, invoking lemma 'lost unchanged' section 22.2; *CExceptionLogClear*, section 29.8.
[17]Used in: *CStartTo*, alternative to lemma 'logs unchanged', section 29.3.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\Phi BOp} \\
\Delta BetweenWorld \\
\Delta ConPurse \\
m?, m! : MESSAGE \\
name? : NAME \\
\hline
m? \in ether \\
name? \in \operatorname{dom} conAuthPurse \\
\theta ConPurse = conAuthPurse\ name? \\
conAuthPurse' = conAuthPurse \oplus \{name? \mapsto \theta ConPurse'\} \\
archive' = archive \\
ether' = ether \cup \{m!\} \\
\hline
\end{array}
$$

The predicate ensures the following properties common to all promoted operations:

- $m? \in ether$
  the input message is in the *ether*, which ensures it was either previously sent by another purse (*req*, *val*, *ack*, etc.), in the ether since initialisation (*startFrom*, *startTo*, etc.), or input by a special global operation (that is, *AuthoriseExLogClear*).
- $name? \in \operatorname{dom} conAuthPurse$
  the purse is in the world of authentic purses.
- $\theta ConPurse = conAuthPurse\ name?$
  The before state of *ConPurse* we are operating on is the state of the purse identified by *name*?
- $conAuthPurse' = conAuthPurse \oplus \{name? \mapsto \theta ConPurse'\}$
  The after state of the purse system has *name*? updated to the after state of *ConPurse* (which particular state depends on the particular operation details) and all other purses are unchanged [18].
- $archive' = archive$
  The archive remains unchanged.
- $ether' = ether \cup \{m!\}$
  the output message is recorded by the *ether*.

[18] Used in *Req* proof, section 18.7.2.

5.5 Ignore, Increase and Abort

There are various general behaviours that operations may engage in: ignore the input and do nothing; ignore the input but increase the sequence number; ignore the input but abort the current payment transaction.

Ignoring is modelled as an unchanging world:

$$Ignore \mathrel{\widehat{=}} [\, \Xi BetweenWorld;\ name? : NAME;\ m?, m! : MESSAGE \mid m! = \bot \,]$$

Increase has been modelled at the purse level, and is now promoted and totalised:

$$
\begin{aligned}
Increase \mathrel{\widehat{=}}\ & Ignore \\
& \lor (\, \exists \Delta ConPurse \bullet \Phi BOp \land IncreasePurseOkay \,)
\end{aligned}
$$

Abort has been modelled at the purse level, and is now promoted and totalised:

$$
\begin{aligned}
Abort \mathrel{\widehat{=}}\ & Ignore \\
& \lor (\, \exists \Delta ConPurse \bullet AbortPurseOkay \land [\, \Phi BOp \mid m! = \bot \,] \,)
\end{aligned}
$$

5.6 Promoted operations

We promote the individual purse operations, and make them total by disjoining them with the operation defined above that does nothing.

5.6.1 Value transfer operations

The promoted start operations are:

$$
\begin{aligned}
StartFrom \mathrel{\widehat{=}}\ & Ignore \\
& \lor Abort \\
& \lor (\, \exists \Delta ConPurse \bullet \Phi BOp \land StartFromPurseOkay \,)
\end{aligned}
$$

$$
\begin{aligned}
StartTo \mathrel{\widehat{=}}\ & Ignore \\
& \lor Abort \\
& \lor (\, \exists \Delta ConPurse \bullet \Phi BOp \land StartToPurseOkay \,)
\end{aligned}
$$

For use in the proofs, we also promote the *Eafrom* part of the operations on their own:

$$StartFromEafromOkay \mathrel{\widehat=} \exists \Delta ConPurse \bullet \\ \qquad \Phi BOp \land StartFromPurseEafromOkay$$

$$StartToEafromOkay \mathrel{\widehat=} \exists \Delta ConPurse \bullet \\ \qquad \Phi BOp \land StartToPurseEafromOkay$$

The promoted protocol operations are:

$$Req \mathrel{\widehat=} Ignore \lor (\exists \Delta ConPurse \bullet \Phi BOp \land ReqPurseOkay)$$

$$Val \mathrel{\widehat=} Ignore \lor (\exists \Delta ConPurse \bullet \Phi BOp \land ValPurseOkay)$$

$$Ack \mathrel{\widehat=} Ignore \lor (\exists \Delta ConPurse \bullet \Phi BOp \land AckPurseOkay)$$

5.6.2 Exception log operations

The promoted log enquiry operation is:

$$ReadExceptionLog \mathrel{\widehat=} Ignore \\ \quad \lor (\exists \Delta ConPurse \bullet \Phi BOp \land ReadExceptionLogPurseOkay)$$

The promoted exception log clear operation is:

$$ClearExceptionLog \mathrel{\widehat=} Ignore \\ \quad \lor Abort \\ \quad \lor (\exists \Delta ConPurse \bullet \Phi BOp \land ClearExceptionLogPurseOkay)$$

For use in the proofs, we also promote the *Eafrom* part of the operations on their own:

$$ReadExceptionLogEafromOkay \mathrel{\widehat=} \exists \Delta ConPurse \bullet \\ \qquad \Phi BOp \land ReadExceptionLogPurseEafromOkay$$

$$ClearExceptionLogEafromOkay \mathrel{\widehat=} \exists \Delta ConPurse \bullet \\ \qquad \Phi BOp \land ClearExceptionLogPurseEafromOkay$$

5.7 Operations at the world level only

There are some operations on the world that do not have equivalents on individual purses. These are not implemented by the target of evaluation, but need to be implemented by some manual means or external system.

To retain the simplicity of our proof rules, these operations take the same input and outputs as all the purse operations.

5.7.1 Exception Log clear authorisation

The message to clear an exception log can be created only for log details which are already recorded in the archive. The clear code of the message is based on the selected logs in the archive. The exception log clear message couples this clear code with the name of a purse. This supports constraint B–15 which requires that this operation not put a clear message into the ether if the relevant logs have not been archived.

AuthoriseExLogClearOkay

$$\Delta BetweenWorld \\ m?, m! : MESSAGE \\ name? : NAME$$

$$conAuthPurse' = conAuthPurse$$

$$\exists pds : \mathbb{P}_1 PayDetails \bullet \\ \quad \{name?\} \times pds \subseteq archive \\ \quad \land m! = exceptionLogClear(name?, image\ pds)$$

$$ether' = ether \cup \{m!\}$$

$$archive = archive'$$

$$AuthoriseExLogClear \mathrel{\widehat=} Ignore \lor AuthoriseExLogClearOkay$$

Exception logs must be kept for all time to ensure that all value remains accounted for. The operation to clear purses of their exception logs must be supported by a mechanism to store the cleared logs. This is what the archive supplies.

The purse supports the *ReadExceptionLog* operation, which puts an exception log record into the *ether* as a message. As the system implementers have no control over the *ether*, we have modelled it as lossy at the concrete level, allowing for messages to be lost from the *ether* at any time. The *archive* is a *secure* store for information, and to support the security of the purse there must be a manual mechanism to move log messages from the *ether* into the *archive* for safe keeping. This is modelled by the *Archive* operation, and is implemented by some mechanism external to the target of evaluation.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Archive}} \\
\hline
\Delta \mathit{BetweenWorld} \\
m?, m! : \mathit{MESSAGE} \\
\mathit{name}? : \mathit{NAME} \\
\hline
\mathit{conAuthPurse}' = \mathit{conAuthPurse} \\
\mathit{ether}' = \mathit{ether} \\
\mathit{archive} \subseteq \\
\quad \mathit{archive}' \subseteq \\
\quad\quad \mathit{archive} \cup \{ \mathit{log} : \mathit{NAME} \times \mathit{PayDetails} \mid \\
\quad\quad\quad\quad \mathit{exceptionLogResult}\ \mathit{log} \in \mathit{ether} \} \\
m! = \perp \\
\hline
\end{array}
$$

This operation non-deterministically copies some exception log information from messages in the *ether* into the *archive*. It ignores its inputs. As one possible behaviour is to move *no* messages into the archive, it can behave exactly like *Ignore*. The operation is therefore total, and we do not need to disjoin it with *Ignore*.

5.8 Forging messages

If arbitrary messages can be sent, then obviously the security can be compromised. We can build into the definition of the *ether* that it is possible to forge only some kinds of messages. The only messages it is possible to forge are

- replays of earlier valid messages (added to the *ether* during an earlier operation)
- unprotected messages (modelled by being in the initial *ether*, and hence being replayable at any time)
- messages it is possible to detect are forged (modelled by the $\perp$ message, present in the initial *ether*)

This allows us to capture the encryption properties of messages: a message encapsulating arbitrary details cannot be forged by a third party.

5.9 The complete protocol

The complete transfer at the between and concrete levels can be described, informally, by the following sequence of operations:

$$\mathit{StartFrom} \fatsemi \mathit{StartTo} \fatsemi \mathit{Req} \fatsemi \mathit{Val} \fatsemi \mathit{Ack}$$

Other operations may be interleaved in an actual transfer.

The refinement proof in the following sections demonstrates that none of the individual concrete operations violates the security policy.

Chapter 6

Between model, initialisation and finalisation

6.1 Initialisation

As with the abstract case, we set up a particular initial between state. We do not want to model adding new authentic purses to the system, since some of the operations involved are outside the security boundary. So we allow the world to be 'switched off' and a new world 'switched on', where the new world consists of the old world as it was, plus the new purses. So our initial state must allow purses to be part-way through transactions.

We set constraints on the initial state of the between system to say that there are all the request messages in the *ether*, any current transactions must be valid, and there are no future messages.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{BetweenInitState}} \\
\hline
\textit{BetweenWorld}' \\
\hline
\{\textit{readExceptionLog}, \perp\} \\
\cup \\
\bigcup\{\, \textit{cpd} : \textit{CounterPartyDetails} \bullet \{\textit{startFrom}\ \textit{cpd}, \textit{startTo}\ \textit{cpd}\}\,\} \\
\subseteq \textit{ether}' \\
\hline
\end{array}
$$

The initial *ether* contains (or may be considered to contain) the following messages:

- the log enquiry and $\perp$ messages (hence a purse can always have a forged message sent to it)
- all possible start messages, even those referring to a non-authentic purse

- no future messages (ensured by the constraints in *BetweenWorld′*)

So any purse, at any time, can be sent a read log message, or an instruction to start a transfer; this saves us having to model the IFD sending these messages. Since the IFD does not authenticate start messages, we cannot insist on authentic purses at this point.

The inability to forge messages means that a *req* message always mentions an authentic *to* purse, and a *val* message an authentic *from* purse. So a *val* message sent on receipt of a *req* will mention authentic *to* and *from* purses.

We must also initialise our concrete inputs, since they are different from the global inputs. This defines how concrete inputs are interpreted.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{BetwInitIn}} \\
\hline
g? : AIN \\[4pt]
m? : MESSAGE \\
name? : NAME \\
\hline
m? \in \operatorname{ran} req \Rightarrow \\
\quad g? = transfer(\, \mu\, TransferDetails \mid \\
\qquad from = (req^{\sim}\, m?).from \\
\qquad \land\ to = (req^{\sim}\, m?).to \\
\qquad \land\ value = (req^{\sim}\, m?).value\,) \\[4pt]
m? \notin \operatorname{ran} req \Rightarrow g? = aNullIn \\
\hline
\end{array}
$$

6.2 Finalisation

Finalisation maps a *BetweenWorld* to a *GlobalWorld*, to specify how the various concrete state components are observed abstractly.

We finalise by choosing to assume that all the transactions in *maybeLost* actually are lost. (In some sense, finalisation treats incomplete transactions as if they would 'abort'.)

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{BetwFinState}} \\
\hline
BetweenWorld \\
GlobalWorld \\
\hline
\operatorname{dom} gAuthPurse = \operatorname{dom} conAuthPurse \\[4pt]
\forall\, name : \operatorname{dom} conAuthPurse \bullet \\[4pt]
\quad (gAuthPurse\ name).balance = (conAuthPurse\ name).balance \\[4pt]
\quad \land\ (gAuthPurse\ name).lost = \\
\qquad sumValue(\, (definitelyLost \cup maybeLost) \\
\qquad\qquad\quad \cap \{\, ld : PayDetails \mid ld.from = name \,\}\,) \\
\hline
\end{array}
$$

There is a simple relationship between concrete and global *balance* components. The global *lost* component is related to the concrete *maybeLost* and *definitelyLost* logs (the function *sumValue* is defined in section D.3).

We must also finalise our concrete outputs, since they are different from the global outputs. This defines how concrete outputs are interpreted.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{BetwFinOut}} \\
\hline
g! : AOUT \\
m! : MESSAGE \\
\hline
g! = aNullOut \\
\hline
\end{array}
$$

All concrete outputs are interpreted as the single abstract output, *aNullOut*.

Chapter 7

Concrete model: implementation

7.1 Concrete World State

The $\mathcal{C}$ world state has the same components as the $\mathcal{B}$ state; we decorate with a subscript zero to distinguish like-named $\mathcal{B}$ and $\mathcal{C}$ components.

Since $\Delta ConWorld_0$ has components dashed-then-subscripted, whereas we require subscripted-then-dashed, we defined our own Δ and Ξ schemas.

$$\Delta ConWorld0 \mathrel{\widehat{=}} ConWorld_0 \land ConWorld_0'$$

$$\Xi ConWorld0 \mathrel{\widehat{=}} [\, \Delta ConWorld0 \mid \theta ConWorld_0 = \theta ConWorld_0' \,]$$

7.2 Framing Schema

The concrete world $\mathcal{C}$ has the same operations as the $\mathcal{B}$ model.

The world we promote to is *ConWorld*, not *BetweenWorld*. (Remember *ConWorld* has the same structure as *BetweenWorld*, but none of the constraints about future messages.) We are also allowed to 'lose' messages from the public *ether*, which models the fact that the *ether* may be implemented as a lossy medium.

So the $\mathcal{C}$ framing schema is used to promote the purse operations.

$$
\begin{array}{|l}
\hline
\Phi COp \\
\hline
\Delta ConWorld0 \\
\Delta ConPurse \\
m?, m! : MESSAGE \\
name? : NAME \\
\hline
m? \in ether_0 \\
name? \in \mathrm{dom}\ conAuthPurse_0 \\
\theta ConPurse = conAuthPurse_0\ name? \\
conAuthPurse'_0 = conAuthPurse_0 \oplus \{name? \mapsto \theta ConPurse'\} \\
archive'_0 = archive_0 \\
ether'_0 \subseteq ether_0 \cup \{m!\} \\
\hline
\end{array}
$$

7.3 Ignore, Increase and Abort

The $\mathcal{B}$ operations *Ignore*, *Increase* and *Abort* have C equivalents, working on the C world instead of the $\mathcal{B}$ world. These operations are not named operations of the purse, i.e. they are not visible at the purse interface. We define them so that they can be used as *components* in C purse operations.

$$CIgnore \mathrel{\widehat=} [\, \Xi ConWorld0;\ name? : NAME;\ m?, m! : MESSAGE \mid m! = \bot \,]$$

$$
\begin{aligned}
CIncrease \mathrel{\widehat=}\ & CIgnore \\
& \lor (\, \exists \Delta ConPurse \bullet \Phi COp \land IncreasePurseOkay \,)
\end{aligned}
$$

$$
\begin{aligned}
CAbort \mathrel{\widehat=}\ & CIgnore \\
& \lor (\, \exists \Delta ConPurse \bullet AbortPurseOkay \land [\, \Phi COp \mid m! = \bot \,] \,)
\end{aligned}
$$

All subsequent operations defined in this chapter correspond to the actual operations of the purse.

7.4 Promoted operations

As with the $\mathcal{B}$ promoted operations, the C promoted operations are made total by disjoining with *CIgnore*.

7.4.1 Value transfer operations

The promoted start operations are:

$$
\begin{aligned}
CStartFrom \mathrel{\widehat=}\ & CIgnore \\
& \lor CAbort \\
& \lor (\, \exists \Delta ConPurse \bullet \Phi COp \land StartFromPurseOkay \,)
\end{aligned}
$$

$$
\begin{aligned}
CStartTo \mathrel{\widehat=}\ & CIgnore \\
& \lor CAbort \\
& \lor (\, \exists \Delta ConPurse \bullet \Phi COp \land StartToPurseOkay \,)
\end{aligned}
$$

The promoted protocol operations are:

$$CReq \mathrel{\widehat=} CIgnore \lor (\, \exists \Delta ConPurse \bullet \Phi COp \land ReqPurseOkay \,)$$

$$CVal \mathrel{\widehat=} CIgnore \lor (\, \exists \Delta ConPurse \bullet \Phi COp \land ValPurseOkay \,)$$

$$CAck \mathrel{\widehat=} CIgnore \lor (\, \exists \Delta ConPurse \bullet \Phi COp \land AckPurseOkay \,)$$

7.4.2 Exception log operations

The promoted log enquiry operation is:

$$
\begin{aligned}
CReadExceptionLog \mathrel{\widehat=}\ & CIgnore \\
& \lor (\, \exists \Delta ConPurse \bullet \Phi COp \land ReadExceptionLogPurseOkay \,)
\end{aligned}
$$

The promoted clear operation is:

$$
\begin{aligned}
CClearExceptionLog \mathrel{\widehat=}\ & CIgnore \\
& \lor CAbort \\
& \lor (\, \exists \Delta ConPurse \bullet \Phi COp \land ClearExceptionLogPurseOkay \,)
\end{aligned}
$$

7.5 Operations at the world level only

As with the $\mathcal{B}$ model, there are some operations that act on the world, rather than on individual purses. These operations are specified exactly as they are in the $\mathcal{B}$ model, but acting on *ConWorld* instead of *BetweenWorld*.

7.5.1 Exception Log clear authorisation

The message to clear an exception log is generated external to the model.

$$
\begin{array}{l}
CAuthoriseExLogClear \mathrel{\widehat=} CIgnore \\
\qquad \lor (\exists \Xi ConPurse \bullet [\Phi COp \mid (\exists lds : \mathbb{P}_1\ PayDetails \mid \\
\qquad\qquad \{name?\} \times lds \subseteq archive_0 \bullet \\
\qquad\qquad\qquad m! = exceptionLogClear(name?, image\ lds))])
\end{array}
$$

The operation to move exception log information from the *ether* to the *archive* is

$$
\begin{array}{|l}
\multicolumn{1}{l}{CArchive} \\
\hline
\Delta ConWorld0 \\
m?, m! : MESSAGE \\
name? : NAME \\
\hline
conAuthPurse'_0 = conAuthPurse_0 \\
ether'_0 \subseteq ether_0 \\
archive_0 \subseteq \\
\quad archive'_0 \subseteq \\
\qquad archive_0 \cup \{ log : NAME \times PayDetails \mid \\
\qquad\qquad exceptionLogResult\ log \in ether_0 \} \\
m! = \perp \\
\hline
\end{array}
$$

7.6 Initial state

The initial state of the C world has an ether that is a subset of one that satisfies the 'no future messages' constraints placed on the $\mathcal{B}$ world (the subset is needed because the C ether is lossy).

$$
\begin{array}{|l}
\multicolumn{1}{l}{ConInitState} \\
\hline
ConWorld'_0 \\
\hline
\exists BetweenWorld' \mid BetweenInitState \bullet \\
\quad conAuthPurse'_0 = conAuthPurse' \\
\quad \land archive'_0 = archive' \\
\quad \land \{\perp\} \subseteq ether'_0 \subseteq ether' \\
\hline
\end{array}
$$

7.7 Finalisation

The $\mathcal{B}$ finalisation is defined for any *ConWorld*; we reuse it for the C finalisation.

$$
\begin{array}{|l}
\multicolumn{1}{l}{ConFinState} \\
\hline
AuxWorld_0 \\
GlobalWorld \\
\hline
\mathrm{dom}\ gAuthPurse = \mathrm{dom}\ conAuthPurse_0 \\
\forall name : \mathrm{dom}\ conAuthPurse_0 \bullet \\
\quad (gAuthPurse\ name).balance \\
\quad = (conAuthPurse_0\ name).balance \\
\quad \land (gAuthPurse\ name).lost = \\
\qquad sumValue((definitelyLost_0 \cup maybeLost_0) \\
\qquad\qquad \cap \{ ld : PayDetails \mid ld.from = name \}) \\
\hline
\end{array}
$$

Chapter 8

Model consistency proofs

8.1 Introduction

In order to increase confidence that the specifications written are not meaningless, it is wise to prove some properties of them.

The least that should be done is to demonstrate that the constraints on the state and those defining each operation do not reduce to *false*. So for each model, the consistency proof obligations are:

- Show it is possible for at least one state to exist (which demonstrates that the state invariant is not contradictory). If we choose this state to be the initial state, we also demonstrate that initialisation is not vacuous, too.

  $$\vdash \exists\, State' \bullet StateInit$$

- Show that each operation does not have an empty precondition (which demonstrates that no operation definition is contradictory).

  $$\vdash \exists\, State;\ Input \bullet \mathrm{pre}\ Op$$

  In fact, here we show that all our operations are total, which is the much stronger condition

  $$\vdash \forall\, State;\ Input \bullet \mathrm{pre}\ Op$$

We present these proofs for each of our three models below.

8.2 Abstract model consistency proofs

8.2.1 Existence of initial abstract state

$\vdash \exists AbWorld' \bullet AbInitState$

Proof:
It is sufficient to find an explicit abstract world that satisfies the constraints of *AbInitState*. Consider the abstract world with the components:

$$abAuthPurse' = \emptyset$$

This satisfies the constraints of *AbWorld*, so is clearly a suitable initial state.
■ 8.2.1

8.2.2 Totality of abstract operations

AbIgnore is total.
Proof:

$$\begin{aligned}
&\text{pre } AbIgnore\\
&\quad = \text{pre } [\, \Delta AbWorld;\ a? : AIN;\ a! : AOUT \mid\\
&\qquad abAuthPurse' = abAuthPurse\\
&\qquad \wedge a! = aNullOut\,] && [\text{defn. } AbIgnore]\\
&\quad = [\, AbWorld;\ a? : AIN \mid\\
&\qquad \exists AbWorld';\ a! : AOUT \mid\\
&\qquad\quad abAuthPurse' = abAuthPurse\\
&\qquad\quad \wedge a! = aNullOut\,] && [\text{defn. pre}\,]\\
&\quad = [\, AbWorld;\ a? : AIN \mid\\
&\qquad \exists abAuthPurse' : NAME \twoheadrightarrow\!\!\!\!\!\shortmid\ AbPurse;\ a! : AOUT \mid\\
&\qquad\quad abAuthPurse' = abAuthPurse\\
&\qquad\quad \wedge a! = aNullOut\,]\\
&&& [\textit{one point} \text{ rule}]\\
&\quad = [\, AbWorld;\ a? : AIN\,]
\end{aligned}$$

■
All the abstract operations are total.
Proof:

They are total by construction. They are all of the form $AbOpOkay \vee AbIgnore$, so:

$$\begin{aligned}
&\text{pre } AbOp\\
&\quad = \text{pre } (AbOpOkay \vee AbIgnore)\\
&\quad = \text{pre } AbOpOkay \vee \text{pre } AbIgnore\\
&\quad = \text{pre } AbOpOkay \vee [\, AbWorld;\ a? : AIN\,]\\
&\quad = [\, AbWorld;\ a? : AIN\,]
\end{aligned}$$

■
■ 8.2.2
■ 8.2

8.3 Between model consistency proofs

8.3.1 Existence of between initial state

$\vdash \exists BetweenWorld' \bullet BetweenInitState$

Proof:
It is sufficient to find an explicit between world that satisfies the constraints of *BetweenWorldInit*.
A world of no purses, an *ether* that consists of exactly the messages explicitly allowed of *BetweenWorldInit*, and an empty *archive*, is sufficient.

$$\begin{aligned}
&conAuthPurse' = \emptyset\\
&ether' = \{readExceptionLog, \bot\}\\
&\qquad \cup \bigcup \{\, cpd : CounterPartyDetails \bullet \{startFrom\ cpd, startTo\ cpd\}\,\}\\
&archive' = \emptyset
\end{aligned}$$

This satisfies the constraints in *ConWorld*. It also satisfies the extra constraints of *BetweenWorld*: all the quantifiers are over empty sets (of purses or messages) and hence are trivially true.
■ 8.3.1

8.3.2 Totality of between operations

All between operations are total.
Proof:

They all offer the option of *Ignore* (explicitly by disjunction, except for *Archive*, which offers it implicitly). *Ignore* is the total identity operation.

■ 8.3.2

■ 8.3

8.4 Concrete model consistency proofs

8.4.1 Existence of concrete initial state

$$\vdash \exists\, ConWorld_0' \bullet ConInitState$$

Proof:
The concrete state is identical to the between state, except for fewer constraints. Therefore as a between state exists, so does a concrete one.

■ 8.4.1

8.4.2 Totality of concrete operations

All concrete operations are total.
Proof:
The concrete operations are identical to the between ones. Therefore if the between operations are total, so are the concrete ones.

■ 8.4.2

■ 8.4

■ 8

Part II

First Refinement: $\mathcal{A}$ to $\mathcal{B}$

Chapter 9

Refinement Proof Rules

9.1 Security of the implementation

We prove the concrete model $\mathcal{C}$ is secure with respect to the abstract model $\mathcal{A}$ in two stages. We first show (in this part) that $\mathcal{B}$ refines $\mathcal{A}$ then we show (in part III) that $\mathcal{C}$ refines $\mathcal{B}$.

To show that $\mathcal{B}$ refines $\mathcal{A}$ we show that every (promoted) $\mathcal{B}$ operation correctly refines some $\mathcal{A}$ operation.

Much of what the $\mathcal{B}$ (and $\mathcal{C}$) operations achieve is invisible at the $\mathcal{A}$ level, so many $\mathcal{B}$ operations are refinements of *AbIgnore* (abstractly 'do nothing'). Some of the $\mathcal{B}$ operations that are refinements of *AbIgnore* do serve to resolve abstract non-determinism.

The refinements are

$$AbTransfer \sqsubseteq Req$$

$$\begin{aligned} AbIgnore \sqsubseteq\ & StartFrom \\ & \lor StartTo \\ & \lor Val \\ & \lor Ack \\ & \lor ReadExceptionLog \\ & \lor ClearExceptionLog \\ & \lor AuthoriseExLogClear \\ & \lor Archive \\ & \lor Ignore \\ & \lor Increase \\ & \lor Abort \end{aligned}$$

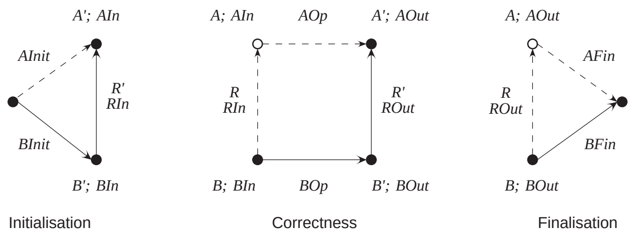

Figure 9.1: A summary of the backward proof rules. The hypothesis is the existence of the lower (solid) path. The proof obligation is to demonstrate the existence of an upper (dashed) path.

Each of these refinements must be proved correct.

For the $\mathcal{A}$ to $\mathcal{B}$ refinement proofs, the following set of 'upward' or 'backward' proof rules are sufficient to show the refinement [Woodcock & Davies 1996]. For the $\mathcal{B}$ to $\mathcal{C}$ refinement proofs, the 'downward' or 'forward' proof rules are sufficient to show the refinement.

These rules are expressed in terms of a 'concrete' (lower) and 'abstract' (upper) model. In this first refinement the 'abstract' model is $\mathcal{A}$ and the 'concrete' model is $\mathcal{B}$. In the second refinement the 'abstract' model is now $\mathcal{B}$ and the 'concrete' model is $\mathcal{C}$.

9.2 Backwards rules proof obligations

Appendix A describes the syntax for theorems, and how we lay out the proofs. The backward proof rules are summarised in figure 9.1, and described below.

9.2.1 Initialisation

We start from some global state G, and *initialise* it to an abstract initial state A' and concrete initial state B'. These must be related by the retrieve.

$$\vdash \forall\, G;\ GIn;\ B';\ BIn;\ A';\ AIn \mid BInitState \land BInitIn \land R' \land RIn \bullet \\ AInitState \land AInitIn$$

Given any global initial state G, if we initialise it with $BInit$ to B', then retrieve B' to A', we must get the same abstract initial state as if we had initialised directly to A' using $AInit$.

This can be simplified to:

$$BInitState;\ R' \vdash AInitState$$
$$BInitIn;\ RIn \vdash AInitIn$$

9.2.2 Finalisation

We start from some abstract final state A and concrete final state B, related by the retrieve, and *finalise* them to the *same* global final state G'.

$$\vdash \forall\, G';\ GOut;\ B;\ BOut \mid BFinState \land BFinOut \bullet \\ \exists\, A;\ AOut \bullet R \land ROut \land AFinState \land AFinOut$$

Given any concrete final state B that finalises with $BFin$ to G', then it is possible to find a corresponding abstract final state A, that both retrieves from B and finalises with $AFin$ to the same G'.

This can be simplified to:

$$BFinState \vdash \exists\, A \bullet R \land AFinState$$
$$BFinOut \vdash \exists\, AOut \bullet ROut \land AFinOut$$

9.2.3 Applicability

$$\vdash \forall\, B;\ BIn \mid (\,\forall\, A;\ AIn \mid R \land RIn \bullet \operatorname{pre} AOp\,) \bullet \operatorname{pre} BOp$$

For each operation: if we are in a concrete state, and if all the abstract states to which it retrieves satisfy the precondition of the abstract operation, then we must also satisfy the precondition of the corresponding concrete operation.

For our case, AOp is total (this needs to be proved for each of the abstract operations — see section 8.2.2). So $\operatorname{pre} AOp = \mathit{true}$. So

$$\begin{aligned}&(\,\forall\, A;\ AIn \mid R \land RIn \bullet \operatorname{pre} AOp\,)\\ &\quad\Rightarrow (\,\forall\, A;\ AIn \bullet R \land RIn \Rightarrow \operatorname{pre} AOp\,)\\ &\quad\Rightarrow (\,\forall\, A;\ AIn \bullet R \land RIn \Rightarrow \mathit{true}\,)\\ &\quad\Rightarrow (\,\forall\, A;\ AIn \bullet \mathit{true}\,)\\ &\quad\Rightarrow \mathit{true}\end{aligned}$$

So, for total abstract operations, the applicability proof obligation reduces to

$$B;\ BIn \vdash \operatorname{pre} BOp$$

That is, a proof that BOp is total, too. This is discharged in section 8.3.2.

9.2.4 Correctness

$$\begin{array}{l} \vdash \forall B;\ BIn \mid (\ \forall A;\ AIn \mid R \land RIn \bullet \mathrm{pre}\ AOp\ ) \bullet \\ \quad (\ \forall A';\ AOut;\ B';\ BOut \mid BOp \land R' \land ROut \bullet \\ \qquad (\ \exists A;\ AIn \bullet R \land RIn \land AOp\ )\ ) \end{array}$$

For each operation: if we start in a concrete state corresponding to the precondition of the abstract operation (the applicability condition ensures we then satisfy the concrete operation's precondition), and do the concrete operation, and then retrieve to the abstract state, then we end up in a state that we could have reached doing the abstract operation.

Using pre $AOp = true$ (proved during applicability), this reduces to

$$\begin{array}{l} \vdash \forall B;\ BIn \bullet (\ \forall A';\ AOut;\ B';\ BOut \mid BOp \land R' \land ROut \bullet \\ \qquad (\ \exists A;\ AIn \bullet R \land RIn \land AOp\ )\ ) \end{array}$$

Moving the quantifier into the hypothesis:

$$\begin{array}{l} B;\ BIn;\ A';\ AOut;\ B';\ BOut \mid BOp \land R' \land ROut \\ \qquad \vdash \exists A;\ AIn \bullet R \land RIn \land AOp \end{array}$$

Then rearranging the schema predicates from the predicate part to the declaration part, and removing the redundant declarations, gives the final form we use:

$$BOp;\ R';\ ROut \vdash \exists A;\ AIn \bullet R \land RIn \land AOp$$

Chapter 10

$\mathcal{A}$ to $\mathcal{B}$ retrieve relation

The purpose of the retrieve relation is to capture the details of the various states the concrete world can be in, and which abstract state(s) these correspond to, and the relationships between the concrete and abstract inputs and outputs.

For the first refinement, we talk of *Rab*: the Retrieve from $\mathcal{A}$ to $\mathcal{B}$. Later, for the second refinement, we talk of *Rbc*: the Retrieve from $\mathcal{B}$ to $\mathcal{C}$.

10.1 Retrieve state

The domains of the $\mathcal{B}$ and $\mathcal{A}$ 'world' functions define the authentic purses.

```
┌─ AbstractBetween ─────────────────────────
│ AbWorld
│ BetweenWorld
├──────────
│ dom abAuthPurse = dom conAuthPurse
└───────────────────────────────────────────
```

$\mathcal{A}$ *balance* and *lost* are related to $\mathcal{B}$ *balance* and *exLogs*. The relationship is relational, not functional, and highly non-deterministic part-way through a transaction.

10.1.1 Exposing *chosenLost*

chosenLost is a non-deterministic choice of a subset of all the *maybeLost* values that we 'choose' to say will be lost.

RabCl

$$
\begin{array}{l}
\mathit{AbstractBetween} \\
\mathit{chosenLost} : \mathbb{P}\,\mathit{PayDetails} \\
\hline
\mathit{chosenLost} \subseteq \mathit{maybeLost} \\
\forall\, \mathit{name} : \operatorname{dom} \mathit{conAuthPurse} \bullet \\
\quad (\mathit{abAuthPurse}\ \mathit{name}).\mathit{lost} = \\
\qquad \mathit{sumValue}((\mathit{definitelyLost} \cup \mathit{chosenLost}) \\
\qquad\quad \cap \{\, \mathit{pd} : \mathit{PayDetails} \mid \mathit{pd}.\mathit{from} = \mathit{name} \,\}) \\
\quad \wedge\ (\mathit{abAuthPurse}\ \mathit{name}).\mathit{balance} = \\
\qquad (\mathit{conAuthPurse}\ \mathit{name}).\mathit{balance} \\
\qquad + \mathit{sumValue}((\mathit{maybeLost} \setminus \mathit{chosenLost}) \\
\qquad\quad \cap \{\, \mathit{pd} : \mathit{PayDetails} \mid \mathit{pd}.\mathit{to} = \mathit{name} \,\})
\end{array}
$$

The predicate links the $\mathcal{B}$ and $\mathcal{A}$ values[1]:

- For a purse *name*, its *lost* value is the sum of the values in all those transactions that are definitely lost or that we have chosen to assume lost with *name* as the *from* purse. (Note the deliberate similarity of this definition and that in *BetwFinState*.)
- The $\mathcal{A}$ *balance* of a purse is its $\mathcal{B}$ balance plus the value of all those transactions we have chosen to assume will not be lost, with *name* as the *to* purse. (For a give *name*, there is at most one such transaction.)

A consequence of this relationship is that the abstract *lost* and *balance* values of a purse can depend on the corresponding values of *more than one* concrete purse.

10.1.2 Hiding *chosenLost*

The retrieve relation is then *RabCl* with the non-deterministic choice *chosenLost* hidden[2]:

$$\mathit{Rab} \mathrel{\widehat{=}} \exists\, \mathit{chosenLost} : \mathbb{P}\,\mathit{PayDetails} \bullet \mathit{RabCl}$$

We define the retrieve in this way because in the proof we need to have direct access to *chosenLost*.

[1] It is valid to apply *sumValue* in this predicate, because both *definitelyLost* and *maybeLost* are finite. *definitelyLost* is finite because of *BetweenWorld* constraint B–13. *maybeLost* is finite because *toInEpv* is finite: each *pd* in the set comprehension for *toInEpv* comes from a distinct purse in *conAuthPurse*, which itself is a finite function.

[2] We use this form to simplify the general correctness proofs, section 14.4.3.

10.1.3 Exposing *pdThis*

In the proof, we find that we wish to focus on a single *pd* (any *pd*). We define a new schema, *RabClPd*, identical to *RabCl* except for an extra declaration of a *pd*.

RabClPd

$$
\begin{array}{l}
\mathit{RabCl} \\
\mathit{pdThis} : \mathit{PayDetails}
\end{array}
$$

We split the predicate part of *RabClPd* into two cases that partition the possibilities:

- $\forall\, \mathit{name} : \operatorname{dom} \mathit{conAuthPurse} \mid \mathit{name} \notin \{\mathit{pdThis}.\mathit{from}, \mathit{pdThis}.\mathit{to}\}$ purses not involved in the *pdThis* transaction.
- $\forall\, \mathit{name} : \operatorname{dom} \mathit{conAuthPurse} \mid \mathit{name} \in \{\mathit{pdThis}.\mathit{from}, \mathit{pdThis}.\mathit{to}\}$ purses involved in the *pdThis* transaction.

In all cases the purses other than the *from* and *to* purses retrieve their *balance* and *lost* values in the same way, so we factor this part of the predicate out into a separate schema, *OtherPursesRab*, which we include with the remaining part of the predicate.

OtherPursesRab

$$
\begin{array}{l}
\mathit{AbstractBetween} \\
\mathit{chosenLost} : \mathbb{P}\,\mathit{PayDetails} \\
\mathit{pdThis} : \mathit{PayDetails} \\
\hline
\forall\, \mathit{name} : \operatorname{dom} \mathit{conAuthPurse} \mid \mathit{name} \notin \{\mathit{pdThis}.\mathit{from}, \mathit{pdThis}.\mathit{to}\} \bullet \\
\quad (\mathit{abAuthPurse}\ \mathit{name}).\mathit{lost} = \\
\qquad \mathit{sumValue}((\mathit{definitelyLost} \cup \mathit{chosenLost}) \\
\qquad\quad \cap \{\, \mathit{pd} : \mathit{PayDetails} \mid \mathit{pd}.\mathit{from} = \mathit{name} \,\}) \\
\quad \wedge\ (\mathit{abAuthPurse}\ \mathit{name}).\mathit{balance} = \\
\qquad (\mathit{conAuthPurse}\ \mathit{name}).\mathit{balance} \\
\qquad + \mathit{sumValue}((\mathit{maybeLost} \setminus \mathit{chosenLost}) \\
\qquad\quad \cap \{\, \mathit{pd} : \mathit{PayDetails} \mid \mathit{pd}.\mathit{to} = \mathit{name} \,\})
\end{array}
$$

We split *RabClPd* into four cases that partition the possibilities:

- *RabOkayClPd*: $\mathit{pdThis} \in \mathit{maybeLost} \setminus \mathit{chosenLost}$ half way through a transaction that will succeed. Since *maybeLost* refers only to authentic purses, we know that $\{pdThis.from, pdThis.to\} \subseteq \operatorname{dom} conAuthPurse$, and so the remaining quantifier is reduced to these two cases.

- $RabWillBeLostClPd : pdThis \in chosenLost$ half way through a transaction that will lose the value (the *to* purse has not yet aborted, but we choose that it will, rather than receive the *val*). Since $chosenLost \subseteq maybeLost$ refers only to authentic purses, we know that $\{pdThis.from, pdThis.to\} \subseteq \operatorname{dom} conAuthPurse$, and so the remaining quantifier is reduced to these two cases.
- $RabHasBeenLostClPd : pdThis \in definitelyLost$ half way through a transaction that has lost the value (the *to* purse has already moved on). Since *definitelyLost* refers only to authentic purses, we know that $\{pdThis.from, pdThis.to\} \subseteq \operatorname{dom} conAuthPurse$, and so the remaining quantifier is reduced to these two cases.
- $RabEndClPd : pdThis \notin definitelyLost \cup maybeLost$ At the beginning or end of a transaction, so there is no non-determinism in the *lost* or *balance* components. A general *pdThis* may refer to non-authentic purses, so the quantifier is reduced no further.

In the later proofs of operations that change purse status (*Abort*, *Req*, *Val* and *Ack*), we argue how the relevant *pd* moves in and out of the sets *maybeLost* and *definitelyLost*, and thereby choose the appropriate one of the four cases of the retrieve to use before and after the operation.

We perform this split by systematically subtracting out the chosen *pd* from the *lost* and *balance* expressions. If the *pd* was in fact in the relevant set, we then have to add the subtracted value back in, otherwise we do nothing, since we have made no change to the expression.

$$
\begin{array}{|l}
\textit{RabOkayClPd} \\
\hline
AbstractBetween \\
chosenLost : \mathbb{P}\, PayDetails \\
pdThis : PayDetails \\
\hline
chosenLost \subseteq maybeLost \\
pdThis \in maybeLost \setminus chosenLost \\
(abAuthPurse\ pdThis.from).balance = \\
\quad (conAuthPurse\ pdThis.from).balance \\
\quad + sumValue(((maybeLost \setminus chosenLost) \\
\qquad \cap \{\, pd : PayDetails \mid pd.to = pdThis.from \,\}) \\
\qquad \setminus \{pdThis\}) \\
(abAuthPurse\ pdThis.to).balance = \\
\quad pdThis.value \\
\quad + (conAuthPurse\ pdThis.to).balance \\
\quad + sumValue(((maybeLost \setminus chosenLost) \\
\qquad \cap \{\, pd : PayDetails \mid pd.to = pdThis.to \,\}) \\
\qquad \setminus \{pdThis\}) \\
\forall\, name : \{pdThis.from, pdThis.to\} \bullet \\
\quad (abAuthPurse\ name).lost = \\
\qquad sumValue(((definitelyLost \cup chosenLost) \\
\qquad\quad \cap \{\, pd : PayDetails \mid pd.from = name \,\}) \\
\qquad\quad \setminus \{pdThis\}) \\
OtherPursesRab \\
\hline
\end{array}
$$

In the *Okay* case, *pdThis* is not lost, so its value has to be added back into the to purse's *balance* component.

$$
\begin{array}{l}
\textit{RabWillBeLostClPd} \\
\hline
AbstractBetween \\
chosenLost : \mathbb{P}\ PayDetails \\
pdThis : PayDetails \\
\hline
chosenLost \subseteq maybeLost \\
pdThis \in chosenLost \\
(abAuthPurse\ pdThis.from).lost = \\
\quad pdThis.value \\
\quad + sumValue(((definitelyLost \cup chosenLost) \\
\qquad \cap \{ pd : PayDetails \mid pd.from = pdThis.from \}) \\
\qquad \setminus \{pdThis\}) \\
(abAuthPurse\ pdThis.to).lost = \\
\quad sumValue(((definitelyLost \cup chosenLost) \\
\qquad \cap \{ pd : PayDetails \mid pd.from = pdThis.to \}) \\
\qquad \setminus \{pdThis\}) \\
\forall name : \{pdThis.from, pdThis.to\} \bullet \\
\quad (abAuthPurse\ name).balance = \\
\qquad (conAuthPurse\ name).balance \\
\qquad + sumValue(((maybeLost \setminus chosenLost) \\
\qquad \cap \{ pd : PayDetails \mid pd.to = name \}) \\
\qquad\quad \setminus \{pdThis\}) \\
OtherPursesRab
\end{array}
$$

In the *WillBeLost* case, *pdThis* is chosen lost, so its value has to be added back into the from purse's *lost* component.

$$
\begin{array}{l}
\textit{RabHasBeenLostClPd} \\
\hline
AbstractBetween \\
chosenLost : \mathbb{P}\ PayDetails \\
pdThis : PayDetails \\
\hline
chosenLost \subseteq maybeLost \\
pdThis \in definitelyLost \\
(abAuthPurse\ pdThis.from).lost = \\
\quad pdThis.value \\
\quad + sumValue(((definitelyLost \cup chosenLost) \\
\qquad \cap \{ pd : PayDetails \mid pd.from = pdThis.from \}) \\
\qquad \setminus \{pdThis\}) \\
(abAuthPurse\ pdThis.to).lost = \\
\quad sumValue(((definitelyLost \cup chosenLost) \\
\qquad \cap \{ pd : PayDetails \mid pd.from = pdThis.to \}) \\
\qquad \setminus \{pdThis\}) \\
\forall name : \{pdThis.from, pdThis.to\} \bullet \\
\quad (abAuthPurse\ name).balance = \\
\qquad (conAuthPurse\ name).balance \\
\qquad + sumValue(((maybeLost \setminus chosenLost) \\
\qquad\quad \cap \{ pd : PayDetails \mid pd.to = name \}) \\
\qquad\quad \setminus \{pdThis\}) \\
OtherPursesRab
\end{array}
$$

In the *HasBeenLost* case, *pdThis* is definitely lost, so its value has to be added back into the from purse's *lost* component.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{RabEndClPd}} \\ \hline
\mathit{AbstractBetween} \\
\mathit{chosenLost} : \mathbb{P}\, \mathit{PayDetails} \\
\mathit{pdThis} : \mathit{PayDetails} \\ \hline
\mathit{chosenLost} \subseteq \mathit{maybeLost} \\
\mathit{pdThis} \notin \mathit{definitelyLost} \cup \mathit{maybeLost} \\
\forall\, \mathit{name} : \mathrm{dom}\, \mathit{conAuthPurse} \cap \{\mathit{pdThis}.\mathit{from}, \mathit{pdThis}.\mathit{to}\} \bullet \\
\quad (\mathit{abAuthPurse}\ \mathit{name}).\mathit{lost} = \\
\qquad \mathit{sumValue}(((\mathit{definitelyLost} \cup \mathit{chosenLost}) \\
\qquad\quad \cap \{\, \mathit{pd} : \mathit{PayDetails} \mid \mathit{pd}.\mathit{from} = \mathit{name} \,\}) \\
\qquad\quad \setminus \{\mathit{pdThis}\}) \\
\quad \wedge\ (\mathit{abAuthPurse}\ \mathit{name}).\mathit{balance} = \\
\qquad (\mathit{conAuthPurse}\ \mathit{name}).\mathit{balance} \\
\qquad + \mathit{sumValue}(((\mathit{maybeLost} \setminus \mathit{chosenLost}) \\
\qquad\quad \cap \{\, \mathit{pd} : \mathit{PayDetails} \mid \mathit{pd}.\mathit{to} = \mathit{name} \,\}) \\
\qquad\quad \setminus \{\mathit{pdThis}\}) \\
\mathit{OtherPursesRab} \\ \hline
\end{array}
$$

In the *End* case, *pdThis* is in neither component, so its value does not have to be added back in anywhere.

10.1.4 Partition

We have the identity[3]:

$$
\begin{array}{l}
\mathit{RabClPd} \\
\vdash \\
\mathit{RabClPd} \Leftrightarrow \\
\quad (\mathit{RabOkayClPd} \\
\qquad \vee\ \mathit{RabWillBeLostClPd} \\
\qquad \vee\ \mathit{RabHasBeenLostClPd} \\
\qquad \vee\ \mathit{RabEndClPd})
\end{array}
$$

Proof:
The four cases differ in the predicate on *pdThis*, which together *partition* the possibilities. It is obvious that the four cases cover the possibilities. We use Lemma 'lost', which says that *definitelyLost* and *maybeLost* are disjoint, to show that the four cases are non-overlapping.

[3] Used in: *Req* **check-operation**, splitting into four cases, section 18.6.

■ 10.1.4

10.1.5 Quantified forms

Because the introduction of the *pd* in *RabClPd* is arbitrary, we have the following identities:

$$\mathit{RabCl} \vdash \mathit{RabCl} \Leftrightarrow (\,\forall\, \mathit{pdThis} : \mathit{PayDetails} \bullet \mathit{RabClPd}\,)$$

and

$$\mathit{RabCl} \vdash \mathit{RabCl} \Leftrightarrow (\,\exists\, \mathit{pdThis} : \mathit{PayDetails} \bullet \mathit{RabClPd}\,)$$

Proof:
That both these identities hold may seem odd, but can be intuitively understood by looking at a similar, smaller example. Consider a non-empty subset of $\mathbb{N}$ called X. Then it is certainly true that

$$\exists\, x : X \bullet X = X \setminus \{x\} \cup \{x\}$$

and also

$$\forall\, x : X \bullet X = X \setminus \{x\} \cup \{x\}$$

■ 10.1.5

We have just chosen to extract an arbitrary element from the set for special naming. We do the same with *RabCl*, selecting an arbitrary *pdThis* for special naming, but without changing the meaning of the schema. This means that we can split up *RabCl* into a collection of four disjunctions on a *pd* in different ways as the proof dictates[4].

10.1.6 The full Retrieve state relation

We also define versions of these schemas with the *pdThis* and *chosenLost* hidden (so they have the same signature as *Rab*):

$$\mathit{RabOkay} \mathrel{\widehat{=}} \mathit{RabOkayClPd} \setminus (\mathit{pdThis}, \mathit{chosenLost})$$

$$\mathit{RabWillBeLost} \mathrel{\widehat{=}} \mathit{RabWillBeLostClPd} \setminus (\mathit{pdThis}, \mathit{chosenLost})$$

$$\mathit{RabHasBeenLost} \mathrel{\widehat{=}} \mathit{RabHasBeenLostClPd} \setminus (\mathit{pdThis}, \mathit{chosenLost})$$

$$\mathit{RabEnd} \mathrel{\widehat{=}} \mathit{RabEndClPd} \setminus (\mathit{pdThis}, \mathit{chosenLost})$$

[4] Used in: lemma 'deterministic', exposing *pdThis* (twice), section 14.4.3.

10.2 Retrieve inputs

Each $\mathcal{A}$ operation has the same type of input, an *AIN*. Each $\mathcal{B}$ operation has the same type of input, a *NAME* and a *MESSAGE*. The input part of the retrieve captures the relationship between these $\mathcal{A}$ and $\mathcal{B}$ inputs.

$$RabIn \mathrel{\widehat{=}} BetwInitIn[a?/g?]$$

The $\mathcal{B}$ inputs are related to $\mathcal{A}$ inputs in the following manner:

RI–1 *Req*: the $\mathcal{A}$ transfer details are in the *req*

RI–2 All other $\mathcal{B}$ inputs: the $\mathcal{A}$ input is *aNullIn*.

10.3 Retrieve outputs

The output retrieve is particularly simple: all $\mathcal{B}$ outputs retrieve to the single $\mathcal{A}$ output.

$$RabOut \mathrel{\widehat{=}} BetwFinOut[a!/g!]$$

Chapter 11

$\mathcal{A}$ to $\mathcal{B}$ initialisation proof

11.1 Proof obligations

The requirement is to prove that the between initial state correctly refines the abstract initial state, and the between inputs correctly refine the abstract inputs. That is,

$$BetweenInitState;\ Rab' \vdash AbInitState$$

$$BetwInitIn;\ RabIn \vdash AbInitIn$$

11.2 Proof of initial state

We successively thin the hypothesis to expose the consequent.

$$\begin{array}{ll} BetweenWorldInit \land Rab' & [\text{hyp}] \\ \quad \Rightarrow Rab' & [\text{thin}] \\ \quad \Rightarrow AbWorld' & [\text{thin}] \\ \quad \Rightarrow AbInitState & [\text{defn } AbInitState] \end{array}$$

■ 11.2

11.3 Proof of initial inputs

Expand *RabIn* and *AbInitIn*.

$$BetwInitIn;\ BetwInitIn[a?/g?] \vdash a? = g?$$

BetwInitIn defines $g?$ as a total function of $(m?, name?)$; call it f. Thin.

$$
\begin{array}{l}
g?, a? : AIN \mid \exists f : MESSAGE \times NAME \rightarrow AIN \bullet \\
\quad \forall m : MESSAGE;\ n : NAME \bullet \\
\quad\quad g? = f(m, n) \land a? = f(m, n) \\
\vdash a? = g?
\end{array}
$$

Simplify and thin.

$$g?, a? : AIN \mid g? = a? \vdash a? = g?$$

■ 11.3
■ 11

Chapter 12

$\mathcal{A}$ to $\mathcal{B}$ finalisation proof

12.1 Proof obligations

The requirement is to prove that the between final state correctly refines the abstract final state, and the between outputs correctly refine the abstract outputs. That is,

$$BetwFinOut \vdash \exists a! : AOUT \bullet RabOut \land AbFinOut$$

$$BetwFinState \vdash \exists AbWorld \bullet Rab \land AbFinState$$

This proof obligation is summarised in figure 12.1.

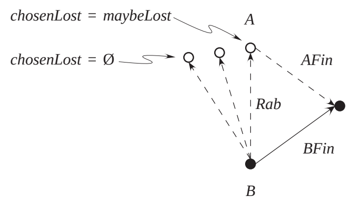

Figure 12.1: Backwards rules finalisation proof obligation

12.2 Output proof

Expand *RabOut* and *AbFinOut*.

$$BetwFinOut \vdash \exists\, a! : AOUT \bullet BetwFinOut[a!/g!] \land a! = g!$$

[*one point*] away the $a!$ in the consequent

$$BetwFinOut \vdash BetwFinOut[g!/g!]$$

■ 12.2

12.3 State proof

We [*cut*] in an *AbWorld*, and put it equal to the *GlobalWorld*.

$$\begin{array}{l} BetwFinState;\ AbWorld \mid abAuthPurse = gAuthPurse \\ \vdash \\ \exists\, AbWorld \bullet Rab \land AbFinState \end{array}$$

Cutting in this new hypothesis requires us to discharge a side-lemma about the existence of such an *AbWorld*. This is trivial to do, by the [*one point*] rule.

We use [*consq exists*] to remove the existential quantifier in the consequent, by using the value just cut in:

$$\begin{array}{l} BetwFinState;\ AbWorld \mid abAuthPurse = gAuthPurse \\ \vdash \\ Rab \land AbFinState \end{array}$$

We prove each of the conjuncts in the consequent separately [*consq conj*], dropping unneeded hypotheses as appropriate [*thin*].

12.3.1 Case *AbFinState*

$$BetwFinState;\ AbWorld \mid abAuthPurse = gAuthPurse \vdash AbFinState$$

The predicates in *AbFinState* occur in the hypothesis, so are satisfied trivially.

■ 12.3.1

12.3.2 Case *Rab*

We expand out *Rab* into its conjuncts:

$$BetwFinState;\ AbWorld \mid abAuthPurse = gAuthPurse \vdash Rab$$

Retrieve of equality

We have the equation

$$\operatorname{dom} abAuthPurse = \operatorname{dom} conAuthPurse$$

which can be shown from the equality of *gAuthPurse* and *conAuthPurse* in *BFinState*, and between *gAuthPurse* and *abAuthPurse* in the hypothesis.

Similarly, in each case the part of the retrieve to be proven has an equality between the abstract and concrete. We show this holds from an equality in that component between global and concrete in *BetwFinState*, and and equality between global and abstract in the hypothesis.

■ 12.3.2

Case *Rab*

$$BetwFinState;\ AbWorld \mid abAuthPurse = gAuthPurse \vdash Rab$$

Expanding *BetwFinState*, thinning unwanted predicates, substituting for *global*, and expanding *Rab*, we get:

$$\begin{array}{l} AuxWorld;\ AbWorld \mid \\ \quad \forall\, name : \operatorname{dom} conAuthPurse \bullet \\ \qquad (abAuthPurse\ name).lost = \\ \qquad\quad sumValue((definitelyLost \cup maybeLost) \\ \qquad\qquad \cap \{\, pd : PayDetails \mid pd.from = name \,\}) \\ \qquad \land (abAuthPurse\ name).balance = (conAuthPurse\ name).balance \\ \vdash \\ \exists\, chosenLost : \mathbb{P}\, maybeLost \bullet \\ \quad \forall\, name : \operatorname{dom} conAuthPurse \bullet \\ \qquad (abAuthPurse\ name).lost = \\ \qquad\quad sumValue((definitelyLost \cup chosenLost) \\ \qquad\qquad \cap \{\, pd : PayDetails \mid pd.from = name \,\}) \\ \qquad \land (abAuthPurse\ name).balance = \\ \qquad\quad (conAuthPurse\ name).balance \\ \qquad\quad + sumValue((maybeLost \setminus chosenLost) \\ \qquad\qquad \cap \{\, pd : PayDetails \mid pd.to = name \,\}) \end{array}$$

We [*one point*] away the *chosenLost* in the consequent by putting it equal to *maybeLost* (having [*cut*] in such a value and proved it exists). We also simplify the equations, now that *maybeLost* \ *chosenLost* is empty:

$$
\begin{array}{l}
AuxWorld;\ AbWorld;\ chosenLost : \mathbb{P}\, PayDetails \mid \\
\quad chosenLost = maybeLost \\
\quad \land\ (\ \forall\ name : \mathrm{dom}\ conAuthPurse \bullet \\
\qquad (abAuthPurse\ name).lost = \\
\qquad\quad sumValue((definitelyLost \cup maybeLost) \\
\qquad\qquad \cap \{\ pd : PayDetails \mid pd.from = name\ \}) \\
\qquad \land\ (abAuthPurse\ name).balance \\
\qquad\quad = (conAuthPurse\ name).balance\ ) \\
\vdash \\
\forall\ name : \mathrm{dom}\ conAuthPurse \bullet \\
\quad (abAuthPurse\ name).lost = \\
\qquad sumValue((definitelyLost \cup maybeLost) \\
\qquad\quad \cap \{\ pd : PayDetails \mid pd.from = name\ \}) \\
\quad \land\ (abAuthPurse\ name).balance = (conAuthPurse\ name).balance
\end{array}
$$

The consequent also appears as an hypothesis, so the proof is complete.

■ 12.3.2
■ 12.3.2
■ 12.3
■ 12

Chapter 13

$\mathcal{A}$ to $\mathcal{B}$ applicability proofs

13.1 Proof obligation

In section 9.2.3 we showed that it is sufficient to prove totality of the concrete operations.

13.2 Proof

Totality for each between operation was shown in the specification consistency proofs, section 8.3.2.

■ 13

Chapter 14

Lemmas for the $\mathcal{A}$ to $\mathcal{B}$ correctness proofs

14.1 Introduction

The correctness proof obligation, to be discharged for each abstract operation AOp, where $AOp \sqsubseteq BOpFull = BOp_1 \lor BOp_2 \lor \ldots$ is the corresponding refinement, is:

$$BOpFull;\ Rab';\ RabOut \vdash \exists\, AbWorld;\ a? : AIN \bullet Rab \land RabIn \land AOp$$

This proof obligation is summarised in figure 14.1. There are multiple lower paths both because the concrete operation is non-deterministic, and because the retrieve is non-deterministic. For each lower path triple of (B, B', A'), we have to find an A that ensures the existence of an upper path; it does not have to be the same A in each case.

There are various classes of $\mathcal{B}$ operation depending on which $\mathcal{A}$ operation is being refined. There are commonalities in the proof structures for these classes. This chapter develops general mechanisms and lemmas to facilitate proving most operations. This fits into the following main areas

- lemma 'multiple refinement': When the $\mathcal{B}$ operation that refines an $\mathcal{A}$ operation in a disjunction of several individual $\mathcal{B}$ operations, the proof obligation can be split into one for each individual $\mathcal{B}$ operation.
- lemma 'ignore': The ignore branch, and any 'abort' branch, of each $\mathcal{B}$ operation need be proved once only.
- lemma 'deterministic': A simplification of all correctness proofs, by exposing the non-determinism in the retrieve, to the three cases **exists-pd**, **exists-chosenLost**, and **check-operation** (with the introduction of two ar-

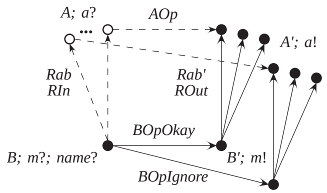

Figure 14.1: The correctness proof. The hypothesis is the existence all of the lower (solid) paths. The proof obligation is to demonstrate the existence of an upper (dashed) path in each case.

bitrary predicates $\mathcal{P}$ and $\mathcal{Q}$, instantiated differently depending on the particular operation).

- lemma 'lost unchanged': Where *maybeLost* and *definitelyLost* are unchanged, the **exists-pd** and **exists-chosenLost** obligations can be automatically discharged.
- lemma 'AbIgnore': A further simplification of the **check-operation** proof obligation, for the operations that refine *AbIgnore*, to **check-operation-ignore**.
- proof that concrete *Ignore* refines *AbIgnore*
- proof that concrete *Abort* refines *AbIgnore*
- lemma 'abort backward': For an operation expressed as *Abort* composed with a simpler version of the operation, we need prove only that the simpler operation is a refinement

The lemmas developed in this chapter are collected together in Appendix C for ease of reference.

14.2 Lemma 'multiple refinement'

In most cases of *AOp*, the corresponding *BOpFull* is a disjunction of many individual $\mathcal{B}$ operations, $BOp_1 \lor BOp_2 \lor \ldots$ whose differences are invisible abstractly. For example, *AbIgnore* is refined by a disjunction of several separate operations.

We use the inference rule [*hyp disj*] to split these large disjunctions into separate proof obligations for each of the $\mathcal{B}$ operations.

14.3 Lemma 'ignore': separating the branches

Each between operation *BOp* is promoted from *BOpPurseOkay*, disjoined with *Ignore*, and sometimes with *Abort*. Call the first disjunction *BOpOkay*:

$$BOpOkay \mathrel{\widehat{=}} \exists\, \Delta ConPurse \bullet \Phi BOp \land BOpPurseOkay$$

We use the inference rule [*hyp disj*], to split the correctness proof into two (or three) parts, one for each disjunct, each of which must be proved.

$$Abort;\ Rab';\ RabOut \vdash \exists\, AbWorld;\ a? : AIN \bullet Rab \land RabIn \land AOp$$
$$Ignore;\ Rab';\ RabOut \vdash \exists\, AbWorld;\ a? : AIN \bullet Rab \land RabIn \land AOp$$
$$BOpOkay;\ Rab';\ RabOut \vdash \exists\, AbWorld;\ a? : AIN \bullet Rab \land RabIn \land AOp$$

All the abstract operations include an option of failing (equivalent to the concrete *Ignore*), which results in no change to the abstract state. We can therefore strengthen the conclusion of the *Ignore* and *Abort* theorems and prove

$$Ignore;\ Rab';\ RabOut \vdash \exists\, AbWorld;\ a? : AIN \bullet Rab \land RabIn \land AbIgnore$$
$$Abort;\ Rab';\ RabOut \vdash \exists\, AbWorld;\ a? : AIN \bullet Rab \land RabIn \land AbIgnore$$

These are independent of the particular operation *AOp*. Thus we need prove these theorems only once (which we do in sections 14.7 and 14.8). To prove the correctness of *BOp* we need additionally to prove the remaining *BOpOkay* theorem.

14.4 Lemma 'deterministic': simplifying the *Okay* branch

The *Okay* branch of the correctness proof is, in general,

$$BOpOkay;\ Rab';\ RabOut \vdash \exists\, AbWorld;\ a? : AIN \bullet Rab \land RabIn \land AOp$$

In order to find an *AbWorld* that is appropriate, we expose the non-determinism in the retrieve. The non-determinism occurs in the *Rab* branch of the retrieve in terms of uncertainty about which transactions still in process will terminate successfully, and which will terminate with a lost value.

We also expose the transaction that is currently in progress, to make it available to the proof.

14.4.1 Choosing an input

We choose a value of *a*? that is consistent with *RabIn*. Since *RabIn* is functional from *m*? and *name*? to *a*?, we know this choice of *a*? is uniquely determined. We [*cut*] this value for *a*? into the hypothesis, and remove the quantifier on *a*? by the [*consq exists*] rule.

We note that *RabIn* in the consequent is independent of the choice of *AbWorld*, so can be pulled out of that quantifier.

$$\begin{array}{l} BOpOkay;\ RabOut;\ Rab';\ a? : AIN \mid RabIn \\ \vdash \\ RabIn \land (\exists AbWorld \bullet Rab \land AOp) \end{array}$$

We split the proof into two on the conjunction in the consequent [*consq conj*], one for *RabIn*, one for $\exists AbWorld \bullet Rab \land AOp$.

RabIn is trivially satisfied by this choice of *a*? in the hypothesis.

The declaration of *a*? in *RabIn* allows us to drop the explicit declaration in the hypothesis, giving

$$BOpOkay;\ RabOut;\ Rab';\ RabIn \vdash \exists AbWorld \bullet Rab \land AOp$$

14.4.2 Cutting in $\Delta ConPurse$

It helps to work with the unpromoted form of the operation. We do this by expanding *BOpOkay*, according to its promoted definition, And [*cut*]ting $\Delta ConPurse$ into the hypothesis such that *BOpPurseOkay* and ΦBOp hold. (The side-lemma is satisfied from the expanded definition of *BOpOkay* in the hypothesis; which states that such a $\Delta ConPurse$ exists.)

$$\begin{array}{l} (\exists \Delta ConPurse \bullet \Phi BOp \land BOpPurseOkay); \\ \qquad RabOut;\ Rab';\ RabIn;\ \Delta ConPurse \mid \\ \quad \Phi BOp \land BOpPurseOkay \\ \vdash \\ \exists AbWorld \bullet Rab \land AOp \end{array}$$

We rearrange the hypothesis, moving ΦBOp and *BOpPurseOkay* from the predicate part to the declaration part. Since ΦBOp declares $\Delta ConPurse$, we remove the latter. We [*thin*] the hypothesis of the expanded definition of *BOpOkay*.

$$\Phi BOp;\ BOpPurseOkay;\ RabOut;\ Rab';\ RabIn \vdash \exists AbWorld \bullet Rab \land AOp$$

14.4.3 Exposing *chosenLost* and *pdThis*

We need to make some of the internal components visible to the proof to enable us to break the proof into sections.

We replace Rab' with the quantified form of $RabCl'$ (section 10.1.2), giving

$$\begin{array}{l} \Phi BOp;\ BOpPurseOkay;\ RabOut; \\ \qquad (\exists chosenLost' : \mathbb{P}\, PayDetails \bullet RabCl');\ RabIn \\ \vdash \\ \exists AbWorld \bullet Rab \land AOp \end{array}$$

We now use [*hyp exists*] to remove the quantification, giving us

$$\begin{array}{l} \Phi BOp;\ BOpPurseOkay;\ RabOut;\ RabCl';\ RabIn \\ \vdash \\ \exists AbWorld \bullet Rab \land AOp \end{array}$$

Next, we [*cut*] in a declaration of an arbitrary payment detail *pdThis*. In practice, this is the *pd* for the payment being processed by *BOpOkay*, but in this general manipulation we don't have enough information to specify this. We therefore constrain the *pdThis* with some arbitrary predicate $\mathcal{P}$.

This generates a non-trivial lemma, **exists-pd**, to be proved in each specific case, as

$$\begin{array}{l} \Phi BOp;\ BOpPurseOkay;\ RabOut;\ RabCl';\ RabIn \\ \vdash \\ \exists pdThis : PayDetails \bullet \mathcal{P} \end{array}$$

and leaves our proof obligation as

$$\begin{array}{l} \Phi BOp;\ BOpPurseOkay;\ RabOut;\ RabCl';\ RabIn;\ pdThis : PayDetails \mid \\ \quad \mathcal{P} \\ \vdash \\ \exists AbWorld \bullet Rab \land AOp \end{array}$$

In the hypothesis we rewrite $RabCl'$ as the universally quantified form of *RabClPd'* (section 10.1.5).

$$\begin{array}{l} \Phi BOp;\ BOpPurseOkay;\ RabOut; \\ \qquad (\forall pdThis' : PayDetails \bullet RabClPd'); \\ \qquad RabIn;\ pdThis : PayDetails \mid \\ \quad \mathcal{P} \\ \vdash \\ \exists AbWorld \bullet Rab \land AOp \end{array}$$

Rather than hypothesising this is true for all *pdThis′* s, we choose a particular value in the quantification. (This is valid, [*hyp uni*], because assuming it true for only a particular value is weaker than assuming it is true for *all* values.) The value we choose for *pdThis′* is that of the value *pdThis*. This substitutes the value *pdThis* for *pdThis′* in the *Rab′* schema. This gives

$$\begin{array}{l} \Phi BOp;\ BOpPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\ RabIn; \\ \qquad\quad pdThis : PayDetails\ | \\ \quad \mathcal{P} \\ \vdash \\ \exists AbWorld \bullet Rab \land AOp \end{array}$$

The declaration in *RabClPd′* allows us to drop the explicit declaration of *pdThis*. So we rewrite this more simply as

$$\begin{array}{l} \Phi BOp;\ BOpPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\ RabIn\ | \\ \quad \mathcal{P} \\ \vdash \\ \exists AbWorld \bullet Rab \land AOp \end{array}$$

In the consequent we do a similar thing: expose *chosenLost*, and rewrite *Rab* as the existentially quantified form of *RabClPd* (section 10.1.5)

$$\begin{array}{l} \Phi BOp;\ BOpPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\ RabIn\ | \\ \quad \mathcal{P} \\ \vdash \\ \exists AbWorld \bullet \\ \quad (\ \exists chosenLost : \mathbb{P}\, PayDetails;\ pd : PayDetails \\ \qquad \bullet RabClPd[pd/pdThis]\ ) \\ \quad \land AOp \end{array}$$

We strengthen the consequent by adding the requirement that the value of *pd* claimed to exist on the right hand side is actually equal to the value *pdThis* declared on the left hand side. Similarly, we constrain *chosenLost* sufficiently. This we do by adding one requirement we always need (namely, that *chosenLost* $\subseteq$ *maybeLost*), and one arbitrary predicate $\mathcal{Q}$, as we did with *pdThis*. This predicate is instantiated to some specific predicate each time this general manipulation is invoked.

$$\begin{array}{l} \Phi BOp;\ BOpPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\ RabIn\ | \\ \quad \mathcal{P} \\ \vdash \\ \exists AbWorld \bullet \\ \quad (\ \exists chosenLost : \mathbb{P}\, PayDetails;\ pd : PayDetails \bullet \\ \qquad pd = pdThis \land \mathcal{Q} \\ \qquad \land chosenLost \subseteq maybeLost \\ \qquad \land RabClPd[pd/pdThis]\ ) \\ \quad \land AOp \end{array}$$

We can remove the *pd* in the consequent with the [*one point*] rule, because we have an explicit value for it (namely, *pdThis*).

$$\begin{array}{l} \Phi BOp;\ BOpPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\ RabIn\ | \\ \quad \mathcal{P} \\ \vdash \\ \exists AbWorld \bullet \\ \quad (\ \exists chosenLost : \mathbb{P}\, PayDetails \bullet \\ \qquad \mathcal{Q} \land chosenLost \subseteq maybeLost \\ \qquad \land RabClPd\ ) \\ \quad \land AOp \end{array}$$

We [*cut*] into the hypothesis a *chosenLost* with the same properties as it has in the consequent (that is, the predicate $\mathcal{Q} \land chosenLost \subseteq maybeLost$). This generates a side lemma that such a value exists, **exists-chosenLost**, which must be discharged in each specific case, as

$$\begin{array}{l} \Phi BOp;\ BOpPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\ RabIn\ | \\ \quad \mathcal{P} \\ \vdash \\ \exists chosenLost : \mathbb{P}\, PayDetails \bullet \mathcal{Q} \land chosenLost \subseteq maybeLost \end{array}$$

This leaves:

$$\begin{array}{l}
\Phi BOp;\ BOpPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\ RabIn; \\
\qquad chosenLost : \mathbb{P}\ PayDetails \mid \\
\quad \mathcal{P} \wedge \mathcal{Q} \wedge chosenLost \subseteq maybeLost \\
\vdash \\
\exists\ AbWorld \bullet \\
\quad (\ \exists\ chosenLost : \mathbb{P}\ PayDetails \bullet \\
\qquad \mathcal{Q} \wedge chosenLost \subseteq maybeLost \\
\qquad \wedge\ RabClPd\ ) \\
\quad \wedge\ AOp
\end{array}$$

We remove the existential quantification using the [*consq exists*] for *chosenLost*:

$$\begin{array}{l}
\Phi BOp;\ BOpPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\ RabIn; \\
\qquad chosenLost : \mathbb{P}\ PayDetails \mid \\
\quad \mathcal{P} \wedge \mathcal{Q} \wedge chosenLost \subseteq maybeLost \\
\vdash \\
\exists\ AbWorld \bullet RabClPd \wedge AOp
\end{array}$$

We break this into two parts, separating the two retrieves in the consequent from *AOp*. We then prove each part.

Cut in *AbWorld* such that *RabClPd* holds. This creates a side lemma to prove that such an *AbWorld* exists, consisting of just the retrieve. (This is discharged in section 14.4.4.)

We are left with

$$\begin{array}{l}
\Phi BOp;\ BOpPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis']; \\
\qquad AbWorld;\ RabClPd;\ RabIn;\ chosenLost : \mathbb{P}\ PayDetails \mid \\
\quad \mathcal{P} \wedge \mathcal{Q} \wedge chosenLost \subseteq maybeLost \\
\vdash \\
RabClPd \wedge AOp
\end{array}$$

We discharge the retrieves in the consequent directly from the hypothesis, and remove *chosenLost* and *chosenLost* ⊆ *maybeLost* as these already occur in *RabClPd*, leaving

$$\begin{array}{l}
\Phi BOp;\ BOpPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis']; \\
\qquad AbWorld;\ RabClPd;\ RabIn \mid \\
\quad \mathcal{P} \wedge \mathcal{Q} \\
\vdash \\
AOp
\end{array}$$

■ 14.4.3

14.4.4 The existence of *AbWorld*

We have to prove the side condition generated when we cut in an *AbWorld* (above).

$$\begin{array}{l}
\Phi BOp;\ BOpPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\ RabIn; \\
\qquad chosenLost : \mathbb{P}\ PayDetails \mid \\
\quad \mathcal{P} \wedge \mathcal{Q} \wedge chosenLost \subseteq maybeLost \\
\vdash \\
\exists\ AbWorld \bullet RabClPd
\end{array}$$

We can prove this by invoking lemma 'AbWorldUnique' (section C.15), provided we can show that the constraints of the hypothesis of that lemma hold.

Certainly we have *BetweenWorld* (from Φ*BOp*), a *pdThis* and a *chosenLost* such that the constraint *chosenLost* ⊆ *maybeLost* holds. This is sufficient to invoke the lemma.

■ 14.4.4

14.4.5 Statement of lemma 'deterministic'

We summarise the results that section 14.4 has developed as a lemma.

Lemma 14.1 (deterministic) The correctness proof for a general *Okay* branch consists of the following three proof obligations:

exists-pd:

$$\begin{array}{l}
\Phi BOp;\ BOpPurseOkay;\ RabOut;\ RabCl';\ RabIn \\
\vdash \\
\exists\ pdThis : PayDetails \bullet \mathcal{P}
\end{array}$$

exists-chosenLost:

$$\begin{array}{l}
\Phi BOp;\ BOpPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\ RabIn \mid \\
\quad \mathcal{P} \\
\vdash \\
\exists\ chosenLost : \mathbb{P}\ PayDetails \bullet \mathcal{Q} \wedge chosenLost \subseteq maybeLost
\end{array}$$

check-operation:

$$\begin{array}{l}
\Phi BOp;\ BOpPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis']; \\
\qquad AbWorld;\ RabClPd;\ RabIn \mid \\
\quad \mathcal{P} \wedge \mathcal{Q} \\
\vdash \\
AOp
\end{array}$$

■

■ 14.4

14.5 Lemma 'lost unchanged'

Many operations do not change *maybeLost* or *definitelyLost*. We call a general such operation $BOp\Xi Lost$.

Lemma 14.2 (lost unchanged) For $BOp\Xi Lost$ operations, where $maybeLost = maybeLost'$ and $definitelyLost' = definitelyLost$, the proof obligations **exists-pd** and **exists-chosenLost** are satisfied automatically by the instantiation of the predicates $\mathcal{P}$ and $\mathcal{Q}$ as:

$$\mathcal{P} \Leftrightarrow true$$

$$\mathcal{Q} \Leftrightarrow chosenLost = chosenLost'$$

leaving the remaining **check-operation** proof obligation as

$$\begin{array}{l} \Phi BOp;\ BOp\Xi LostPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\\ \qquad AbWorld;\ RabClPd;\ RabIn\ |\\ \quad chosenLost = chosenLost'\\ \quad \wedge\ maybeLost = maybeLost'\\ \quad \wedge\ definitelyLost' = definitelyLost\\ \vdash\\ AOp \end{array}$$

■

14.5.1 Proof

We add the hypotheses $maybeLost = maybeLost'$ and $definitelyLost' = definitelyLost$ to the proof obligations for these $BOp\Xi Lost$ operations.

exists-pd

$$\begin{array}{l} \Phi BOp;\ BOp\Xi LostPurseOkay;\ RabOut;\ RabCl';\ RabIn\ |\\ \quad maybeLost' = maybeLost\\ \quad \wedge\ definitelyLost' = definitelyLost\\ \vdash\\ \exists\, pdThis : PayDetails \bullet true \end{array}$$

This is trivially true.

■ 14.5.1

exists-chosenLost

$$\begin{array}{l} \Phi BOp;\ BOp\Xi LostPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\\ \quad RabIn\ |\\ \quad maybeLost' = maybeLost\\ \quad \wedge\ definitelyLost' = definitelyLost\\ \vdash\\ \exists\, chosenLost : \mathbb{P}\, PayDetails \bullet\\ \quad chosenLost = chosenLost' \wedge chosenLost \subseteq maybeLost \end{array}$$

We apply the [*one point*] rule to remove the existential quantifier in the consequent, substitute for *maybeLost*, and [*thin*].

$$RabClPd'[pdThis/pdThis'] \vdash chosenLost' \subseteq maybeLost'$$

The hypothesis $RabClPd'[pdThis/pdThis']$ has $chosenLost' \subseteq maybeLost'$.

■ 14.5.1

■ 14.5

14.5.2 Sufficient conditions for invoking lemma 'lost unchanged'

Since ΦBOp gives us that *archive* is unchanged, sufficient conditions for invoking lemma 'lost unchanged' are that the operation in question changes neither the purse's status (hence no movement into or out of *epv* or *epa*) nor its exception log (hence no change to *from* logs or *to* logs).

14.6 Lemma '*AbIgnore*': Operations that refine *AbIgnore*

As shown in section 14.2, to prove the refinement of the abstract identity operation *AbIgnore*, we can separately prove correctness for each of the between operations *StartFrom*, *StartTo*, *Val*, *Ack*, *ReadExceptionLog*, *ClearExceptionLog*, *AuthoriseExLogClear*, *Archive*, *Ignore*, *Increase*, and *Abort*.

For those which are structured as promoted operations (that is, all except *Archive* and *Ignore*), consider a general such operation, call it *BOpIg*. We note that all *BOpIg* operations have the properties:

- *BOpIg* is a promoted operation, and thus alters only one concrete purse. It has the form

$$\exists\, \Delta ConPurse \bullet \Phi BOp \wedge BOpIgPurse$$

- for any purse, the *name* is unchanged (by definition of the single purse operations)
- the domain of *conAuthPurse* is unchanged (by construction of the promotion)
- for any purse, either *nextSeqNo* is unchanged, or increased.

$$\forall\, BOpIgPurse \bullet nextSeqNo \leq nextSeqNo'$$

We use these properties to simplify the proof obligation for the *BOpIg* operations.

We invoke lemma 'deterministic' (section 14.4) to reduce the *BOpIg* proof obligation to **exists-pd**, **exists-chosenLost** and **check-operation**:

$$\begin{array}{l}
\Phi BOp;\ BOpIgPurse;\ RabOut;\ RabClPd'[pdThis/pdThis'];\\
\qquad AbWorld;\ RabClPd;\ RabIn \mid\\
\quad \mathcal{P} \land \mathcal{Q}\\
\vdash\\
AbIgnore
\end{array}$$

Lemma 14.3 (*AbIgnore*) For a *BOpIg* operation, the **check-operation** proof obligation reduces to **check-operation-ignore**[1]:

$$\begin{array}{l}
\Phi BOp;\ BOpIgPurse;\ RabClPd'[pdThis/pdThis'];\ AbWorld;\ RabClPd \mid\\
\quad \mathcal{P} \land \mathcal{Q}\\
\vdash\\
\forall\, n : \mathrm{dom}\, abAuthPurse \bullet\\
\quad (abAuthPurse'\ n).lost = (abAuthPurse\ n).lost\\
\quad \land\ (abAuthPurse'\ n).balance = (abAuthPurse\ n).balance
\end{array}$$

■

Proof:

We take the **check-operation** proof obligation, and expand *AbIgnore*. The *BOpIgPurse* operations have certain properties in common; we explicitly state these in the hypothesis.

[1]Used in: *Ignore*, 14.7.2.

$$\begin{array}{l}
\Phi BOp;\ BOpIgPurse;\ RabOut;\ RabClPd'[pdThis/pdThis'];\\
\qquad AbWorld;\ RabClPd;\ RabIn \mid\\
\quad \mathcal{P} \land \mathcal{Q}\\
\quad \land\ name' = name\\
\quad \land\ nextSeqNo' \geq nextSeqNo\\
\vdash\\
AbOp \land abAuthPurse' = abAuthPurse
\end{array}$$

We use [*consq conj*] to split this proof into two parts. The *AbOp* part is trivial: there are no constraints. This leaves the other conjunct to be proven, which is rewritten as follows:

$$\begin{array}{l}
\Phi BOp;\ BOpIgPurse;\ RabOut;\ RabClPd'[pdThis/pdThis'];\\
\qquad AbWorld;\ RabClPd;\ RabIn \mid\\
\quad \mathcal{P} \land \mathcal{Q}\\
\quad \land\ name' = name\\
\quad \land\ nextSeqNo' \geq nextSeqNo\\
\vdash\\
\forall\, n : \mathrm{dom}\, abAuthPurse \bullet abAuthPurse'\ n = abAuthPurse\ n
\end{array}$$

We prove this component by component. From ΦBOp in the hypothesis, all concrete purses other than purse *name*? remain unchanged. For the purse *name*?, we also have the equality of the pre and post states of *name*. This leaves the components *balanace* and *lost*. We use this with [*consq conj*] to reduce our proof requirement to the following:

$$\begin{array}{l}
\Phi BOp;\ BOpIgPurse;\ RabOut;\ RabClPd'[pdThis/pdThis'];\\
\qquad AbWorld;\ RabClPd;\ RabIn \mid\\
\quad \mathcal{P} \land \mathcal{Q}\\
\quad \land\ name' = name\\
\quad \land\ nextSeqNo' \geq nextSeqNo\\
\vdash\\
\forall\, n : \mathrm{dom}\, abAuthPurse \bullet\\
\quad (abAuthPurse'\ n).balance = (abAuthPurse\ n).balance\\
\quad \land\ (abAuthPurse'\ n).lost = (abAuthPurse\ n).lost
\end{array}$$

We then [*thin*] the hypothesis to get the following, which proves the *AbIgnore* lemma.

$$\begin{array}{l}
\Phi BOp;\ BOpIgPurse;\ RabClPd'[pdThis/pdThis'];\ AbWorld;\ RabClPd \mid \\
\quad \mathcal{P} \land \mathcal{Q} \\
\vdash \\
\forall\, n : \mathrm{dom}\ abAuthPurse \bullet \\
\quad (abAuthPurse'\ n).balance = (abAuthPurse\ n).balance \\
\quad \land\ (abAuthPurse'\ n).lost = (abAuthPurse\ n).lost
\end{array}$$

■ 14.6

14.7 *Ignore* refines *AbIgnore*

As we saw at the end of section 14.3, by splitting up promoted operations, we have generated a requirement to prove the correctness of the *Ignore* branch once only. We do that here.

14.7.1 Invoking lemma 'deterministic'

Lemma 'deterministic' (section 14.4.5) cannot be applied as-is, because *Ignore* is not written as a promotion (in order to ensure it is total). However, the arguments to split the proof obligation into three parts follow in exactly the same manner even if the unpromoted purse is not exposed. The proof obligations simply have *BOpOkay* in the hypothesis, instead of ΦBOp; *BOpPurseOkay*. We use that form to simplify the *Ignore* proof obligation to three parts, and then invoke lemma 'lost unchanged' to discharge the first two obligations. We similarly use lemma 'AbIgnore' to simplify the third proof obligation to **check-operation-ignore**.

14.7.2 check-operation-ignore

$$\begin{array}{l}
Ignore;\ RabClPd'[pdThis/pdThis'];\ AbWorld;\ RabClPd \mid \\
\quad chosenLost = chosenLost' \\
\quad \land\ maybeLost = maybeLost' \\
\quad \land\ definitelyLost = definitelyLost' \\
\vdash \\
\forall\, n : \mathrm{dom}\ abAuthPurse \bullet \\
\quad (abAuthPurse'\ n).balance = (abAuthPurse\ n).balance \\
\quad \land\ (abAuthPurse'\ n).lost = (abAuthPurse\ n).lost
\end{array}$$

The proof of this is immediate: *Ignore* changes no values, *definitelyLost*, *maybeLost* and *chosenLost* do not change, from the hypothesis; so the abstract *balance* and *lost*, which depend only on these unchanging values, are unchanged.

■ 14.7.2

■ 14.7

14.8 *Abort* refines *AbIgnore*

As we saw at the end of section 14.3, by splitting up promoted operations, we have generated a requirement to prove the correctness of the *Abort* branch once only. We do that here. We cast it as a lemma, because we also use it to simplify the proofs of operations that first abort (lemma 'abort backward').

Lemma 14.4 (*Abort* refines *AbIgnore*) Concrete *Abort* refines abstract *Ignore*.[2]

$$Abort;\ Rab';\ RabOut \vdash \exists\, AbWorld;\ a? : AIN \bullet Rab \land RabIn \land AbIgnore$$

■

Proof:

Abort is written as a disjunction between *Ignore* and a promoted *AbortPurseOkay*. We use lemma 'ignore' (section 14.3) to simplify the proof obligation to the correctness of *Ignore* (which we discharge in section 14.7), and the *Okay* branch, which we prove here.

14.8.1 Invoking lemma 'deterministic'

We use lemma 'deterministic' (section 14.4.5) to simplify the proof obligations and then lemma 'AbIgnore' (section 14.6) to simplify the **check-operation** step.

We have to instantiate the predicates $\mathcal{P}$ and $\mathcal{Q}$.

$\mathcal{P}$ is a predicate identifying the *pdThis* involved in the transaction. This is the *pdAuth* stored in the aborting purse, unless the aborting purse is in *eaFrom*, in which case we don't have a defined transaction. We cater for the case of no transaction in the $\mathcal{Q}$ predicate, so $\mathcal{P}$ can safely be defined as

$$\mathcal{P} \Leftrightarrow pdThis = pdAuth$$

$\mathcal{Q}$ is a predicate on *chosenLost*. The after set *chosenLost'* either has *pdThis* removed (if the transaction moves it from *chosenLost* to *definitelyLost*), or is

[2] Used in proof of lemma abort, 14.9

unchanged (because *pdThis* was not in *chosenLost* to start with) or is unchanged because there was no transaction to abort. Hence

$$
\begin{aligned}
&\mathcal{Q} \Leftrightarrow \\
&\quad (pdThis \in maybeLost \wedge chosenLost = chosenLost' \cup \{pdThis\}) \\
&\quad \vee (pdThis \notin maybeLost \wedge status \neq eaFrom \wedge \\
&\qquad chosenLost = chosenLost') \\
&\quad \vee (status = eaFrom \wedge chosenLost = chosenLost')
\end{aligned}
$$

14.8.2 exists-pd

The unpromoted operation *AbortPurseOkay* is incomplete. The output, $m! = \perp$, is not provided until promotion.

$$
\begin{aligned}
&\Phi BOp;\ AbortPurseOkay;\ RabOut;\ RabCl';\ RabIn \mid m! = \perp \\
&\vdash \\
&\exists\, pdThis : PayDetails \bullet pdThis = pdAuth
\end{aligned}
$$

This is immediate by the one point rule.
■ 14.8.2

14.8.3 Three cases

We split the remaining two proofs, of **exists-chosenLost** and **check-operation**, into three cases each, for each of the three disjuncts of $\mathcal{Q}$. We start by arguing the behaviour of *maybeLost* and *definitelyLost* in the three cases.

- **Case 1: aborted transaction in 'limbo'**: The aborting purse is the *to* purse in *epv*; the corresponding *from* purse is in *epa* or has logged. Hence aborting the transaction will definitely lose the value.

  $$pdThis \in maybeLost$$

- **Case 2: aborted transaction not in 'limbo'**: The aborting purse is not the *to* purse in *epv*, or the corresponding *from* purse is not in *epa* and has not logged. The transaction has either not got far enough to lose anything, or has progressed sufficiently far that the value was already either successfully transferred or definitely lost.

  $$pdThis \notin maybeLost \wedge status \neq eaFrom$$

- **Case 3: no transaction to abort**: The aborting purse is in *eaFrom*, so has no defined transaction. Nothing is aborted, so no value is lost.

  $$status = eaFrom$$

Case 1: old transaction in limbo

$$pdThis \in (fromInEpa \cup fromLogged) \cap toInEpv$$

We argue about the behaviour of *maybeLost* and *definitelyLost* using the fact that the purse is the *to* purse initially in *epv* in the aborting transaction, and it logs the old transaction and moves to *eaFrom*. We argue that the transaction *pdThis*, initially in *maybeLost* by construction, is moved into *definitelyLost'* by this case of the *Abort* operation. The transaction was far enough progressed that value may be lost, and it is lost in this case.

Behaviour of *fromInEpa* and *fromLogged* *pdThis* is in *toInEpv* (by our case assumption), so the only purse undergoing any change (*name*?) is the *to* purse; hence there can be no change to the status or logs of any *from* purse. Hence

$$fromInEpa = fromInEpa'$$

$$fromLogged = fromLogged'$$

Behaviour of *toInEpv* *pdThis* is in *toInEpv* (by our case assumption); *pdThis* is not in *toInEpv'* (*Abort* puts the purse into *eaFrom*); all other purses and transactions remain unchanged. So

$$toInEpv = toInEpv' \cup \{pdThis\}$$

Behaviour of *toLogged* *pdThis* is not in *toLogged* (using lemma 'notLogged-AndIn' with $pdThis \in toInEpv$); *pdThis* is in *toLogged'* (the purse makes a *to* log when it aborts from *epv*); all other purses and transactions remain unchanged. So

$$toLogged = toLogged' \setminus \{pdThis\}$$

Behaviour of *definitelyLost*

$$\begin{aligned}
& \mathit{definitelyLost} \\
& \quad = \mathit{toLogged} \cap (\mathit{fromLogged} \cup \mathit{fromInEpa}) && [\text{defn } \mathit{definitelyLost}] \\
& \quad = (\mathit{toLogged}' \setminus \{\mathit{pdThis}\}) \cap (\mathit{fromLogged}' \cup \mathit{fromInEpa}') && [\text{above}] \\
& \quad = (\mathit{toLogged}' \cap (\mathit{fromLogged}' \cup \mathit{fromInEpa}')) \setminus \{\mathit{pdThis}\} && [\text{rearrange}] \\
& \quad = \mathit{definitelyLost}' \setminus \{\mathit{pdThis}\} && [\text{defn } \mathit{definitelyLost}']
\end{aligned}$$

Behaviour of *maybeLost*

$$\begin{aligned}
& \mathit{maybeLost} \\
& \quad = (\mathit{fromInEpa} \cup \mathit{fromLogged}) \cap \mathit{toInEpv} && [\text{defn } \mathit{maybeLost}] \\
& \quad = (\mathit{fromInEpa}' \cup \mathit{fromLogged}') \cap (\mathit{toInEpv}' \cup \{\mathit{pdThis}\}) && [\text{above}] \\
& \quad = ((\mathit{fromInEpa}' \cup \mathit{fromLogged}') \cap \mathit{toInEpv}') \\
& \qquad \cup ((\mathit{fromInEpa}' \cup \mathit{fromLogged}') \cap \{\mathit{pdThis}\}) && [\text{Spivey}] \\
& \quad = ((\mathit{fromInEpa}' \cup \mathit{fromLogged}') \cap \mathit{toInEpv}') \\
& \qquad \cup \{\mathit{pdThis}\} && [\text{case assumption}] \\
& \quad = \mathit{maybeLost}' \cup \{\mathit{pdThis}\} && [\text{defn } \mathit{maybeLost}']
\end{aligned}$$

Case 2: old transaction not in limbo

$$\mathit{pdThis} \notin (\mathit{fromInEpa} \cup \mathit{fromLogged}) \cap \mathit{toInEpv} \wedge \mathit{status} \neq \mathit{eaFrom}$$

We argue that the transaction *pdThis* is not moved into or out of *maybeLost* or *definitelyLost* by this case of the *Abort* operation.

Behaviour of $\mathit{fromInEpa} \cup \mathit{fromLogged}$ If *pdThis* is in *fromInEpa* it is also in $\mathit{fromLogged}'$ (the purse is in *epa*, so it makes a *from* log when it aborts); if *pdThis* is in *fromLogged* it is also in $\mathit{fromLogged}'$ (logs cannot be removed); if *pdThis* is not in $\mathit{fromInEpa} \cup \mathit{fromLogged}$ it is not in $\mathit{fromLogged}'$ (the purse is not in *epa*, so does not make a *from* log when it aborts), and not in $\mathit{fromInEpa}'$ (because it ends in *eaFrom*); all other purses and transactions remain unchanged. So

$$\mathit{fromInEpa} \cup \mathit{fromLogged} = \mathit{fromInEpa}' \cup \mathit{fromLogged}'$$

Behaviour of *definitelyLost* The cases allowed by our case assumption are:

- *pdThis* refers to the *to* purse in *epv*, hence is not in

  $$\mathit{fromInEpa} \cup \mathit{fromLogged}$$

  and hence not in *definitelyLost*. Also it is not in $\mathit{fromInEpa}' \cup \mathit{fromLogged}'$, and hence not in $\mathit{definitelyLost}'$. So *definitelyLost* is unchanged.
- *pdThis* refers to the *to* purse, but not in *epv*, or *pdThis* refers to the *from* purse. Hence *toLogged* is unchanged, since no *to* log is written, and logs cannot be lost. Also $\mathit{fromInEpa} \cup \mathit{fromLogged}$ is unchanged. So *definitelyLost* is unchanged.

So

$$\mathit{definitelyLost}' = \mathit{definitelyLost}$$

Behaviour of *maybeLost* The cases allowed by our case assumption are:

- *pdThis* refers to the *to* purse in *epv*, hence is not in

  $$\mathit{fromInEpa} \cup \mathit{fromLogged}$$

  and hence not in *maybeLost*. Also it is not in $\mathit{fromInEpa}' \cup \mathit{fromLogged}'$, and hence not in $\mathit{maybeLost}'$. So *maybeLost* is unchanged.
- *pdThis* refers to the *to* purse, but not in *epv*, or *pdThis* refers to the *from* purse. Hence *toInEpv* is unchanged, since no purse moves out of or in to *epv*. Also $\mathit{fromInEpa} \cup \mathit{fromLogged}$ is unchanged. So *maybeLost* is unchanged.

So

$$\mathit{maybeLost}' = \mathit{maybeLost}$$

Case 3: no transaction to abort

$$\mathit{status} = \mathit{eaFrom}$$

From *AbortPurseOkay*, no purses change state and no logs are written. Therefore, *definitelyLost* and *maybeLost* don't change.

$$\mathit{definitelyLost}' = \mathit{definitelyLost}$$

$$\mathit{maybeLost}' = \mathit{maybeLost}$$

14.8.4 exists-chosenLost

We now use the behaviour of *maybeLost* and *definitelyLost* in the three cases to prove **exists-chosenLost**.

$$
\begin{array}{l}
\Phi BOp;\ AbortPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\ RabIn\ |\\
\quad m! = \bot\\
\quad \land\ pdThis = pdAuth\\
\vdash\\
\exists\ chosenLost : \mathbb{P}\ PayDetails \bullet\\
\quad (pdThis \in maybeLost \land chosenLost = chosenLost' \cup \{pdThis\}\\
\qquad \lor\ pdThis \notin maybeLost \land status \neq eaFrom\\
\qquad\qquad \land\ chosenLost = chosenLost'\\
\qquad \lor\ status = eaFrom \land chosenLost = chosenLost'\ )\\
\quad \land\ chosenLost \subseteq maybeLost
\end{array}
$$

We push the existential quantifier in the consequent into the predicates:

$$
\begin{array}{l}
\Phi BOp;\ AbortPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\ RabIn\ |\\
\quad m! = \bot\\
\quad \land\ pdThis = pdAuth\\
\vdash\\
pdThis \in maybeLost\\
\quad \land\ (\ \exists\ chosenLost : \mathbb{P}\ PayDetails \bullet\\
\qquad chosenLost = chosenLost' \cup \{pdThis\}\\
\qquad \land\ chosenLost \subseteq maybeLost\ )\\
\lor\ pdThis \notin maybeLost \land status \neq eaFrom\\
\quad \land\ (\ \exists\ chosenLost : \mathbb{P}\ PayDetails \bullet\\
\qquad chosenLost = chosenLost'\\
\qquad \land\ chosenLost \subseteq maybeLost\ )\\
\lor\ status = eaFrom\\
\quad \land\ (\ \exists\ chosenLost : \mathbb{P}\ PayDetails \bullet\\
\qquad chosenLost = chosenLost'\\
\qquad \land\ chosenLost \subseteq maybeLost\ )
\end{array}
$$

In each case, we [*one point*] away the *chosenLost* because the predicate includes an explicit definition for it.

$$
\begin{array}{l}
\Phi BOp;\ AbortPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\ RabIn\ |\\
\quad m! = \bot\\
\quad \land\ pdThis = pdAuth\\
\vdash\\
pdThis \in maybeLost\\
\quad \land\ chosenLost' \cup \{pdThis\} \subseteq maybeLost\\
\lor\ pdThis \notin maybeLost \land status \neq eaFrom\\
\quad \land\ chosenLost' \subseteq maybeLost\\
\lor\ status = eaFrom\\
\quad \land\ chosenLost' \subseteq maybeLost
\end{array}
$$

In each case, the predicate is of the form $(a \land b)$, and we argue below that $a \Rightarrow b$. This allows us to replace $(a \land b)$ with a. If we do this, we obtain

$$
\begin{array}{l}
\Phi BOp;\ AbortPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\ RabIn\ |\\
\quad m! = \bot\\
\quad \land\ pdThis = pdAuth\\
\vdash\\
pdThis \in maybeLost\\
\lor\ pdThis \notin maybeLost \land status \neq eaFrom\\
\lor\ status = eaFrom
\end{array}
$$

which is *true*. We now carry out the argument as described above for each of the three disjuncts.

Case 1: old transaction in limbo

We must show that under the assumptions of this lemma and in this case

$$
\begin{array}{l}
pdThis \in maybeLost \Rightarrow\\
\quad chosenLost' \cup \{pdThis\} \subseteq maybeLost
\end{array}
$$

This follows by:

$$
\begin{array}{lr}
chosenLost' \cup \{pdThis\} & \\
\quad \subseteq maybeLost' \cup \{pdThis\} & \text{[hypothesis]}\\
\quad \subseteq maybeLost & \text{[previous argument for case 1]}
\end{array}
$$

■ 14.8.4

Case 2: old transaction not in limbo

We must show that under the assumptions of this lemma and in this case

$$\begin{array}{l} pdThis \notin maybeLost \land status \neq eaFrom \Rightarrow \\ \quad chosenLost' \subseteq maybeLost \end{array}$$

This follows by

$$chosenLost' \subseteq maybeLost' \qquad \text{[hypothesis]}$$

$$\Rightarrow chosenLost' \subseteq maybeLost \qquad \text{[previous argument for case 2]}$$

■ 14.8.4

Case 3: no transaction to abort

We must show that under the assumptions of this lemma and in this case

$$\begin{array}{l} status = eaFrom \Rightarrow \\ \quad chosenLost' \subseteq maybeLost \end{array}$$

This follows by

$$chosenLost' \subseteq maybeLost' \qquad \text{[hypothesis]}$$

$$\Rightarrow chosenLost' \subseteq maybeLost \qquad \text{[previous argument for case 3]}$$

■ 14.8.4
■ 14.8.4

14.8.5 check-operation-ignore

We now use the behaviour of *maybeLost* and *definitelyLost* in the three cases to prove **check-operation-ignore**.

$$\begin{array}{l} \Phi BOp;\ AbortPurseOkay;\ RabClPd'[pdThis/pdThis'];\\ \quad AbWorld;\ RabClPd \mid \\ \quad pdThis = pdAuth \\ \quad \land\ (pdThis \in maybeLost \land chosenLost = chosenLost' \cup \{pdThis\} \\ \qquad \lor\ pdThis \notin maybeLost \land status \neq eaFrom \\ \qquad\qquad \land\ chosenLost = chosenLost' \\ \qquad \lor\ status = eaFrom \land chosenLost = chosenLost') \\ \vdash \\ \forall\, n : \mathrm{dom}\, abAuthPurse \bullet \\ \quad (abAuthPurse'\ n).balance = (abAuthPurse\ n).balance \\ \quad \land\ (abAuthPurse'\ n).lost = (abAuthPurse\ n).lost \end{array}$$

We can prove this for each of the three disjuncts in the hypothesis by [*hyp disj*].

Case 1: old transaction in limbo

lost is a function of $definitelyLost \cup chosenLost$. The *pdThis* moves from *chosenLost* to *definitelyLost'*, so the union is unchanged.

balance is a function of $maybeLost \setminus chosenLost$. The *pdThis* moves from *chosenLost*, and hence from *maybeLost*, so the difference is unchanged.

■ 14.8.5

Case 2+3: old transaction not in limbo or no transaction

From $chosenLost = chosenLost'$ and the arguments above, all the relevant sets are unchanging, so *lost* and *balalnce* are unchanging.

■ 14.8.5
■ 14.8.5
■ 14.8

14.9 Lemma 'abort backward': operations that first abort

Some of the concrete operations are written as a composition of *AbortPurseOkay* with a simpler operation starting from *eaFrom* (*StartFrom*, *StartTo*, *ReadExceptionLog*, *ExceptionLogClear*).

Lemma 14.5 (abort backward) Where a concrete operation is written as a composition of *AbortPurseOkay* and a simpler operation starting from *eaFrom*, it is sufficient to prove that the promotion of the simpler operation alone refines the relevant abstract operation.

$$\begin{array}{l} \exists \Delta ConPurse \bullet \Phi BOp \land (AbortPurseOkay \mathbin{⨟} BOpPurseEafromOkay); \\ \quad\quad Rab'; RabOut; \\ \quad\quad (\forall BOpEafromOkay; Rab'; RabOut \bullet \\ \quad\quad\quad \exists AbWorld; a? : AIN \bullet Rab \land RabIn \land AOp) \\ \vdash \\ \exists AbWorld; a? : AIN \bullet Rab \land RabIn \land AOp \end{array}$$

■

Proof

- Use lemma 'promoted composition' (section C.11) to rewrite the promotion of the composition to a composition of promotions, yielding

$$\begin{array}{l} (AbortOkay \mathbin{⨟} BOpEafromOkay); \\ \quad\quad Rab'; RabOut; \\ \quad\quad (\forall BOpEafromOkay; Rab'; RabOut \bullet \\ \quad\quad\quad \exists AbWorld; a? : AIN \bullet Rab \land RabIn \land AOp) \\ \vdash \\ \exists AbWorld; a? : AIN \bullet Rab \land RabIn \land AOp \end{array}$$

- If $BOp1$ refines $AOp1$ and $BOp2$ refines $AOp2$, then $BOp1 \mathbin{⨟} BOp2$ refines $AOp1 \mathbin{⨟} AOp2$ (invoke lemma 'compose backward', section C.9).
- Take $BOp1 = AbortOkay$, $AOp1 = AbIgnore$, and invoke lemma '*Abort* refines *AbIgnore*' (section 14.8), to discharge this proof.
- Take $BOp2 = BOpEafromOkay$, $AOp2 = AOp$, and note that we have that *BOp* refines *AOp* in the hypothesis.
- Note that $AbIgnore \mathbin{⨟} AOp = AOp$, to reduce this expression in the consequent.

■ 14.9

14.10 Summary of lemmas

In section 9.2.4 we reduced the refinement correctness proof for an operation to:

$$BOp; Rab'; RabOut \vdash \exists AbWorld; a? : AIN \bullet Rab \land RabIn \land AOp$$

We then built up a set of lemmas which may be used to simplify this proof requirement.

AOp and *BOp* are often disjunctions of simpler operations, and lemmas 'multiple refinement' (section 14.2) and 'ignore' (section 14.3) are used to prove that any *Ignore* or *Abort* branches of *BOp* need be proved once only for all *BOp*s. These two branches are proved in lemmas later on, after further simplification for a general disjunct (*Ignore*, *Abort* or *Okay*) of *BOp*. This simplification starts with lemma 'deterministic' (section 14.4) which removes the $\exists AbWorld$ in the consequent of the correctness obligation. In doing so, it requires us to prove three side-lemmas (**exists-pd**, **exists-chosenLost**, **check-operation**). Lemma 'lost unchanged' (section 14.5) allows the side-lemmas **exists-pd** and **exists-chosenLost** to be discharged immediately given certain conditions. Lemma '*AbIgnore*' (section 14.6) then provides a simplification of the side-lemma **check-operation** when *AOp* is *AbIgnore*.

We can now prove that the *Ignore* and *Abort* branches of *BOp* are correct with respect to *AOp*. Section 14.7 proves that *Ignore* refines *AbIgnore*, and lemma '*Abort* refines *AbIgnore*' (section 14.8) handles the *Abort* branch. With lemmas 'multiple refinement' and 'ignore', this has now proved the correctness of the *Ignore* and *Abort* branches of all *BOp*.

Where the *Okay* branch of an operation is composed of *Abort* followed by the 'active' operation, lemma 'abort backward' gives us that we only need to prove the 'active' part.

Returning to the proof obligation written above, any of the *Ignore* or *Abort* branches of a *BOp* operation are dealt with by the lemmas. This leaves the *Okay* branch (if this contains an initial *Abort*, this can be ignored — from lemma 'abort backward' we need only prove the non-aborting part). Usually, we then apply lemma 'deterministic' yielding a number of side-lemmas. These may sometimes be further simplified using lemmas 'lost unchanged' and '*AbIgnore*'. The remaining proof is then particular to the *BOp*.

Chapter 15

Correctness of *Increase*

15.1 Proof obligation

We have to prove the correct refinement of each abstract operation. In section 9.2.4 we give a general simplification of the correctness proof. We use lemma 'multiple refinement' (section 14.2) to split the proof obligation for each $\mathcal{A}$ operation into one for each individual $\mathcal{B}$ operation.

This chapter proves the $\mathcal{B}$ operation.

- We use lemma 'ignore' (see section 14.3) to simplify the proof obligation by proving the correctness of *Ignore* (in section 14.7), leaving the *Okay* branch to be proven here.
- We use lemma 'deterministic' (section C.1) to reduce the proof obligation to the three cases **exists-pd**, **exists-chosenLost**, and **check-operation**.
- Since this operation leaves the sets *maybeLost* and *definitelyLost* unchanged, we use lemma 'lost unchanged' (section C.2) to discharge the **exists pd-and exists chosenLost-obligations** automatically.
- Since this operation refines *AbIgnore*, we use lemma '*AbIgnore*' (from section C.3) to simplify **check-operation** to **check-operation-ignore**.

15.2 Invoking lemma 'lost unchanged'

Section 14.5.2 gives sufficient conditions to be able to invoke lemma 'lost unchanged'. These are that the unpromoted operation changes neither the *status* nor the exception log of the purse. *Increase* includes Ξ*ConPurseIncrease*, which says exactly that. We can therefore invoke lemma 'Lost unchanged'.

15.3 check-operation-ignore

$$
\begin{array}{l}
\Phi BOp;\ IncreasePurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\\
\qquad AbWorld;\ RabClPd;\ RabIn \mid\\
\quad chosenLost' = chosenLost\\
\quad \land\ maybeLost' = maybeLost\\
\quad \land\ definitelyLost' = definitelyLost\\
\vdash\\
\forall\ n : \mathrm{dom}\ abAuthPurse \bullet\\
\quad (abAuthPurse'\ n).balance = (abAuthPurse\ n).balance\\
\quad \land\ (abAuthPurse'\ n).lost = (abAuthPurse\ n).lost
\end{array}
$$

Proof: We have that *maybeLost* and *definitelyLost* are unchanged from the hypothesis. This shows that the *balance* and *lost* components of all the abstract purses remain unchanged.

■ 15.3

■ 15

Chapter 16

Correctness of *StartFrom*

16.1 Proof obligation

We have to prove the correct refinement of each abstract operation. In section 9.2.4 we give a general simplification of the correctness proof. We use lemma ‘multiple refinement’ (section 14.2) to split the proof obligation for each $\mathcal{A}$ operation into one for each individual $\mathcal{B}$ operation.

This chapter proves the $\mathcal{B}$ operation.

- We use lemma ‘ignore’ (see section 14.3) to simplify the proof obligation by proving the correctness of *Ignore* (in section 14.7), and *Abort* (in section 14.8), leaving the *Okay* branch to be proven here.
- Since the *Okay* branch of this operation is expressed as a promotion of *AbortPurseOkay* composed with a simpler *EafromPurseOkay* operation, we use lemma ‘abort backward’ (section C.5), and prove only that the promotion of the simpler operation is a refinement.
- We use lemma ‘deterministic’ (section C.1) to reduce the proof obligation to the three cases **exists-pd**, **exists-chosenLost**, and **check-operation**.
- Since this operation refines *AbIgnore*, we use lemma ‘*AbIgnore*’ (from section C.3) to simplify **check-operation** to **check-operation-ignore**.

16.2 Instantiating lemma 'deterministic'

We take the *pdThis* to be the *pdAuth* created by the start operation, and *chosenLost* to be unchanging.

$$\mathcal{P} \Leftrightarrow pdThis = (conAuthPurse'\ name?).pdAuth$$
$$\mathcal{Q} \Leftrightarrow chosenLost = chosenLost'$$

16.3 Behaviour of *maybeLost* and *definitelyLost*

We argue that *pdThis* is not in *fromInEpa* or *fromLogged* before or after the operation, where $pdThis = (conAuthPurse'\ pdThis.from).pdAuth$.

First, before the operation the purse is in *eaFrom*, and after it is in *epr*, and hence *pdThis* can never be in *fromInEpa*.

From *BetweenWorld* constraint B–7 if *pdThis* were in *fromLogged′* then we would have

$$(conAuthPurse\ name?).pdAuth.fromSeqNo > pdThis.fromSeqNo$$

but we know these two *pdAuth*s are equal, so *pdThis* cannot be in *fromLogged′*. If the log isn't there after the operation, it certainly isn't there before, so *pdThis* is not in *toLogged* either.

Only the *from* purse changes in this operation, so the sets *toInEpv* and *toLogged* can't change. Hence

$$toInEpv' = toInEpv$$
$$toLogged' = toLogged$$
$$fromInEpa' = fromInEpa$$
$$fromLogged' = fromLogged$$

It follows that *maybeLost* is unchanged:

$$\begin{aligned} & maybeLost' \\ & \quad = toInEpv' \cap (fromInEpa' \cup fromLogged') \\ & \quad = toInEpv \cap (fromInEpa \cup fromLogged) \\ & \quad = maybeLost \end{aligned}$$

Also, *definitelyLost* is unchanged:

$$\begin{aligned} & definitelyLost' \\ & \quad = toLogged' \cap (fromInEpa' \cup fromLogged') \\ & \quad = toLogged \cap (fromInEpa \cup fromLogged) \\ & \quad = definitelyLost \end{aligned}$$

16.4 exists-pd

$$\begin{aligned} & \Phi BOp;\ StartFromPurseEafromOkay;\ RabOut;\ RabCl';\ RabIn \\ & \vdash \\ & \exists\, pdThis : PayDetails \bullet pdThis = (conAuthPurse'\ name?).pdAuth \end{aligned}$$

Proof

Use the [*one point*] rule with the expression for *pdThis* in the quantifier.
■ 16.4

16.5 exists-chosenLost

$$\begin{aligned} & \Phi BOp;\ StartFromPurseEafromOkay;\ RabOut; \\ & \quad RabClPd'[pdThis / pdThis'];\ RabIn \mid \\ & \quad pdThis = (conAuthPurse'\ name?).pdAuth \\ & \vdash \\ & \exists\, chosenLost : \mathbb{P}\, PayDetails \bullet \\ & \quad chosenLost = chosenLost' \\ & \quad \wedge chosenLost \subseteq maybeLost \end{aligned}$$

Proof:

We use the [*one point*] rule on *chosenLost* to give

$$\begin{aligned} & \Phi BOp;\ StartFromPurseEafromOkay;\ RabOut; \\ & \quad RabClPd'[pdThis / pdThis'];\ RabIn \mid \\ & \quad pdThis = (conAuthPurse'\ name?).pdAuth \\ & \vdash \\ & chosenLost' \subseteq maybeLost \end{aligned}$$

We then have

$$\begin{aligned} & chosenLost' \subseteq maybeLost' && [RabClPd'] \\ & \quad \subseteq maybeLost && [\text{unchanging } maybeLost] \end{aligned}$$

■ 16.5

16.6 check-operation

$$
\begin{array}{l}
\Phi BOp;\ StartFromPurseEafromOkay;\ RabClPd'[pdThis/pdThis'];\\
\quad\quad AbWorld;\ RabClPd \mid\\
\quad pdThis = (conAuthPurse'\ name?).pdAuth\\
\quad \land\ chosenLost = chosenLost'\\
\vdash\\
\forall\ n : \mathrm{dom}\ abAuthPurse \bullet\\
\quad (abAuthPurse'\ n).balance = (abAuthPurse\ n).balance\\
\quad \land\ (abAuthPurse'\ n).lost = (abAuthPurse\ n).lost
\end{array}
$$

Proof:
From *Rab*, we have that *lost* is a function of $definitelyLost \cup chosenLost$, which is unchanging, and that *balance* is a function of $maybeLost \setminus chosenLost$, which is also unchanging.

■ 16.6
■ 16

Chapter 17

Correctness of *StartTo*

17.1 Proof obligation

We have to prove the correct refinement of each abstract operation. In section 9.2.4 we give a general simplification of the correctness proof. We use lemma 'multiple refinement' (section 14.2) to split the proof obligation for each $\mathcal{A}$ operation into one for each individual $\mathcal{B}$ operation.

This chapter proves the $\mathcal{B}$ operation.

- We use lemma 'ignore' (see section 14.3) to simplify the proof obligation by proving the correctness of *Ignore* (in section 14.7), and *Abort* (in section 14.8), leaving the *Okay* branch to be proven here.
- Since the *Okay* branch of this operation is expressed as a promotion of *AbortPurseOkay* composed with a simpler *EafromPurseOkay* operation, we use lemma 'abort backward' (section C.5), and prove only that the promotion of the simpler operation is a refinement.
- We use lemma 'deterministic' (section C.1) to reduce the proof obligation to the three cases **exists-pd**, **exists-chosenLost**, and **check-operation**.
- Since this operation refines *AbIgnore*, we use lemma '*AbIgnore*' (from section C.3) to simplify **check-operation** to **check-operation-ignore**.

17.2 Instantiating lemma 'deterministic'

We take *pdThis* to be the *pdAuth* created by the start operation, and *chosenLost* to be unchanging.

$$\mathcal{P} \Leftrightarrow pdThis = (conAuthPurse'\ name?).pdAuth$$

$$\mathcal{Q} \Leftrightarrow chosenLost = chosenLost'$$

17.3 Behaviour of *maybeLost* and *definitelyLost*

We argue that *pdThis* is not in any of the before sets *fromInEpa*, *fromLogged*, *toInEpv*, or *toLogged*, where we have
$pdThis = (conAuthPurse'\ name?).pdAuth$.

$$\begin{aligned}
&(conAuthPurse\ name?).nextSeqNo && \text{[defn. } StartTo\text{]}\\
&\quad = (conAuthPurse'\ name?).pdAuth.toSeqNo\\
&\Rightarrow (conAuthPurse\ name?).nextSeqNo && \text{[defn. } pdThis\text{]}\\
&\quad = pdThis.toSeqNo\\
&\Rightarrow req\ pdThis \notin ether && \text{[}BetweenWorld \text{ constraint B–2]}\\
&\Rightarrow pdThis \notin fromInEpa \cup fromLogged && \text{[}BetweenWorld \text{ constraint B–12]}\\
&\quad \land pdThis \notin toInEpv \cup toLogged && \text{[}BetweenWorld \text{ constraint B–10]}
\end{aligned}$$

The operation moves one purse from *eaFrom* into *epv*; no logs are written. Hence *pdThis* is in *toInEpv′*, but not newly added to any of the other after sets. So

$$\begin{aligned}
&toInEpv' = toInEpv \cup \{pdThis\}\\
&toLogged' = toLogged\\
&fromInEpa' = fromInEpa\\
&fromLogged' = fromLogged
\end{aligned}$$

It follows that *maybeLost* is unchanged:

$$\begin{aligned}
&maybeLost'\\
&\quad = toInEpv' \cap (fromInEpa' \cup fromLogged')\\
&\quad = (toInEpv \cup \{pdThis\} \cap (fromInEpa \cup fromLogged)\\
&\quad = maybeLost \cup (\{pdThis\} \cap (fromInEpa \cup fromLogged))\\
&\quad = maybeLost
\end{aligned}$$

Also, *definitelyLost* is unchanged:

$$\begin{aligned}
&definitelyLost'\\
&\quad = toLogged' \cap (fromInEpa' \cup fromLogged')\\
&\quad = toLogged \cap (fromInEpa \cup fromLogged)\\
&\quad = definitelyLost
\end{aligned}$$

17.4 exists-pd

$$\begin{aligned}
&\Phi BOp;\ StartToPurseEafromOkay;\ RabOut;\ RabCl';\ RabIn\\
&\vdash\\
&\exists\ pdThis : PayDetails \bullet pdThis = (conAuthPurse'\ name?).pdAuth
\end{aligned}$$

Proof:
Use the [*one point*] rule with the expression for *pdThis* in the quantifier.
■ 17.4

17.5 exists-chosenLost

$$\begin{aligned}
&\Phi BOp;\ StartToPurseEafromOkay;\ RabOut;\ RabClPd'[pdThis/pdThis']\\
&\quad RabIn\ |\\
&\quad pdThis = (conAuthPurse'\ name?).pdAuth\\
&\vdash\\
&\exists\ chosenLost : \mathbb{P}\ PayDetails \bullet\\
&\quad chosenLost = chosenLost'\\
&\quad \land chosenLost \subseteq maybeLost
\end{aligned}$$

Proof:
We apply the [*one point*] rule for *chosenLost* in the consequent to give

$$\begin{aligned}
&\Phi BOp;\ StartToPurseEafromOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\\
&\quad RabIn\ |\\
&\quad pdThis = (conAuthPurse'\ name?).pdAuth\\
&\vdash\\
&chosenLost' \subseteq maybeLost
\end{aligned}$$

$$\begin{aligned}
&chosenLost' \subseteq maybeLost' && [RabClPd']\\
&\quad \subseteq maybeLost && \text{[unchanging } maybeLost\text{]}
\end{aligned}$$

■ 17.5

17.6 check-operation

$$
\begin{array}{l}
\Phi BOp;\ StartToPurseEafromOkay;\ RabClPd'[pdThis/pdThis'];\\
\qquad AbWorld;\ RabClPd \mid \\
\quad pdThis = (conAuthPurse'\ name?).pdAuth \\
\quad \land\ chosenLost = chosenLost' \\
\vdash \\
\forall\ n : \mathrm{dom}\ abAuthPurse \bullet \\
\quad (abAuthPurse'\ n).balance = (abAuthPurse\ n).balance \\
\quad \land\ (abAuthPurse'\ n).lost = (abAuthPurse\ n).lost
\end{array}
$$

Proof:
From *Rab*, we have that *lost* is a function of $definitelyLost \cup chosenLost$, which is unchanging, and that *balance* is a function of $maybeLost \setminus chosenLost$, which is also unchanging.

■ 17.6

■ 17

Chapter 18

Correctness of *Req*

18.1 Proof obligation

We have to prove the correct refinement of each abstract operation. In section 9.2.4 we give a general simplification of the correctness proof. We use lemma 'multiple refinement' (section 14.2) to split the proof obligation for each $\mathcal{A}$ operation into one for each individual $\mathcal{B}$ operation.

This chapter proves the $\mathcal{B}$ operation.

- We use lemma 'ignore' (see section 14.3) to simplify the proof obligation by proving the correctness of *Ignore* (in section 14.7), leaving the *Okay* branch to be proven here.
- We use lemma 'deterministic' (section C.1) to reduce the proof obligation to the three cases **exists-pd**, **exists-chosenLost**, and **check-operation**.

18.2 Instantiating lemma 'deterministic'

We must instantiate two general predicates relating to *pdThis* and *chosenLost*. The choices for these predicates are based on the fact that the important transaction is the one referred to by the *req* message being consumed by the *ReqOkay* operation, and that before the operation, the set of transactions chosen to be lost should be all those chosen to be lost after the operation, but specifically excluding the transaction *pdThis*. Thus

$$\mathcal{P} \Leftrightarrow req^{\sim}m? = pdThis$$

$$\mathcal{Q} \Leftrightarrow chosenLost = chosenLost' \setminus \{pdThis\}$$

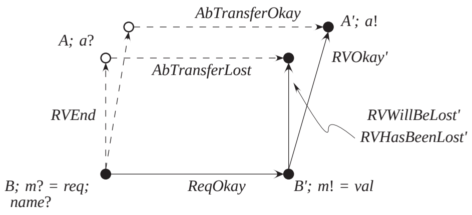

Figure 18.1: The correctness proof for *Req*.

18.3 Discussion

The correctness proof for *Req* is summarised in figure 18.1. There are three cases:

- The *to* purse for the transaction is in *epv*, and we choose that the transfer will succeed.
  Before the operation, $pdThis \notin maybeLost \cup definitelyLost$, and the appropriate retrieve is *RabEnd*.
  After the operation, $pdThis \in maybeLost' \setminus chosenLost'$, and the appropriate retrieve is *RabOkay'*; the abstract operation is *AbTransferOkay*.
- The *to* purse is in *epv*, and we choose the transfer will fail (the *to* purse will move out of *epv* before receiving the *val*).
  Before, $pdThis \notin maybeLost \cup definitelyLost$, and the appropriate retrieve is *RabEnd'*.
  After, $pdThis \in chosenLost'$, and the appropriate retrieve is *RabWillBeLost'*; the abstract operation is *AbTransferLost*.
- The *to* purse has already moved out of *epv*, so will not receive the *val*: the transfer has failed.
  Before, $pdThis \notin maybeLost \cup definitelyLost$, and the appropriate retrieve is *RabEnd*.
  After, $pdThis \in definitelyLost'$, and the appropriate retrieve is *RabHasBeenLost'*; the abstract operation is *AbTransferLost*.

The following proof establishes that these are indeed the only cases, and that *ReqOkay* correctly refines *AbTransfer* in each case.

18.4 exists-pd

$$
\begin{array}{l}
\Phi BOp;\ ReqPurseOkay;\ RabOut;\ RabCl';\ RabIn \\
\vdash \\
\exists\, pdThis : PayDetails \bullet req^{\sim} m? = pdThis
\end{array}
$$

Proof:
We discharge this by removing the existential for *pdThis* because we have an explicit equation for it, using the [*one point*] rule.
■ 18.4

18.5 exists-chosenlost

$$
\begin{array}{l}
\Phi BOp;\ ReqPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\ RabIn \mid \\
\quad req^{\sim} m? = pdThis \\
\vdash \\
\exists\, chosenLost : \mathbb{P}\, PayDetails \bullet \\
\quad chosenLost = chosenLost' \setminus \{pdThis\} \\
\quad \land chosenLost \subseteq maybeLost
\end{array}
$$

Proof:
That we can construct a *chosenLost* as the set difference is true because set difference is always defined. That the subset constraint holds follows as below:

$$
\begin{array}{ll}
chosenLost' \subseteq maybeLost' & [RabClPd'] \\
chosenLost' \setminus \{pdThis\} \subseteq maybeLost' \setminus \{pdThis\} & [\text{property of set minus}] \\
chosenLost \subseteq maybeLost' \setminus \{pdThis\} & [\text{eqn for } chosenLost] \\
chosenLost \subseteq maybeLost & [\text{lemma 'not lost before', section C.14 }]
\end{array}
$$

■ 18.5

18.6 check-operation

$$
\begin{array}{l}
\Phi BOp;\ ReqPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis']; \\
\qquad AbWorld;\ RabClPd;\ RabIn \mid \\
\quad req^{\sim} m? = pdThis \\
\quad \land chosenLost = chosenLost' \setminus \{pdThis\} \\
\vdash \\
AbTransfer
\end{array}
$$

Proof:
We invoke lemma 'not lost before' to add constraints on *maybeLost* and *definitelyLost* to the hypothesis. This allows us to further alter the hypothesis by replacing *RabClPd* with *RabEndClPd*.

$$\begin{array}{l} \Phi BOp;\ ReqPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\\ \quad\quad AbWorld;\ RabEndClPd;\ RabIn \mid \\ \quad req{\sim}m? = pdThis\\ \quad \land\ chosenLost = chosenLost' \setminus \{pdThis\}\\ \quad \land\ maybeLost = maybeLost' \setminus \{pdThis\}\\ \quad \land\ definitelyLost = definitelyLost' \setminus \{pdThis\}\\ \vdash\\ AbTransfer \end{array}$$

We use [*hyp disj*] to split *RabClPd'*[...] into four separate cases (section 10.1.4) to prove (using identity in section 10.1.5). In each case, we strengthen the consequent by choosing an appropriate disjunct of *AbTransfer*.

- **case 1:** We choose that the value is not lost, so the corresponding abstract operation is *AbTransferOkay*

$$\begin{array}{l} \Phi BOp;\ ReqPurseOkay;\ RabOut;\ RabOkayClPd'[pdThis/pdThis'];\\ \quad\quad AbWorld;\ RabEndClPd;\ RabIn \mid \\ \quad req{\sim}m? = pdThis\\ \quad \land\ chosenLost = chosenLost' \setminus \{pdThis\}\\ \quad \land\ maybeLost = maybeLost' \setminus \{pdThis\}\\ \quad \land\ definitelyLost = definitelyLost' \setminus \{pdThis\}\\ \vdash\\ AbTransferOkay \end{array}$$

- **case 2:** We choose that the value will be lost, so the corresponding abstract operation is *AbTransferLost*

$$\begin{array}{l} \Phi BOp;\ ReqPurseOkay;\ RabOut;\\ \quad RabWillBeLostClPd'[pdThis/pdThis'];\\ \quad\quad AbWorld;\ RabEndClPd;\ RabIn \mid \\ \quad req{\sim}m? = pdThis\\ \quad \land\ chosenLost = chosenLost' \setminus \{pdThis\}\\ \quad \land\ maybeLost = maybeLost' \setminus \{pdThis\}\\ \quad \land\ definitelyLost = definitelyLost' \setminus \{pdThis\}\\ \vdash\\ AbTransferLost \end{array}$$

- **case 3:** We say that the value has already been lost, so the corresponding abstract operation is *AbTransferLost*

$$\begin{array}{l} \Phi BOp;\ ReqPurseOkay;\ RabOut;\\ \quad RabHasBeenLostClPd'[pdThis/pdThis'];\\ \quad\quad AbWorld;\ RabEndClPd;\ RabIn \mid \\ \quad req{\sim}m? = pdThis\\ \quad \land\ chosenLost = chosenLost' \setminus \{pdThis\}\\ \quad \land\ maybeLost = maybeLost' \setminus \{pdThis\}\\ \quad \land\ definitelyLost = definitelyLost' \setminus \{pdThis\}\\ \vdash\\ AbTransferLost \end{array}$$

- **case 4:** The fourth case is impossible. We choose *RabEndClPd'*, and prove that the hypothesis is contradictory, so the choice of corresponding abstract operation is unimportant.

$$\begin{array}{l} \Phi BOp;\ ReqPurseOkay;\ RabOut;\ RabEndClPd'[pdThis/pdThis'];\\ \quad\quad AbWorld;\ RabEndClPd;\ RabIn \mid \\ \quad req{\sim}m? = pdThis\\ \quad \land\ chosenLost = chosenLost' \setminus \{pdThis\}\\ \quad \land\ maybeLost = maybeLost' \setminus \{pdThis\}\\ \quad \land\ definitelyLost = definitelyLost' \setminus \{pdThis\}\\ \vdash\\ AbTransfer \end{array}$$

We now have four independent cases to prove. The next four sections each prove one case.

18.7 case 1: *ReqOkay* and *RabOkayClPd'*

$$\begin{array}{l} \Phi BOp;\ ReqPurseOkay;\ RabOut;\ RabOkayClPd'[pdThis/pdThis'];\\ \quad\quad AbWorld;\ RabEndClPd;\ RabIn \mid \\ \quad req{\sim}m? = pdThis\\ \quad \land\ chosenLost = chosenLost' \setminus \{pdThis\}\\ \quad \land\ maybeLost = maybeLost' \setminus \{pdThis\}\\ \quad \land\ definitelyLost = definitelyLost' \setminus \{pdThis\}\\ \vdash\\ AbTransferOkay \end{array}$$

18.7.1 The behaviour of *maybeLost* and *definitelyLost*

We argue that the transaction *pdThis* is initially not in *maybeLost* or *definitely-Lost*, and is moved into $maybeLost' \setminus chosenLost'$ by this case of the *ReqOkay* operation. The transaction initially was not far enough progressed to have the potential of being lost; afterwards it has progressed far enough that it may be lost, but we are actually on the branch that will succeed.

We have from *RabOkayClPd′* that

$$pdThis \in maybeLost' \setminus chosenLost'$$

Therefore $pdThis \notin chosenLost'$ (by the definition of set minus) and $pdThis \notin definitelyLost'$ (by lemma 'lost'). So we have

$$definitelyLost = definitelyLost'$$

$$maybeLost = maybeLost' \setminus \{pdThis\}$$

$$chosenLost = chosenLost'$$

18.7.2 AbTransferOkay

In this section we prove that an *AbWorld* that has the correct retrieve properties also satisfies *AbTransferOkay*. Recall that our proof obligation is

$$\begin{array}{l}
\Phi BOp;\ ReqPurseOkay;\ RabOut;\ RabOkayClPd'[pdThis/pdThis'];\\
\quad AbWorld;\ RabEndClPd;\ RabIn \mid\\
\quad req^{\sim} m? = pdThis\\
\quad \land\ chosenLost = chosenLost' \setminus \{pdThis\}\\
\quad \land\ maybeLost = maybeLost' \setminus \{pdThis\}\\
\quad \land\ definitelyLost = definitelyLost' \setminus \{pdThis\}\\
\vdash\\
AbTransferOkay
\end{array}$$

Each element of *AbWorld* is defined by an explicit equation in *RabEndClPd*, and we show that this value satisfies *AbTransferOkay* by showing each predicate holds.

A–1 *AbOp*: This trivial: *AbOp* imposes no constraints.

A–2 *AbWorldSecureOp*

- $a? \in \operatorname{ran} transfer$
  true by construction of $a?$ from $m?$ in *RabIn*.

- no purses other than *from*? and *to*? change
  For *balance* and *lost* we show that *RabEndClPd* and

  $$RabOkayClPd'[pdThis/pdThis']$$

  are essentially the same. This is immediate because in both cases the relevant predicates are captured in the same schema *OtherPursesRab*.

A–3 $Authentic[from?/name?], Authentic[to?/name?]$
We have $pdThis \in maybeLost'$, hence it is in both $authenticFrom'$ and in $authenticTo'$. Hence, by ΦBOp and *AbstractBetween*, it is also in both *authenticFrom* and in *authenticTo*.

A–4 *SufficientFundsProperty*
true from *ConPurse* constraint P–2b

A–5 $to? \neq from?$
true because *pdThis* is a *PayDetails*.

A–6 $abAuthPurse'\ from? = \ldots, abAuthPurse'\ to? = \ldots$
Each of the four elements (*from* and *to* purses, each with *balance* and *lost*) are handled below, followed by all the other elements in one section.

The *from* purse's *balance* component

$$\begin{array}{ll}
(abAuthPurse\ pdThis.from).balance & \\
\quad = (conAuthPurse\ pdThis.from).balance & \\
\qquad + sumValue(((maybeLost \setminus chosenLost) & \\
\qquad\qquad \cap \{ pd : PayDetails \mid pd.to = pdThis.from \}) & \\
\qquad\quad \setminus \{pdThis\}) & [RabEndClPd] \\
 & \\
\quad = (conAuthPurse\ pdThis.from).balance & \\
\qquad + sumValue((((maybeLost' \setminus \{pdThis\}) \setminus chosenLost') & \\
\qquad\qquad \cap \{ pd : PayDetails \mid pd.to = pdThis.from \}) & \\
\qquad\quad \setminus \{pdThis\}) & [\text{section 18.7.1}] \\
 & \\
\quad = (conAuthPurse\ pdThis.from).balance & \\
\qquad + sumValue(((maybeLost' \setminus chosenLost') & \\
\qquad\qquad \cap \{ pd : PayDetails \mid pd.to = pdThis.from \}) & \\
\qquad\quad \setminus \{pdThis\}) & [\text{rearranging}]
\end{array}$$

$$
\begin{aligned}
&= pdThis.value + (conAuthPurse'\ pdThis.from).balance \\
&\quad + sumValue(((maybeLost' \setminus chosenLost') \\
&\qquad \cap \{ pd : PayDetails \mid pd.to = pdThis.from \}) \\
&\quad \setminus \{pdThis\}) && [ReqPurseOkay] \\
&= pdThis.value + (abAuthPurse'\ pdThis.from).balance \\
& && [RabOkayClPd'[\ldots]]
\end{aligned}
$$

So

$$(abAuthPurse'\ from?).balance = (abAuthPurse\ from?).balance - value?$$

The *from* purse's *lost* component

$$
\begin{aligned}
&(abAuthPurse\ pdThis.from).lost \\
&= sumValue(((definitelyLost \cup chosenLost) \\
&\qquad \cap \{ pd : PayDetails \mid pd.from = pdThis.from \}) \\
&\quad \setminus \{pdThis\}) && [RabEndClPd] \\
&= sumValue(((definitelyLost' \cup chosenLost') \\
&\qquad \cap \{ pd : PayDetails \mid pd.from = pdThis.from \}) \\
&\quad \setminus \{pdThis\}) && [\text{section 18.7.1}] \\
&= (abAuthPurse'\ pdThis.from).lost && [RabOkayClPd'[\ldots]]
\end{aligned}
$$

The *to* purse's *balance* component

$$
\begin{aligned}
&(abAuthPurse\ pdThis.to).balance \\
&= (conAuthPurse\ pdThis.to).balance \\
&\quad + sumValue(((maybeLost \setminus chosenLost) \\
&\qquad \cap \{ pd : PayDetails \mid pd.to = pdThis.to \}) \\
&\quad \setminus \{pdThis\}) && [RabEndClPd] \\
&= (conAuthPurse\ pdThis.to).balance \\
&\quad + sumValue((((maybeLost' \setminus \{pdThis\}) \setminus chosenLost') \\
&\qquad \cap \{ pd : PayDetails \mid pd.to = pdThis.to \}) \\
&\quad \setminus \{pdThis\}) && [\text{section 18.7.1}]
\end{aligned}
$$

$$
\begin{aligned}
&= (conAuthPurse\ pdThis.to).balance \\
&\quad + sumValue(((maybeLost' \setminus chosenLost') \\
&\qquad \cap \{ pd : PayDetails \mid pd.to = pdThis.to \}) \\
&\quad \setminus \{pdThis\}) && [\text{rearranging}] \\
&= (conAuthPurse'\ pdThis.to).balance \\
&\quad + sumValue(((maybeLost' \setminus chosenLost') \\
&\qquad \cap \{ pd : PayDetails \mid pd.to = pdThis.to \}) \\
&\quad \setminus \{pdThis\}) && [\Phi BOp] \\
&= (abAuthPurse'\ pdThis.to).balance + pdThis.value \\
& && [RabOkayClPd'[\ldots]]
\end{aligned}
$$

From the form of $(abAuthPurse'\ pdThis.to).balance = pdThis.value + n$ in *AbTransferOkay*, we see that this last subtraction gives a positive result. So

$$(abAuthPurse'\ to?).balance = (abAuthPurse\ to?).balance + value?$$

The *to* purse's *lost* component

$$
\begin{aligned}
&(abAuthPurse\ pdThis.to).lost \\
&= sumValue(((definitelyLost \cup chosenLost) \\
&\qquad \cap \{ pd : PayDetails \mid pd.from = pdThis.to \}) \\
&\quad \setminus \{pdThis\}) && [RabEndClPd] \\
&= sumValue(((definitelyLost' \cup chosenLost') \\
&\qquad \cap \{ pd : PayDetails \mid pd.from = pdThis.to \}) \\
&\quad \setminus \{pdThis\}) && [\text{section 18.7.1}] \\
&= (abAuthPurse'\ pdThis.to).lost && [RabOkayClPd'[\ldots]]
\end{aligned}
$$

The remaining *from* and *to* purse components

These are unchanging, by $\Xi ConPurseReq$, and that the retrieves each define a unique abstract world.

■ 18.7.2

■ 18.7

18.8 case 2: *ReqOkay* and *RabWillBeLostPd′*

$$
\begin{array}{l}
\Phi BOp;\ ReqPurseOkay;\ RabOut;\ RabWillBeLostClPd'[pdThis/pdThis'];\\
\quad AbWorld;\ RabEndClPd;\ RabIn \mid\\
\quad req^{\sim} m? = pdThis\\
\quad \land\ chosenLost = chosenLost' \setminus \{pdThis\}\\
\quad \land\ maybeLost = maybeLost' \setminus \{pdThis\}\\
\quad \land\ definitelyLost = definitelyLost' \setminus \{pdThis\}\\
\vdash\\
AbTransferLost
\end{array}
$$

18.8.1 The behaviour of *maybeLost* and *definitelyLost*

We argue that the transaction *pd* is initially not in *maybeLost* or *definitelyLost*, and is moved into *chosenLost′* by this case of the *ReqOkay* operation. The transaction initially was not far enough progressed to have the potential of being lost; afterwards it has progressed far enough that it may be lost, and we choose that it will be lost.

We have from *RabWillBeLostClPd′*[...] that

$$pdThis \in chosenLost'$$

Therefore

$$pdThis \in maybeLost'$$

because $chosenLost' \subseteq maybeLost'$. But we can say that $pdThis \notin definitelyLost'$ (by lemma 'lost'). So we have

$$definitelyLost = definitelyLost'$$

$$maybeLost = maybeLost' \setminus \{pdThis\}$$

$$chosenLost = chosenLost' \setminus \{pdThis\}$$

18.8.2 AbTransferLost

In this section we prove that an *AbWorld* that has the correct retrieve properties also satisfies *AbTransferLost*. Recall, our proof obligation is

$$
\begin{array}{l}
\Phi BOp;\ ReqPurseOkay;\ RabOut;\ RabWillBeLostClPd'[pdThis/pdThis'];\\
\quad AbWorld;\ RabEndClPd;\ RabIn \mid\\
\quad req^{\sim} m? = pdThis\\
\quad \land\ chosenLost = chosenLost' \setminus \{pdThis\}\\
\quad \land\ maybeLost = maybeLost' \setminus \{pdThis\}\\
\quad \land\ definitelyLost = definitelyLost' \setminus \{pdThis\}\\
\vdash\\
AbTransferLost
\end{array}
$$

Each element of *AbWorld* is defined by an explicit equation in *RabEndClPd*, and we show that this value satisfies *AbTransferLost* by showing each predicate holds.

A–1 *AbOp*: This trivial: *AbOp* imposes no constraints.

A–2 *AbWorldSecureOp*
- $a? \in \operatorname{ran} transfer$
  true by construction of $a?$
- no purses other than $from?$ and $to?$ change
  For *balance* and *lost* we show that *RabEndClPd* and *RabWillBeLost-ClPd′*[*pdThis*/*pdThis′*] are essentially the same. This is immediate because in both cases the relevant predicates are captured in the same schema *OtherPursesRab*.

A–3 $Authentic[from?/name?]$, $Authentic[to?/name?]$
We have $pdThis \in maybeLost'$, hence it is in both *authenticFrom′* and in *authenticTo′*. Hence, by ΦBOp and *AbstractBetween*, it is also in both *authenticFrom* and in *authenticTo*.

A–4 *SufficientFundsProperty*
true from *ConPurse* constraint P–2b

A–5 $to? \neq from?$
true because *pdThis* is a *PayDetails*.

A–6 $abAuthPurse'\ from? = \ldots, abAuthPurse'\ to? = \ldots$
Each of the four elements (*from* and *to* purses, each with *balance* and *lost*) are handled below, followed by all the other elements in one section.

The *from* purse's *balance* component

$$
\begin{aligned}
&(abAuthPurse\ pdThis.from).balance\\
&\quad = (conAuthPurse\ pdThis.from).balance\\
&\qquad + sumValue(((maybeLost \setminus chosenLost)\\
&\qquad\qquad \cap \{\, pd : PayDetails \mid pd.to = pdThis.from \,\})\\
&\qquad\quad \setminus \{pdThis\}) && [RabEndClPd]\\[1ex]
&\quad = (conAuthPurse\ pdThis.from).balance\\
&\qquad + sumValue((((maybeLost' \setminus \{pdThis\}) \setminus chosenLost' \setminus \{pdThis\})\\
&\qquad\qquad \cap \{\, pd : PayDetails \mid pd.to = pdThis.from \,\})\\
&\qquad\quad \setminus \{pdThis\}) && [\text{section 18.8.1}]\\[1ex]
&\quad = (conAuthPurse\ pdThis.from).balance\\
&\qquad + sumValue(((maybeLost' \setminus chosenLost')\\
&\qquad\qquad \cap \{\, pd : PayDetails \mid pd.to = pdThis.from \,\})\\
&\qquad\quad \setminus \{pdThis\}) && [\text{rearranging}]\\[1ex]
&\quad = pdThis.value + (conAuthPurse'\ pdThis.from).balance\\
&\qquad + sumValue(((maybeLost' \setminus chosenLost')\\
&\qquad\qquad \cap \{\, pd : PayDetails \mid pd.to = pdThis.from \,\})\\
&\qquad\quad \setminus \{pdThis\}) && [ReqPurseOkay]\\
&\quad = pdThis.value + (abAuthPurse'\ pdThis.from).balance\\
&&& [RabWillBeLostClPd'[\ldots]]
\end{aligned}
$$

So

$$(abAuthPurse'\ from?).balance = (abAuthPurse\ from?).balance - value?$$

The *from* purse's *lost* component

$$
\begin{aligned}
&(abAuthPurse\ pdThis.from).lost\\
&\quad = sumValue(((definitelyLost \cup chosenLost)\\
&\qquad\qquad \cap \{\, pd : PayDetails \mid pd.from = pdThis.from \,\})\\
&\qquad \setminus \{pdThis\}) && [RabEndClPd]\\[1ex]
&\quad = sumValue(((definitelyLost' \cup chosenLost' \setminus \{pdThis\})\\
&\qquad\qquad \cap \{\, pd : PayDetails \mid pd.from = pdThis.from \,\})\\
&\qquad \setminus \{pdThis\}) && [\text{section 18.8.1}]
\end{aligned}
$$

$$
\begin{aligned}
&\quad = sumValue(((definitelyLost' \cup chosenLost')\\
&\qquad\qquad \cap \{\, pd : PayDetails \mid pd.from = pdThis.from \,\})\\
&\qquad \setminus \{pdThis\}) && [\text{rearrange}]\\
&\quad = (abAuthPurse'\ pdThis.from).lost - pdThis.value\\
&&& [RabWillBeLostClPd'[\ldots]]
\end{aligned}
$$

The *to* purse's *balance* component

$$
\begin{aligned}
&(abAuthPurse\ pdThis.to).balance\\
&\quad = (conAuthPurse\ pdThis.to).balance\\
&\qquad + sumValue(((maybeLost \setminus chosenLost)\\
&\qquad\qquad \cap \{\, pd : PayDetails \mid pd.to = pdThis.to \,\})\\
&\qquad\quad \setminus \{pdThis\}) && [RabEndClPd]\\[1ex]
&\quad = (conAuthPurse\ pdThis.to).balance\\
&\qquad + sumValue((((maybeLost' \setminus \{pdThis\}) \setminus chosenLost' \setminus \{pdThis\})\\
&\qquad\qquad \cap \{\, pd : PayDetails \mid pd.to = pdThis.to \,\})\\
&\qquad\quad \setminus \{pdThis\}) && [\text{section 18.8.1}]\\[1ex]
&\quad = (conAuthPurse\ pdThis.to).balance\\
&\qquad + sumValue(((maybeLost' \setminus chosenLost')\\
&\qquad\qquad \cap \{\, pd : PayDetails \mid pd.to = pdThis.to \,\})\\
&\qquad\quad \setminus \{pdThis\}) && [\text{rearranging}]\\[1ex]
&\quad = (conAuthPurse'\ pdThis.to).balance\\
&\qquad + sumValue(((maybeLost' \setminus chosenLost')\\
&\qquad\qquad \cap \{\, pd : PayDetails \mid pd.to = pdThis.to \,\})\\
&\qquad\quad \setminus \{pdThis\}) && [\Phi BOp]\\
&\quad = (abAuthPurse'\ pdThis.to).balance && [RabWillBeLostClPd'[\ldots]]
\end{aligned}
$$

The *to* purse's *lost* component

$$
\begin{aligned}
&(abAuthPurse\ pdThis.to).lost\\
&\quad = sumValue(((definitelyLost \cup chosenLost)\\
&\qquad\qquad \cap \{\, pd : PayDetails \mid pd.from = pdThis.to \,\})\\
&\qquad \setminus \{pdThis\}) && [RabEndClPd]
\end{aligned}
$$

$$= sumValue(((definitelyLost' \cup chosenLost' \setminus \{pdThis\}) \cap \{ pd : PayDetails \mid pd.from = pdThis.to \}) \setminus \{pdThis\}) \quad \text{[section 18.8.1]}$$

$$= sumValue(((definitelyLost' \cup chosenLost') \cap \{ pd : PayDetails \mid pd.from = pdThis.to \}) \setminus \{pdThis\}) \quad \text{[rearrange]}$$

$$= (abAuthPurse'\ pdThis.to).lost \quad [RabWillBeLostClPd'[\ldots]]$$

The remaining *from* and *to* purse components

These are unchanging, by $\Xi ConPurseReq$, and that the retrieves each define a unique abstract world.

■ 18.8.2

■ 18.8

18.9 case 3: *ReqOkay* and *RabHasBeenLostPd'*

$$\begin{array}{l} \Phi BOp;\ ReqPurseOkay;\ RabOut;\ RabHasBeenLostClPd'[pdThis/pdThis'];\\ \quad AbWorld;\ RabEndClPd;\ RabIn \mid \\ \quad req\sim m? = pdThis \\ \quad \land chosenLost = chosenLost' \setminus \{pdThis\} \\ \quad \land maybeLost = maybeLost' \setminus \{pdThis\} \\ \quad \land definitelyLost = definitelyLost' \setminus \{pdThis\} \\ \vdash \\ AbTransferLost \end{array}$$

18.9.1 The behaviour of *maybeLost* and *definitelyLost*

We argue that the transaction *pd* is initially not in *maybeLost* or *definitelyLost*, and is moved into *definitelyLost'* by this case of the *ReqOkay* operation. The transaction initially was not far enough progressed to have the potential of being lost; afterwards it has progressed far enough that it has in fact been lost.

We have from *RabHasBeenLostClPd'* that

$$pdThis \in definitelyLost'$$

Therefore $pdThis \notin maybeLost'$ (by lemma 'lost'), and also $pdThis \notin chosenLost'$

(because this is a subset of *maybeLost'*). So we have

$$definitelyLost = definitelyLost' \setminus \{pdThis\}$$

$$maybeLost = maybeLost'$$

$$chosenLost = chosenLost'$$

18.9.2 AbTransferLost

In this section we prove that an *AbWorld* that has the correct retrieve properties also satisfies *AbTransferLost*. Recall, our proof obligation is

$$\begin{array}{l} \Phi BOp;\ ReqPurseOkay;\ RabOut;\ RabHasBeenLostClPd'[pdThis/pdThis'];\\ \quad AbWorld;\ RabEndClPd;\ RabIn \mid \\ \quad req\sim m? = pdThis \\ \quad \land chosenLost = chosenLost' \setminus \{pdThis\} \\ \quad \land maybeLost = maybeLost' \setminus \{pdThis\} \\ \quad \land definitelyLost = definitelyLost' \setminus \{pdThis\} \\ \vdash \\ AbTransferLost \end{array}$$

Each element of *AbWorld* is defined by an explicit equation in *RabEndClPd*, and we show that this value satisfies *AbTransferLost* by showing each predicate holds.

A–1 *AbOp*: This trivial: *AbOp* imposes no constraints.

A–2 *AbWorldSecureOp*

- $a? \in \operatorname{ran} transfer$
  true by construction of $a?$
- no purses other than *from*? and *to*? change
  For *balance* and *lost* we show that *RabEndClPd* and *RabHasBeenLostClPd'*[*pdThis*/*pdThis*] are essentially the same. This is immediate because in both cases the relevant predicates are captured in the same schema *OtherPursesRab*.

A–3 *Authentic*[*from*?/*name*?], *Authentic*[*to*?/*name*?]
We have $pdThis \in maybeLost'$, hence it is in both *authenticFrom'* and in *authenticTo'*. Hence, by ΦBOp and *AbstractBetween*, it is also in both *authenticFrom* and in *authenticTo*.

A–4 *SufficientFundsProperty*
true from *ConPurse* constraint P–2b

A–5 $to? \neq from?$
true because $pdThis$ is a $PayDetails$.

A–6 $abAuthPurse'\ from? = \ldots, abAuthPurse'\ to? = \ldots$
Each of the four elements (*from* and *to* purses, each with *balance* and *lost*) are handled below, followed by all the other elements in one section.

The *from* purse's *balance* component

$$
\begin{aligned}
&(abAuthPurse\ pdThis.from).balance \\
&\quad = (conAuthPurse\ pdThis.from).balance \\
&\qquad + sumValue(((maybeLost \setminus chosenLost) \\
&\qquad\qquad \cap \{ pd : PayDetails \mid pd.to = pdThis.from \}) \\
&\qquad\quad \setminus \{pdThis\}) && [RabEndClPd] \\
&\quad = (conAuthPurse\ pdThis.from).balance \\
&\qquad + sumValue(((maybeLost' \setminus chosenLost') \\
&\qquad\qquad \cap \{ pd : PayDetails \mid pd.to = pdThis.from \}) \\
&\qquad\quad \setminus \{pdThis\}) && [\text{section } 18.9.1] \\
&\quad = pdThis.value + (conAuthPurse'\ pdThis.from).balance \\
&\qquad + sumValue(((maybeLost' \setminus chosenLost') \\
&\qquad\qquad \cap \{ pd : PayDetails \mid pd.to = pdThis.from \}) \\
&\qquad\quad \setminus \{pdThis\}) && [ReqPurseOkay] \\
&\quad = pdThis.value + (abAuthPurse'\ pdThis.from).balance \\
&&& [RabHasBeenLostClPd'[\ldots]]
\end{aligned}
$$

So

$$(abAuthPurse'\ from?).balance = (abAuthPurse\ from?).balance - value?$$

The *from* purse's *lost* component

$$
\begin{aligned}
&(abAuthPurse\ pdThis.from).lost \\
&\quad = sumValue(((definitelyLost \cup chosenLost) \\
&\qquad\qquad \cap \{ pd : PayDetails \mid pd.from = pdThis.from \}) \\
&\qquad \setminus \{pdThis\}) && [RabEndClPd] \\
&\quad = sumValue(((definitelyLost' \setminus \{pdThis\} \cup chosenLost') \\
&\qquad\qquad \cap \{ pd : PayDetails \mid pd.from = pdThis.from \}) \\
&\qquad \setminus \{pdThis\}) && [\text{section } 18.9.1] \\
&\quad = sumValue(((definitelyLost' \cup chosenLost') \\
&\qquad\qquad \cap \{ pd : PayDetails \mid pd.from = pdThis.from \}) \\
&\qquad \setminus \{pdThis\}) && [\text{rearrange}] \\
&\quad = (abAuthPurse'\ pdThis.from).lost - pdThis.value \\
&&& [RabHasBeenLostClPd'[\ldots]]
\end{aligned}
$$

The *to* purse's *balance* component

$$
\begin{aligned}
&(abAuthPurse\ pdThis.to).balance \\
&\quad = (conAuthPurse\ pdThis.to).balance \\
&\qquad + sumValue(((maybeLost \setminus chosenLost) \\
&\qquad\qquad \cap \{ pd : PayDetails \mid pd.to = pdThis.to \}) \\
&\qquad\quad \setminus \{pdThis\}) && [RabEndClPd] \\
&\quad = (conAuthPurse\ pdThis.to).balance \\
&\qquad + sumValue(((maybeLost' \setminus chosenLost') \\
&\qquad\qquad \cap \{ pd : PayDetails \mid pd.to = pdThis.to \}) \\
&\qquad\quad \setminus \{pdThis\}) && [\text{section } 18.9.1] \\
&\quad = (conAuthPurse'\ pdThis.to).balance \\
&\qquad + sumValue(((maybeLost' \setminus chosenLost') \\
&\qquad\qquad \cap \{ pd : PayDetails \mid pd.to = pdThis.to \}) \\
&\qquad\quad \setminus \{pdThis\}) && [\Phi BOp] \\
&\quad = (abAuthPurse'\ pdThis.to).balance && [RabHasBeenLostClPd'[\ldots]]
\end{aligned}
$$

The *to* purse's *lost* component

$$
\begin{aligned}
&(abAuthPurse\ pdThis.to).lost \\
&\quad = sumValue(((definitelyLost \cup chosenLost) \\
&\qquad\qquad \cap \{ pd : PayDetails \mid pd.from = pdThis.to \}) \\
&\qquad \setminus \{pdThis\}) && [RabEndClPd] \\
&\quad = sumValue(((definitelyLost' \setminus \{pdThis\} \cup chosenLost') \\
&\qquad\qquad \cap \{ pd : PayDetails \mid pd.from = pdThis.to \}) \\
&\qquad \setminus \{pdThis\}) && [\text{section } 18.9.1]
\end{aligned}
$$

$$= sumValue(((definitelyLost' \cup chosenLost') \cap \{ pd : PayDetails \mid pd.from = pdThis.to \}) \setminus \{pdThis\}) \quad \text{[rearrange]}$$

$$= (abAuthPurse'\ pdThis.to).lost \quad [RabHasBeenLostClPd'[\ldots]]$$

The remaining *from* and *to* purse components

These are unchanging, by $\Xi ConPurseReq$, and that the retrieves each define a unique abstract world.

- ■ 18.9.2
- ■ 18.9

18.10 case 4: *ReqOkay* and *RabEndPd'*

$$\Phi BOp;\ ReqPurseOkay;\ RabOut;\ RabEndClPd'[pdThis/pdThis'];\ AbWorld;\ RabEndClPd;\ RabIn \mid$$
$$req^{\sim} m? = pdThis$$
$$\land\ chosenLost = chosenLost' \setminus \{pdThis\}$$
$$\land\ maybeLost = maybeLost' \setminus \{pdThis\}$$
$$\land\ definitelyLost = definitelyLost' \setminus \{pdThis\}$$
$$\vdash$$
$$AbTransfer$$

We show that $RabEndClPd'[\ldots]$ is *false* under *ReqOkay*, and then proceed by [*contradiction*], because this shows the antecedent of the theorem is false, and hence the theorem is true.

$$\Phi BOp;\ ReqPurseOkay;\ RabOut;\ AbWorld';\ pdThis : PayDetails;\ chosenLost' : \mathbb{P}\, PayDetails \mid$$
$$req^{\sim} m? = pdThis$$
$$\vdash$$
$$\neg\ RabEndClPd'[pdThis/pdThis']$$

It suffices to show that $pdThis \in definitelyLost' \cup maybeLost'$. We have

$$definitelyLost' \cup maybeLost'$$
$$= (fromInEpa' \cup fromLogged') \cap (toInEpv' \cup toLogged')$$

ReqPurseOkay gives us that the after state of the purse is *epa*; *pdThis* is in

authenticFrom, from ΦBOp; hence *pdThis* is in $fromInEpa'$. So it is sufficient to show either *pdThis* is in $toInEpv'$ or in $toLogged'$.

We know from the existence of the *req*, with *BetweenWorld* constraint B–1, that $pdThis \in authenticTo$. There is no *ack* in the $ether'$:

$$pdThis \in fromInEpr \quad \text{[precondition } ReqPurseOkay\text{]}$$
$$\Rightarrow ack\ pdThis \notin ether \quad [BetweenWorld \text{ constraint B–9}]$$
$$\Rightarrow ack\ pdThis \notin ether' \quad \text{[defn. } ReqPurseOkay \text{ and } \Phi BOp\text{]}$$

Hence

$$req\ pdThis \in ether' \quad \text{[precondition } ReqPurseOkay\text{]}$$
$$\land\ ack\ pdThis \notin ether' \quad \text{[above]}$$
$$\Rightarrow pdThis \in toInEpv' \cup toLogged' \quad [BetweenWorld \text{ constraint B–10}]$$

as required.

- ■ 18.10
- ■ 18.6
- ■ 18

Chapter 19

Correctness of *Val*

19.1 Proof obligation

We have to prove the correct refinement of each abstract operation. In section 9.2.4 we give a general simplification of the correctness proof. We use lemma 'multiple refinement' (section 14.2) to split the proof obligation for each $\mathcal{A}$ operation into one for each individual $\mathcal{B}$ operation.

This chapter proves the $\mathcal{B}$ operation.

- We use lemma 'ignore' (see section 14.3) to simplify the proof obligation by proving the correctness of *Ignore* (in section 14.7), leaving the *Okay* branch to be proven here.
- We use lemma 'deterministic' (section C.1) to reduce the proof obligation to the three cases **exists-pd**, **exists-chosenLost**, and **check-operation**.
- Since this operation refines *AbIgnore*, we use lemma '*AbIgnore*' (from section C.3) to simplify **check-operation** to **check-operation-ignore**.

19.2 Instantiating lemma 'deterministic'

The choices for the predicates relating to *pdThis* and *chosenLost* are based on the fact that the important transaction is the one stored in the purse performing the *ValOkay* operation, and that before the operation, the set of transactions chosen to be lost should be all those chosen to be lost after the operation. Thus

$$\mathcal{P} \Leftrightarrow pdThis = (conAuthPurse\ name?).pdAuth$$

$$\mathcal{Q} \Leftrightarrow chosenLost = chosenLost'$$

19.3 exists-pd

ΦBOp; $ValPurseOkay$; $RabOut$; $RabCl'$; $RabIn$
$\vdash$
$\exists\, pdThis : PayDetails \bullet pdThis = (conAuthPurse\ name?).pdAuth$

Proof:
This is immediate by the [*one point*] rule, as we have an explicit definition of *pdThis*.

■ 19.3

19.4 exists-chosenlost

ΦBOp; $ValPurseOkay$; $RabOut$; $RabClPd'[pdThis/pdThis']$; $RabIn$ |
$\quad pdThis = (conAuthPurse\ name?).pdAuth$
$\vdash$
$\exists\, chosenLost : \mathbb{P}\, PayDetails \bullet$
$\quad chosenLost = chosenLost'$
$\quad \wedge\ chosenLost \subseteq maybeLost$

Proof:
We can [*one point*] away the quantification because we have an explicit definition of *chosenLost* (as *chosenLost′*). We show that the constraint holds by

$$
\begin{array}{ll}
chosenLost = chosenLost' & \text{[defn.]} \\
\quad \subseteq maybeLost' & [RabClPd'[\ldots]] \\
\quad \subseteq maybeLost \setminus \{pdThis\} & \text{[see 19.6.7]} \\
\quad \subseteq maybeLost & \text{[defn. } \setminus\text{]}
\end{array}
$$

■ 19.4

19.5 check-operation

ΦBOp; $ValPurseOkay$; $RabClPd'[pdThis/pdThis']$; $AbWorld$; $RabClPd$ |
$\quad pdThis = (conAuthPurse\ name?).pdAuth$
$\quad \wedge\ chosenLost = chosenLost'$
$\vdash$
$\forall\, n : \mathrm{dom}\ abAuthPurse \bullet$
$\quad (abAuthPurse'\ n).balance = (abAuthPurse\ n).balance$
$\quad \wedge\ (abAuthPurse'\ n).lost = (abAuthPurse\ n).lost$

We prove this first by investigating the way in which the key sets *definitelyLost* and *maybeLost* are modified by the operation. Having got equations for these changes, we then look at the equations for the components *balance* and *lost* for two types of purses: the *to* purse in the transaction *pdThis*, and all other purses.

19.6 Behaviour of *maybeLost* and *definitelyLost*

We argue that the transaction *pdThis* is initially in *maybeLost*, and is moved out of it, but not into *definitelyLost′*, by the *ValOkay* operation. This operation determines that the transaction is successful.

19.6.1 fromLogged

No logs change, so

$$fromLogged' = fromLogged$$

19.6.2 toLogged

No logs change, so

$$toLogged' = toLogged$$

After the operation the purse is in *eaTo*, and *pdThis* is in *authenticTo*, from ΦBOp, hence $pdThis \in toInEapayee'$. Lemma ‘notLoggedAndIn’ (section C.12) gives us:

$$pdThis \notin toLogged'$$

19.6.3 toInEpv

From the precondition of *ValPurseOkay* we know the purse is in *epv*, and we know that the name of this purse is equal to *pdThis.to*. After the operation, this purse is in *eaTo* (that is, not in *epv*). No other purses change.

$$toInEpv' = toInEpv \setminus \{pdThis\}$$
$$toInEpv = toInEpv' \cup \{pdThis\}$$

19.6.4 fromInEpa

Only the *to* purse changes.

$$fromInEpa' = fromInEpa$$

19.6.5 definitelyLost

$$\begin{aligned} & definitelyLost' \\ & \quad = toLogged' \cap (fromLogged' \cup fromInEpa') && \text{[defn]} \\ & \quad = toLogged \cap (fromLogged \cup fromInEpa) && \text{[above]} \\ & \quad = definitelyLost && \text{[defn]} \end{aligned}$$

19.6.6 chosenLost

$$chosenLost' = chosenLost$$

by choice. So

$$definitelyLost \cup chosenLost = definitelyLost' \cup chosenLost'$$

19.6.7 maybeLost

$$\begin{aligned} & maybeLost' \\ & \quad = (fromInEpa' \cup fromLogged') \cap toInEpv' && \text{[defn]} \\ & \quad = (fromInEpa \cup fromLogged) \cap (toInEpv \setminus \{pdThis\}) && \text{[above]} \\ & \quad = ((fromInEpa \cup fromLogged) \cap toInEpv) \setminus \{pdThis\} && \text{[Spivey]} \\ & \quad = maybeLost \setminus \{pdThis\} && \text{[defn]} \end{aligned}$$

$$\begin{aligned} & val \in ether \land to.status = epv && \text{[precondition } ValPurseOkay\text{]} \\ & \quad \Rightarrow pdThis \in fromInEpa \cup fromLogged && \text{[B–11]} \\ & \quad \Rightarrow pdThis \in maybeLost && [toInEpv\text{, defn } maybeLost] \end{aligned}$$

$$\begin{aligned} & pdThis \in maybeLost && \text{[above]} \\ & \land\ pdThis \notin chosenLost' && \text{[because } pdThis \notin maybeLost'\text{]} \\ & \quad \Rightarrow pdThis \in maybeLost \land pdThis \notin chosenLost \\ & \quad \Rightarrow pdThis \in maybeLost \setminus chosenLost \end{aligned}$$

Also

$$maybeLost \setminus chosenLost = (maybeLost' \setminus chosenLost') \cup \{pdThis\}$$

19.7 Clarifying the hypothesis

We can show that the hypothesis is actually stronger than it looks, in that we can replace *RabClPd* with *RabOkayClPd* and replace *RabClPd′* with *RabEndClPd′*. This is because $pdThis \in maybeLost \setminus chosenLost$, implying that *RabOkayClPd* holds.

$pdThis \notin maybeLost'$ (see construction of $maybeLost'$) and so it cannot be in $chosenLost'$. $pdThis \notin maybeLost'$ and so it cannot be in $maybeLost' \setminus chosenLost'$. $pdThis \notin definitelyLost'$ because it is not in $toLogged'$.

This implies that $RabEndClPd'[\ldots]$ holds. So we have to prove

$$\begin{aligned} & \Phi BOp;\ ValPurseOkay;\ RabEndClPd'[pdThis/pdThis'];\\ & \qquad AbWorld;\ RabOkayClPd \mid \\ & \quad pdThis = (conAuthPurse\ name?).pdAuth \\ & \quad \land\ chosenLost = chosenLost' \\ & \vdash \\ & \forall\, n : \operatorname{dom} abAuthPurse \bullet \\ & \quad (abAuthPurse'\ n).balance = (abAuthPurse\ n).balance \\ & \quad \land\ (abAuthPurse'\ n).lost = (abAuthPurse\ n).lost \end{aligned}$$

We do this for each of the three components, for all the purses other than the *to* purse engaged in this transaction, and for exactly the *to* purse in this transaction.

19.7.1 Case *balance* component for non-*pdThis.to* purse

$$\begin{aligned} & \forall\, n : \operatorname{dom} abAuthPurse \mid n \neq pdThis.to \bullet \\ & \quad (abAuthPurse'\ n).balance \\ & \qquad = (conAuthPurse'\ n).balance \\ & \qquad\quad + sumValue(((maybeLost' \setminus chosenLost') \\ & \qquad\quad \cap \{pd : PayDetails \mid pd.to = n\}) \setminus \{pdThis\}) \\ & && [RabEndClPd'[pdThis/pdThis']] \\ & \quad = (conAuthPurse'\ n).balance \\ & \qquad\quad + sumValue((((maybeLost' \setminus chosenLost') \cup \{pdThis\}) \\ & \qquad\quad \cap \{pd : PayDetails \mid pd.to = n\}) \setminus \{pdThis\}) \\ & && \text{[union and subtraction cancel]} \end{aligned}$$

$$
\begin{aligned}
&= (conAuthPurse'\ n).balance \\
&\quad + sumValue(((maybeLost \setminus chosenLost) \\
&\quad \cap \{pd : PayDetails \mid pd.to = n\}) \setminus \{pdThis\}) \\
&\hfill [\text{equation earlier}] \\
&= (conAuthPurse\ n).balance \\
&\quad + sumValue(((maybeLost \setminus chosenLost) \\
&\quad \cap \{pd : PayDetails \mid pd.to = n\}) \setminus \{pdThis\}) \\
&\hfill [\Phi BOp] \\
&= (abAuthPurse\ n).balance \qquad [RabOkayClPd]
\end{aligned}
$$

■ 19.7.1

19.7.2 Case *lost* component for non-*pdThis.to* purse

In this case the defining equations in the retrieve depend upon *definitelyLost* ∪ *chosenLost*, which we derived as unchanging earlier. ΦBOp does not change the concrete values, so the abstract values do not change either.

■ 19.7.2

19.7.3 Case *balance* component for *pdThis.to* purse

$$
\begin{aligned}
&(abAuthPurse'\ pdThis.to).balance \\
&= (conAuthPurse'\ pdThis.to).balance \\
&\quad + sumValue(((maybeLost' \setminus chosenLost') \\
&\qquad \cap \{pd : PayDetails \mid pd.to = pdThis.to\}) \setminus \{pdThis\}) \\
&\hfill [RabEndClPd'[\ldots]] \\
&= (conAuthPurse'\ pdThis.to).balance \\
&\quad + sumValue((((maybeLost' \setminus chosenLost') \cup \{pdThis\}) \\
&\qquad \cap \{pd : PayDetails \mid pd.to = pdThis.to\}) \setminus \{pdThis\}) \\
&\hfill [\text{union and subtraction cancel}] \\
&= (conAuthPurse'\ pdThis.to).balance \\
&\quad + sumValue(((maybeLost \setminus chosenLost) \\
&\qquad \cap \{pd : PayDetails \mid pd.to = pdThis.to\}) \setminus \{pdThis\}) \\
&\hfill [\text{equation earlier}]
\end{aligned}
$$

$$
\begin{aligned}
&= (conAuthPurse\ pdThis.to).balance + pdThis.value \\
&\quad + sumValue(((maybeLost \setminus chosenLost) \\
&\qquad \cap \{pd : PayDetails \mid pd.to = pdThis.to\}) \setminus \{pdThis\}) \\
&\hfill [ValPurseOkay] \\
&= (abAuthPurse\ pdThis.to).balance \qquad [RabOkayClPd]
\end{aligned}
$$

■ 19.7.3

19.7.4 Case *lost* component for *pdThis.to* purse

In this case the defining equations in the retrieve depend upon *definitelyLost* ∪ *chosenLost*, which we derived as unchanging earlier. *ValOkay* does not change the concrete values, so the abstract values do not change either.

■ 19.7.4

■ 19.7

■ 19

Chapter 20

Correctness of *Ack*

20.1 Proof obligation

We have to prove the correct refinement of each abstract operation. In section 9.2.4 we give a general simplification of the correctness proof. We use lemma 'multiple refinement' (section 14.2) to split the proof obligation for each $\mathcal{A}$ operation into one for each individual $\mathcal{B}$ operation.

This chapter proves the $\mathcal{B}$ operation.

- We use lemma 'ignore' (see section 14.3) to simplify the proof obligation by proving the correctness of *Ignore* (in section 14.7), leaving the *Okay* branch to be proven here.
- We use lemma 'deterministic' (section C.1) to reduce the proof obligation to the three cases **exists-pd**, **exists-chosenLost**, and **check-operation**.
- Since this operation refines *AbIgnore*, we use lemma '*AbIgnore*' (from section C.3) to simplify **check-operation** to **check-operation-ignore**.

20.2 Instantiating lemma 'deterministic'

We must instantiate two general predicates relating to *pdThis* and *chosenLost*. The choices for these predicates are based on the fact that the important transaction is the one stored in the purse performing the *AckOkay* operation, and that before the operation, the set of transactions chosen to be lost should be all those chosen to be lost after the operation, because this operation plays no part in deciding which transactions succeed and which ones lose. Thus

$$\mathcal{P} \Leftrightarrow pdThis = (conAuthPurse\ name?).pdAuth$$

$$\mathcal{Q} \Leftrightarrow chosenLost = chosenLost'$$

20.3 exists-pd

$$\begin{array}{l} \Phi BOp;\ AckPurseOkay;\ RabOut;\ RabCl';\ RabIn \\ \vdash \\ \exists\, pdThis : PayDetails \bullet pdThis = (conAuthPurse\ name?).pdAuth \end{array}$$

Proof:
This is immediate by [*one point*] rule, as we have an explicit definition of *pdThis*.
■ 20.3

20.4 exists-chosenlost

$$\begin{array}{l} \Phi BOp;\ AckPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\ RabIn \mid \\ \quad pdThis = (conAuthPurse\ name?).pdAuth \\ \vdash \\ \exists\, chosenLost : \mathbb{P}\, PayDetails \bullet \\ \quad chosenLost = chosenLost' \\ \quad \wedge\ chosenLost \subseteq maybeLost \end{array}$$

Proof:
We can [*one point*] away the quantification because we have an explicit definition of *chosenLost* (as *chosenLost'*). We show that the constraint holds by

$$\begin{array}{lr} chosenLost = chosenLost' & [\text{def}] \\ \quad \subseteq maybeLost' & [RabClPd'[\ldots]] \\ \quad \subseteq maybeLost & [\text{see } 20.6.6] \end{array}$$

■ 20.4

20.5 check-operation

$$\begin{array}{l} \Phi BOp;\ AckPurseOkay;\ RabClPd'[pdThis/pdThis'];\ AbWorld;\ RabClPd \mid \\ \quad pdThis = (conAuthPurse\ name?).pdAuth \\ \quad \wedge\ chosenLost = chosenLost' \\ \vdash \\ \forall\, n : \mathrm{dom}\, abAuthPurse \bullet \\ \quad (abAuthPurse'\ n).balance = (abAuthPurse\ n).balance \\ \quad \wedge\ (abAuthPurse'\ n).lost = (abAuthPurse\ n).lost \end{array}$$

Proof:
We prove this by investigating the way in which the key sets *definitelyLost* and *maybeLost* are modified by the operation.

20.6 Behaviour of *maybeLost* and *definitelyLost*

We argue that the transaction *pd* is initially in neither *maybeLost* nor *definitelyLost*, and is not moved into either of them by the *AckOkay* operation. The transaction was initially far enough along to have already succeeded.

20.6.1 Behaviour of *fromLogged*

From ΦBOp, which says that only the purse *name*? changes, and then only according to *AckPurseOkay*, and from the definition of *AckPurseOkay*, in which $exLog' = exLog$, we can see that

$$fromLogged' = fromLogged$$

20.6.2 Behaviour of *toLogged*

Exactly as we argued for *fromLogged*,

$$toLogged' = toLogged$$

20.6.3 Behaviour of *toInEpv*

If $toInEpv' \neq toInEpv$, there must be some *pd* in one and not in the other. From the definition of *toInEpv*, this means that for some purse that changes, either before or after the operation its status must equal *epv*. That is,

$$\begin{array}{l} (conAuthPurse\ pd.to).status = epv \\ \vee \\ (conAuthPurse'\ pd.to).status = epv \end{array}$$

From ΦBOp we have that the only purse that changes is $name?$. From *AckPurse-Okay* we have that

$$(conAuthPurse\ name?).status = epa$$
$$(conAuthPurse'\ name?).status = eaFrom$$

(neither equal to epv). Therefore, no such pd exists, and we have

$$toInEpv' = toInEpv$$

20.6.4 Behaviour of *fromInEpa*

If $fromInEpa' \neq fromInEpa$, there must be some pd in one and not in the other. From the definition of *fromInEpa*, this means that for some purse that changes, either before or after the operation its status must equal epa. That is,

$$\begin{array}{l}(conAuthPurse\ pd.from).status = epa \\ \vee \\ (conAuthPurse'\ pd.from).status = epa\end{array}$$

The only name that changes is $name?$, and from *AckPurseOkay* we have that

$$(conAuthPurse\ name?).status = epa$$
$$(conAuthPurse'\ name?).status = eaFrom$$

Therefore, we have

$$\begin{array}{l}fromInEpa' = fromInEpa \setminus \{\, pd : PayDetails \mid pd.from = name? \\ \qquad\qquad\qquad \wedge\ (conAuthPurse\ name?).status = epa \\ \qquad\qquad\qquad \wedge\ (conAuthPurse\ name?).pdAuth = pd \,\}\end{array}$$

In fact, the last predicate in this set limits the pd to a single value, equal to $pdThis$, so we have

$$fromInEpa' = fromInEpa \setminus \{pdThis\}$$

We now build up the two sets *definitelyLost* and *maybeLost*.

20.6.5 Behaviour of *definitelyLost*

$$\begin{array}{ll}definitelyLost' = toLogged' \cap (fromLogged' \cup fromInEpa') & [\text{defn}] \\ \quad = toLogged & [\text{above identities}] \\ \qquad \cap\ (fromLogged \cup (fromInEpa \setminus \{pdThis\})) & \\ \quad = toLogged & [pdThis \notin fromLogged, \text{ see below}] \\ \qquad \cap\ ((fromLogged \cup fromInEpa) \setminus \{pdThis\}) & \\ \quad = (fromLogged \cup fromInEpa) & [\text{algebra}] \\ \qquad \cap\ (toLogged \setminus \{pdThis\}) & \\ \quad = (fromLogged \cup fromInEpa) \cap toLogged & [pdThis \notin toLogged, \text{ see below}] \\ \quad = definitelyLost & [\text{defn}]\end{array}$$

We have $pdThis \notin fromLogged$, from the fact that $pdThis \in fromInEpa$ (because the before purse state is epa, and ΦBOp gives $pdThis \in authenticFrom$), and using lemma 'notLoggedAndIn'.

We have $pd \notin toLogged$:

$$\begin{array}{ll}ack\ pd \in ether & [\text{precondition } AckPurseOkay] \\ \quad \Rightarrow pd \notin toInEpv \cup toLogged & [BetweenWorld \text{ constraint B–10}] \\ \quad \Rightarrow pd \notin toLogged & [\text{law}]\end{array}$$

Thus we have

$$definitelyLost' = definitelyLost$$

20.6.6 Behaviour of *maybeLost*

$$\begin{array}{ll}maybeLost' = (fromInEpa' \cup fromLogged') \cap toInEpv' & [\text{defn.}] \\ \quad = (fromInEpa \cup (fromLogged \setminus \{pdThis\})) \cap toInEpv & \\ & [\text{above identities}] \\ \quad = ((fromInEpa \cup fromLogged) \setminus \{pdThis\}) \cap toInEpv & \\ & [pdThis \notin fromLogged, \text{ as above}] \\ \quad = (fromInEpa \cup fromLogged) \cap (toInEpv \setminus \{pdThis\}) & [\text{algebra}] \\ \quad = (fromInEpa \cup fromLogged) \cap toInEpv & [pdThis \notin toInEpv, \text{ see below}] \\ \quad = maybeLost & [\text{defn.}]\end{array}$$

We have $pdThis \notin toInEpv$:

$$
\begin{aligned}
& ack\ pd \in ether && \text{[precondition } AckOkay\text{]} \\
& \quad \Rightarrow pdThis \notin toInEpv \cup toLogged && \text{[}BetweenWorld\text{ constraint B–10]} \\
& \quad \Rightarrow pdThis \notin toInEpv && \text{[law]}
\end{aligned}
$$

Thus we have

$$maybeLost' = maybeLost$$

20.7 Finishing proof of check-operation

The above shows that none of the three sets *definitelyLost*, *maybeLost* or *chosenLost* changes. As *AckOkay* does not alter any concrete *balance* or *lost*, and given that the abstract values are defined solely in terms of these (unchanging) values, it follows that the abstract values don't change, thus discharging the **check-operation** proof obligation.

■ 20.5

■ 20

Chapter 21

Correctness of *ReadExceptionLog*

21.1 Proof obligation

We have to prove the correct refinement of each abstract operation. In section 9.2.4 we give a general simplification of the correctness proof. We use lemma 'multiple refinement' (section 14.2) to split the proof obligation for each $\mathcal{A}$ operation into one for each individual $\mathcal{B}$ operation.

This chapter proves the $\mathcal{B}$ operation.

- We use lemma 'ignore' (see section 14.3) to simplify the proof obligation by proving the correctness of *Ignore* (in section 14.7), and *Abort* (in section 14.8), leaving the *Okay* branch to be proven here.
- Since the *Okay* branch of this operation is expressed as a promotion of *AbortPurseOkay* composed with a simpler *EafromPurseOkay* operation, we use lemma 'abort backward' (section C.5), and prove only that the promotion of the simpler operation is a refinement.
- We use lemma 'deterministic' (section C.1) to reduce the proof obligation to the three cases **exists-pd**, **exists-chosenLost**, and **check-operation**.
- Since this operation leaves the sets *maybeLost* and *definitelyLost* unchanged, we use lemma 'lost unchanged' (section C.2) to discharge the **exists pd-and exists chosenLost-obligations** automatically.
- Since this operation refines *AbIgnore*, we use lemma '*AbIgnore*' (from section C.3) to simplify **check-operation** to **check-operation-ignore**.

21.2 Invoking lemma 'lost unchanged'

We have the constraint $\Xi ConPurse$ in the definition of *ReadExceptionLogPurseEafromOkay*. From ΦBOp and $\Xi ConPurse$, we know that *archive* and *conAuthPurse* remain unchanged, as do *definitelyLost* and *maybeLost*. Hence we can invoke lemma 'Lost unchanged'.

21.3 check-operation-ignore

$$
\begin{array}{l}
\Phi BOp;\ ReadExceptionLogPurseEafromOkay; \\
\qquad RabOut;\ RabClPd'[pdThis/pdThis']; \\
\qquad AbWorld;\ RabClPd;\ RabIn \mid \\
\quad chosenLost' = chosenLost \\
\quad \land\ maybeLost' = maybeLost \\
\quad \land\ definitelyLost' = definitelyLost \\
\vdash \\
\forall\, n : \mathrm{dom}\, abAuthPurse \bullet \\
\quad (abAuthPurse'\ n).balance = (abAuthPurse\ n).balance \\
\quad \land\ (abAuthPurse'\ n).lost = (abAuthPurse\ n).lost
\end{array}
$$

Proof:
We have that *maybeLost* and *definitelyLost* are unchanged from the hypothesis. Hence the *balance* and *lost* components of all the abstract purses remain unchanged, satisfying our proof requirement.

■ 21.3

■ 21

Chapter 22

Correctness of *ClearExceptionLog*

22.1 Proof obligation

We have to prove the correct refinement of each abstract operation. In section 9.2.4 we give a general simplification of the correctness proof. We use lemma 'multiple refinement' (section 14.2) to split the proof obligation for each $\mathcal{A}$ operation into one for each individual $\mathcal{B}$ operation.

This chapter proves the $\mathcal{B}$ operation.

- We use lemma 'ignore' (see section 14.3) to simplify the proof obligation by proving the correctness of *Ignore* (in section 14.7), and *Abort* (in section 14.8), leaving the *Okay* branch to be proven here.
- Since the *Okay* branch of this operation is expressed as a promotion of *AbortPurseOkay* composed with a simpler *EafromPurseOkay* operation, we use lemma 'abort backward' (section C.5), and prove only that the promotion of the simpler operation is a refinement.
- We use lemma 'deterministic' (section C.1) to reduce the proof obligation to the three cases **exists-pd**, **exists-chosenLost**, and **check-operation**.
- Since this operation leaves the sets *maybeLost* and *definitelyLost* unchanged, we use lemma 'lost unchanged' (section C.2) to discharge the **exists pd-and exists chosenLost-obligations** automatically.
- Since this operation refines *AbIgnore*, we use lemma '*AbIgnore*' (from section C.3) to simplify **check-operation** to **check-operation-ignore**.

22.2 Invoking lemma ‘Lost unchanged’

The purse’s exception log is cleared, so we cannot use the ‘sufficient conditions’ to invoke lemma ‘lost unchanged’: we need first to show that *fromLogged* and *toLogged* are unchanged.

We have from the operation definition that the exception log details in the purse that are to be cleared match the ones in the *exceptionLogClear* message. We have, from constraint B–15 that the log details in the message are already in the *archive*. So deleting them from the purse will not change *allLogs*. But *fromLogged* and *toLogged* partition *allLogs*, so these do not change either.

Hence we can invoke lemma ‘Lost unchanged’.

22.3 check-operation-ignore

$$
\begin{array}{l}
\Phi BOp;\ ClearExceptionLogPurseEafromOkay; \\
\qquad RabOut;\ RabClPd'[pdThis/pdThis']; \\
\qquad AbWorld;\ RabClPd;\ RabIn\ | \\
\quad chosenLost' = chosenLost \\
\quad \land\ maybeLost' = maybeLost \\
\quad \land\ definitelyLost' = definitelyLost \\
\vdash \\
\forall\ n : \mathrm{dom}\ abAuthPurse \bullet \\
\quad (abAuthPurse'\ n).balance = (abAuthPurse\ n).balance \\
\quad \land\ (abAuthPurse'\ n).lost = (abAuthPurse\ n).lost
\end{array}
$$

Proof:
We have that *maybeLost* and *definitelyLost* are unchanged from the hypothesis. Hence the *balance* and *lost* components of all the abstract purses remain unchanged.

■ 22.3

■ 22

Chapter 23

Correctness of *AuthoriseExLogClear*

23.1 Proof obligation

We have to prove the correct refinement of each abstract operation. In section 9.2.4 we give a general simplification of the correctness proof. We use lemma ‘multiple refinement’ to split the proof obligation for each $\mathcal{A}$ operation into one for each individual $\mathcal{B}$ operation.

This chapter proves the $\mathcal{B}$ operation.

- We use lemma ‘ignore’ to simplify the proof obligation further to proving the correctness of *Ignore* (section 14.7), leaving the *Okay* branch to be proven.

We cannot use any of the other simplifications directly for *AuthoriseExLogClear*, since it cannot be written as a promotion. So the correctness proof obligation for *AuthoriseExLogClear* is

$$
\begin{array}{l}
AuthoriseExLogClearOkay;\ Rab';\ RabOut \\
\vdash \\
\exists\ AbWorld;\ a? : AIN \bullet Rab \land RabIn \land AbIgnore
\end{array}
$$

23.2 Proof

First we choose an input. We argue exactly as in section 14.4.1 to reduce the obligation to:

$$
\begin{array}{l}
AuthoriseExLogClearOkay;\ Rab';\ RabOut;\ RabIn \\
\vdash \\
\exists\ AbWorld \bullet Rab \land AbIgnore
\end{array}
$$

We [*cut*] in a before *AbWorld* equal to the after *AbWorld'* in *Rab'* (the side lemma is trivial), and use [*consq exists*] to remove the quantifier from the consequent.

$$\begin{array}{l} \mathit{AuthoriseExLogClearOkay};\ \mathit{Rab}';\ \mathit{RabOut};\ \mathit{RabIn};\ \mathit{AbWorld} \mid \\ \quad \theta \mathit{AbWorld} = \theta \mathit{AbWorld}' \\ \vdash \\ \mathit{Rab} \wedge \mathit{AbIgnore} \end{array}$$

AbIgnore is certainly satisfied by the equal abstract before and after worlds.

It remains to show that *Rab* is satisfied. The only difference between the concrete before and after worlds, as given by *AuthoriseExLogClearOkay*, is the addition of an *exceptionLogClear* message in the *ether*. But *Rab* does not depend on *exceptionLogClear* messages, and so we can deduce *Rab* directly from *Rab'*

■ 23.2

■ 23

Chapter 24

Correctness of *Archive*

24.1 Proof obligation

We have to prove the correct refinement of each abstract operation. In section 9.2.4 we give a general simplification of the correctness proof. We use lemma 'multiple refinement' to split the proof obligation for each $\mathcal{A}$ operation into one for each individual $\mathcal{B}$ operation.

This chapter proves the $\mathcal{B}$ operation.

We cannot use any more of the usual simplifications directly for *Archive*, since it cannot be written as a promotion. So the correctness proof obligation for *Archive* is

$$\mathit{Archive};\ \mathit{Rab}';\ \mathit{RabOut} \vdash \exists\, \mathit{AbWorld};\ a? : \mathit{AIN} \bullet \mathit{Rab} \wedge \mathit{RabIn} \wedge \mathit{AbIgnore}$$

24.2 Proof

First we choose an input. We argue exactly as in section 14.4.1 to reduce the obligation to:

$$\mathit{Archive};\ \mathit{Rab}';\ \mathit{RabOut};\ \mathit{RabIn} \vdash \exists\, \mathit{AbWorld} \bullet \mathit{Rab} \wedge \mathit{AbIgnore}$$

We [*cut*] in a before *AbWorld* equal to the after *AbWorld'* in *Rab'* (the side lemma is trivial), and use [*consq exists*] to remove the quantifier from the consequent.

$$\begin{array}{l} \mathit{Archive};\ \mathit{Rab}';\ \mathit{RabOut};\ \mathit{RabIn};\ \mathit{AbWorld} \mid \\ \quad \theta \mathit{AbWorld} = \theta \mathit{AbWorld}' \\ \vdash \\ \mathit{Rab} \wedge \mathit{AbIgnore} \end{array}$$

AbIgnore is certainly satisfied by the equal abstract before and after worlds.

It remains to show that *Rab* is satisfied. The only difference between the concrete before and after worlds, as given by *Archive*, is the inclusion of some log details in the *archive*. We have, from *BetweenWorld* constraint B–14, that the log details added to the archive from the *exceptionLogResult* message are already in *allLogs*. So, although the *archive* grows, the operation does not add any new logs to the *world*. Thus *fromLogged* and *toLogged* don't change. Hence *maybeLost* and *definitelyLost* don't change. Therefore, nothing that *Rab* relies upon changes in the concrete world, and so we can deduce *Rab* directly from Rab'.

■ 24.2

■ 24

Part III

Second Refinement: $\mathcal{B}$ to $\mathcal{C}$

Chapter 25

Refinement Proof Rules

25.1 Security of the implementation

We prove the concrete model C is secure with respect to the between model $\mathcal{B}$ by showing that every concrete operation correctly refines a between operation. The concrete and between operations are similarly-named.

The full list of refinements is:

$StartTo \sqsubseteq CStartTo$

$StartFrom \sqsubseteq CStartFrom$

$Req \sqsubseteq CReq$

$Val \sqsubseteq CVal$

$Ack \sqsubseteq CAck$

$ReadExceptionLog \sqsubseteq CReadExceptionLog$

$ClearExceptionLog \sqsubseteq CClearExceptionLog$

$AuthoriseExLogClear \sqsubseteq CAuthoriseExLogClear$

$Archive \sqsubseteq CArchive$

$Abort \sqsubseteq CAbort$

$Increase \sqsubseteq CIncrease$

$Ignore \sqsubseteq CIgnore$

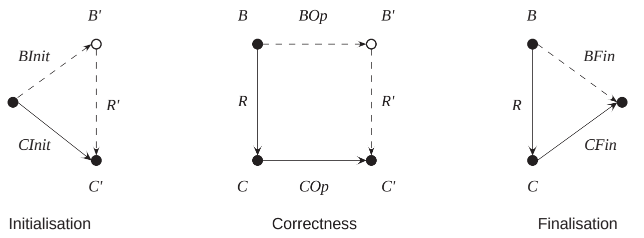

Figure 25.1: A summary of the forward proof rules. The hypothesis is the existence of the lower (solid) path. The proof obligation is to demonstrate the existence of an upper (dashed) path.

25.2 Forwards rules proof obligations

Each of these refinements must be proved correct. [Spivey 1992b, Chapter 5] presents the theorems that need to be proved for the most commonly-occurring case of non-determinism, sometimes called 'downward' or 'forward' conditions, where the abstract and concrete inputs and outputs are identical. These, augmented with a finalisation proof, are appropriate for the $\mathcal{B}$ to C refinement proofs.

The forward rules are summarised in figure 25.1. Note how the paths are different from the backward case (figure 9.1) because of the direction of the R arrows.

25.2.1 Retrieve

The retrieve relation has one part that links the abstract and concrete states.

25.2.2 Initialisation

$$CInit \vdash \exists B' \bullet BInit \land R'$$

25.2.3 Finalisation

$$R;\ CFin \vdash BFin$$

25.2.4 Applicability

$$R;\ BIn \mid \text{pre } BOp \vdash \text{pre } COp$$

25.2.5 Correctness

$$R;\ COp \mid \text{pre } BOp \vdash \exists B' \bullet R' \land BOp$$

We can simplify the correctness condition because we know that all the between operations are total, i.e.

$$\text{pre } BOp = true$$

This was proved earlier, in section 8.3.2.

We can therefore simplify the correctness condition to

$$R;\ COp \vdash \exists B' \bullet R' \land BOp$$

Chapter 26

$\mathcal{B}$ to $\mathcal{C}$ retrieve relation

26.1 Retrieve state

The $\mathcal{B}$ and $\mathcal{C}$ worlds are identical, except that the $\mathcal{C}$ world can 'lose' *ether* messages.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Rbc}} \\
\hline
\textit{BetweenWorld} \\
\textit{ConWorld}_0 \\
\hline
\textit{conAuthPurse}_0 = \textit{conAuthPurse} \\
\textit{ether}_0 \subseteq \textit{ether} \\
\textit{archive}_0 = \textit{archive} \\
\hline
\end{array}
$$

The subscript zero on the concrete world serves to distinguish like-named between and concrete components.

Chapter 27

Initialisation, Finalisation, and Applicability

27.1 Initialisation proof

$$ConInitState \vdash \exists\, BetweenWorld' \bullet BetweenInitState \wedge Rbc'$$

Proof:
We expand *ConInitState* in the hypothesis according to its definition.

$$\begin{array}{l}
ConWorld_0' \mid \\
\quad (\, \exists\, BetweenWorld' \mid BetweenInitState \bullet \\
\quad\quad conAuthPurse_0' = conAuthPurse' \\
\quad\quad \wedge\ archive' = archive' \\
\quad\quad \wedge\ \{\perp\} \subseteq ether_0' \subseteq ether'\ ) \\
\vdash \\
\exists\, BetweenWorld' \bullet BetweenInitState \wedge Rbc'
\end{array}$$

From the definition of Rbc', we can see that the consequent follows directly from the hypothesis.

■ 27.1

27.2 Finalisation proof

$$Rbc;\ ConFinState \vdash BetwFinState$$

Proof:
We have defined *ConFinState* and *BetwFinState* to have the same mathematical form.

Rbc in the hypothesis requires the concrete and between purse states and archives to be identical, and allows the between *ether* to be bigger than the concrete *ether*.

Finalisation of the purses depends only on the purse states (identical by hypothesis) and on the sets *definitelyLost* and *maybeLost*. These sets themselves depend only on purse states and on the archive (also identical for concrete and between worlds by the retrieve in the hypothesis). As result, *gAuthPurse* for between finalisation is identical to that for concrete finalisation.

■ 27.2

27.3 Applicability proofs

Applicability follows automatically from the totality of the concrete operations as shown in section 8.4.

■ 27.3

Chapter 28

Lemmas for the $\mathcal{B}$ to $\mathcal{C}$ correctness proofs

28.1 Specialising the proof rules

For each concrete operation *COp* and corresponding between operation *BOp* we have to show

$$Rbc;\ COp \vdash \exists\, BetweenWorld' \bullet Rbc' \land BOp$$

Many operations are defined as the disjunction of other operations. A *COp* will have the same branches as a corresponding *BOp*: a *CIgnore* branch, and either a *CAbort* or *COpOkay* branch, or both. We split the proof obligation into *CIgnore*, *CAbort* and *COpOkay* branches, as we did in section 14.3. This gives some or all of the following proof requirements, depending on which branches are in *COp*:

$$Rbc;\ CIgnore \vdash \exists\, BetweenWorld' \bullet Rbc' \land Ignore$$
$$Rbc;\ CAbort \vdash \exists\, BetweenWorld' \bullet Rbc' \land Abort$$
$$Rbc;\ COpOkay \vdash \exists\, BetweenWorld' \bullet Rbc' \land BOpOkay$$

The correctness of the *CIgnore* branch is dealt with below in section 28.2. We then develop the correctness proof for the *CAbort* and *COpOkay* branches, and introduce a lemma applicable to certain operations. Following this, we present the proof of correctness of two common branches — *CIncrease* and *CAbort*.

28.2 Correctness of *CIgnore*

The correctness of the *CIgnore* branch follows trivially by choosing

$$\theta BetweenWorld' = \theta BetweenWorld$$

■ 28.2

28.3 Correctness of a branch of the operation

28.3.1 Choosing *BetweenWorld'*

In choosing *BetweenWorld'*, we base our choice of the *conAuthPurse'* and *archive'* components on *Rbc'*, and our choice of the *ether'* component on *BOpOkay'*.

We have $conAuthPurse'_0$ and $archive'_0$ in the hypothesis, and we use this to provide the value for *conAuthPurse'* and *archive'*, respectively (this satisfies the constraint on *conAuthPurse'* and *archive'* in *Rbc'*).

$$conAuthPurse' = conAuthPurse'_0$$
$$archive' = archive'_0$$

m! and *ether* are declared in the hypothesis, and *ether'* can be constructed deterministically from these (note that the following construction satisfies the relevant constraint in *BOpOkay* — either in ΦBOp or explicitly as in *Archive*).

$$ether' = ether \cup \{m!\}$$

We need to show that the chosen *BetweenWorld'* and *m*! satisfy each of the conjuncts in the consequent (retrieve *Rbc'* and operation *BOpOkay*).

We also need to show that this choice is indeed an after *BetweenWorld'* (that it satisfies the constraints on *BetweenWorld* specified in section 5.3).

28.3.2 Case *BOpOkay*

From the choice of *ether'* above, the relevant constraint on *ether'* in *BOpOkay* is satisfied by construction.

At most one purse changes in *COpOkay*. Let us call this new purse value *p*. This gives

$$conAuthPurse'_0 = conAuthPurse_0 \oplus \{p\}$$
$$conAuthPurse'_0 = conAuthPurse \oplus \{p\} \qquad [Rbc]$$
$$conAuthPurse' = conAuthPurse \oplus \{p\} \qquad [\text{choice of } conAuthPurse']$$

This satisfies the constraint on *conAuthPurse'* in *BOpOkay* (where at most one purse changes in an identical manner to *COpOkay*).

archive' is a function of *archive* and *m*!, defined in *BOpOkay*. Call this function *f*:

$$f : Logbook \times MESSAGE \rightarrow Logbook$$

Because *COpOkay* is defined in an analogous way, *f* also relates $archive'_0$ to $archive_0$ and *m*!.

From the hypothesis we have *COpOkay* and *Rbc*, and with our choice of *archive'* we have, respectively

$$archive'_0 = f(archive_0, m!)$$
$$\wedge\ archive = archive_0$$
$$\wedge\ archive' = archive'_0$$

Substituting the latter two equations into the first gives the predicate in *BOpOkay*.

Thus, the *BOpOkay* constraints on all the components of our chosen *BetweenWorld'* are satisfied under the correctness hypothesis and choice of *BetweenWorld'*.

■ 28.3.2

28.3.3 Case *Rbc'*

Both the *conAuthPurse'* and *archive'* components of *BetweenWorld'* satisfy *Rbc'* from the choice of *BetweenWorld'*.

All *COpOkay* operations constrain *ether'* as

$$ether'_0 \subseteq ether_0 \cup \{m!\}$$

either through ΦCOp, or explicitly in *CArchive.* Hence for *ether'* we have

$$\begin{aligned} & ether' \\ & \quad = ether \cup \{m!\} && [\text{choice of } ether'] \\ & \quad \supseteq ether_0 \cup \{m!\} && [Rbc] \\ & \quad \supseteq ether'_0 && [COpOkay] \end{aligned}$$

This satisfies the constraint on *ether'* in *Rbc'*.

28.3.4 Case 'obey constraints'

We know from the hypothesis that the before *BetweenWorld* satisfies the constraints, so we need check only that the chosen message *m*!, and any change of purse state during the operation, maintains this constraint.

Lemma 28.1 (constraint) If an operation obeys the following properties, then it preserves the *BetweenWorld* constraints:

- it does not change purse status or current transaction details (*pdAuth*)
- it does not change *allLogs*
- it does not change the payment detail messages, exception log read messages or exception log clear messages in the *ether* (either by not emitting such a message, or by emitting an already existing message)
- no sequence number decreases (all concrete operations have the property, so it is automatically satisfied)

■

Proof:
The *BetweenWorld* constraints refer only to certain *ether* messages (*req*, *val*, *ack*, *exceptionLogResult* and *exceptionLogClear*), and relate their presence or absence to purse status (*status*, *pdAuth* and *nextSeqNo*) and *allLogs*. From the hypothesis we can invoke lemma 'logs unchanged' (section C.7) to say that, as *allLogs* does not change, not does *alLogs*. So operations that do not change the purse status, do not change *allLogs*, and do not emit any relevant new messages, will automatically preserve the constraints.
■ 28.3.4
Even when lemma 'constraint' does not apply, we know from the form of the operation that at most one purse changes, and one message is emitted. As at most one purse changes, the proof that the *BetweenWorld* constraints are preserved need refer only to this purse; the constraints hold on the other purses before the operation by hypothesis, and so they hold afterward, too.

28.3.5 Summary of *ConOkay* proof obligation

For each operation, we have to show that either lemma 'constraint' holds or that the choice of *BetweenWorld′* obeys the constraints (see section 5.3).

28.4 Correctness of *CIncrease*

CIncrease does not change *status* or *pdAuth*, does not log, and no relevant message is emitted to the *ether*, so lemma 'constraint' (section C.6) is applicable.
■ 28.4

28.5 Correctness of *CAbort*

Lemma 'constraint' is not applicable, because *CAbort* moves one purse into *eaFrom*, and it may not have been in this state before, and it may log a pending transaction. Therefore we have to show that our chosen *BetweenWorld′* obeys the constraints.

One $\perp$ message is emitted, and (possibly) one log is recorded.

B–1 *req* $\Rightarrow$ authentic *to* purse. No new *req* messages.

B–2 No future *req*s. No new *req* messages.

B–3 No future *val*s. No new *val* messages.

B–4 No future *ack*s. No new *ack* messages.

B–5 No future *from* logs. The purse moves into *eaFrom*, possibly logging a transaction, and possibly increasing *nextSeqNo*. This does not invalidate this constraint for any previous logs. To create a new *from* log, the purse would have had to have been in *epa* (from *LogIfNecessary*). Hence, using *ConPurse* constraint P–**??**, we have

$$pdAuth.fromSeqNo < nextSeqNo$$

From *AbortPurse*, we also have

$$nextSeqNo \leq nextSeqNo'$$

This gives

$$pdAuth.fromSeqNo < nextSeqNo'$$

The *pdAuth* is logged when the pre-state purse is in *epa*, and thus the new log obeys the constraint.

B–6 No future *to* logs. The purse moves into *eaFrom*, possibly logging a transaction, and possibly increasing *nextSeqNo*. This does not invalidate this constraint for any previous logs. To create a new *to* log, the purse would have had to have been in *epv* (from *LogIfNecessary*). Hence, using *ConPurse* constraint P–**??**, we have

$$pdAuth.toSeqNo < nextSeqNo$$

From *AbortPurse*, we also have

$$nextSeqNo \leq nextSeqNo'$$

This gives

$$pdAuth.toSeqNo < nextSeqNo'$$

The *pdAuth* is logged when the pre-state purse is in *epv*, and thus the new log obeys the constraint.

B–7 *from* in $\{epr, epa\}$, so no future *from* logs. The purse moves into *eaFrom*, so no new purses in *epr* or *epa*.

B–8 *to* in $\{epv, eaTo\}$, so no future *to* logs. The purse moves into *eaFrom*, so no new purses in *epv* or *eaTo*.

B–9 $epr \Rightarrow \neg\, val \wedge \neg\, ack$. The purse moves into *eaFrom*, and so does not move into *epr*.

B–10 $req \wedge \neg\, ack \Leftrightarrow toInEpv \vee toLogged$.

- case $\Rightarrow$:
  No new *req* messages; no *ack* messages removed from the *ether*.
  The purse may have moved out of *epv*, but in such a case *LogIfNecessary* says that it logs, hence re-establishing the condition.
- case $\Leftarrow$:
  No purses newly in *epv*.
  There might be a new *to* log, in which case we must show there was a *req*, but no *ack* before. A *to* log can be made only by a purse moving out of *epv*. Then the *BetweenWorld* constraint B–10, on *toInEpv*, before the operation gives us the required *req* and lack of *ack*.

B–11 $epv \wedge val \Rightarrow fromInEpa \vee fromLogged$. No purses newly in *epv*; no new *val* messages.
The purse may have moved out of *epa*. But in such a case *LogIfNecessary* says that it logs, hence re-establishing the condition.

B–12 $fromInEpa \vee fromLogged \Rightarrow req$. No purses newly in *epa*.
There might be a new *from* log, in which case we must show there was a *req* before. A *from* log can be made only by a purse moving out of *epa*. Then the *BetweenWorld* constraint B–12, on *fromInEpa*, before the operation gives us the required *req*.

B–13 *toLogged* finite. At most one *to* log written, so finite before gives finite after.

B–14 *exceptionLogResult*s in *allLogs*. No new exception log result messages.

B–15 Cleared logs archived. No *exceptionLogClear* messages are added, and the archive is unchanged.

B–16 *req* for each log. If there are no new logs, then the constraint holds from the pre-state.
If a transaction exception is logged, then the purse status must have been either *epv* or *epa*. From constraints B–10 and B–12, there was a *req* in the pre-state *ether* for the transaction which was logged. This *req* will also be in the post-state *ether*.

■ 28.5

28.6 Lemma 'logs unchanged'

Lemma 28.2 (logs unchanged) When the *archive* and the individual purse logs do not change, and when no new *req* messages are added to the *ether*, the set of *PayDetails* representing all the logs does not change either.

$$\begin{array}{l}
BOpOkay \mid archive' = archive \\
\quad \wedge\ req \rhd ether' = req \rhd ether \\
\quad \wedge\ \forall\, n : \mathrm{dom}\, conAuthPurse \bullet \\
\qquad (conAuthPurse'\, n).exLog = (conAuthPurse\, n).exLog \\
\vdash \\
allLogs' = allLogs \\
\wedge\ toLogged' = toLogged \\
\wedge\ fromLogged' = fromLogged
\end{array}$$

■

Proof:

$$
\begin{aligned}
& allLogs = archive \\
& \quad \cup \{ n : \operatorname{dom} conAuthPurse;\ ld : PayDetails \mid \\
& \qquad ld \in (conAuthPurse\ n).exLog \} && \text{[defn]} \\
& \quad = archive' \\
& \quad \cup \{ n : \operatorname{dom} conAuthPurse';\ ld : PayDetails \mid \\
& \qquad ld \in (conAuthPurse'\ n).exLog \} && \text{[assumption and } \Phi BOp\text{]} \\
& \quad = allLogs' && \text{[defn]}
\end{aligned}
$$

$$
\begin{aligned}
& allLogs = \{ n : \operatorname{dom} conAuthPurse;\ pd : PayDetails \mid \\
& \qquad n \mapsto pd \in allLogs \land req\ pd \in ether \} && \text{[defn]} \\
& \quad = \{ n : \operatorname{dom} conAuthPurse';\ pd : PayDetails \mid \\
& \qquad n \mapsto pd \in allLogs' \land req\ pd \in ether' \} && \text{[assumption and above]} \\
& \quad = allLogs' && \text{[defn]}
\end{aligned}
$$

The arguments for *toLogged* and *fromLogged* follow in exactly the same way.
■ 28.6

28.7 Lemma 'abort forward': operations that first abort

Some concrete operations are written as a composition of *Abort* and a simpler operation starting from *eaFrom* (*StartFrom*, *StartTo*, *ReadExceptionLog*, *ClearExceptionLog*, etc.).

Lemma 28.3 (abort forward) Where a C operation is written as a composition of *CAbort* and a simpler operation starting from *eaFrom*, and the corresponding $\mathcal{B}$ operation is structured analogously, it is sufficient to prove that the simpler C operation refines the corresponding $\mathcal{B}$ operation.

$$
\begin{aligned}
& (CAbort \fatsemi COpEafrom);\ Rbc; \\
& \qquad (\forall\ COpEafrom;\ Rbc \bullet \exists\ BetweenWorld' \bullet Rbc' \land BOpEafrom) \\
& \vdash \\
& \exists\ BetweenWorld' \bullet Rbc' \land (Abort \fatsemi BOpEafrom)
\end{aligned}
$$

■

Proof We have already proved in section 28.5 that *CAbort* refines *Abort*. Adding this to our hypothesis, we get

$$
\begin{aligned}
& (CAbort \fatsemi COpEafrom);\ Rbc; \\
& \qquad (\forall\ CAbort;\ Rbc \bullet \exists\ BetweenWorld' \bullet Rbc' \land Abort); \\
& \qquad (\forall\ COpEafrom;\ Rbc \bullet \exists\ BetweenWorld' \bullet Rbc' \land BOpEafrom) \\
& \vdash \\
& \exists\ BetweenWorld' \bullet Rbc' \land (Abort \fatsemi BOpEafrom)
\end{aligned}
$$

The hypothesis is now in precisely the form required to use lemma 'compose forward', (section C.10) and we do so to prove the consequent.
■ 28.7

Chapter 29

Correctness proofs

29.1 Introduction

Many of the following arguments are about constraints of the form

$$antecedent \Rightarrow consequent$$

The correctness arguments are of three kinds:

B–1 Argue that the operation leaves the truth values of both antecedent and consequent unaltered, so that the truth before the operation establishes the truth afterwards.

B–2 The operation might make the antecedent true after when it was false before, by adding a new message to a set, or moving a purse into a set. In this case it is necessary to show that the consequent is true after.

B–3 The operation might make the consequent false after when it was true before, by moving a purse out of a set. In this case it is necessary to show that the antecedent is false after.

Note that we do not need to argue that a constraint cannot be changed by *removing* a message: messages stay in the *ether* once there.

29.2 Correctness of *CStartFrom*

StartFromOkay comprises *AbortPurse* followed by *StartFromEafromPurseOkay* at the unpromoted level. As a result, we can apply lemma 'abort forward' (section C.8), leaving us to prove the correctness of *StartFromEafromPurseOkay*.

Lemma 'constraint' is not applicable, because *StartFromEafromPurseOkay* changes status: it moves the purse from *eaFrom* into *epr*. Therefore we have to show that our chosen *BetweenWorld'* obeys the constraints.

One $\perp$ message is emitted, and no logs are recorded.

We can invoke lemma 'logs unchanged', section C.7, because no new *req* messages are produced, no new purse logs are produced, and the *archive* does not change. Therefore, the sets *allLogs*, *fromLogged* and *toLogged* remain unchanged.

B–1 *req* $\Rightarrow$ authentic *to* purse. No new *req* messages.

B–2 No future *req*s. No new *req* messages.

B–3 No future *val*s. No new *val* messages.

B–4 No future *ack*s. No new *ack* messages.

B–5 No future *from* logs. No new logs.

B–6 No future *to* logs. No new logs.

B–7 *from* in $\{epr, epa\}$ $\Rightarrow$ no future *from* logs. There are no new logs, but the purse moves into *epr*, so we must prove that the constraint for this purse holds (for all other purses in *epr*, the constraint holds beforehand, and so holds afterwards). In *StartFrom*, the post-state *pdAuth'.fromSeqNo* is equal to pre-state *nextSeqNo*. Coupling this with constraint B–5 we have

$$\forall\, pd : fromLogged \mid pd.from = name? \bullet \\ pd.fromSeqNo < (conAuthPurse'\ pd.from).pdAuth.fromSeqNo$$

Since the logs don't change we have

$$\forall\, pd : fromLogged' \mid pd.from = name? \bullet \\ pd.fromSeqNo < (conAuthPurse'\ pd.from).pdAuth.fromSeqNo$$

which proves the constraint for purse *name?*.

B–8 *to* in $\{epv, eaTo\}$ $\Rightarrow$ no future *to* logs. No new logs, and the purse moves into *epr*.

B–9 *epr* $\Rightarrow \neg$ *val* $\wedge \neg$ *ack*. The purse moves into *epr*, so it is necessary to show there was no *val* or *ack* before.
The *pd* we are considering is given by

$$pd == (conAuthPurse'\ name?).pdAuth$$

Noting that *pd.from* = *name?*, the definition of *StartFrom* then gives us that

$$\begin{aligned} &(conAuthPurse\ name?).nextSeqNo \\ &\qquad = (conAuthPurse'\ name?).pdAuth.fromSeqNo \\ &\Rightarrow (conAuthPurse\ pd.from).nextSeqNo = pd.fromSeqNo \\ &\Rightarrow val\ pd \notin ether && [BetweenWorld \text{ constraint B–3}] \\ &\quad \wedge ack\ pd \notin ether && [BetweenWorld \text{ constraint B–4}] \end{aligned}$$

B–10 *req* $\wedge \neg$ *ack* $\Leftrightarrow$ *toInEpv* $\vee$ *toLogged*.

- case $\Rightarrow$:
  No new *req* messages. The purse moved from *eaFrom* to *epr* without generating new logs. Hence, true before implies true after.
- case $\Leftarrow$:
  No purses newly in *epv* and no new logs. No *ack*s added to the *ether*.

B–11 *epv* $\wedge$ *val* $\Rightarrow$ *fromInEpa* $\vee$ *fromLogged*. No purses newly in *epv*; no new *val* messages. The purse did not move out of *epa*.

B–12 *fromInEpa* $\vee$ *fromLogged* $\Rightarrow$ *req*. No purses newly in *epa*; no new logs.

B–13 *toLogged* finite. No new logs.

B–14 *exceptionLogResult*s in *allLogs*. No new log result messages.

B–15 Cleared logs archived. No new *exceptionLogClear* messages.

B–16 *req* for each log. No new elements added to *fromLogged* or *toLogged*.

■ 29.2

29.3 Correctness of *CStartTo*

StartToOkay is composed of *AbortPurse* followed by *StartToEafromPurseOkay* at the unpromoted level. As a result, we can apply lemma 'abort forward' (section C.8), leaving us to prove the correctness of *StartToEafromPurseOkay*.

Lemma 'constraint' is not applicable, because *StartToEafromPurseOkay* moves one purse into *epv*, and it was not in this state before. Therefore we have to show that our chosen *BetweenWorld'* obeys the constraints.

One *req* message is emitted, and no new logs are recorded. We cannot invoke lemma 'logs unchanged' because we do have a new *req* message, but constraint B–16 gives us the same result. This is not a circular argument.

B–1 *req* $\Rightarrow$ authentic *to* purse. One new *req*, which refers to the *name?* purse as the *to* purse. ΦBOp states that this purse is authentic.

B–2 No future *req*s. *StartToPurseEafromOkay* emits one *req* message, which has its *nextSeqNo* in it by construction. It also increases *nextSeqNo*. The *req* message meets the constraints because the referenced *to* purse (itself) has a larger *nextSeqNo* after the operation.

B–3 No future *val*s. No new *val* messages.

B–4 No future *ack*s. No new *ack* messages.

B–5 No future *from* logs. No new logs.

B–6 No future *to* logs. No new logs.

B–7 *from* in $\{epr, epa\}$ $\Rightarrow$ no future *from* logs. There are no new logs and the purse moves into *epv*, so this constraint does not apply to this purse.

B–8 *to* in $\{epv, eaTo\}$ $\Rightarrow$ no future *to* logs. There are no new logs, but the purse moves into *epv*, so we must prove that the constraint for this purse holds (for all other purses in *epv*, the constraint holds beforehand, and so holds afterwards). In *StartTo*, the post-state $pdAuth'.toSeqNo$ is equal to pre-state *nextSeqNo*. Coupling this with constraint B–6 we have

$$\begin{aligned}&\forall\, pd : toLogged \mid pd.to = name? \bullet\\ &\quad pd.toSeqNo < (conAuthPurse'\ pd.to).pdAuth.toSeqNo\end{aligned}$$

Since the logs don't change, we have

$$\begin{aligned}&\forall\, pd : toLogged' \mid pd.to = name? \bullet\\ &\quad pd.toSeqNo < (conAuthPurse'\ pd.to).pdAuth.toSeqNo\end{aligned}$$

which proves the constraint for purse *name?*.

B–9 *epr* $\Rightarrow \neg$ *val* $\wedge \neg$ *ack*. No purses newly in *epr*; no new *val*s or *ack*s.

B–10 *req* $\wedge \neg$ *ack* $\Leftrightarrow$ *toInEpv* $\vee$ *toLogged*. We claim that there is a new *req* for which there is no *ack* in the ether, and the purse moves into *epv*. As a result, we prove the consequent for each implication direction.

- case $\Rightarrow$:
  We must prove *toInEpv* $\vee$ *toLogged*. The purse moves into *epv*, thus establishing the consequent.
- case $\Leftarrow$:
  The purse moves into *epv*, so we must show that there is a *req*, but no *ack*, for the purse's $pdAuth'$. From *StartTo*, we have $m! = req\ pdAuth'$, so the *req* is in the ether. It is then necessary to show there is no *ack* before. The *pd* we are considering is given by

  $$pd == (conAuthPurse'\ name?).pdAuth$$

  Noting that $pd.to = name?$, the definition of *StartTo* gives us that

  $$\begin{aligned}&(conAuthPurse\ name?).nextSeqNo\\ &\qquad = (conAuthPurse'\ name?).pdAuth.toSeqNo\\ &\Rightarrow (conAuthPurse\ pd.to).nextSeqNo = pd.toSeqNo\\ &\Rightarrow ack\ pd \notin ether \qquad [BetweenWorld\ \text{constraint B–4}]\end{aligned}$$

  Hence, we have the corresponding *req* but no *ack*.

B–11 *epv* $\wedge$ *val* $\Rightarrow$ *fromInEpa* $\vee$ *fromLogged*. To prove this constraint, we demonstrate that the antecedent is false: the purse moves into *epv*, so we must show that there is no *val* before. The *pd* we are considering is given by

$$pd == (conAuthPurse'\ name?).pdAuth$$

Noting that $pd.to = name?$, the definition of *StartTo* gives us that

$$\begin{aligned}&(conAuthPurse\ name?).nextSeqNo\\ &\qquad = (conAuthPurse'\ name?).pdAuth.toSeqNo\\ &\Rightarrow (conAuthPurse\ pd.to).nextSeqNo = pd.toSeqNo\\ &\Rightarrow val\ pd \notin ether \qquad [BetweenWorld\ \text{constraint B–3}]\end{aligned}$$

Hence, there is no *val* before, and no *val* is emitted by this operation.

B–12 *fromInEpa* $\vee$ *fromLogged* $\Rightarrow$ *req*. No purses newly in *epa*; no new logs.

B–13 *toLogged* finite. No new logs.

B–14 Read exception record messages are logged. No new log result messages.

B–15 Cleared logs archived. No new *exceptionLogClear* messages.

B–16 *req* for each log. No new elements added to *fromLogged* or *toLogged*.

■ 29.3

29.4 Correctness of *CReq*

Lemma 'constraint' is not applicable, because a purse moves from *epr* to *epa* and emits a *val* message. Therefore we have to show that our chosen *BetweenWorld′* obeys the constraints.

We can invoke lemma 'logs unchanged', section C.7, because no new *req* messages are produced, no new purse logs are produced, and the *archive* does not change. Therefore, the sets *allLogs*, *fromLogged* and *toLogged* remain unchanged.

B–1 *req* ⇒ authentic *to* purse. No new *req* messages.

B–2 No future *req*s. No new *req* messages.

B–3 No future *val*s. *Req* puts a *val* in the *ether′*. Let *pd* be the pay details of the *val*. Hence,

$$pd == (conAuthPurse\ name?).pdAuth$$
$$m? = req\ pd$$
$$m! = val\ pd$$

To show that the new *val* message upholds this constraint, we have to demonstrate that this is not a future message with respect to purse *name*?:

$$pd.toSeqNo < (conAuthPurse'\ pd.to).nextSeqNo$$
$$pd.fromSeqNo < (conAuthPurse'\ pd.from).nextSeqNo$$

Since *req pd* is in the ether, from B–2 we can then satisfy the requirement for the *to* sequence number. Since the pre-state *status* was *epr*, using purse constraint P–2c we know that

$$pd.fromSeqNo < nextSeqNo$$

Since *Req* does not alter *nextSeqNo*, we thus have

$$pd.fromSeqNo < (conAuthPurse'\ pd.from).nextSeqNo$$

B–4 No future *ack*s. No new *ack* messages.

B–5 No future *from* logs. No new logs.

B–6 No future *to* logs. No new logs.

B–7 *from* in {*epr*, *epa*} ⇒ no future *from* logs. No new logs. The *from* purse moves from *epr* into *epa*. *BetweenWorld* constraint B–7 held on *epr*.

B–8 *to* in {*epv*, *eaTo*} ⇒ no future *to* logs. No new logs; no purses newly in *epv* or *eaTo*.

B–9 *epr* ⇒ ¬ *val* ∧ ¬ *ack*. No purses newly in *epr*; no new *ack*s. We need to show the emitted *val* does not have the same *pd* as the stored *pdAuth* of any purse currently in *epr*. It has the same *pd* as the *pdAuth* stored in the purse from which it was emitted, which moved from *epr* and is now in *epa*. No other purse can also have this *pdAuth*, because *pdAuth* includes the *name* of the purse (*ConPurse* constraint P–2a), and purse names are unique.

B–10 *req* ∧ ¬ *ack* ⇔ *toInEpv* ∨ *toLogged*.

- case ⇒: No new *req* or *ack* messages.
- case ⇐: No purses newly in *epv*; no new logs.

B–11 *epv* ∧ *val* ⇒ *fromInEpa* ∨ *fromLogged*. The *from* purse emits a *val*. It also moves into *epa*, thereby establishing the constraint.

B–12 *fromInEpa* ∨ *fromLogged* ⇒ *req*. The purse moves into *epa*. The operation precondition gives the presence of the required *req*.

B–13 *toLogged* finite. No new logs.

B–14 Read exception record messages are logged. No new log result messages.

B–15 Cleared logs archived. No new *exceptionLogClear* messages.

B–16 *req* for each log. No new elements added to *fromLogged* or *toLogged*.

■ 29.4

29.5 Correctness of *CVal*

Lemma 'constraint' is not applicable, because a purse moves from *epv* to *eaPayee* and emits an *ack* message. Therefore we have to show that our chosen *BetweenWorld′* obeys the constraints.

We can invoke lemma 'logs unchanged', section C.7, because no new *req* messages are produced, no new purse logs are produced, and the *archive* does not change. Therefore, the sets *allLogs*, *fromLogged* and *toLogged* remain unchanged.

B–1 *req* ⇒ authentic *to* purse. No new *req* messages.

B–2 No future *req*s. *Val* emits no new *req* messages.

B–3 No future *val*s. *Val* emits no new *val* messages.

B–4 No future *ack*s. *ValOkay* puts an *ack* in the *ether*′, but it has the same *pd* as the *val* read from the *ether*, which obeys *BetweenWorld* constraint B–3. So the *ack*'s *pd* obeys the constraint.

B–5 No future *from* logs. No new logs.

B–6 No future *to* logs. No new logs.

B–7 *from* in $\{epr, epa\} \Rightarrow$ no future *from* logs. No new logs; no purses newly in *epr* or *epa*.

B–8 *to* in $\{epv, eaTo\} \Rightarrow$ no future *to* logs. No new logs.
The *to* purse moves from *epv* into *eaTo*. *BetweenWorld* constraint B–8 held on *epv*.

B–9 $epr \Rightarrow \neg\, val \wedge \neg\, ack$. No purses newly in *epr*.
We need to show the emitted *ack* does not have the same *pd* as any purse currently in *epr*. It has the same *pd* as the *val* message, and so *BetweenWorld* constraint B–9 on *val* gives us the required condition.

B–10 $req \wedge \neg\, ack \Leftrightarrow toInEpv \vee toLogged$.

- case $\Rightarrow$: *ValOkay* emits an *ack*, making the antecedent false.
- case $\Leftarrow$: From lemma 'notLoggedAndIn', section C.12, the purse cannot be in *toLogged*. *ValOkay* moves the purse out of *epv* without logging, making the antecedent false.

B–11 $epv \wedge val \Rightarrow fromInEpa \vee fromLogged$. No purses newly in *epv*; no new *val* messages; no purses leaving *epa*, no changing logs.

B–12 $fromInEpa \vee fromLogged \Rightarrow req$. No purses newly in *epa*; no new logs.

B–13 *toLogged* finite. No new logs.

B–14 Read exception record messages are logged. No new log result messages.

B–15 Cleared logs archived. No new *exceptionLogClear* messages.

B–16 *req* for each log. No new elements added to *fromLogged* or *toLogged*.

■ 29.5

29.6 Correctness of *CAck*

Lemma 'constraint' is not applicable, because a purse moves from *epa* to *eaPayer*. Therefore we have to show that our chosen *BetweenWorld*′ obeys the constraints.

It emits a $\perp$ message. We can invoke lemma 'logs unchanged', section C.7, because no new *req* messages are produced, no new purse logs are produced, and the *archive* does not change. Therefore, the sets *allLogs*, *fromLogged* and *toLogged* remain unchanged.

B–1 $req \Rightarrow$ authentic *to* purse. No new *req* messages.

B–2 No future *req*s. No new *req* messages.

B–3 No future *val*s. No new *val* messages.

B–4 No future *ack*s. No new *ack* messages.

B–5 No future *from* logs. No new logs.

B–6 No future *to* logs. No new logs.

B–7 *from* in $\{epr, epa\} \Rightarrow$ no future *from* logs. No purses newly in *epr* or *epa*.

B–8 *to* in $\{epv, eaTo\} \Rightarrow$ no future *to* logs. No purses newly in *epv* or *eaTo*.

B–9 $epr \Rightarrow \neg\, val \wedge \neg\, ack$. No purses newly in *epr*; no new *val*s or *ack*s.

B–10 $req \wedge \neg\, ack \Leftrightarrow toInEpv \vee toLogged$.

- case $\Rightarrow$: No new *req*s; no new *ack*s; no purses moving out of *epv*, no logs lost.
- case $\Leftarrow$: No purses newly in *epv*; no new logs.

B–11 $epv \wedge val \Rightarrow fromInEpa \vee fromLogged$. No purses newly in *epv*; no new *val*s.
The purse moves out of *epa* without logging, so we need to show that the antecedent is false for this purse. It is sufficient to show the antecedent is false before the operation (since the operation does not change it). There is an *ack* message, *AckOkay*'s input, so *BetweenWorld* constraint B–10 gives us $pd \notin toInEpv$.

B–12 $fromInEpa \vee fromLogged \Rightarrow req$. No purses newly in *epa*; no new logs.

B–13 *toLogged* finite. No new logs.

B–14 Read exception record messages are logged. No new log result messages.

B–15 Cleared logs archived. No new *exceptionLogClear* messages.

B–16 *req* for each log. No new elements added to *fromLogged* or *toLogged*.

■ 29.6

29.7 Correctness of *CReadExceptionLog*

ReadExceptionLogOkay is composed of *AbortPurse* followed by *ReadExceptionLogEafromPurseOkay* at the unpromoted level. As a result, we can apply lemma 'abort forward' (section C.8), leaving us to prove the correctness of *ReadExceptionLogEafromPurseOkay*.

This operation does not change any purse, but it does emit an *exceptionLogResult* message. As a result, lemma 'constraint' is not applicable.

We can invoke lemma 'logs unchanged', section C.7, because no new *req* messages are produced, no new purse logs are produced, and the *archive* does not change. Therefore, the sets *allLogs*, *fromLogged* and *toLogged* remain unchanged.

B–1 *req* $\Rightarrow$ authentic *to* purse. No new *req* messages.

B–2 No future *req*s. No new *req* messages.

B–3 No future *val*s. No new *val* messages.

B–4 No future *ack*s. No new *ack* messages.

B–5 No future *from* logs. No new logs.

B–6 No future *to* logs. No new logs.

B–7 *from* in $\{epr, epa\}$ $\Rightarrow$ no future *from* logs. No purses newly in *epr* or *epa*.

B–8 *to* in $\{epv, eaTo\}$ $\Rightarrow$ no future *to* logs. No purses newly in *epv* or *eaTo*.

B–9 *epr* $\Rightarrow \neg\, val \wedge \neg\, ack$. No purses newly in *epr*; no new *val*s or *ack*s.

B–10 $req \wedge \neg\, ack \Leftrightarrow toInEpv \vee toLogged$.

- case $\Rightarrow$: No new *req*s; no new *ack*s; no purses moving out of *epv*, no logs lost.
- case $\Leftarrow$: No purses newly in *epv*; no new logs.

B–11 $epv \wedge val \Rightarrow fromInEpa \vee fromLogged$. No purses newly in *epv*; no new *val*s; no purse moves out of *epa*; no logs lost.

B–12 $fromInEpa \vee fromLogged \Rightarrow req$. No purses newly in *epa*; no new logs.

B–13 *toLogged* finite. No new logs.

B–14 Read exception record messages are logged. There may be a new *exceptionLogResult* message. If this is so, then we must show that this refers to a stored exception log record. From *ReadExceptionLogPurseEafromOkay*, we have

$$m! \in \{\perp\} \cup \{ld : exLog' \bullet exceptionLogResult(name, ld)\}$$

Hence, if there is an *exceptionLogResult* message, it refers to an exception record which is in the log of purse *name*?, and so is in *allLogs*′. This upholds the constraint.

B–15 Cleared logs archived. No new *exceptionLogClear* messages.

B–16 *req* for each log. No new elements added to *fromLogged* or *toLogged*.

■ 29.7

29.8 Correctness of *CClearExceptionLog*

ClearExceptionLogOkay is composed of *AbortPurse* followed by *ClearExceptionLogEafromPurseOkay* at the unpromoted level. As a result, we can apply lemma 'abort forward' (section C.8), leaving us to prove the correctness of *ClearExceptionLogEafromPurseOkay*.

The operation changes only one purse, and emits a $\perp$ message. The only change to the purse is that its exception log is cleared. However, we have the pre-condition that the input message matches the the exception log (*exLog*). The input message comes from the ether, and hence from constraint B–15 we know that the purse's exception log must have already been recorded in the archive. In this way, clearing the purse's log does not affect *allLogs*. So lemma 'constraint' (section C.6) is applicable.

■ 29.8

29.9 Correctness of *CAuthoriseExLogClear*

Lemma 'constraint' is not applicable, because an *exceptionLogClear* message is emitted to the ether. So, we must show that the constraints hold afterwards.

No purses are changed.

We can invoke lemma 'logs unchanged', section C.7, because no new *req* messages are produced, no new purse logs are produced, and the *archive* does not change. Therefore, the sets *allLogs*, *fromLogged* and *toLogged* remain unchanged.

B–1 *req* $\Rightarrow$ authentic *to* purse. No new *req* messages.

B–2 No future *req*s. No new *req* messages.

B–3 No future *val*s. No new *val* messages.

B–4 No future *ack*s. No new *ack* messages.

B–5 No future *from* logs. No new logs.

B–6 No future *to* logs. No new logs.

B–7 *from* in $\{epr, epa\} \Rightarrow$ no future *from* logs. No purses newly in *epr* or *epa*.

B–8 *to* in $\{epv, eaTo\} \Rightarrow$ no future *to* logs. No purses newly in *epv* or *eaTo*.

B–9 $epr \Rightarrow \neg\ val \wedge \neg\ ack$. No purses newly in *epr*; no new *val*s or *ack*s.

B–10 $req \wedge \neg\ ack \Leftrightarrow toInEpv \vee toLogged$.

- case $\Rightarrow$: No new *req*s; no new *ack*s; no purses moving out of *epv*; no logs lost.
- case $\Leftarrow$: No purses newly in *epv*; no new logs.

B–11 $epv \wedge val \Rightarrow fromInEpa \vee fromLogged$. No purses newly in *epv*; no new *val*s; no purse moves out of *epa*; no logs lost.

B–12 $fromInEpa \vee fromLogged \Rightarrow req$. No purses newly in *epa*; no new logs.

B–13 *toLogged* finite. No new logs.

B–14 Read exception record messages are logged. No new exception log read messages.

B–15 Cleared logs archived. There is a new *exceptionLogClear* message. However, the operation contains the pre-condition that the log records for which the message is generated must be in the archive. Hence, the constraint is upheld.

B–16 *req* for each log. No new elements added to *fromLogged* or *toLogged*.

29.10 Correctness of *CArchive*

This operation archives the contents of some of the *exceptionLogResult* messages in the ether. It does not change any purse, or change the ether.

From B–14, we know that those exception records referred to by the *exceptionLogResult* messages are already in *allLogs*. As a result, adding them to *archive* does not change *allLogs*. This operation does not change any purse, and does not emit a payment details message. So lemma 'constraint' is applicable.

■ 29.10

■ 29

Chapter 30

Summary

The proofs presented in this report constitute a proof that the architectural design given by the *C* model is *secure* with respect to the security properties as described in the Formal Security Policy Model (the $\mathcal{A}$ model) and the Security Properties.

We have presented the proofs in a logical sequence, but even so, it can be hard to be sure that no steps have been missed. The following table gives a hierarchical view of the proof, showing at each level how a proof goal is satisfied by a number of subgoals. Each line in the table is one proof goal, together with a section reference for where that proof goal is addressed.

If the proof goal has child goals (goals one level of indent deeper) then the section reference explains how it is that the goal can be satisfied by its collection of subgoals. For example, goal 1.4 (AbTransfer upholds properties) is proved by proving three subgoals: 1.4.1 (SP 1), 1.4.2 (SP 2.1) and 1.4.3 (SP 6.2). The reference for goal 1.4 is to section 2.4, where it is argued that we have only to prove the three SPs 1, 2.1 and 6.2 because all other SPs can be proved trivially.

If a goal has no further subgoals, its section reference is the proof of this goal directly.

It can be seen that all proof goals have section references, and all steps have been addressed.

System secure	by definition
1. Abstract preserves security properties	by definition
1.1. AbIgnore upholds properties	2.4
1.2. AbTransfer upholds properties	2.4
1.2.1. SP 1	2.4
1.2.1.1. Okay	2.4.1
1.2.1.2. Lost	2.4.3
1.2.2. SP 2.1	2.4
1.2.2.1. Okay	2.4.2
1.2.2.2. Lost	2.4.4
2. Concrete preserves security properties	by definition
2.1. Each concrete operation upholds properties	2.4
3. Abstract operations are total	8.2.2
4. A is refined by B	by definition
4.1. Init	by definition
4.1.1. state initialisation	11.2
4.1.2. input initialisation	11.3
4.2. Applicability	9.2.3
4.2.1. pre AOp = true	8.2.2
4.2.2. simpler applicability	by definition
4.2.2.1. pre BOp = true	8.3.2
4.3. Correctness	9.2.4
4.3.1. pre AOp = true	8.2.2
4.3.2. simpler correctness	by definition
4.3.2.1. AbTransfer	9 and 14.3
4.3.2.1.1. Ignore	14.7
4.3.2.1.2. Okay and Lost	C.1
4.3.2.1.2.1. exists-pd	18.4
4.3.2.1.2.2. exists-chosenLost	18.5
4.3.2.1.2.3. check-operation	18.6
4.3.2.2. AbIgnore	9 and 14.2
4.3.2.2.1. StartFrom	14.3
4.3.2.2.1.1. Ignore	14.7
4.3.2.2.1.2. Abort	14.8
4.3.2.2.1.3. Okay	C.5
4.3.2.2.1.3.1. Abort	14.8

4.3.2.2.1.3.2. EaPayer operation	C.1
4.3.2.2.1.3.2.1. exists-pd	16.4
4.3.2.2.1.3.2.2. exists-chosenLost	16.5
4.3.2.2.1.3.2.3. check-operation	C.3
4.3.2.2.1.3.2.3.1. check-operation-ignore	16.6
4.3.2.2.2. StartTo	14.3
4.3.2.2.2.1. Ignore	14.7
4.3.2.2.2.2. Abort	14.8
4.3.2.2.2.3. Okay	C.5
4.3.2.2.2.3.1. Abort	14.8
4.3.2.2.2.3.2. EaPayer operation	C.1
4.3.2.2.2.3.2.1. exists-pd	17.4
4.3.2.2.2.3.2.2. exists-chosenLost	17.5
4.3.2.2.2.3.2.3. check-operation	C.3
4.3.2.2.2.3.2.3.1. check-operation-ignore	17.6
4.3.2.2.3. Val	14.3
4.3.2.2.3.1. Ignore	14.7
4.3.2.2.3.2. Okay	C.1 and 19.2
4.3.2.2.3.2.1. exists-pd	19.3
4.3.2.2.3.2.2 exists-chosenLost	19.4
4.3.2.2.3.2.3. check-operation	C.3
4.3.2.2.3.2.3.1. check-operation-ignore	19.5 and on
4.3.2.2.4. Ack	14.3
4.3.2.2.4.1. Ignore	14.7
4.3.2.2.4.2. Okay	C.1 and 20.2
4.3.2.2.4.2.1. exists-pd	20.3
4.3.2.2.4.2.2 exists-chosenLost	20.4
4.3.2.2.4.2.3. check-operation	C.3
4.3.2.2.4.2.3.1. check-operation-ignore	20.5 and on

4.3.2.2.5. ReadExceptionLog	14.3
4.3.2.2.5.1. Ignore	14.7
4.3.2.2.5.2. Okay	C.5
4.3.2.2.5.2.1. Abort	14.8
4.3.2.2.5.2.2. EaPayer operation	C.1 and 21
4.3.2.2.5.2.2.1. lemma lost unchanged	C.2
4.3.2.2.5.2.2.2. check-operation	C.3
4.3.2.2.5.2.2.2.1. check-operation-ignore	21.3
4.3.2.2.6. ClearExceptionLog	14.3
4.3.2.2.6.1. Ignore	14.7
4.3.2.2.6.2. Abort	14.8
4.3.2.2.6.3. Okay	C.5
4.3.2.2.6.3.1. Abort	14.8
4.3.2.2.6.3.2. EaPayer operation	C.1 and 22
4.3.2.2.6.3.2.1. lemma lost unchanged	C.2
4.3.2.2.6.3.2.2. check-operation	C.3
4.3.2.2.6.3.2.2.1. check-operation-ignore	22.3
4.3.2.2.7. AuthoriseExLogClear	14.3
4.3.2.2.7.1. Ignore	14.7
4.3.2.2.7.2. Okay	23.2
4.3.2.2.8. Archive	24.2
4.3.2.2.9. Ignore	14.7
4.3.2.2.10. Increase	15.3
4.3.2.2.11. Abort	14.8
4.4. Finalisation	by definition
4.4.1. output finalisation	12.2
4.4.2. state finalisation	12.3
5. B is refined by C	established rules 25.2
5.1. Init	27.1
5.2. Applicability	27.3
5.2.1. pre COp = true	8.4.2
5.3. Correctness	25.2.5
5.3.1. Simpler correctness	25
5.3.1.1. StartTo is refined	28.1
5.3.1.1.1. Okay branch	29.3 and C.10
5.3.1.1.1.1. Eafrom branch	29.3
5.3.1.1.1.2. Abort branch	28.5

5.3.1.1.2. CIgnore branch	28.2
5.3.1.1.3. CAbort branch	28.5
5.3.1.2. StartFrom is refined	28.1
5.3.1.2.1. Okay branch	29.2 and C.10
5.3.1.2.1.1. Eafrom branch	29.2
5.3.1.2.1.2. Abort branch	28.5
5.3.1.2.2. CIgnore branch	28.2
5.3.1.2.3. CAbort branch	28.5
5.3.1.3. Req is refined	28.1
5.3.1.3.1. Okay branch	29.4
5.3.1.3.2. CIgnore branch	28.2
5.3.1.4. Val is refined	28.1
5.3.1.4.1. Okay branch	29.5
5.3.1.4.2. CIgnore branch	28.2
5.3.1.5. Ack is refined	28.1
5.3.1.5.1. Okay branch	29.6
5.3.1.5.2. CIgnore branch	28.2
5.3.1.6. ReadExceptionLog is refined	28.1
5.3.1.6.1. Okay branch	29.7 and C.10
5.3.1.6.1.1. Eafrom branch	29.7
5.3.1.6.1.2. Abort branch	28.5
5.3.1.6.2. CIgnore branch	28.2
5.3.1.7. ClearExceptionLog is refined	28.1
5.3.1.7.1. Okay branch	29.8 and C.10
5.3.1.7.1.1. Eafrom branch	29.8
5.3.1.7.1.2. Abort branch	28.5
5.3.1.7.2. CIgnore branch	28.2
5.3.1.7.3. CAbort branch	28.5
5.3.1.8. AuthoriseExLogClear is refined	28.1
5.3.1.8.1. Okay branch	29.9
5.3.1.8.2. CIgnore branch	28.2
5.3.1.9. Archive is refined	29.10
5.3.2. Totality of BOp	8.3.2
5.4. Finalisation	27.2

Part IV

Appendices

Appendix A

Proof Layout

A.1 Notation

The notation

$$Abs \sqsubseteq Conc$$

says the the *Abs* operation is refined by the *Conc* operation.

In order to prove that *Abs* is indeed validly refined by *Conc*, we need to prove various 'correctness conditions', expressed as theorems (section 9).

That the predicate

$$\forall D \mid P \bullet Q$$

is always true is expressed as the theorem

$$\vdash \forall D \mid P \bullet Q$$

which is equivalent to

$$D \mid P \vdash Q$$

This can be read as a theorem that states that, under hypothesis $D \mid P$ (declarations D constrained by predicates P), consequent Q (a predicate) has been proved to hold. $D \mid P$ is usually written as a schema text, and Q may be written using a schema as predicate.

A.2 Labelling proof steps

In labelling various steps of the proofs below, we use the following notation.

- [defn P]: from the definition of the schema predicate P
- [hyp]: from the hypothesis of the theorem
- [prop x]: from a property of the Z operator x
- [*name*]: use of inference rule *name*

Appendix B

Inference rules

The proofs presented are rigorous, but informal, in that they have not been checked by a machine proof-checker.

We present below the sort of inference rules we have used. Such explicit use of inference rules improves the readability of the proofs by showing exactly what steps of mathematical reasoning are being made. These inference rules are not intended as a definition of the logic being used, but as guidance about the reasoning steps.

The inference rule

$$\frac{P1 \quad P2 \quad \ldots \quad Pn}{C} \quad [\ \mathit{rulename}\ ]$$

says that conclusion C can be inferred if every premiss Pi can be proved. (The rule name is used for labelling proof steps.)

The inference rule

$$\frac{P1, P2, \ldots, Pn}{C} \quad [\ \mathit{rulename}\ ]$$

says that conclusion C can be inferred if any premiss Pi can be proved.

B.1 Universal quantifier becomes hypothesis

$$\frac{S \vdash P}{\vdash \forall S \bullet P} \quad [\ \mathit{uni\ hyp}\ ]$$

B.2 Disjunction in the hypothesis

Given an hypothesis containing a disjunct, it is sufficient to prove the theorem for each case.

$$\frac{R \vdash P \quad S \vdash P}{R \lor S \vdash P} \quad [\ \mathit{hyp\ disj}\ ]$$

B.3 Disjunction in the consequent

Given a consequent containing a disjunct, it is sufficient to prove the theorem for only one case (since this is a harder thing to prove).

$$\frac{R \vdash P, R \vdash Q}{R \vdash P \lor Q} \quad [\ \mathit{consq\ disj}\ ]$$

B.4 Conjunction in the consequent

Given a consequent containing a conjunct, it is sufficient to prove the theorem for each case separately.

$$\frac{R \vdash P \quad R \vdash Q}{R \vdash P \land Q} \quad [\ \mathit{consq\ conj}\ ]$$

We can add conjuncts to the consequent (since this is a harder thing to prove).

$$\frac{R \vdash P \land Q}{R \vdash P} \quad [\ \mathit{strengthen\ consq}\ ]$$

B.5 Cut for lemmas

Cut is a way to introduce new hypotheses, and discharge them as lemmas.

$$\frac{R;\ D \mid Q \vdash P \quad R \vdash \exists D \bullet Q}{R \vdash P} \quad [\ \mathit{cut}\ ]$$

B.6 Thin

We can remove assumptions.

$$\frac{\vdash R}{P \vdash R} \quad [\ \mathit{thin}\ ]$$

B.7 Universal Quantification

Universals can be replaced by a particular choice in the hypothesis

$$\frac{x_1 \in X \Rightarrow P(x_1) \vdash R}{\forall x : X \bullet P(x) \vdash R} \quad [\ \mathit{hyp\ uni}\ ]$$

B.8 Negation

In order to prove something, you can assume its negation.

$$\frac{\neg P \vdash}{\vdash P} \quad [\ \mathit{negation}\ ]$$

B.9 Contradiction

If R can be proved, assuming its negation allows you to prove anything (because *false* $\Rightarrow$ *anything*).

$$\frac{\vdash R}{\neg R \vdash \mathit{anything}} \quad [\ \mathit{contradiction}\ ]$$

B.10 One Point Rule

In order to prove there exists a value with a property, it is enough to exhibit such a value.

$$\frac{\vdash P[t/x]}{\vdash \exists x \bullet P \land x = t} \quad [\ \mathit{one\ point}\ ]$$

provided x is not free in t.

B.11 Derived Rules

We find it useful to derive some compound rules. These make the proofs in the body of the document easier to follow, and can themselves be proved from the inference rules above.

B.11.1 One point cut

$$\frac{P \vdash Q}{P \vdash \exists P \bullet Q} \quad [\ \textit{consq exists}\ ]$$

and very similarly

$$\frac{P \vdash Q}{P \vdash (\exists P) \land Q} \quad [\ \textit{consq exists}\ ]$$

B.11.2 Existential in the hypothesis

$$\frac{x : X;\ D \mid P \vdash}{D \mid \exists x : X \bullet P \vdash} \quad [\ \textit{hyp exists}\ ]$$

B.12 Proof of the Derived Rules

We derive each of the derived rules above from the main inference rules.

B.12.1 Derivation of One point cut

We can derive the first one-point cut rule ([*consq exists*]) as follows. First, we expand P into a declaration D and a predicate p.

$D \mid p \vdash \exists D \bullet p \land q$	[starting point]
$D \mid p \vdash \exists D' \bullet p[D'/D] \land q[D'/D]$	[rename bound declaration]
$D \mid p \vdash \exists D' \bullet p[D'/D] \land q[D'/D] \land D' = D$	[strengthen consequent]
$D \mid p \vdash p[D'/D][D/D'] \land q[D'/D][D/D']$	[one point rule]
$D \mid p \vdash p \land q$	[simplify renaming]
$D \mid p \vdash q$	[discharge p from hyp]

The second onepoint-cut rule follows exactly the same way, except that q is not bound by the existential, and so none of the renamings alters it.

B.12.2 Derivation of existential in the hypothesis

$D \mid (\exists x : X \bullet P) \vdash$	[starting point]
$D;\ x : X \mid P \land (\exists x : X \bullet P) \vdash \quad D \mid (\exists x : X \bullet P) \vdash \exists x : X \bullet P$	[cut in $x : X \mid P$]
$D;\ x : X \mid P \land (\exists x : X \bullet P) \vdash$	[discharge side lemma from hyp]
$D;\ x : X \mid P \vdash$	[*thin*]

as required.

Appendix C

Lemmas and their proofs

C.1 Lemma 'deterministic'

Lemma 1 (deterministic) The correctness proof for a general *Okay* branch consists of the following three proof obligations: [1]
exists-pd:

$$
\begin{array}{l}
\Phi BOp;\ BOpPurseOkay;\ RabOut;\ RabCl';\ RabIn \\
\vdash \\
\exists\, pdThis : PayDetails \bullet \mathcal{P}
\end{array}
$$

exists-chosenLost:

$$
\begin{array}{l}
\Phi BOp;\ BOpPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\ RabIn \mid \\
\quad \mathcal{P} \\
\vdash \\
\exists\, chosenLost : \mathbb{P}\, PayDetails \bullet \mathcal{Q} \land chosenLost \subseteq maybeLost
\end{array}
$$

check-operation:

$$
\begin{array}{l}
\Phi BOp;\ BOpPurseOkay;\ RabOut;\ RabClPd'[pdThis/pdThis'];\\
\qquad AbWorld;\ RabClPd;\ RabIn \mid \\
\quad \mathcal{P} \land \mathcal{Q} \\
\vdash \\
AOp
\end{array}
$$

■

[1] Used in: lemma '*AbIgnore*', section 14.6; lemma '*Ignore*', section 14.7; lemma '*Abort* refines *AbIgnore*', section 14.8; used to simplify every $\mathcal{A}$-$\mathcal{B}$ operation proof.

Proof:
See section 14.4.5.
■ C.1

C.2 Lemma 'lost unchanged'

Lemma 2 (lost unchanged) For $BOp\Xi Lost$ operations, where we have that $maybeLost' = maybeLost$ and $definitelyLost' = definitelyLost$, the proof obligations **exists-pd** and **exists-chosenLost** are satisfied automatically by the instantiation of the predicates $\mathcal{P}$ and $\mathcal{Q}$ as: [2]

$$\mathcal{P} \Leftrightarrow true$$

$$\mathcal{Q} \Leftrightarrow chosenLost = chosenLost'$$

■

Proof:
See section 14.5
■ C.2

C.3 Lemma '*AbIgnore*'

Consider an operation *BOpIg* which refines *AbIgnore*. The operation should have the following properties.

- *BOpIg* is a promoted operation, and thus alters only one concrete purse.
- for any purse, the *name* is unchanged.
- the domain of *conAuthPurse* is unchanged (by construction of the promotion)
- for any purse, either *nextSeqNo* is unchanged, or increased.

Where these properties hold for *BOpIg*, we can apply lemma *AbIgnore*.

Lemma 3 (*AbIgnore*) For a *BOpIg* operation, the **check-operation** proof obligation reduces to [3]

$$\begin{array}{l}
\Phi BOp;\ BOpIgPurse;\ RabClPd'[pdThis/pdThis'];\ AbWorld;\ RabClPd \mid \\
\quad \mathcal{P} \land \mathcal{Q} \\
\vdash \\
\forall n : \mathrm{dom}\, abAuthPurse \bullet \\
\quad (abAuthPurse'\ n).lost = (abAuthPurse\ n).lost \\
\quad \land (abAuthPurse'\ n).balance = (abAuthPurse\ n).balance
\end{array}$$

■

Proof:
See section 14.6.
■ C.3

[2]Used in *ExceptionLogEnquiry*, chapter 21; *ExceptionLogClear*, chapter 22.

C.4 Lemma '*Abort* refines *AbIgnore*'

Lemma 4 (*Abort* refines *AbIgnore*) Concrete *Abort* refines abstract *AbIgnore*.[4]

$$Abort;\ Rab';\ RabOut \vdash \exists\, AbWorld;\ a? : AIN \bullet Rab \land RabIn \land AbIgnore$$

■

Proof:
See section 14.8.
■ C.4

C.5 Lemma 'abort backward'

Lemma 5 (abort backward) Where a concrete operation is written as a composition of *AbortPurseOkay* and a simpler operation starting from *eaFrom*, it is sufficient to prove that the promotion of the simpler operation alone refines

[3]Used in: '*Ignore*', section 14.7; lemma '*Abort* refines *AbIgnore*', section 14.8; used to simplify every $\mathcal{A}$-$\mathcal{B}$ operation proof that refines *AbIgnore*.
[4]Used in: lemma 'abort backward', section C.5

the relevant abstract operation. [5]

$$
\begin{aligned}
&(\exists \Delta ConPurse \bullet \Phi BOp \land (AbortPurseOkay \fatsemi BOpPurseEafromOkay)); \\
&\qquad Rab'; \; RabOut; \\
&\qquad (\forall BOpEafromOkay; \; Rab'; \; RabOut \bullet \\
&\qquad\quad \exists AbWorld; \; a? : AIN \bullet Rab \land RabIn \land AOp) \\
&\vdash \\
&\exists AbWorld; \; a? : AIN \bullet Rab \land RabIn \land AOp
\end{aligned}
$$

■

Proof:
See section 14.9.
■ C.5

C.6 Lemma 'constraint'

Lemma 6 (constraint) If an operation does not change purse status and does not change the presence of payment detail messages in the ether (either by not emitting such a message, or by emitting an already existing message), then it preserves the *BetweenWorld* constraints. [6] ■

Proof:
See section 28.3.4.
■ C.6

C.7 Lemma 'logs unchanged'

Lemma 7 (logs unchanged) When the *archive* and the individual purse logs do not change, and when no new *req* messages are added to the *ether*, the set of

[5] Used in: *StartFrom*, section 16; *StartTo*, section 17; *ClearExceptionLog*, section 22; *ReadExceptionLog*, section 21

[6] Used in: *Increase*, section 28.4; *CClearExceptionLog*, section 29.8; *CArchive*, section 29.10.

PayDetails representing all the logs does not change either. [7]

$$
\begin{aligned}
&BOpOkay \mid archive' = archive \\
&\quad \land (\operatorname{ran} req) \cap ether' = (\operatorname{ran} req) \cap ether \bullet \\
&\quad \land \forall n : \operatorname{dom} conAuthPurse \bullet \\
&\qquad (conAuthPurse' \, n).exLog = (conAuthPurse \, n).exLog \\
&\vdash \\
&allLogs' = allLogs \\
&\land toLogged' = toLogged \\
&\land fromLogged' = fromLogged
\end{aligned}
$$

■

Proof:
See section 28.6.
■ C.7

C.8 Lemma 'abort forward'

Lemma 8 (abort forward) Where a C operation is written as a composition of *CAbort* and a simpler operation starting from *eaFrom*, and the corresponding $\mathcal{B}$ operation is structured similarly, it is sufficient to prove that the simpler C operation refines corresponding $\mathcal{B}$ operation [8].

$$
\begin{aligned}
&(CAbort \fatsemi COpEafrom); \; Rbc; \\
&\qquad (\forall COpEafrom; \; Rbc \bullet \exists BetweenWorld' \bullet Rbc' \land BOpEafrom) \\
&\vdash \\
&\exists BetweenWorld' \bullet Rbc' \land (Abort \fatsemi BOpEafrom)
\end{aligned}
$$

■

Proof:
See section 28.7.
■ C.8

[7] Used in: lemma 'constraint', section 28.3.4; *CStartFrom*, section 29.2; *CReq*, section 29.4; *CVal*, section 29.5; *CAck*, section 29.6; *CReadExceptionLog*, section 29.7; *CAuthoriseExLogClear*, section 29.9.

[8] Used in: *CStartFrom*, section 29.2; *CStartTo*, section 29.3; *CReadExceptionLog*, section 29.7; *CClearExceptionLog*, section 29.8.

C.9 Lemma 'compose backward'

Lemma C.1 (compose backward) If, under the backwards refinement rules, a concrete operation COp_1 is a refinement of abstract operation AOp_1, and COp_2 is a refinement of AOp_2, then their composition is a refinement of the abstract composition [9].

$$\begin{array}{l} (COp_1 \fatsemi COp_2);\ R';\ ROut; \\ \qquad (\forall\ COp_1;\ R';\ ROut \bullet (\exists\ A;\ AIn \bullet R \wedge RIn \wedge AOp_1\ )\ ); \\ \qquad (\forall\ COp_2;\ R';\ ROut \bullet (\exists\ A;\ AIn \bullet R \wedge RIn \wedge AOp_2\ )\ ) \\ \vdash \\ \exists\ A;\ AIn \bullet R \wedge RIn \wedge (AOp_1 \fatsemi AOp_2) \end{array}$$

■

Proof:
This result is reasonably self-evident, from the definition of refinement in terms of complete programs. We show that the particular form of the theorem holds here.

Without loss of generality, assume that the concrete and abstract state schemas have a single component, c and a respectively. (A multi-component state is isomorphic to a single component state consisting of all the multi-components bundled into a single schema or Cartesian product.)

Expand the compositions, and rename the quantified variables in the hypothesis.

$$\begin{array}{l} (\exists\ C_0 \bullet COp_1[c_0/c'] \wedge COp_2[c_0/c]\ );\ R';\ ROut; \\ \quad (\forall\ COp_1[c_0/c'];\ R_0;\ ROut \bullet (\exists\ A;\ AIn \bullet R \wedge RIn \wedge AOp_1[a_0/a']\ )\ ); \\ \quad (\forall\ COp_2[c_0/c];\ R';\ ROut \bullet (\exists\ A_0;\ AIn \bullet R_0 \wedge RIn \wedge AOp_2[a_0/a]\ )\ ) \\ \vdash \\ \exists\ A;\ AIn \bullet R \wedge RIn \wedge (\exists\ A_0 \bullet AOp_1[a_0/a'] \wedge AOp_2[a_0/a]\ ) \end{array}$$

Use [*hyp exists*] to drop the $\exists$ in the hypothesis, then simplify.

$$\begin{array}{l} COp_1[c_0/c'];\ COp_2[c_0/c];\ R';\ ROut; \\ \quad (\forall\ COp_1[c_0/c'];\ R_0;\ ROut \bullet \\ \qquad (\exists\ A;\ AIn \bullet R \wedge RIn \wedge AOp_1[a_0/a']\ )\ ); \\ \qquad (\forall\ COp_2[c_0/c];\ R';\ ROut \bullet \\ \qquad\quad (\exists\ A_0;\ AIn \bullet R_0 \wedge RIn \wedge AOp_2[a_0/a]\ )\ ) \\ \vdash \\ \exists\ A;\ AIn \bullet R \wedge RIn \wedge (\exists\ A_0 \bullet AOp_1[a_0/a'] \wedge AOp_2[a_0/a]\ ) \end{array}$$

[9] Used in: lemma 'abort backward', section C.5.

Use $D \wedge (\forall\ D \bullet P) \Rightarrow P$ to simplify the second universal quantifier in the hypothesis.

$$\begin{array}{l} COp_1[c_0/c'];\ COp_2[c_0/c];\ R';\ ROut; \\ \quad (\forall\ COp_1[c_0/c'];\ R_0;\ ROut \bullet \\ \quad (\exists\ A;\ AIn \bullet R \wedge RIn \wedge AOp_1[a_0/a']\ )\ )\ | \\ \quad \exists\ A_0;\ AIn \bullet R_0 \wedge RIn \wedge AOp_2[a_0/a] \\ \vdash \\ \exists\ A;\ AIn \bullet R \wedge RIn \wedge (\exists\ A_0 \bullet AOp_1[a_0/a'] \wedge AOp_2[a_0/a]\ ) \end{array}$$

Use [*hyp exists*] to drop the $\exists$ in the hypothesis, then simplify.

$$\begin{array}{l} COp_1[c_0/c'];\ COp_2[c_0/c];\ R_0;\ R';\ ROut;\ RIn;\ AOp_2[a_0/a]; \\ \quad (\forall\ COp_1[c_0/c'];\ R_0;\ ROut \bullet \\ \qquad (\exists\ A;\ AIn \bullet R \wedge RIn \wedge AOp_1[a_0/a']\ )\ ) \\ \vdash \\ \exists\ A;\ AIn \bullet R \wedge RIn \wedge (\exists\ A_0 \bullet AOp_1[a_0/a'] \wedge AOp_2[a_0/a]\ ) \end{array}$$

Repeat the previous three steps to simplify the remaining quantifier in the hypothesis.

$$\begin{array}{l} COp_1[c_0/c'];\ COp_2[c_0/c];\ R;\ R_0;\ R';\ ROut;\ RIn; \\ AOp_1[a_0/a'];\ AOp_2[a_0/a] \\ \vdash \\ \exists\ A;\ AIn \bullet R \wedge RIn \wedge (\exists\ A_0 \bullet AOp_1[a_0/a'] \wedge AOp_2[a_0/a]\ ) \end{array}$$

Move the inner $\exists$ in the consequent outwards.

$$\begin{array}{l} COp_1[c_0/c'];\ COp_2[c_0/c];\ R;\ R_0;\ R';\ ROut;\ RIn; \\ AOp_1[a_0/a'];\ AOp_2[a_0/a] \\ \vdash \\ \exists\ A;\ A_0;\ AIn \bullet R \wedge RIn \wedge AOp_1[a_0/a'] \wedge AOp_2[a_0/a] \end{array}$$

All the terms are in the hypothesis.
■ C.9

C.10 Lemma 'compose forward'

Lemma C.2 (compose forward) If, under the forwards refinement rules, concrete operation COp_1 is a refinement of abstract operation AOp_1, and COp_2 is a refinement of AOp_2, then their composition is a refinement of the abstract composition [10].

$$
\begin{array}{l}
(COp_1 \fatsemi COp_2);\ R; \\
\quad\quad (\forall\ COp_1;\ R \bullet (\exists\ A' \bullet R' \land AOp_1)); \\
\quad\quad (\forall\ COp_2;\ R \bullet (\exists\ A' \bullet R' \land AOp_2)) \\
\vdash \\
\exists\ A' \bullet R' \land (AOp_1 \fatsemi AOp_2)
\end{array}
$$

■

Proof:

Follows as for lemma 'compose backward', above.

■ C.10

C.11 Lemma 'promoted composition'

Lemma C.3 (promoted composition) The promotion of the composition of two operations is equal to the composition of the promotions of the two operations [11].

Assume the existence of a local state *Local*, which, without loss of generality we assume has a single variable x; a global state *Global*, with a standard promotion framing schema, Φ

$$
\begin{array}{|l}
\hline
\textit{Local} \\
x : X \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Global} \\
locals : NAME \twoheadrightarrow Local \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\Phi \\
\Delta Global \\
\Delta Local \\
n? : NAME \\
\hline
n? \in \operatorname{dom} locals \\
locals\ n? = \theta Local \\
locals' = locals \oplus \{ n? \mapsto \theta Local' \} \\
\hline
\end{array}
$$

[10]Used in: lemma 'abort forward', section 28.7.
[11]Used in: lemma 'abort backward', section C.5

$$
\begin{array}{l}
\Phi;\ Op_1;\ Op_2 \\
\vdash \\
\exists\ \Delta Local \bullet \Phi \land (Op_1 \fatsemi Op_2) \\
\quad = (\exists\ \Delta Local \bullet \Phi \land Op_1) \fatsemi (\exists\ \Delta Local \bullet \Phi \land Op_2)
\end{array}
$$

■

Proof:

We prove this by expanding the definition of composition as an existential quantification, and then showing that this quantification and the quantification used in the promotion commute.

Expand the composition on the right hand side, and then expand the definition of Φ.

$$
\begin{array}{l}
(\exists\ \Delta Local \bullet \Phi \land Op_1) \fatsemi (\exists\ \Delta Local \bullet \Phi \land Op_2) \\
\quad = \exists\ Global_0 \bullet (\exists\ \Delta Local \bullet \Phi[locals_0 / locals'] \land Op_1) \\
\quad\quad \land (\exists\ \Delta Local \bullet \Phi[locals_0 / locals] \land Op_2) \\
\quad = \exists\ Global_0 \bullet \\
\quad\quad (\exists\ \Delta Local \bullet \\
\quad\quad\quad [locals;\ locals_0 : NAME \twoheadrightarrow Local \mid \\
\quad\quad\quad\quad n? \in \operatorname{dom} locals \\
\quad\quad\quad\quad \land\ locals\ n? = \theta Local \\
\quad\quad\quad\quad \land\ locals_0 = locals \oplus \{ n? \mapsto \theta Local' \}] \\
\quad\quad\quad \land\ Op_1) \\
\quad\quad \land (\exists\ \Delta Local \bullet \\
\quad\quad\quad [locals_0;\ locals' : NAME \twoheadrightarrow Local \mid \\
\quad\quad\quad\quad n? \in \operatorname{dom} locals_0 \\
\quad\quad\quad\quad \land\ locals_0\ n? = \theta Local \\
\quad\quad\quad\quad \land\ locals' = locals_0 \oplus \{ n? \mapsto \theta Local' \}] \\
\quad\quad\quad \land\ Op_2)
\end{array}
$$

Rename the after state in the first operation to $Local_a$ and the before state in the second operation to $Local_b$. Choosing different names makes it easier to combine the schemas across the quantifiers.

$$\begin{array}{l}
= \exists\, Global_0 \bullet \\
\quad (\, \exists\, Local;\ Local_a \bullet \\
\qquad [\, locals;\ locals_0 : NAME \pfun Local \mid \\
\qquad\quad n? \in \operatorname{dom} locals \\
\qquad\quad \land\ locals\ n? = \theta Local \\
\qquad\quad \land\ locals_0 = locals \oplus \{n? \mapsto \theta Local_a\}\,] \\
\qquad \land\ Op_1[x_a / x']\,) \\
\quad \land\ (\, \exists\, Local_b;\ Local' \bullet \\
\qquad [\, locals_0;\ locals' : NAME \pfun Local \mid \\
\qquad\quad n? \in \operatorname{dom} locals_0 \\
\qquad\quad \land\ locals_0\ n? = \theta Local_b \\
\qquad\quad \land\ locals' = locals_0 \oplus \{n? \mapsto \theta Local'\}\,] \\
\qquad \land\ Op_2[x_b / x]\,)
\end{array}$$

Combine all these as a single schema, putting the quantifications into the predicate.

$$\begin{array}{l}
= [\, locals;\ locals' : NAME \pfun Local \mid \\
\quad \exists\, local_0;\ Local;\ Local';\ Local_a;\ Local_b \bullet \\
\qquad n? \in \operatorname{dom} locals \\
\qquad \land\ locals\ n? = \theta Local \\
\qquad \land\ locals_0 = locals \oplus \{n? \mapsto \theta Local_a\} \\
\qquad \land\ n? \in \operatorname{dom} locals_0 \\
\qquad \land\ locals_0\ n? = \theta Local_b \\
\qquad \land\ locals' = locals_0 \oplus \{n? \mapsto \theta Local'\} \\
\qquad \land\ Op_1[x_a / x'] \\
\qquad \land\ Op_2[x_b / x]\,]
\end{array}$$

We can remove the quantification of $local_0$ because we have a full definition of it in terms of other variables. This leaves the following equations relating the remaining variables.

$$\begin{array}{l}
= [\, locals;\ locals' : NAME \pfun Local \mid \\
\quad \exists\, Local;\ Local';\ Local_a;\ Local_b \bullet \\
\qquad n? \in \operatorname{dom} locals \\
\qquad \land\ locals\ n? = \theta Local \\
\qquad \land\ \theta Local_b = \theta Local_a \\
\qquad \land\ locals' = locals \oplus \{n? \mapsto \theta Local'\} \\
\qquad \land\ Op_1[x_a / x'] \\
\qquad \land\ Op_2[x_b / x]\,]
\end{array}$$

Using the equation that $\theta Local_b = \theta Local_a$, rename $Local_a$ and $Local_b$ both to $Local_0$.

$$\begin{array}{l}
= [\, locals;\ locals' : NAME \pfun Local \mid \\
\quad \exists\, Local;\ Local';\ Local_0 \bullet \\
\qquad n? \in \operatorname{dom} locals \\
\qquad \land\ locals\ n? = \theta Local \\
\qquad \land\ locals' = locals \oplus \{n? \mapsto \theta Local'\} \\
\qquad \land\ Op_1[x_0 / x'] \\
\qquad \land\ Op_2[x_0 / x]\,]
\end{array}$$

Redistribute the quantifications

$$\begin{array}{l}
= \exists\, Local;\ Local' \bullet \\
\quad [\, locals;\ locals' : NAME \pfun Local \mid \\
\qquad n? \in \operatorname{dom} locals \\
\qquad \land\ locals\ n? = \theta Local \\
\qquad \land\ locals' = locals \oplus \{n? \mapsto \theta Local'\} \\
\qquad \land\ (\, \exists\, Local_0 \bullet Op_1[x_0 / x'] \land Op_2[x_0 / x]\,)\,]
\end{array}$$

and rewrite in terms of composition

$$= \exists\, Local;\ Local' \bullet \Phi \land (Op_1 \fatsemi Op_2)$$

$$= \exists\, \Delta Local \bullet \Phi \land (Op_1 \fatsemi Op_2)$$

This is the left hand side of the equation, and hence the proof is complete.
■ C.11

C.12 Lemma 'notLoggedAndIn'

Lemma C.4 (notLoggedAndIn) If a purse is engaged in a transaction, it does not have a log for that transaction [12].

$$\begin{array}{l}
BetweenWorld \\
\vdash \\
(fromInEpr \cup fromInEpa) \cap fromLogged = \varnothing \\
\land\ (toInEpv \cup toInEapayee) \cap toLogged = \varnothing
\end{array}$$

■

[12] Used in: *Val*, behaviour of *toLogged*, section 19.6.2; *Ack*, behaviour of *definitelyLost*, section 20.6.5; *CVal*, B–10, section 29.5; lemma 'lost', section C.13; lemma 'not lost before', section C.14.

Proof:
Consider the *to* purse case. We consider the *pd* stored in the *to* purse, so

$$\begin{array}{l} pd \in (toInEpv \cup toInEapayee) \Rightarrow \\ \quad pd.toSeqNo = (conAuthPurse\ pd.to).pdAuth.toSeqNo \end{array}$$

We have, from *BetweenWorld* constraint B–8, that

$$pd \in toLogged \Rightarrow pd.toSeqNo < (conAuthPurse\ pd.to).pdAuth.toSeqNo$$

Hence there can be no *pd* in both sets.

The arguments for the *from* cases follow similarly, from *BetweenWorld* constraint B–7.

■ C.12

C.13 Lemma 'lost'

Lemma C.5 (lost) The sets *definitelyLost* and *maybeLost* are disjoint: a *pd* can never be in both. [13]

$$BetweenWorld \vdash definitelyLost \cap maybeLost = \varnothing$$

■

Proof:

$$\begin{array}{ll} definitelyLost \cap maybeLost & \\ \quad = toLogged \cap (fromLogged \cup fromInEpa) & \\ \qquad \cap (fromInEpa \cup fromLogged) \cap toInEpv & [\text{defn.}] \\ \quad = toLogged \cap toInEpv \cap (fromLogged \cup fromInEpa) & [\text{rearranging}] \\ \quad = \varnothing & [\text{Lemma 'notLoggedAndIn' (section C.12)}] \end{array}$$

■ C.13

[13] Used in: *Req*, case 1, section 18.7.1; *Req*, case 2, section 18.8.1; *Req*, case 3, section 18.9.1.

C.14 Lemma 'not lost before'

Lemma C.6 (not lost before) *pdThis* is not lost before the *Req* operation, although it maybe lost after. [14]

$$\begin{array}{l} \Phi BOp;\ ReqPurseOkay;\ pdThis : PayDetails \mid (req^{\sim} m?) = pdThis \\ \vdash \\ definitelyLost = definitelyLost' \setminus \{pdThis\} \\ \wedge\ maybeLost = maybeLost' \setminus \{pdThis\} \end{array}$$

■

Proof:
From the definition of the way the state changes in *ReqOkay* we can say that the following sets are the same before and afterward:

$$\begin{array}{l} fromLogged = fromLogged' \\ \wedge\ toLogged = toLogged' \\ \wedge\ toInEpv = toInEpv' \end{array}$$

For the set *fromInEpa*, we know from *ReqOkay* that beforehand this *pdThis* was *not* in the set and afterward it *was*. So

$$\begin{array}{l} pdThis \in fromInEpa' \\ \wedge\ fromInEpa = fromInEpa' \setminus \{pdThis\} \end{array}$$

From Lemma 'notLoggedAndIn' (section C.12), we have:

$$pdThis \in fromInEpa' \Rightarrow pdThis \notin fromLogged'$$

Reminding ourselves of the definitions of *definitelyLost* and using the identities above, we have

$$\begin{array}{ll} definitelyLost & \\ \quad = toLogged \cap (fromLogged \cup fromInEpa) & [\text{defn}] \\ \quad = toLogged' \cap (fromLogged' \cup fromInEpa' \setminus \{pdThis\}) & [\text{above}] \\ \quad = toLogged' \cap (fromLogged' \cup fromInEpa') \setminus \{pdThis\} & \\ & [pdThis \notin fromLogged'] \\ \quad = (toLogged' \cap (fromLogged' \cup fromInEpa')) \setminus \{pdThis\} & [\text{Spivey}] \\ \quad = definitelyLost' \setminus \{pdThis\} & [\text{defn}] \end{array}$$

[14] Used in: *Req*, **exists-chosenLost**, section 18.5; *Req*, **check-operation**, section 18.6.

Similarly for *maybeLost*:

$$
\begin{aligned}
& maybeLost \\
& \quad = (fromInEpa \cup fromLogged) \cap toInEpv && \text{[defn]} \\
& \quad = ((fromInEpa' \setminus \{pdThis\}) \cup fromLogged') \cap toInEpv' && \text{[above]} \\
& \quad = ((fromInEpa' \cup fromLogged') \setminus \{pdThis\}) \cap toInEpv' \\
& && [pdThis \notin fromLogged'] \\
& \quad = ((fromInEpa' \cup fromLogged') \cap toInEpv') \setminus \{pdThis\} && \text{[prop } \setminus\text{]} \\
& \quad = maybeLost' \setminus \{pdThis\} && \text{[def]}
\end{aligned}
$$

■ C.14

C.15 Lemma '*AbWorld* unique'

Lemma C.7 (*AbWorld* unique) Given *BetweenWorld* and a choice of which transactions will be lost, there is always exactly one *AbWorld* that retrieves.[15]

$$
\begin{aligned}
& BetweenWorld;\ chosenLost : \mathbb{P}\, PayDetails;\ pdThis : PayDetails \mid \\
& \quad chosenLost \subseteq maybeLost \\
& \vdash \\
& \exists_1\, AbWorld \bullet RabClPd
\end{aligned}
$$

■

Proof:
Each element of *AbWorld* has an explicit equation in *Rab* defining it uniquely in terms of *BeweenWorld* components. The components are entirely independent, and the only constraint that ties any together is that on *chosenLost* and *maybeLost*, which we have directly in the hypothesis.

The constraints required of any *AbWorld* can be shown to hold as follows:

- $abAuthPurse : NAME \twoheadrightarrow\!\!\!\!\!\!\;\shortmid\; AbPurse$
  conAuthPurse is a finite function. From the retrieve *AbstractBetween* the domain of *abAuthPurse* equals the domain of *conAuthPurse*, and so is finite, too.

■ C.15
■ C

[15]Used in: lemma 'deterministic', section 14.4.4.

Appendix D

Auxiliary toolkit definitions

D.1 Total abstract balance

The function *totalAbBalance* returns the total value held in a finite collection of purses.

$$
\begin{array}{|l}
totalAbBalance : (NAME \twoheadrightarrow\!\!\!\!\!\shortmid\; AbPurse) \to \mathbb{N} \\
\hline
totalAbBalance\, \emptyset = 0 \\
\forall\, w : NAME \twoheadrightarrow\!\!\!\!\!\shortmid\; AbPurse;\ n : NAME;\ AbPurse \mid n \notin \operatorname{dom} w \bullet \\
\quad totalAbBalance(\{n \mapsto \theta AbPurse\} \cup w) = \\
\quad\quad balance + totalAbBalance\, w
\end{array}
$$

This recursive definition is valid, because it is finite, and hence bounded.

D.2 Total lost value

The function *totalLost* returns the total value lost by a finite collection of purses.

$$
\begin{array}{|l}
totalLost : (NAME \twoheadrightarrow\!\!\!\!\!\shortmid\; AbPurse) \to \mathbb{N} \\
\hline
totalLost\, \emptyset = 0 \\
\forall\, w : NAME \twoheadrightarrow\!\!\!\!\!\shortmid\; AbPurse;\ n : NAME;\ AbPurse \mid n \notin \operatorname{dom} w \bullet \\
\quad totalLost(\{n \mapsto \theta AbPurse\} \cup w) = lost + totalLost\, w
\end{array}
$$

This recursive definition is valid, because it is finite, and hence bounded.

D.3 Summing values

We define the sum of the values in a set of exception logs, or a set of payment details. This recursive definition is valid, because it is finite, and hence bounded.

$$sumValue : \mathbb{F}\ PayDetails \rightarrow \mathbb{N}$$

$$sumValue\ \emptyset = 0$$

$$\forall\ pds : \mathbb{F}\ PayDetails;\ PayDetails \mid \theta PayDetails \notin pds \bullet$$
$$\quad sumValue(\{\theta PayDetails\} \cup pds) = value + sumValue\ pds$$

Bibliography

[Barden *et al.* 1994]
Rosalind Barden, Susan Stepney, and David Cooper. *Z in Practice.* BCS Practitioners Series. Prentice Hall, 1994.

[Flynn *et al.* 1990]
Mike Flynn, Tim Hoverd, and David Brazier. Formaliser—an interactive support tool for Z. In John E. Nicholls, editor, *Z User Workshop: Proceedings of the 4th Annual Z User Meeting, Oxford 1989*, Workshops in Computing, pages 128–141. Springer Verlag, 1990.

[Spivey 1992a]
J. Michael Spivey. *The fUZZ Manual.* Computer Science Consultancy, 2nd edition, 1992. ftp://ftp.comlab.ox.ac.uk/pub/Zforum/fuzz.

[Spivey 1992b]
J. Michael Spivey. *The Z Notation: a Reference Manual.* Prentice Hall, 2nd edition, 1992.

[Stepney]
Susan Stepney. Formaliser Home Page. http://public.logica.com/˜formaliser/.

[Woodcock & Davies 1996]
Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall, 1996.

Index