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
Contents

1 Introduction 1
  1.1 The application 1
  1.2 Overview of model and proof structure 3
  1.3 Rationale for model structure 5
  1.4 Rationale for proof structure 6
  1.5 Status 7

2 Models 9

2.1 Introduction 11

3 SPs 11
  2.1 Introduction 11
  2.2 Abstract model SPs 11
  2.3 Concrete model SPs 13
  2.4 SPs and the models 13

3 A model 17
  3.1 Introduction 17
  3.2 The abstract state 17
  3.3 Secure operations 18
  3.4 Abstract initial state 21
  3.5 Abstract finalisation 22

4 Purse model 23
  4.1 Overview 23
  4.2 Status 23
16 StartFrom 121
  16.1 Proof obligation 121
  16.2 Instantiating lemma 'deterministic' 122
  16.3 Behaviour of maybeLost and definitelyLost 122
  16.4 exists-pd 123
  16.5 exists-chosenLost 123
  16.6 check-operation 124

17 StartTo 125
  17.1 Proof obligation 125
  17.2 Instantiating lemma 'deterministic' 126
  17.3 Behaviour of maybeLost and definitelyLost 126
  17.4 exists-pd 127
  17.5 exists-chosenLost 127
  17.6 check-operation 128

18 Req 129
  18.1 Proof obligation 129
  18.2 Instantiating lemma 'deterministic' 129
  18.3 Discussion 130
  18.4 exists-pd 131
  18.5 exists-chosenLost 131
  18.6 check-operation 131
  18.7 case 1: ReqOkay and RabOkayClPd' 133
  18.8 case 2: ReqOkay and RabWillBeLostPd' 138
  18.9 case 3: ReqOkay and RabHasBeenLostPd' 142
  18.10 case 4: ReqOkay and RabEndPd' 146

19 Val 149
  19.1 Proof obligation 149
  19.2 Instantiating lemma 'deterministic' 149
  19.3 exists-pd 150
  19.4 exists-chosenLost 150
  19.5 check-operation 150
  19.6 Behaviour of maybeLost and definitelyLost 151
  19.7 Clarifying the hypothesis 153

20 Ack 157
  20.1 Proof obligation 157
  20.2 Instantiating lemma 'deterministic' 157
  20.3 exists-pd 158
  20.4 exists-chosenLost 158
  20.5 check-operation 159
  20.6 Behaviour of maybeLost and definitelyLost 159
  20.7 Finishing proof of check-operation 162

21 ReadExceptionLog 163
  21.1 Proof obligation 163
  21.2 Invoking lemma 'lost unchanged' 164
  21.3 check-operation-ignore 164

22 ClearExceptionLog 165
  22.1 Proof obligation 165
  22.2 Invoking lemma 'Lost unchanged' 166
  22.3 check-operation-ignore 166

23 AuthoriseExLogClear 167
  23.1 Proof obligation 167
  23.2 Proof 167

24 Archive 169
  24.1 Proof obligation 169
  24.2 Proof 169

III Second Refinement: B to C 171

25 B to C rules 173
  25.1 Security of the implementation 173
  25.2 Forwards rules proof obligations 174

26 Rbc 177
  26.1 Retrieve state 177

27 Initialisation, Finalisation, and Applicability 179
  27.1 Initialisation proof 179
  27.2 Finalisation proof 179
  27.3 Applicability proofs 180

28 B to C lemmas 181
  28.1 Specialising the proof rules 181
  28.2 Correctness of C\_ignore 182
28.3 Correctness of a branch of the operation 182
28.4 Correctness of CIncrease 185
28.5 Correctness of CAbort 185
28.6 Lemma ‘logs unchanged’ 187
28.7 Lemma ‘abort forward’: operations that first abort 188

29 Correctness proofs 191
29.1 Introduction 191
29.2 Correctness of CStartFrom 191
29.3 Correctness of CStartTo 193
29.4 Correctness of CReq 196
29.5 Correctness of CVal 197
29.6 Correctness of CAck 198
29.7 Correctness of CReadExceptionLog 200
29.8 Correctness of CClearExceptionLog 201
29.9 Correctness of CAuthoriseExLogClear 201
29.10 Correctness of CArchive 202

30 Summary 203

IV Appendices 209

A Proof Layout 211
A.1 Notation 211
A.2 Labelling proof steps 211

B Inference rules 213
B.1 Universal quantifier becomes hypothesis 213
B.2 Disjunction in the hypothesis 214
B.3 Disjunction in the consequent 214
B.4 Conjunction in the consequent 214
B.5 Cut for lemmas 215
B.6 Thin 215
B.7 Universal Quantification 215
B.8 Negation 215
B.9 Contradiction 216
B.10 One Point Rule 216
B.11 Derived Rules 216
B.12 Proof of the Derived Rules 217

C Lemmas 219
C.1 Lemma ‘deterministic’ 219
C.2 Lemma ‘lost unchanged’ 220
C.3 Lemma ‘AbIgnore’ 220
C.4 Lemma ‘Abort refines AbIgnore’ 221
C.5 Lemma ‘abort backward’ 221
C.6 Lemma ‘constraint’ 222
C.7 Lemma ‘logs unchanged’ 222
C.8 Lemma ‘abort forward’ 223
C.9 Lemma ‘compose backward’ 224
C.10 Lemma ‘compose forward’ 225
C.11 Lemma ‘promoted composition’ 226
C.12 Lemma ‘notLoggedAndIn’ 229
C.13 Lemma ‘lost’ 230
C.14 Lemma ‘not lost before’ 231
C.15 Lemma ‘AbWorld unique’ 232

D Toolkit 235
D.1 Total abstract balance 235
D.2 Total lost value 235
D.3 Summing values 236
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
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 fuzz [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).
1.3. RATIONALE FOR MODEL STRUCTURE

- Concrete model, C: Our final model is the concrete level model, which forms the Formal Architectural Design. This model, C, is structured as a promoted state-and-operations model, very similar to 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 A–B: The security policy is proved to hold for B by proving that B is a refinement of A. This forms the first part of Explanation of Consistency:
  - The retrieve relation Rab, relating the B and A worlds, is defined (Chapter 10).
  - The security policy is shown to hold for B by proof that B refines A, using the "backward" proof rules (Part II). This proof comprises the bulk of the proof work.

- Security proof B–C: The security policy is proved to hold for C by proving that C is a refinement of B (and hence of A, by transitivity of refinement). This forms the remaining part of Explanation of Consistency:
  - The retrieve relation Rbc, relating the C and B worlds, is defined (Chapter 26).
  - The security policy is shown to hold for C by proof that C refines 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.
CHAPTER 1. INTRODUCTION

1.4 Rationale for proof structure

Imagine two specifications \( A \) 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 \( A \). In this sense, \( A \) can simulate any behaviour of \( C \). If this is the case, then we say that \( C \) is a refinement of \( A \).

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, focusing 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 \( A \) and \( C \) only in terms of inputs and outputs, nondeterminism present in \( A \) 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 10] 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 \( A \) 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 \( B \) model, which is a constrained promotion, and hence can contain the predicates needed for the backward proofs. We then prove a refinement from \( A \) to \( B \) using the backward rules. But now the constrained promotion \( B \) 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 \( B \) to \( C \).

1.5 Status

The specification and theorems have been parsed and type-checked using fuzz [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 fuzz’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.$^1$

\[
\begin{align*}
\Delta_{\text{AbWorld}} \triangleq & \text{totalAbBalance}_{\text{ab/AuthPurse}}' - \text{totalAbBalance}_{\text{ab/AuthPurse}} \\
\text{NoValueCreation} \quad & \text{requires that the sum of the before balances is greater or equal to the sum of the after balances. The abstract model enforces a stronger condition: that transfers change only the purses involved in the transfer and only by the amount stated in the transfer.}
\end{align*}
\]

$^1$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.
CHAPTER 2. SPS

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.

\[ \text{AllValueAccounted} \]
\[ \Delta \text{AbWorld} \]
\[ \text{totalAbBalance}_{\text{abAuthPurse}}' = \text{totalLost}_{\text{abAuthPurse}}' = \text{totalAbBalance}_{\text{abAuthPurse}} + \text{totalLost}_{\text{abAuthPurse}} \]

2.2.3 Authentic purses

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

\[ \text{Authentic} \]
\[ \Delta \text{AbWorld} \]
\[ \text{name}? \in \text{dom } \text{abAuthPurse} \]

2.2.4 Sufficient funds

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

\[ \text{SufficientFundsProperty} \]
\[ \Delta \text{AbWorld} \]
\[ \text{TransferDetails}? \]
\[ \text{value}? \leq (\text{abAuthPurse from}?).\text{balance} \]

2.3 Concrete model SPs

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

2.4. SPS AND THE MODELS

2.4.1 Transfer okay, no value creation

\[ \text{AbTransferOkayTD} \Rightarrow \neg \text{NoValueCreation} \]

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.

\[ \text{LogIfNecessary} \]
\[ \Delta \text{ConPurse} \]
\[ \text{exLog}' = \text{exLog} \cup (\text{if } \text{status} \in \{\text{epv, epa}\} \text{then } \{\text{pdAuth}\} \text{else } \emptyset) \]

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:

\[ \text{AbTransferOkay} \Rightarrow \neg \text{NoValueCreation} \land \text{AllValueAccounted} \]
\[ \text{AbTransferLost} \Rightarrow \neg \text{NoValueCreation} \land \text{AllValueAccounted} \]
\[ \text{AbIgnore} \Rightarrow \neg \text{NoValueCreation} \land \text{AllValueAccounted} \]

In the proofs below, we use the TD form of the definitions, by cutting in the appropriate TransferDetails.

2.4.1 Transfer okay, no value creation

\[ \text{AbTransferOkayTD} \Rightarrow \neg \text{NoValueCreation} \]

1Used in the definition of: AbortPurse, section 4.8.2.
Proof:

2.4.2 Transfer okay, all value accounted

\[
\text{totalAbBalance} \triangleright \text{abAuthPurse}' = \text{totalAbBalance}((\text{from},\text{to}) \triangleright \text{abAuthPurse}') + (\text{abAuthPurse}' \triangleright \text{from}'\text{.balance} + (\text{abAuthPurse}' \triangleright \text{to}'\text{.balance})) \]

\[
\text{totalAbBalance}((\text{from},\text{to}) \triangleright \text{abAuthPurse}') = \text{totalAbBalance}((\text{from},\text{to}) \triangleright \text{abAuthPurge}) + (\text{abAuthPurse}' \triangleright \text{from}'\text{.balance} + (\text{abAuthPurse}' \triangleright \text{to}'\text{.balance} + \text{value}))) [\text{AbTransferOkay}]
\]

\[
\text{totalAbBalance} \triangleright \text{abAuthPurse} = \text{totalAbBalance} \triangleright \text{abAuthPurse}
\]

\[\text{totalAbBalance} \triangleright \text{abAuthPurse} \leq \text{totalAbBalance} \triangleright \text{abAuthPurse} \]

2.4.4 Transfer lost, all value accounted

\[\text{AbTransferLostTD} \triangleright \text{AbValueAccounted} \]

Proof:

\[
\text{totalAbBalance} \triangleright \text{abAuthPurse'} + \text{totalLost} \triangleright \text{abAuthPurse'}
\]

\[
\text{totalAbBalance}((\text{from},\text{to}) \triangleright \text{abAuthPurse'}) = \text{totalAbBalance}((\text{from},\text{to}) \triangleright \text{abAuthPurse}) + (\text{abAuthPurse'} \triangleright \text{from}'\text{.balance} + (\text{abAuthPurse'} \triangleright \text{to}'\text{.balance})) \]

\[
\text{totalAbBalance}((\text{from},\text{to}) \triangleright \text{abAuthPurse}) = \text{totalAbBalance}((\text{from},\text{to}) \triangleright \text{abAuthPurge}) + (\text{abAuthPurse} \triangleright \text{from}'\text{.balance} + (\text{abAuthPurse} \triangleright \text{to}'\text{.balance} + \text{value})) [\text{AbTransferOkay}]
\]

\[
\text{totalAbBalance} \triangleright \text{abAuthPurse} = \text{totalAbBalance} \triangleright \text{abAuthPurse}
\]

\[\text{totalAbBalance} \triangleright \text{abAuthPurse} \leq \text{totalAbBalance} \triangleright \text{abAuthPurse} \]

2.4.3 Transfer lost, no value creation

\[\text{AbTransferLostTD} \triangleright \text{NoValueCreation} \]

Proof:

\[
\text{totalAbBalance} \triangleright \text{abAuthPurse'}
\]

\[
\text{totalAbBalance}((\text{from},\text{to}) \triangleright \text{abAuthPurse'}) = \text{totalAbBalance}((\text{from},\text{to}) \triangleright \text{abAuthPurge}) + (\text{abAuthPurse'} \triangleright \text{from}'\text{.balance} + (\text{abAuthPurse'} \triangleright \text{to}'\text{.balance})) [\text{totalAbBalance}]
\]

\[
\text{totalAbBalance}((\text{from},\text{to}) \triangleright \text{abAuthPurge}) = \text{totalAbBalance}((\text{from},\text{to}) \triangleright \text{abAuthPurse}) + (\text{abAuthPurse} \triangleright \text{from}'\text{.balance} + (\text{abAuthPurse} \triangleright \text{to}'\text{.balance} + \text{value})) [\text{AbTransferLost}]
\]

\[
\text{totalAbBalance} \triangleright \text{abAuthPurse} = \text{totalAbBalance} \triangleright \text{abAuthPurse}
\]

\[\text{totalAbBalance} \triangleright \text{abAuthPurse} \leq \text{totalAbBalance} \triangleright \text{abAuthPurse} \]

2.4.4 Transfer lost, all value accounted

\[\text{AbTransferLostTD} \triangleright \text{AbValueAccounted} \]

Proof:

\[
\text{totalAbBalance} \triangleright \text{abAuthPurse'} + \text{totalLost} \triangleright \text{abAuthPurse'}
\]

\[
\text{totalAbBalance}((\text{from},\text{to}) \triangleright \text{abAuthPurse'}) = \text{totalAbBalance}((\text{from},\text{to}) \triangleright \text{abAuthPurse}) + (\text{abAuthPurse'} \triangleright \text{from}'\text{.balance} + (\text{abAuthPurse'} \triangleright \text{to}'\text{.balance})) [\text{totalAbBalance}]
\]

\[
\text{totalAbBalance}((\text{from},\text{to}) \triangleright \text{abAuthPurse}) = \text{totalAbBalance}((\text{from},\text{to}) \triangleright \text{abAuthPurse}) + (\text{abAuthPurse} \triangleright \text{from}'\text{.balance} + (\text{abAuthPurse} \triangleright \text{to}'\text{.balance} + \text{value})) [\text{AbTransferLost}]
\]

\[
\text{totalAbBalance} \triangleright \text{abAuthPurse} = \text{totalAbBalance} \triangleright \text{abAuthPurse}
\]

\[\text{totalAbBalance} \triangleright \text{abAuthPurse} \leq \text{totalAbBalance} \triangleright \text{abAuthPurse} \]
2.4.5 Transfer ignore

\( \text{AbIgnore } \vdash \text{NoValueCreation } \land \text{AllValueAccounted} \)

Proof: 
Follows directly from the definition of \text{AbIgnore}, which changes none of the relevant values.

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 \text{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.)

\[ \text{AbPurse} \overset{\text{def}}{=} \{ \text{balance}, \text{lost} : \mathbb{N} \} \]

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.

\[
\text{TransferDetails}
\begin{array}{l}
\text{from, to : NAME} \\
\text{value : N}
\end{array}
\]

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

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.

\[\text{AbWorld} \equiv [\ \text{abAuthPurse} : \text{NAME} \rightarrow \text{AbPurse}]\]

3.3 Secure operations

Having defined our abstract world, \(\text{AbWorld}\), we now define operations on the world that respect the relevant SPs. We call these secure operations. They comprise:

- \(\text{AbIgnore}\): securely do nothing
- \(\text{AbTransfer}\): 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:

\[\text{AIN} ::= \text{aNullIn} \mid \text{transfer}((\text{TransferDetails}))\]

\[\text{AOUT} ::= \text{aNullOut}\]

Every abstract operation has the following properties:

\[\text{AbOp} \quad \Delta \text{AbWorld} \quad a? : \text{AIN} \quad a! : \text{AOUT}\]

\[a! = \text{aNullOut}\]

The output is always \(\text{aNullOut}\) (that is, we are not interested in the abstract output).

3.3.2 Abstract ignore

Any operation has the option of securely doing nothing.

\[\text{AbIgnore} \quad \text{AbOp} \quad \text{abAuthPurse'} = \text{abAuthPurse}\]

3.3.3 Transfer

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

\[\text{AbPurseTransfer} \equiv \text{AbPurse} \setminus (\text{balance, lost})\]

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

\[\text{AbWorldSecureOp} \quad \text{AbOp} \quad \text{TransferDetails}? \quad a? \in \text{ran transfer}\]

\[\delta \text{TransferDetails}? = \text{transfer}^{-1} a?\]

\[\{\text{from}, \text{to}\} \not\in \text{abAuthPurse'} = \{\text{from}, \text{to}\} \not\in \text{abAuthPurse}\]

\(^1\)SP 3, ‘Authentic purses’, section 2.2.3.
A transfer can securely succeed between two purses if they are distinct, both purses are authentic\(^2\), and the from purse has sufficient funds\(^3\).

\[
\text{AbTransferOkayTD} \\
\text{AbWorldSecureOp} \\
\text{Authentic}[\text{from}?/\text{name}?] \\
\text{Authentic}[\text{to}?/\text{name}?] \\
\text{SufficientFundsProperty} \\
\text{to}? = \text{from}? \\
\text{abAuthPurse}\,\text{from}? = (\mu \Delta \text{AbPurse} | \\
\quad \Delta \text{AbPurse} = \text{abAuthPurse}\,\text{from}? \\
\quad \land \, \text{balance'} = \text{balance} - \text{value}? \\
\quad \land \, \text{lost'} = \text{lost} \\
\quad \land \, \Xi \text{AbPurseTransfer} \\
\quad \land \, \partial \text{AbPurse'} ) \\
\text{abAuthPurse}\,\text{to}? = (\mu \Delta \text{AbPurse} | \\
\quad \Delta \text{AbPurse} = \text{abAuthPurse}\,\text{to}? \\
\quad \land \, \text{balance'} = \text{balance} + \text{value}? \\
\quad \land \, \text{lost'} = \text{lost} \\
\quad \land \, \Xi \text{AbPurseTransfer} \\
\quad \land \, \partial \text{AbPurse'} )
\]

The operation transfers \(\text{value}?\) 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:

\[
\text{AbTransferOkay} \equiv \text{AbTransferOkayTD} \setminus (\text{to}?,\text{from}?,\text{value}?)
\]

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\(^6\).

\[
\text{AbTransferLostTD} \\
\text{AbWorldSecureOp} \\
\text{Authentic}[\text{from}?/\text{name}?] \\
\text{Authentic}[\text{to}?/\text{name}?] \\
\text{SufficientFundsProperty} \\
\text{to}? = \text{from}? \\
\text{abAuthPurse}\,\text{from}? \in (\Delta \text{AbPurse} | \\
\quad \Delta \text{AbPurse} = \text{abAuthPurse}\,\text{from}? \\
\quad \land \, \text{balance'} = \text{balance} - \text{value}? \\
\quad \land \, \text{lost'} = \text{lost} + \text{value}? \\
\quad \land \, \Xi \text{AbPurseTransfer} \\
\quad \land \, \partial \text{AbPurse'} ) \\
\text{abAuthPurse}\,\text{to}? = \text{abAuthPurse}\,\text{to}?
\]

The operation removes \(\text{value}?\) 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:

\[
\text{AbTransferLost} \equiv \text{AbTransferLostTD} \setminus (\text{to}?,\text{from}?,\text{value}?)
\]

The full transfer operation can also securely do nothing, \text{AbIgnore}. The full transfer operation is

\[
\text{AbTransfer} \equiv \text{AbTransferOkay} \lor \text{AbTransferLost} \lor \text{AbIgnore}
\]

### 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 \text{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 \text{AbWorld}’\(^9\).

\[
\text{AbInitState} \equiv \text{AbWorld}'
\]

---

\(^2\)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.
\(^7\)SP 1, ‘No value created’, section 2.2.1.
\(^8\)SP 2, ‘All value accounted’, section 2.2.2.
CHAPTER 3. A MODEL

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 \( A \) model:

\[
\text{AbInitIn} \equiv \{ a?, g? : \text{ABN} \mid a? = g? \}
\]

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:

\[
\text{GlobalWorld} \equiv \{ g\text{AuthPurse} : \text{NAME} \mapsto \text{AbPurse} \}
\]

Finalisation gives the global state corresponding to an abstract state. These are mostly the identity relations in the \( A \) model:

\[
\begin{align*}
\text{AbFinState} & \equiv \text{AbState} \\
\text{AbWorld} & \equiv \text{AbState} \\
\text{GlobalWorld} & \equiv \{ g\text{AuthPurse} : \text{NAME} \mapsto \text{AbPurse} \}
\end{align*}
\]

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

\[
\text{AbFinOut} \equiv \{ a?, g? : \text{AOUT} \mid a? = g? \}
\]

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 NAMEs 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 \( B \) 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.

\[
\text{STATUS} ::= \text{eaFrom} \mid \text{eaTo} \mid \text{epr} \mid \text{epv} \mid \text{epa}
\]

The statuses are: \( \text{eaFrom} \) 'expecting any payer', \( \text{eaTo} \) 'expecting any payee', \( \text{epr} \) 'expecting payment req', \( \text{epv} \) 'expecting payment val', and \( \text{epa} \) '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.
CHAPTER 4. B MODEL, PURSE

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.

<table>
<thead>
<tr>
<th>CounterPartyDetails</th>
</tr>
</thead>
<tbody>
<tr>
<td>name : NAME</td>
</tr>
<tr>
<td>value : N</td>
</tr>
<tr>
<td>nextSeqNo : N</td>
</tr>
</tbody>
</table>

4.3.2 Payment log message details

Purses store current payment details, and exception logs 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.

<table>
<thead>
<tr>
<th>PayDetails</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransferDetails</td>
</tr>
<tr>
<td>fromSeqNo, toSeqNo : N</td>
</tr>
<tr>
<td>from ≠ to</td>
</tr>
</tbody>
</table>

4.4 CLEAR EXCEPTION LOG VALIDATION

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.

\[ [\text{CLEAR}] \]

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).

\( \text{image} \) is a function to calculate the clear code for a given non-empty set of exception records.

\[ \text{image} : P, \text{PayDetails} \rightarrow \text{CLEAR} \]

\( \text{image} \) 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 \( \text{image} \)^2.

\(^1\)Concrete SF 2.2, ‘Exception logging’, section 2.3.1.

\(^2\)In practice, \( \text{image} \) 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:

\[
\text{MESSAGE ::= startFrom}((\text{CounterPartyDetails})) \\
| \text{startTo}((\text{CounterPartyDetails})) \\
| \text{readExceptionLog} \\
| \text{req}((\text{PayDetails})) \\
| \text{val}((\text{PayDetails})) \\
| \text{ack}((\text{PayDetails})) \\
| \text{exceptionLogResult}((\text{NAME} \times \text{PayDetails})) \\
| \text{exceptionLogClear}((\text{NAME} \times \text{CLEAR})) \\
| \bot
\]

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.

\(\bot\), '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. \(\bot\) 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.

\[
\begin{align*}
\text{ConPurse} & \quad \text{balance : N} \\
& \quad \text{exLog : PayDetails} \\
& \quad \text{name : NAME} \\
& \quad \text{nextSeqNo : N} \\
& \quad \text{pdAuth : PayDetails} \\
& \quad \text{status : STATUS} \\
\forall pd : \text{exLog} \land \text{name} \in \{pd.\text{from}, pd.\text{to}\} \\
& \quad \text{status} = \text{epr} \Rightarrow \text{name} = \text{pdAuth.\text{from}} \\
& \quad \land \text{pdAuth.\text{value}} \leq \text{balance} \\
& \quad \land \text{pdAuth.\text{fromSeqNo}} < \text{nextSeqNo} \\
& \quad \text{status} = \text{epv} \Rightarrow \text{pdAuth.\text{toSeqNo}} < \text{nextSeqNo} \\
& \quad \text{status} = \text{epa} \Rightarrow \text{pdAuth.\text{fromSeqNo}} < \text{nextSeqNo}
\end{align*}
\]

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

The predicate on the purse state records the following constraints:

P-1 \(\forall pd : \text{exLog} \land \text{name} \in \{pd.\text{from}, pd.\text{to}\}\) All log details in the exception log refer to this purse, as the from or the to party\(^1\).

P-2 \(\text{status} = \text{epr} \Rightarrow \text{name} = \text{pdAuth.\text{from}} \\
\land \text{pdAuth.\text{value}} \leq \text{balance} \\
\land \text{pdAuth.\text{fromSeqNo}} < \text{nextSeqNo}\)

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 \(\text{status} = \text{epv} \Rightarrow \text{pdAuth.\text{toSeqNo}} < \text{nextSeqNo}\)

\(^1\)Used in: AuxWorld does not add constraints, section 5.2.1.

\(^2\)Used in: CReq, B-9, section 29.4.

\(^3\)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.

\(^4\)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

\[ \text{P-4 status} = \text{cpa} = \text{pdAuth}.\text{fromSeqNo} < \text{nextSeqNo} \]

If the purse is expecting a payment acknowledgement, then its next sequence number is greater than the current transaction’s sequence number.

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).

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.

\[
\text{ConPurseIncrease} \triangleq \text{ConPurse} \setminus \{\text{nextSeqNo}\}
\]

\[
\begin{array}{c}
\text{IncreasePurseOkay} \\
\Delta \text{ConPurse} \\
\text{m?}, \text{m!} : \text{MESSAGE} \\
\hline
\text{\text{ConPurseIncrease}} \\
\text{nextSeqNo}' \geq \text{nextSeqNo} \\
\text{m!} = \bot
\end{array}
\]

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.

\[
\text{ConPurseAbort} \triangleq \text{ConPurse} \setminus \{\text{nextSeqNo}, \text{exLog}, \text{pdAuth}, \text{status}\}
\]

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

\[\text{Concrete SP 2.2, 'Exception logging', section 2.3.1.}\]
We do not, at this stage, put any restrictions on the output message \( m' \). Later, we either compose \( \text{AbortPurseOkay} \) with another operation, using the latter’s \( m' \), or we promote \( \text{AbortPurseOkay} \) to the world level, where we define \( m' = \bot \).

### 4.9 Value transfer operations

The \( \text{StartTo} \) and \( \text{StartFrom} \) operations, when starting from \( \text{eaFrom} \), change only the sequence number, the stored \( \text{pdAuth} \), and the status of a purse.

\[
\text{ConPurseStart} \triangleq \text{ConPurse} \setminus \{(\text{nextSeqNo}, \text{pdAuth}, \text{status})\}
\]

The \( \text{Req} \) operation change only the balance and the status of a purse.

\[
\text{ConPurseReq} \triangleq \text{ConPurse} \setminus \{(\text{balance}, \text{status})\}
\]

The \( \text{Val} \) operation change only the balance and the status of a purse.

\[
\text{ConPurseVal} \triangleq \text{ConPurse} \setminus \{(\text{balance}, \text{status})\}
\]

The \( \text{Ack} \) operation changes only the status of a purse, and allows the \( \text{pdAuth} \) to change arbitrarily.

\[
\text{ConPurseAck} \triangleq \text{ConPurse} \setminus \{(\text{status}, \text{pdAuth})\}
\]

#### 4.9.1 StartFromPurse

A \( \text{startFrom} \) message is valid only if it refers to a different purse from the receiver, and mentions a value for which the \( \text{from} \) purse has sufficient funds.
4.9.2 StartToPurse

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

\[
\text{ValidStartTo} \quad \text{ConPurse}\ m? : \text{MESSAGE} \\
\text{cpd} : \text{CounterPartyDetails} \\
\begin{align*}
m? & \in \text{ran} \text{startTo} \\
\text{cpd} & = \text{startTo} \ m? \\
\text{cpd}.\text{name} & = \text{name}
\end{align*}
\]

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

\[
\text{StartToPurseEaFromOkay} \quad \text{ΔConPurse}\ m?, \ m! : \text{MESSAGE} \\
\text{cpd} : \text{CounterPartyDetails} \\
\begin{align*}
\text{ValidStartTo} & \quad \text{status} = \text{eaFrom} \\
\text{ΔConPurseStart} & \quad \text{nextSeqNo}' > \text{nextSeqNo} \\
\text{pdAuth}' = (\text{μ PayDetails} | \\
\text{to} = \text{name} \\
\text{from} = \text{cpd}.\text{name} \\
\text{value} = \text{cpd}.\text{value} \\
\text{toSeqNo} = \text{nextSeqNo} \\
\text{fromSeqNo} = \text{cpd}.\text{nextSeqNo} ) \\
\text{status}' & = \text{epv} \\
\text{m!} & = \text{req pdAuth}'
\end{align*}
\]

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).

\[
\text{AuthenticReqMessage} \quad \text{ConPurse}\ m? : \text{MESSAGE} \\
m? = \text{req pdAuth}
\]

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

\[
\text{ReqPurseOkay} \quad \text{ΔConPurse}\ m?, \ m! : \text{MESSAGE} \\
\text{AuthenticReqMessage} \\
\text{status} = \text{epr} \\
\text{ΔConPurseReq} \\
\text{balance}' = \text{balance} - \text{pdAuth.value} \\
\text{status}' = \text{epa} \\
\text{m!} = \text{val pdAuth}
\]

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).

\[
\begin{align*}
\text{ValPurse} \\
\text{ConPurse} \\
m? &= \text{val pdAuth}
\end{align*}
\]

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

\[
\begin{align*}
\text{ValPurseOkay} \\
\text{ConPurse} \\
m? &= \text{val pdAuth}
\end{align*}
\]

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

4.9.5 AckPurge

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

\[
\begin{align*}
\text{AckPurse} \\
\text{ConPurse} \\
m? &= \text{ack pdAuth}
\end{align*}
\]

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

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.

\[
\begin{align*}
\text{ReadExceptionLogPurseEafromOkay} \\
\text{ConPurse} \\
m? &= \text{readExceptionLog} \\
\text{status} &= \text{eaFrom} \\
m! &= \text{\text{readExceptionLogResult} (\text{name}, \text{id})}
\end{align*}
\]

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\(^\text{10}\).

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

\[
\begin{align*}
\text{ReadExceptionLogPurseOkay} \\
\text{\text{AbortPurseOkay} \ \text{\&} \ \text{ReadExceptionLogPurseEafromOkay}}
\end{align*}
\]

\(^{10}\)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.

\[
\text{ConPurseClear} \equiv \text{ConPurse} \setminus \{\text{exLog}\}
\]

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:

\[
\begin{align*}
\text{ClearExceptionLogPurseEafromOkay} & \quad \Delta \text{ConPurse} \\
& m? : \text{MESSAGE} \\
& \text{exLog} = \emptyset \\
& m? = \text{exceptionLogClear}(\text{name}, \text{imageexLog}) \\
& \text{status} = \text{eaFrom} \\
& \equiv \text{ConPurseClear} \\
& \text{exLog}' = \emptyset \\
& m! = \bot
\end{align*}
\]

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.
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.

\[ \text{Logbook} : \mathcal{P}(\text{name} \leftrightarrow \text{PayDetails}) \]

\[ \text{Logbook} = \mathcal{P}(\{\text{PayDetails} \cdot \text{from} \to \text{PayDetails}\} \cup \{\text{PayDetails} \cdot \text{to} \to \text{PayDetails}\}) \]

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

\[ \forall n : \text{dom conAuthPurse} \cdot (\text{conAuthPurse } n).\text{name} = n \]

\[ \forall nld : \text{archive} \cdot \text{first } nld \in \text{dom conAuthPurse} \]

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

<table>
<thead>
<tr>
<th>from</th>
<th>epr</th>
<th>eaFrom</th>
<th>(diff trans incl eaFrom)</th>
<th>to eaTo</th>
</tr>
</thead>
<tbody>
<tr>
<td>epv</td>
<td>0</td>
<td>?</td>
<td>0 0</td>
<td>×</td>
</tr>
<tr>
<td>eaTo</td>
<td>0</td>
<td>0</td>
<td>0 1</td>
<td>0</td>
</tr>
</tbody>
</table>

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.

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:

- \textit{allLogs}: All the exception logs; all those logs in the archive, and those still uncleared in purses.
- \textit{authenticFrom}, \textit{authenticTo}: All possible payment details referring to authentic from purses, and authentic to purses.
- \textit{fromLogged}: All those payment details logged by a from purse.
- \textit{toLogged}: All those details logged by a to purse.
- \textit{tolnEpv}: 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.
- \textit{fromlnEpv}: All those details for which the to purse is authentic, and is currently in epv with those details stored.
- \textit{fromlnEpa}: All those details for which the from purse is authentic, and is currently in epa with those details stored.
- \textit{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.
- \textit{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 tolnEpv is a finite set.

We have the identity

\[ \text{AuxWorld} \sim \text{definitelyLost} \cup \text{maybeLost} = \left( \text{fromlnEpa} \cup \text{fromLogged} \right) \cap \left( \text{tolnEpv} \cup \text{toLogged} \right) \]

Figure 5.2: The sets \textit{definitelyLost} (vertical hatching) and \textit{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 \textit{maybeLost} and \textit{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 \textit{AuxWorld} does not add constraints

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

\[ \text{NewVariables} \equiv \exists \text{ConWorld} \cdot \text{AuxWorld} \]

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

\[ \text{ConWorld} \vdash \exists \text{NewVariables} \cdot \text{AuxWorld} \]

\textbf{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.

\[ \text{ConWorld} \vdash \exists \text{NewVariables} \cdot \text{AuxWorld} \land \text{allLogs} \subseteq \text{Logbook} \]
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.

ConWorld
\[ \top \]
\begin{align*}
\neg \text{archive} \cup \{ n : \text{conAuthPurse}; \text{pd} : \text{PayDetails} | \text{pd} \in (\text{conAuthPurse}.\text{exLog}) \} \in \text{Logbook}
\end{align*}

From the definition of archive, archive is in Logbook. From constraint P–1 in ConParse, 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.

5.3 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.

**BetweenWorld**

AuxWorld

\[ \forall \text{pd} : \text{PayDetails} | \text{req pd} \in \text{ether} \Rightarrow \text{pd} \in \text{authenticTo} \]

\[ \forall \text{pd} : \text{PayDetails} | \text{req pd} \in \text{ether} \]
\[ \text{pd}.\text{toSeqNo} \prec (\text{conAuthPurse}.\text{pd}.\text{to}).\text{nextSeqNo} \]

\[ \forall \text{pd} : \text{PayDetails} | \text{val pd} \in \text{ether} \]
\[ \text{pd}.\text{toSeqNo} \prec (\text{conAuthPurse}.\text{pd}.\text{to}).\text{nextSeqNo} \]
\[ \land \text{pd}.\text{fromSeqNo} \prec (\text{conAuthPurse}.\text{pd}.\text{from}).\text{nextSeqNo} \]

\[ \forall \text{pd} : \text{PayDetails} | \text{ack pd} \in \text{ether} \]
\[ \text{pd}.\text{toSeqNo} \prec (\text{conAuthPurse}.\text{pd}.\text{to}).\text{nextSeqNo} \]
\[ \land \text{pd}.\text{fromSeqNo} \prec (\text{conAuthPurse}.\text{pd}.\text{from}).\text{nextSeqNo} \]

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.

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

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.
There is a req of definitelyLost in location 28.5; CAbort, section C.12.

B–3 There are no 'future' val messages 7; 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 6.

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 evp 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 evp 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 evp or eaTo 7: 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 evp), hold a to sequence number strictly less than that purse's stored current transaction sequence number.

B–9 If the from purse is in evp then there is no val message 9 or ack message 9 in the ether.

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

5.4 FRAMING SCHEMA

B–11 If the to purse is in evp 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.

---

1Used in: CStartFrom, B–9, section 29.2; CStartTo, B–11, section 29.3; CVal, B–4, section 29.5.

2Used in: CStartFrom, B–9, section 29.2; CStartTo, B–10, section 29.3.

3Used in: CStartFrom, B–7, section 29.2.

4Used in: CStartTo, B–8, 29.3, 29.3

5Used in: CStartFrom, location of pdThis, section 16.3; CReq, B–7, section 29.4; lemma 'notLoggedAndAval', section C.12.

6Used in: CVal, B–8, section 29.5; lemma 'notLoggedAndAval', section C.12.

7Used in: CVal, B–9, section 29.5.

8Used in: CReq, case 4, section 18.10.

9Used in: CStartTo, location of pdThis, section 17.3; CReq, 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–10, section 28.5; CAck, B–11, section 29.6.
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:

\[ \text{Ignore} \stackrel{\hat{}}{=} [ \exists \text{BetweenWorld}; \text{name}? : \text{NAME}; \text{m?}, \text{m!} : \text{MESSAGE} | \text{m!} = \bot ] \]

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

\[ \text{Increase} \stackrel{\hat{}}{=} \text{Ignore} \lor ( \exists \Delta \text{ConPurse} \cdot \Phi \text{BOp} \land \text{IncreasePurseOkay} ) \]

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

\[ \text{Abort} \stackrel{\hat{}}{=} \text{Ignore} \lor ( \exists \Delta \text{ConPurse} \cdot \text{AbortPurseOkay} \land ( \Phi \text{BOp} | \text{m!} = \bot ) ) \]

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:

\[ \text{StartFrom} \stackrel{\hat{}}{=} \text{Ignore} \lor ( \exists \Delta \text{ConPurse} \cdot \Phi \text{BOp} \land \text{StartFromPurseOkay} ) \]

\[ \text{StartTo} \stackrel{\hat{}}{=} \text{Ignore} \lor ( \exists \Delta \text{ConPurse} \cdot \Phi \text{BOp} \land \text{StartToPurseOkay} ) \]

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

- \( \text{m?} \in \text{ether} \)
  
  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).

- \( \text{name}? \in \text{dom} \text{conAuthPurse} \)
  
  the purse is in the world of authentic purses.

- \( \Phi \text{ConPurse} = \text{conAuthPurse name}? \)
  
  The before state of ConPurse we are operating on is the state of the purse identified by name?

- \( \text{conAuthPurse}' = \text{conAuthPurse} \oplus \{ \text{name}? \to \theta \text{ConPurse}' \} \)
  
  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.

- \( \text{archive}' = \text{archive} \)
  
  The archive remains unchanged.

- \( \text{ether}' = \text{ether} \cup \{ \text{m!} \} \)
  
  the output message is recorded by the ether.

---

18Used in Req proof, section 18.7.2.
For use in the proofs, we also promote the \( Ea \) from part of the operations on their own:

\[
\begin{align*}
\text{StartFromEaOkay} & \equiv \exists \Delta \text{ConPurse} \cdot \\
& \Phi \text{BoP} \land \text{StartFromPurseEaOkay} \\
\text{StartToEaOkay} & \equiv \exists \Delta \text{ConPurse} \cdot \\
& \Phi \text{BoP} \land \text{StartToPurseEaOkay}
\end{align*}
\]

The promoted protocol operations are:

\[
\begin{align*}
\text{Req} & \equiv \text{Ignore} \lor (\exists \Delta \text{ConPurse} \cdot \Phi \text{BoP} \land \text{ReqPurseOkay}) \\
\text{Val} & \equiv \text{Ignore} \lor (\exists \Delta \text{ConPurse} \cdot \Phi \text{BoP} \land \text{ValPurseOkay}) \\
\text{Ack} & \equiv \text{Ignore} \lor (\exists \Delta \text{ConPurse} \cdot \Phi \text{BoP} \land \text{AckPurseOkay})
\end{align*}
\]

5.6.2 Exception log operations

The promoted log enquiry operation is:

\[
\begin{align*}
\text{ReadExceptionLog} & \equiv \text{Ignore} \lor (\exists \Delta \text{ConPurse} \cdot \Phi \text{BoP} \land \text{ReadExceptionLogPurseOkay})
\end{align*}
\]

The promoted exception log clear operation is:

\[
\begin{align*}
\text{ClearExceptionLog} & \equiv \text{Ignore} \lor \text{Abort} \\
& \lor (\exists \Delta \text{ConPurse} \cdot \Phi \text{BoP} \land \text{ClearExceptionLogPurseOkay})
\end{align*}
\]

For use in the proofs, we also promote the \( Ea \) from part of the operations on their own:

\[
\begin{align*}
\text{ReadExceptionLogEaOkay} & \equiv \exists \Delta \text{ConPurse} \cdot \\
& \Phi \text{BoP} \land \text{ReadExceptionLogPurseEaOkay} \\
\text{ClearExceptionLogEaOkay} & \equiv \exists \Delta \text{ConPurse} \cdot \\
& \Phi \text{BoP} \land \text{ClearExceptionLogPurseEaOkay}
\end{align*}
\]

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.
5.8 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 ⊥ 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:

\(\text{StartFrom} \Rightarrow \text{StartTo} \Rightarrow \text{Req} \Rightarrow \text{Val} \Rightarrow \text{Ack}\)

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.
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.

\[
\text{BetweenInitState} = \text{BetweenWorld}' \\
\{ \text{readExceptionLog, } \perp \} \\
\cup \\
\{ \text{cpd} : \text{CounterPartyDetails} \cdot \{ \text{startFrom cpd, startTo cpd} \} \} \\
\subseteq \text{ether}'
\]

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
CHAPTER 6. B INITIAL, FINAL

- 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 req 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.

\[
\text{BetwInitIn}
\]

\[
g? : \text{AIN}
m? : \text{MESSAGE}
\]

\[
m? \in \text{ran req} \Rightarrow
\]

\[
g? = \text{transfer}(\mu \text{TransferDetails} |
\]

\[
\qquad \text{from} = (\text{req} \circ m?), \text{from}
\]

\[
\qquad \land \text{to} = (\text{req} \circ m?), \text{to}
\]

\[
\qquad \land \text{value} = (\text{req} \circ m?), \text{value}
\]

\[
m? \notin \text{ran req} \Rightarrow g? = aNullIn
\]

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'.)

\[
\text{BetwFinState}
\]

\[
\text{BetweenWorld}
\]

\[
\text{GlobalWorld}
\]

\[
\text{dom gAuthPurse} = \text{dom conAuthPurse}
\]

\[
\forall \text{name} : \text{dom conAuthPurse} \
\]

\[
(\text{gAuthPurse name).balance} = (\text{conAuthPurse name}).balance
\]

\[
\land (\text{gAuthPurse name).lost} =
\]

\[
\text{sumValue}\left(\text{definitelyLost } \cup \text{maybeLost}\right)
\]

\[
\cap \{ \text{id} : \text{PayDetails} | \text{id}.\text{from} = \text{name} \}
\]

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 \(\text{sumValue}\) 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.

\[
\text{BetwFinOut}
\]

\[
g? : \text{AOUT}
m? : \text{MESSAGE}
\]

\[
g? = aNullOut
\]

All concrete outputs are interpreted as the single abstract output, aNullOut.
Chapter 7

Concrete model: implementation

7.1 Concrete World State

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

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

\[
\begin{align*}
\Delta ConWorld_0 \triangleq ConWorld_0 \land ConWorld_0' \\
\Xi ConWorld_0 \triangleq [ \Delta ConWorld_0 \mid \emptyset ConWorld_0 = \emptyset ConWorld_0']
\end{align*}
\]

7.2 Framing Schema

The concrete world $C$ has the same operations as the $B$ 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 $C$ framing schema is used to promote the purse operations.
7.3 Ignore, Increase and Abort

The B operations Ignore, Increase and Abort have C equivalents, working on the C world instead of the 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.

\[ \begin{align*} 
\Delta \text{ConWorld}_0 \\
\Delta \text{ConPurse} \\
m?_0, m? : \text{MESSAGE} \\
\text{name}? : \text{NAME} \\
m? \in \text{ether}_0 \\
\text{name}? \in \text{dom} \text{conAuthPurse}_0 \\
\delta \text{ConPurse} = \text{conAuthPurse}_0 \cdot \text{name}? \\
\text{conAuthPurse}'_0 = \text{conAuthPurse}_0 \cdot \{ \text{name}? \mapsto \delta \text{ConPurse}' \} \\
\text{archive}'_0 = \text{archive}_0 \\
\text{ether}'_0 = \text{ether}_0 \cup \{ m! \} 
\end{align*} \]

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

7.4 Promoted operations

As with the B promoted operations, the C promoted operations are made total by disjoining with \( \text{CIgnore} \).

7.5 Operations at the world level only

As with the 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 B model, but acting on \( \text{ConWorld} \) instead of \( \text{BetweenWorld} \).
7.5.1 Exception Log clear authorisation

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

\[
\text{CAuthoriseExLogClear} \equiv \text{Clgnore} \\
\lor (\exists \text{ConPurse} \cdot \left( \phi \text{COp} \mid \left( \exists \text{lds} : \text{P}_1 \text{PayDetails} \mid \text{name} \cdot \text{lds} \subseteq \text{archive}_0 \cdot \text{m}! = \text{exceptionLogClear}(\text{name}, \text{image} \text{lds}) \right) \right)
\]

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

\[
\text{CArchive} \quad \Delta \text{ConWorld}_{0} \\
\text{m}?, \text{m}! : \text{MESSAGE} \\
\text{name} : \text{NAME} \\
\text{conAuthPurse}_0' = \text{conAuthPurse}_0 \\
\text{ether}_0' \subseteq \text{ether}_0 \\
\text{archive}_0' \subseteq \text{archive}_0 \\
\text{archive}_0' \subseteq \text{archive}_0 \cup \left\{ \text{log} : \text{NAME} \times \text{PayDetails} \mid \text{exceptionLogResult} \text{log} \in \text{ether}_0 \right\} \\
\text{m}! = \bot
\]

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 B world (the subset is needed because the C ether is lossy).

\[
\text{ConInitState} \\
\text{ConWorld}_{0} \\
( \exists \text{BetweenWorld} \cdot \text{BetweenInitState} \cdot \text{conAuthPurse}_0' = \text{conAuthPurse}_0 \\
\land \text{archive}_0' = \text{archive}' \\
\land \left\{ . \right\} \subseteq \text{ether}_0' \subseteq \text{ether}'
\]

7.7 Finalisation

The B finalisation is defined for any ConWorld; we reuse it for the C finalisation.

\[
\text{ConFinState} \\
\text{AuxWorld}_{0} \\
\text{GlobalWorld} \\
\text{dom} \text{gAuthPurse} = \text{dom} \text{conAuthPurse}_0 \\
\forall \text{name} : \text{dom} \text{conAuthPurse}_0 \cdot \\
\left( \text{gAuthPurse} \cdot \text{name} \right).\text{balance} = (\text{conAuthPurse}_0 \cdot \text{name}).\text{balance} \\
\land (\text{gAuthPurse} \cdot \text{name}).\text{lost} = \text{sumValue}(\text{definitelyLost}_0 \cup \text{maybeLost}_0) \\
\land (\text{ld} : \text{PayDetails} \mid \text{ld}.\text{from} = \text{name})
\]
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.

\[ \vdash \exists \text{State}\, \cdot \, \text{StateInit} \]

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

\[ \vdash \exists \text{State, Input} \, \cdot \, \text{pre Op} \]

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

\[ \vdash \forall \text{State, Input} \, \cdot \, \text{pre Op} \]

We present these proofs for each of our three models below.
CHAPTER 8. CONSISTENCY

8.2 Abstract model consistency proofs

8.2.1 Existence of initial abstract state

⊢ ∃ AbWorld′ • 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. 

8.2.2 Totality of abstract operations

AbIgnore is total.

Proof:

pre AbIgnore

= pre [ ΔAbWorld; a? : AIN; a! : AOUT ]

\[ abAuthPurse' = abAuthPurse \land a! = aNullOut \]

[defn. AbIgnore]

= [ AbWorld; a? : AIN ]

\[ \exists abAuthPurse' : NAME → AbPurse; a! : AOUT \]

\[ abAuthPurse' = abAuthPurse \land a! = aNullOut \]

[defn. pre ]

= [ AbWorld; a? : AIN ]

\[ \exists conAuthPurse' = \emptyset \]

\[ ether' = \{ readExceptionLog, ⊥ \} \cup \{ cpd : CounterPartyDetails → [ startPoint cpd, startTo cpd ] \} \]

\[ archive' = \emptyset \]

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 Between model consistency proofs

8.3.1 Existence of between initial state

⊢ ∃ BetweenWorld′ • 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.

\[ conAuthPurse' = \emptyset \]

\[ ether' = \{ readExceptionLog, ⊥ \} \]

\[ archive' = \emptyset \]

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.

8.3.2 Totality of between operations

All between operations are total.

Proof:
They all offer the option of \textit{Ignore} (explicitly by disjunction, except for \textit{Archive}, which offers it implicitly). \textit{Ignore} is the total identity operation.

\subsection{8.4 Concrete model consistency proofs}

\subsubsection{Existence of concrete initial state}

\[ \exists \text{ConWorld}_{\gamma} \rightarrow \text{ConInitState} \]

\textbf{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.

\subsubsection{Totality of concrete operations}

All concrete operations are total.

\textbf{Proof:}

The concrete operations are identical to the between ones. Therefore if the between operations are total, so are the concrete ones.
Chapter 9

Refinement Proof Rules

9.1 Security of the implementation

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

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

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

The refinements are

\[
\begin{align*}
A\text{\textunderscore Transfer} & \sqsubseteq \text{Req} \\
A\text{\textunderscore Ignore} & \sqsubseteq \text{StartFrom} \\
& \quad \vee \text{StartTo} \\
& \quad \vee \text{Val} \\
& \quad \vee \text{Ack} \\
& \quad \vee \text{ReadExceptionLog} \\
& \quad \vee \text{ClearExceptionLog} \\
& \quad \vee \text{AuthoriseExLogClear} \\
& \quad \vee \text{Archive} \\
& \quad \vee \text{Ignore} \\
& \quad \vee \text{Increase} \\
& \quad \vee \text{Abort}
\end{align*}
\]
CHAPTER 9. A TO B RULES

Correctness

R' ROut R RIn R
A'; AOut A; AOutA; AInA'; AIn AOp

Initialisation

AInit

BInit

BInitState

R' RIn

Finalisation

AFin

BFin

BFinState

BFinOut B; BOut B; BOutB; BInB'; BIn BOp

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 A to B refinement proofs, the following set of ‘upward’ or ‘backward’ proof rules are sufficient to show the refinement [Woodcock & Davies 1996]. For the B to 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 A and the ‘concrete’ model is B. In the second refinement the ‘abstract’ model is now B and the ‘concrete’ model is 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.

⊢ ∀ G; GIn; B; BIn; A’; AIn | BInitState ∧ BInitIn ∧ R' ∧ RIn •

AInitState ∧ 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.

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’.

⊢ ∀ G’; GOut; B; BOut | BFinState ∧ BFinOut •

∃ A; AOut • R ∧ ROut ∧ AFinState ∧ 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 ⊢ ∃ A • R ∧ AFinState

BFinOut ⊢ ∃ AOut • ROut ∧ AFinOut

9.2.3 Applicability

⊢ ∀ B; BIn | (∀ A; AIn | R ∧ RIn • pre AOp) • 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 pre AOp = true. So

(∀ A; AIn | R ∧ RIn • pre AOp)
⇒ (∀ A; AIn • R ∧ RIn • pre AOp)
⇒ (∀ A; AIn • R ∧ RIn = true)
⇒ (∀ A; AIn • true)
⇒ true

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

B; BIn ⊢ pre BOp

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

⊢ ∀ B; Bln • (∀ A; AIn • R ∧ Rln • pre AOp •

(∀ A′; AOut; B′; BOut | BOp ∧ R′ ∧ ROut •

(∃ A; Aln • R ∧ Rln ∧ AOp )))

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

⊢ ∀ B; Bln • (∀ A′; AOut; B′; BOut | BOp ∧ R′ ∧ ROut •

(∃ A; Aln • R ∧ Rln ∧ AOp ))

Moving the quantifier into the hypothesis:

B; Bln; A′; AOut; B′; BOut | BOp ∧ R′ ∧ ROut •

⊢ (∃ A; Aln • R ∧ Rln ∧ AOp)

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 • (∃ A; Aln • R ∧ Rln ∧ AOp)

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 A to B. Later, for the second refinement, we talk of Rbc: the Retrieve from B to C.

10.1 Retrieve state

The domains of the B and A ‘world’ functions define the authentic purses.

AbstractBetween

AbWorld

BetweenWorld

dom ab/AuthPurse = dom con/AuthPurse

A balance and lost are related to 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.
CHAPTER 10. RAB

AbstractBetween
chosenLost : P PayDetails
chosenLost ⋐ maybeLost

∀ name : dom conAuthPurse •
  (abAuthPurse name).lost =
  sumValue((definitelyLost ⋃ chosenLost)
  ⋂ {pd : PayDetails | pd.from = name})
  ∧ (abAuthPurse name).balance =
  (conAuthPurse name).balance
  + sumValue((maybeLost \ chosenLost)
  ⋂ {pd : PayDetails | pd.to = name})

The predicate links the $B$ and $A$ values:

- 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 $A$ balance of a purse is its $B$ 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:

$\text{Rab} \equiv \exists chosenLost : P PayDetails \cdot RabCl$

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

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

2We 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 $pd$.

$RabClPd$

$pdThis : PayDetails$

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

- $\forall name : dom conAuthPurse \mid name \notin \{pdThis/from, pdThis/to\}$
  purses not involved in the $pdThis$ transaction.
- $\forall name : dom conAuthPurse \mid name \in \{pdThis/from, pdThis/to\}$
  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, $\text{OtherPursesRab}$, which we include with the remaining part of the predicate.

$\text{OtherPursesRab}$

AbstractBetween
chosenLost : P PayDetails
pdThis : PayDetails

∀ name : dom conAuthPurse \mid name \notin \{pdThis/from, pdThis/to\}
  •
  (abAuthPurse name).lost =
  sumValue((definitelyLost ⋃ chosenLost)
  ⋂ {pd : PayDetails | pd/from = name})
  ∧ (abAuthPurse name).balance =
  (conAuthPurse name).balance
  + sumValue((maybeLost \ chosenLost)
  ⋂ {pd : PayDetails | pd/to = name})

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

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

- **RabWillBeLostClPd** : \(\text{pdThis} \in \text{chosenLost} \) halfway 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 null). Since \(\text{chosenLost} \subseteq \text{maybeLost} \) refers only to authentic purses, we know that \(\{\text{pdThis}.\text{from}, \text{pdThis}.\text{to}\} \subseteq \text{dom} \text{conAuthPurse} \), and so the remaining quantifier is reduced to these two cases.

- **RabHas BeenLostClPd** : \(\text{pdThis} \in \text{definitelyLost} \) halfway through a transaction that has lost the value (the to purse has already moved on). Since \(\text{definitelyLost} \) refers only to authentic purses, we know that \(\{\text{pdThis}.\text{from}, \text{pdThis}.\text{to}\} \subseteq \text{dom} \text{conAuthPurse} \), and so the remaining quantifier is reduced to these two cases.

- **RabEndClPd** : \(\text{pdThis} \notin \text{definitelyLost} \cup \text{maybeLost} \) At the beginning or end of a transaction, so there is no non-determinism in the lost or balance components. A general \(\text{pdThis} \) may refer to non-authentic purses, so the quantifier is reduced no further.

In the later proofs of operations that change purse status (\text{Abort}, \text{Req}, \text{Val} and \text{Ack}), we argue how the relevant \(pd\) moves in and out of the sets \(\text{maybeLost}\) and \(\text{definitelyLost}\), 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.
CHAPTER 10. RAB

RabWillBeLostClPd

AbstractBetween
chosenLost : P PayDetails
pdThis : PayDetails

chosenLost ⊆ maybeLost

pdThis ∈ chosenLost

(abAuthPurse pdThis.from).lost =
  pdThis.value
  + sumValue((definitelyLost ⊆ chosenLost)
            ∩ { pd : PayDetails | pd.from = pdThis.from })

(abAuthPurse pdThis.to).lost =
  sumValue((definitelyLost ⊆ chosenLost)
            ∩ { pd : PayDetails | pd.to = pdThis.to })

∀ name : {pdThis.from, pdThis.to} •

(abAuthPurse name).balance =
  (conAuthPurse name).balance
  + sumValue((maybeLost \ chosenLost)
            ∩ { pd : PayDetails | pd.to = name })

OtherPursesRab

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

RabHasBeenLostClPd

AbstractBetween
chosenLost : P PayDetails
pdThis : PayDetails

chosenLost ⊆ maybeLost

pdThis ∈ definitelyLost

(abAuthPurse pdThis.from).lost =
  pdThis.value
  + sumValue((definitelyLost ⊆ chosenLost)
            ∩ { pd : PayDetails | pd.from = pdThis.from })

(abAuthPurse pdThis.to).lost =
  sumValue((definitelyLost ⊆ chosenLost)
            ∩ { pd : PayDetails | pd.to = pdThis.to })

∀ name : {pdThis.from, pdThis.to} •

(abAuthPurse name).balance =
  (conAuthPurse name).balance
  + sumValue((maybeLost \ chosenLost)
            ∩ { pd : PayDetails | pd.to = name })

OtherPursesRab

In the HasBeenLost case, pdThis is definitely lost, so its value has to be added back into the from purse’s lost component.
CHAPTER 10. RAB

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 identity³:

\[ \text{RabClPd} \equiv (\forall \text{pdThis} : \text{PayDetails} \quad \text{RabClPd}) \]

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.

³Used in: Req check operation, splitting into four cases, section 18.6.

10.1.5 Quantified forms

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

\[ \text{RabCl} \vdash \text{RabCl} \equiv (\forall \text{pdThis} : \text{PayDetails} \quad \text{RabClPd}) \]

and

\[ \text{RabCl} \vdash \text{RabCl} \equiv (\exists \text{pdThis} : \text{PayDetails} \quad \text{RabClPd}) \]

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 \( N \) called \( X \). Then it is certainly true that

\[ \exists x : X \quad x = X \setminus \{x\} \cup \{x\} \]

and also

\[ \forall x : X \quad x = X \setminus \{x\} \cup \{x\} \]

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):

\[ \text{RabOkay} \equiv \text{RabOkayClPd} \setminus (\text{pdThis}, \text{chosenLost}) \]
\[ \text{RabWillBeLost} \equiv \text{RabWillBeLostClPd} \setminus (\text{pdThis}, \text{chosenLost}) \]
\[ \text{RabHasBeenLost} \equiv \text{RabHasBeenLostClPd} \setminus (\text{pdThis}, \text{chosenLost}) \]
\[ \text{RabEnd} \equiv \text{RabEndClPd} \setminus (\text{pdThis}, \text{chosenLost}) \]

⁴Used in: lemma ‘deterministic’, exposing pdThis (twice), section 14.4.3.
10.2 Retrieve inputs

Each \(A\) operation has the same type of input, an \(A\text{IN}\). Each \(B\) operation has the same type of input, a NAME and a MESSAGE. The input part of the retrieve captures the relationship between these \(A\) and \(B\) inputs.

\[ \text{RabIn} \approx \text{BetwInitIn}[a?/g?] \]

The \(B\) inputs are related to \(A\) inputs in the following manner:

RI-1 Req: the \(A\) transfer details are in the req
RI-2 All other \(B\) inputs: the \(A\) input is \(a\text{NullIn}\).

10.3 Retrieve outputs

The output retrieve is particularly simple: all \(B\) outputs retrieve to the single \(A\) output.

\[ \text{RabOut} \approx \text{BetwFinOut}[a!/g!] \]

11. \(A\) to \(B\) 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,

\[ \text{BetweenInitState} \land \text{Rab'} \vdash \text{AbInitState} \]

\[ \text{BetwInitIn}; \text{Rab} \vdash \text{AbInitIn} \]

11.2 Proof of initial state

We successively thin the hypothesis to expose the consequent.

\[ \text{BetweenWorldInit} \land \text{Rab'} \vdash \text{AbInitState} \]

\[ \Rightarrow \text{Rab'} \text{ [hyp]} \]

\[ \Rightarrow \text{AbWorld'} \text{ [thin]} \]

\[ \Rightarrow \text{AbInitState} \text{ [defn AbInitState]} \]

11.3 Proof of initial inputs

Expand \(\text{RabIn}\) and \(\text{AbInitIn}\).

\[ \text{BetwInitIn}; \text{BetwInitIn}[a?/g?] \vdash a? = g? \]
BetwInitIn defines \( g? \) as a total function of \((m?, name?)\); call it \( f \). Thin.

\[
g?, a?: AIN | \exists f : MESSAGE \times NAME \rightarrow AIN \bullet \\
\forall m : MESSAGE, n : NAME \bullet \\
g? = f(m, n) \wedge a? = f(m, n)
\]

\[\vdash a? = g?\]

Simplify and thin.

\[
g?, a?: AIN | g? = a? \vdash a? = g?
\]

\(\square\)

### Chapter 12

**A to 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,

\[
\text{BetwFinOut} \vdash \exists a' : AOUT \bullet RabOut \wedge AbFinOut
\]

\[
\text{BetwFinState} \vdash \exists AbWorld \bullet Rab \wedge AbFinState
\]

This proof obligation is summarised in figure 12.1.

![Figure 12.1: Backwards rules finalisation proof obligation](image-url)
CHAPTER 12. A TO B FINALISATION

12.2 Output proof

Expand RabOut and AbFinOut.

\[ \text{BetwFinOut} \vdash \exists \alpha : \text{AOUT} \bullet \text{BetwFinOut}[\alpha! / \gamma!] \land \alpha! = \gamma! \]

[one point] away the \( \alpha! \) in the consequent

\[ \text{BetwFinOut} \vdash \text{BetwFinOut}[\gamma! / \gamma!] \]

\[ \text{BetwFinOut} \]

12.3 State proof

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

\[ \text{BetwFinState}, \text{AbWorld} \mid \text{abAuthPurse} = \text{gAuthPurse} \]

\[ \vdash \exists \text{AbWorld} \bullet \text{Rab} \land \text{AbFinState} \]

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:

\[ \text{BetwFinState}, \text{AbWorld} \mid \text{abAuthPurse} = \text{gAuthPurse} \]

\[ \vdash \text{Rab} \land \text{AbFinState} \]

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

12.3.1 Case AbFinState

\[ \text{BetwFinState}, \text{AbWorld} \mid \text{abAuthPurse} = \text{gAuthPurse} \rightarrow \text{AbFinState} \]

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

12.3.2 Case Rab

We expand out Rab into its conjuncts:

\[ \text{BetwFinState}, \text{AbWorld} \mid \text{abAuthPurse} = \text{gAuthPurse} \rightarrow \text{Rab} \]

12.3. STATE PROOF

Retrieve of equality

We have the equation

\[ \text{dom abAuthPurse} = \text{dom conAuthPurse} \]

which can be shown from the equality of \( g \cdot \text{AuthPurse} \) and \( \text{conAuthPurse} \) in BetwFinState, and between \( g \cdot \text{AuthPurse} \) and \( ab \cdot \text{AuthPurse} \) 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 equality between global and abstract in the hypothesis.

12.3.2 Case Rab

\[ \text{BetwFinState}, \text{AbWorld} \mid \text{abAuthPurse} = \text{gAuthPurse} \rightarrow \text{Rab} \]

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

\[ \text{AuxWorld}, \text{AbWorld} \mid \forall \text{name} : \text{dom conAuthPurse} \bullet \]

\[ \left( (\text{abAuthPurse name}).\text{lost} = \right. \]

\[ \text{sumValue}(\left(\text{definitelyLost} \cup \text{maybeLost}\right)) \]

\[ \land \left\{ (\text{pd} : \text{PayDetails} \mid \text{pd}.\text{from} = \text{name}) \right\} \]

\[ \land \left( (\text{abAuthPurse name}).\text{balance} = \right. \]

\[ \left( \text{conAuthPurse name}.\text{balance} + \text{sumValue}(\left(\text{maybeLost} \setminus \text{chosenLost}\right)) \right. \]

\[ \land \left\{ (\text{pd} : \text{PayDetails} \mid \text{pd}.\text{to} = \text{name}) \right\} \]

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

\[
\begin{align*}
\text{AuxWorld}; \text{AbWorld}; \text{chosenLost} : \mathbb{P} \text{PayDetails} \mid \\
\text{chosenLost} = \text{maybeLost} \\
\land (\forall \text{name} : \text{dom conAuthPurse} \cdot \\
\quad \text{(abAuthPurse name).lost} = \\
\quad \text{sumValue(\text{definitelyLost} \cup \text{maybeLost})} \\
\quad \land \{ \text{pd : PayDetails} \mid \text{pd.from} = \text{name} \}) \\
\land (\text{abAuthPurse name).balance} \\
\quad = (\text{conAuthPurse name).balance})
\end{align*}
\]

\[
\vdash (\forall \text{name} : \text{dom conAuthPurse} \cdot \\
\quad (\text{abAuthPurse name).lost} = \\
\quad \text{sumValue(\text{definitelyLost} \cup \text{maybeLost})} \\
\quad \land \{ \text{pd : PayDetails} \mid \text{pd.from} = \text{name} \}) \\
\land (\text{abAuthPurse name).balance} \\
\quad = (\text{conAuthPurse name).balance})
\]

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

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.
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 $AOp \subset BOpFull = BOp_1 \lor BOp_2 \lor \ldots$ is the corresponding refinement, is:

$BOpFull; \mathcal{R}ab'; \mathcal{R}abOut \leftarrow \exists \mathcal{A}bWorld; a? : \mathcal{A}In \land \mathcal{R}ab \land \mathcal{R}abIn \land AOp$

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 $(\mathcal{B}, \mathcal{B}', \mathcal{A}')$, we have to find an $\mathcal{A}$ that ensures the existence of an upper path; it does not have to be the same $\mathcal{A}$ 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 $\textit{exists-pd}$, $\textit{exists-chosenLost}$, and $\textit{check-operation}$ (with the introduction of two ar-
14.2 Lemma ‘multiple refinement’

In most cases of $AOp$, the corresponding $BOpFull$ is a disjunction of many individual $B$ 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 $B$ operations.

14.3 Lemma ‘ignore’: separating the branches

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

$$BOpOkay \equiv \exists \Delta \text{ComParse} \bullet \Phi BOp \land BOpParseOkay$$

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.

$$\text{Abort; Rab'; RabOut} \vdash \exists \text{AbWorld; a?; Ain} \bullet \text{Rab} \land \text{RabIn} \land AOp$$

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

$$\text{Ignore; Rab'; RabOut} \vdash \exists \text{AbWorld; a?; Ain} \bullet \text{Rab} \land \text{RabIn} \land \text{AbIgnore}$$

$$\text{Abort; Rab'; RabOut} \vdash \exists \text{AbWorld; a?; Ain} \bullet \text{Rab} \land \text{RabIn} \land \text{AbIgnore}$$

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,

$$BOpOkay; Rab'; RabOut \vdash \exists \text{AbWorld; a?; Ain} \bullet \text{Rab} \land \text{RabIn} \land AOp$$

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 \( \text{Rab}n \). Since \( \text{Rab}n \) is functional from \( m? \) and \( \text{name} \) 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 [conseq exists] rule.

We note that \( \text{Rab}n \) in the consequent is independent of the choice of \( \text{AbWorld} \), so can be pulled out of that quantifier.

\[
\begin{align*}
\text{BOpOkay}; \text{RabOut}; a' & : \text{AIN} \mid \text{Rab}n \\
\vdash & \text{Rab}n \land (\exists \text{AbWorld} \cdot \text{Rab} \land \text{AOp})
\end{align*}
\]

We split the proof into two on the conjunction in the consequent [conseq], one for \( \text{Rab}n \), one for \( \exists \text{AbWorld} \cdot \text{Rab} \land \text{AOp} \).

\( \text{Rab}n \) is trivially satisfied by this choice of \( a' \) in the hypothesis. The declaration of \( a' \) in \( \text{Rab}n \) allows us to drop the explicit declaration in the hypothesis, giving

\[
\begin{align*}
\text{BOpOkay}; \text{RabOut}; a' & : \text{AIN} \mid \text{Rab}n \\
\vdash & \exists \text{AbWorld} \cdot \text{Rab} \land \text{AOp}
\end{align*}
\]

14.4.2 Cutting in \( \Delta \text{ConPurse} \)

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

\[
\begin{align*}
(\exists \Delta \text{ConPurse} \cdot \Phi \text{Bo} \land \text{BOpPurseOkay}); & \\
\text{RabOut}; a' & : \text{AIN} \mid \text{Rab}n; \Delta \text{ConPurse} \mid \\
\Phi \text{Bo} & \land \text{BOpPurseOkay} \\
\vdash & \exists \text{AbWorld} \cdot \text{Rab} \land \text{AOp}
\end{align*}
\]

We rearrange the hypothesis, moving \( \Phi \text{Bo} \) and \( \text{BOpPurseOkay} \) from the predicate part to the declaration part. Since \( \Phi \text{Bo} \) declares \( \Delta \text{ConPurse} \), we remove the latter. We [thin] the hypothesis of the expanded definition of \( \text{BOpOkay} \).

\[
\begin{align*}
\Phi \text{Bo}; & \text{BOpPurseOkay}; \text{RabOut}; a' : \text{AIN} \mid \text{Rab}n \\
\vdash & \exists \text{AbWorld} \cdot \text{Rab} \land \text{AOp}
\end{align*}
\]

14.4.3 Exposing \( \text{chosenLost} \) and \( \text{pdThis} \)

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

We replace \( \text{Rab}' \) with the quantified form of \( \text{RabC}' \) (section 10.1.2), giving

\[
\Phi \text{Bo}; \text{BOpPurseOkay}; \text{RabOut}; \\
(\exists \text{chosenLost} : \mathcal{P}; \text{PayDetails} \cdot \text{RabC}'); \text{Rab}n \\
\vdash & \exists \text{AbWorld} \cdot \text{Rab} \land \text{AOp}
\]

We now use \([\text{hyp exists}] \) to remove the quantification, giving us

\[
\Phi \text{Bo}; \text{BOpPurseOkay}; \text{RabOut}; \text{RabC}; \text{Rab}n \\
\vdash & \exists \text{AbWorld} \cdot \text{Rab} \land \text{AOp}
\]

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

This generates a non-trivial lemma, \( \text{exists-pd} \), to be proved in each specific case, as

\[
\Phi \text{Bo}; \text{BOpPurseOkay}; \text{RabOut}; \text{RabC}; \text{Rab}n \\
\vdash & \exists \text{pdThis} : \mathcal{P}, \text{PayDetails} \cdot \mathcal{P}
\]

and leaves our proof obligation as

\[
\Phi \text{Bo}; \text{BOpPurseOkay}; \text{RabOut}; \text{RabC}; \text{Rab}n; \text{pdThis} : \mathcal{P}, \text{PayDetails} \cdot \mathcal{P} \\
\vdash & \exists \text{AbWorld} \cdot \text{Rab} \land \text{AOp}
\]

In the hypothesis we rewrite \( \text{RabC} \) as the universally quantified form of \( \text{RabCIPdf} \) (section 10.1.5).

\[
\Phi \text{Bo}; \text{BOpPurseOkay}; \text{RabOut}; \\
(\forall \text{pdThis} : \mathcal{P}; \text{PayDetails} \cdot \text{RabCIPdf}'); \text{Rab}n; \text{pdThis} : \mathcal{P}, \text{PayDetails} \cdot \mathcal{P} \\
\vdash & \exists \text{AbWorld} \cdot \text{Rab} \land \text{AOp}
\]
Rather than hypothesising this is true for all \(pdThis\)'s, we choose a particular value in the quantification. (This is valid, \([\text{hyp uni}]\), 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

\[
\Phi \text{ BOp}; \text{ BOpPurseOkay}; \text{ RabOut}; \text{ RabCIPd}[pdThis/pdThis']; \text{ RabIn};
\]

\[
pdThis : \text{PayDetails} \mid
\]

\[
\not\vdash
\]

\[
\exists AbWorld \bullet Rab \land AOp
\]

The declaration in \(RabCIPd'\) allows us to drop the explicit declaration of \(pdThis\). So we rewrite this more simply as

\[
\Phi \text{ BOp}; \text{ BOpPurseOkay}; \text{ RabOut}; \text{ RabCIPd}[pdThis/pdThis']; \text{ RabIn} \mid
\]

\[
p \not\vdash
\]

\[
\exists AbWorld \bullet Rab \land AOp
\]

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

\[
\Phi \text{ BOp}; \text{ BOpPurseOkay}; \text{ RabOut}; \text{ RabCIPd}[pdThis/pdThis']; \text{ RabIn} \mid
\]

\[
p \not\vdash
\]

\[
\exists AbWorld \bullet
\]

\[
(\exists chosenLost : \text{PayDetails} \land \text{PayDetails} \bullet
\]

\[
\bullet RabCIPd(pd/pdThis))
\]

\[
\land AOp
\]

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 \(Q\), as we did with \(pdThis\). This predicate is instantiated to some specific predicate each time this general manipu-
This leaves:

\[ \Phi \text{BOp; BOpPurseOkay; RabOut; RabClPd}'[pdThis]/pdThis']; RabIn; chosenLost : P PayDetails | P \land Q \land chosenLost \subseteq maybeLost \]
\[ \vdash \exists AbWorld \bullet (\exists chosenLost : P PayDetails \bullet Q \land chosenLost \subseteq maybeLost \land RabClPd) \land AOp \]

We remove the existential quantification using the [consq exists] for \textit{chosenLost}:

\[ \Phi \text{BOp; BOpPurseOkay; RabOut; RabClPd}'[pdThis]/pdThis']; RabIn; chosenLost : P PayDetails | P \land Q \land chosenLost \subseteq maybeLost \]
\[ \vdash \exists AbWorld \bullet RabClPd \land AOp \]

We break this into two parts, separating the two retrieves in the consequent from \textit{AOp}. We then prove each part.

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

We are left with

\[ \Phi \text{BOp; BOpPurseOkay; RabOut; RabClPd}'[pdThis]/pdThis']; AbWorld; RabClPd; RabIn; chosenLost : P PayDetails | P \land Q \land chosenLost \subseteq maybeLost \]
\[ \vdash RabClPd \land AOp \]

We discharge the retrieves in the consequent directly from the hypothesis, and remove \textit{chosenLost} and \textit{chosenLost} \subseteq \textit{maybeLost} as these already occur in \textit{RabClPd}, leaving

\[ \Phi \text{BOp; BOpPurseOkay; RabOut; RabClPd}'[pdThis]/pdThis']; AbWorld; RabClPd; RabIn | P \land Q \]
\[ \vdash AOp \]

\[ \framebox{14.4.3} \]

14.4. Lemma ‘Deterministic’: Simplifying the Okay Branch

14.4.4 The existence of \textit{AbWorld}

We have to prove the side condition generated when we cut in an \textit{AbWorld} (above).

\[ \Phi \text{BOp; BOpPurseOkay; RabOut; RabClPd}'[pdThis]/pdThis']; RabIn; chosenLost : P PayDetails | P \land Q \land chosenLost \subseteq maybeLost \]
\[ \vdash \exists AbWorld \bullet RabClPd \]

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 \textit{BetweenWorld} (from \textit{BOp}), a \textit{pdThis} and a \textit{chosenLost} such that the constraint \textit{chosenLost} \subseteq \textit{maybeLost} holds. This is sufficient to invoke the lemma.

\[ \framebox{14.4.4} \]

14.4.5 Statement of lemma ‘deterministic’

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

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

\begin{itemize}
  \item \textbf{exists-pd:}
    \[ \Phi \text{BOp; BOpPurseOkay; RabOut; RabCT}; RabIn \]
    \[ \vdash \exists pdThis : PayDetails \bullet P \]
  \item \textbf{exists-chosenLost:}
    \[ \Phi \text{BOp; BOpPurseOkay; RabOut; RabClPd}'[pdThis]/pdThis']; RabIn | P \]
    \[ \vdash \exists chosenLost : P PayDetails \bullet Q \land chosenLost \subseteq \textit{maybeLost} \]
  \item \textbf{check-operation:}
    \[ \Phi \text{BOp; BOpPurseOkay; RabOut; RabClPd}'[pdThis]/pdThis']; AbWorld; RabClPd; RabIn | P \land Q \]
    \[ \vdash AOp \]
\end{itemize}
CHAPTER 14. A TO B LEMMAS

14.4

14.5 Lemma 'lost unchanged'

Many operations do not change maybeLost or definitelyLost. We call a general such operation \( BOp \) \( Lost \).

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

\[
P \iff true
\]

\[
Q \iff chosenLost = chosenLost'
\]

leaving the remaining check-operation proof obligation as

\[
\Phi BOp; BOp \in Lost\text{-}PurseOkay; RabOut; RabClPd' [pdThis/pdThis']; AbWorld; RabClPd; RabIn | chosenLost = chosenLost'
\]

\[
\wedge maybeLost = maybeLost'
\]

\[
\wedge definitelyLost = definitelyLost'
\]

\[
\vdash AOp
\]

14.5.1 Proof

We add the hypotheses \( maybeLost = maybeLost' \) and \( definitelyLost = definitelyLost' \) to the proof obligations for these \( BOp \in Lost \) operations.

\( \exists pd \)

\[
\Phi BOp; BOp \in Lost\text{-}PurseOkay; RabOut; RabClT'; RabIn | maybeLost = maybeLost'
\]

\[
\wedge definitelyLost = definitelyLost'
\]

\[
\vdash \exists pdThis : PayDetails \iff true
\]

This is trivially true.

14.5.2 Sufficient conditions for invoking lemma 'lost unchanged'

Since \( \Phi BOp \) 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 \( BOpplg \). We note that all \( BOpplg \) operations have the properties:

- \( BOpplg \) is a promoted operation, and thus alters only one concrete purse.
- It has the form

\[
\exists \Delta ConPurse \iff \Phi BOp \wedge BOpplgPurse
\]
• 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.

\[ \forall \text{BOpIgPurse} \bullet \text{nextSeqNo} \leq \text{nextSeqNo}' \]

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 \( \text{exists-pd, exists-chosenLost} \) and check-operation:

\[ \Phi \text{BOp}; \text{BOpIgPurse}; \text{RabOut}; \text{RabClPd}'[\text{pdThis}/\text{pdThis'}]; \text{AbWorld}; \text{RabClPd}; \text{RabIn} | P \land Q \]

\[ \vdash \text{AbOp} \land \text{abAuthPurse}' = \text{abAuthPurse} \]

We prove this component by component. From \( \Phi \text{BOp} \) 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 balance and lost. We use this with [consq conj] to reduce our proof requirement to the following:

\[ \forall n : \text{dom abAuthPurse} \bullet (\text{abAuthPurse}' n).\text{lost} = (\text{abAuthPurse} n).\text{lost} \]

\[ \land (\text{abAuthPurse}' n).\text{balance} = (\text{abAuthPurse} n).\text{balance} \]

**Proof:**

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

1 Used in ‘ignore’, 14.7.2.

14.6. LEMMA ‘ABIGNORE’: OPERATIONS THAT REFINE ABIGNORE

these in the hypothesis.

\[ \Phi \text{BOp}; \text{BOpIgPurse}; \text{RabOut}; \text{RabClPd}'[\text{pdThis}/\text{pdThis'}]; \text{AbWorld}; \text{RabClPd}; \text{RabIn} | P \land Q \]

\[ \land \text{name}' = \text{name} \]

\[ \land \text{nextSeqNo}' \geq \text{nextSeqNo} \]

\[ \vdash \text{AbOp} \land \text{abAuthPurse}' = \text{abAuthPurse} \]

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:

\[ \Phi \text{BOp}; \text{BOpIgPurse}; \text{RabOut}; \text{RabClPd}'[\text{pdThis}/\text{pdThis'}]; \text{AbWorld}; \text{RabClPd}; \text{RabIn} | P \land Q \]

\[ \land \text{name}' = \text{name} \]

\[ \land \text{nextSeqNo}' = \text{nextSeqNo} \]

\[ \vdash \forall n : \text{dom abAuthPurse} \bullet \text{abAuthPurse}' n = \text{abAuthPurse} n \]

We prove this component by component. From \( \Phi \text{BOp} \) 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 balance and lost. We use this with [consq conj] to reduce our proof requirement to the following:

\[ \Phi \text{BOp}; \text{BOpIgPurse}; \text{RabOut}; \text{RabClPd}'[\text{pdThis}/\text{pdThis'}]; \text{AbWorld}; \text{RabClPd}; \text{RabIn} | P \land Q \]

\[ \land \text{name}' = \text{name} \]

\[ \land \text{nextSeqNo}' = \text{nextSeqNo} \]

\[ \vdash \forall n : \text{dom abAuthPurse} \bullet (\text{abAuthPurse}' n).\text{balance} = (\text{abAuthPurse} n).\text{balance} \]

\[ \land (\text{abAuthPurse}' n).\text{lost} = (\text{abAuthPurse} n).\text{lost} \]

We then [thin] the hypothesis to get the following, which proves the AbIgnore
106 CHAPTER 14. A TO B LEMMAS

lemma.

\( \Phi; \text{B\text{Op\text{Ig\text{Purse}}}'; \text{Rab\text{Cl\text{Pd'}}} \mid \text{pdThis'/pdThis'}\}; \text{Ab\text{World}; Rab\text{Cl\text{Pd'}}} \mid P \land Q \)

\( \forall n : \text{dom ab\text{Auth\text{Purse}}} \bullet (\text{ab\text{Auth\text{Purse'}}} n).\text{balance} = (\text{ab\text{Auth\text{Purse}}} n).\text{balance} \land (\text{ab\text{Auth\text{Purse'}}} n).\text{lost} = (\text{ab\text{Auth\text{Purse}}} n).\text{lost} \)

\( \text{■} 14.6 \)

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 B\text{Op\text{Okay}} in the hypothesis, instead of \( \Phi; \text{B\text{Op\text{Ig\text{Purse}}}'; \text{Rab\text{Cl\text{Pd'}}} \mid \text{pdThis'/pdThis'}\}; \text{Ab\text{World}; Rab\text{Cl\text{Pd'}}} \mid P \land Q \)

\( \forall n : \text{dom ab\text{Auth\text{Purse}}} \bullet (\text{ab\text{Auth\text{Purse'}}} n).\text{balance} = (\text{ab\text{Auth\text{Purse}}} n).\text{balance} \land (\text{ab\text{Auth\text{Purse'}}} n).\text{lost} = (\text{ab\text{Auth\text{Purse}}} n).\text{lost} \)

\( \text{■} 14.6 \)

14.7.2 check-operation-ignore

Ignore; Rab\text{Cl\text{Pd'}} \mid \text{pdThis'/pdThis'}\}; \text{Ab\text{World}; Rab\text{Cl\text{Pd'}}} \mid \text{chosenLost'} = \text{chosenLost} \land \text{maybeLost'} = \text{maybeLost} \land \text{definitelyLost'} = \text{definitelyLost}

\( \forall n : \text{dom ab\text{Auth\text{Purse}}} \bullet (\text{ab\text{Auth\text{Purse'}}} n).\text{balance} = (\text{ab\text{Auth\text{Purse}}} n).\text{balance} \land (\text{ab\text{Auth\text{Purse'}}} n).\text{lost} = (\text{ab\text{Auth\text{Purse}}} n).\text{lost} \)

\( \text{■} 14.6 \)

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.

Abort; Rab'; RabOut ⊢ \exists \text{Ab\text{World}; ab\text{In; AIN} • Rab ∧ Rab\text{In} ∧ Ab\text{Ignore}}

\( \text{■} \)

Proof:

Abort is written as a disjunction between Ignore and a promoted Ab\text{Abort-PurseOkay}. 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 \( P \) and \( Q \).

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

\( P ⇔ \text{pdThis} = \text{pdAuth} \)

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

\( \text{Used in proof of lemma abort, 14.9} \)
unchanged (because pdThis was not in chosenLost to start with) or is unchanged because there was no transaction to abort. Hence

\[ Q \iff (pdThis \in maybeLost \land chosenLost = chosenLost' \cup \{pdThis\}) \lor (pdThis \notin maybeLost \land status = eaFrom \land chosenLost = chosenLost') \lor (status = eaFrom \land chosenLost = chosenLost') \]

14.8.2 \textit{exists-pd}

The unpromoted operation \textit{AbortPurseOkay} is incomplete. The output, \( m! = \bot \), is not provided until promotion.

\[ \Phi \textit{BOp}; \textit{AbortPurseOkay}; \textit{RabOut}; \textit{RabCl'} ; \textit{RabIn} \mid m! = \bot \]

\[ 3 \textit{pdThis} : \textit{PayDetails} \mid \textit{pdThis} = \textit{pdAuth} \]

This is immediate by the one point rule.

\[ \blacksquare \]

14.8.3 \textit{Three cases}

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

- \textbf{Case 1: aborted transaction in ‘limbo’}: The aborting purse is the \textit{to} purse in \textit{epv}; the corresponding \textit{from} purse is in \textit{epa} or has logged. Hence aborting the transaction will definitely lose the value.

  \[ pdThis \in maybeLost \]

- \textbf{Case 2: aborted transaction not in ‘limbo’}: The aborting purse is not the \textit{to} purse in \textit{epv}, or the corresponding \textit{from} purse is not in \textit{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.

  \[ pdThis \notin maybeLost \land status = eaFrom \]

- \textbf{Case 3: no transaction to abort}: The aborting purse is in \textit{eaFrom}, so has no defined transaction. Nothing is aborted, so no value is lost.

  \[ status = eaFrom \]

\section*{Case 1: old transaction in limbo}

\[ pdThis \in (\text{fromInEpa} \cup \text{fromLogged}) \cap \text{toInEpv} \]

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

\begin{itemize}
  \item \textbf{Behaviour of fromInEpa and fromLogged}: \textit{pdThis} is in \textit{toInEpv} (by our case assumption), so the only purse undergoing any change (\textit{name}?) is the \textit{to} purse; hence there can be no change to the \textit{status} or \textit{logs} of any \textit{from} purse. Hence

    \[ \text{fromInEpa} = \text{fromInEpa}' \]
    \[ \text{fromLogged} = \text{fromLogged}' \]

  \item \textbf{Behaviour of toInEpv}: \textit{pdThis} is in \textit{toInEpv} (by our case assumption); \textit{pdThis} is not in \textit{toInEpv'} (\textit{Abort} puts the purse into \textit{eaFrom}); all other purses and transactions remain unchanged. So

    \[ \text{toInEpv} = \text{toInEpv}' \cup \{pdThis\} \]

  \item \textbf{Behaviour of toLogged}: \textit{pdThis} is not in \textit{toLogged} (using lemma ‘notLogged-AndIn’ with \textit{pdThis} ∈ \textit{toInEpv}); \textit{pdThis} is in \textit{toLogged'} (the purse makes a \textit{to} log when it aborts from \textit{epv}); all other purses and transactions remain unchanged. So

    \[ \text{toLogged} = \text{toLogged}' \setminus \{pdThis\} \]
\end{itemize}
CHAPTER 14. A TO B LEMMAS

Behaviour of $\text{definitelyLost}$

$\text{definitelyLost} \ni \ (\text{fromLogged} \cup \text{fromEpa})$  \hspace{1cm} \text{[defn $\text{definitelyLost}$]}
$\text{toLogged} \cap (\text{fromLogged} \cup \text{fromEpa})$  \hspace{1cm} \text{[above]}
$\text{toLogged} \cap (\text{fromLogged} \cup \text{fromEpa}) \setminus \{\text{pdThis}\}$  \hspace{1cm} \text{[rearrange]}
$\text{definitelyLost} \setminus \{\text{pdThis}\}$

Behaviour of $\text{maybeLost}$

$\text{maybeLost} \ni \ (\text{fromInEpa} \cup \text{fromLogged}) \cap \text{toInEpv}$  \hspace{1cm} \text{[defn $\text{maybeLost}$]}
$\text{toInEpv} \cup \{\text{pdThis}\}$  \hspace{1cm} \text{[case assumption]}
$\text{maybeLost} \cup \{\text{pdThis}\}$

Case 2: old transaction not in limbo

$\text{pdThis} \notin (\text{fromInEpa} \cup \text{fromLogged}) \cap \text{toInEpv}$ \hspace{1cm} \text{status} = \text{eaFrom}$

We argue that the transaction $\text{pdThis}$ is not moved into or out of $\text{maybeLost}$ or $\text{definitelyLost}$ by this case of the Abort operation.

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

So

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

14.8. ABORT REFINES ABIGNORE

Behaviour of $\text{definitelyLost}$ The cases allowed by our case assumption are:

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

So

$\text{definitelyLost'} = \text{definitelyLost}$

Behaviour of $\text{maybeLost}$ The cases allowed by our case assumption are:

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

So

$\text{maybeLost'} = \text{maybeLost}$

Case 3: no transaction to abort

$\text{status} = \text{eaFrom}$

From $\text{AbortPurseOkay}$, no purses change state and no logs are written. Therefore, $\text{definitelyLost}$ and $\text{maybeLost}$ don’t change.

$\text{definitelyLost'} = \text{definitelyLost}$
$\text{maybeLost'} = \text{maybeLost}$
14.8.4 exists-chosenLost

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

ΦBOp; AbortPurseOkay; RabOut; RabClPd[\{pdThis\}]\[rd\]RabIn | 
   m\textsuperscript{t} = \bot 
   ∧ pdThis = pdAuth
   ⊢ 
   \exists \text{chosenLost : P PayDetails} • 
   (pdThis \in \text{maybeLost} ∧ \text{chosenLost} = \text{chosenLost}' ∪ \{pdThis\})
   ∨ pdThis \notin \text{maybeLost} ∧ status = eaFrom
   ∧ \text{chosenLost} = \text{chosenLost}'
   ∨ status = eaFrom ∧ \text{chosenLost} = \text{chosenLost}')
   ∧ \text{chosenLost} \subseteq \text{maybeLost}

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

ΦBOp; AbortPurseOkay; RabOut; RabClPd[\{pdThis\}]\[rd\]RabIn | 
   m\textsuperscript{t} = \bot 
   ∧ pdThis = pdAuth
   ⊢ 
   pdThis \in \text{maybeLost}
   ∨ pdThis \notin \text{maybeLost} ∧ status = eaFrom
   ∨ status = eaFrom ∧ \text{chosenLost} = \text{chosenLost}')
   ∧ \text{chosenLost} \subseteq \text{maybeLost}

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

ΦBOp; AbortPurseOkay; RabOut; RabClPd[\{pdThis\}]\[rd\]RabIn | 
   m\textsuperscript{t} = \bot 
   ∧ pdThis = pdAuth
   ⊢ 
   pdThis \in \text{maybeLost}
   ∨ pdThis \notin \text{maybeLost} ∧ status = eaFrom
   ∨ status = eaFrom ∧ \text{chosenLost} = \text{chosenLost}')
   ∧ \text{chosenLost} \subseteq \text{maybeLost}

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

pdThis \in \text{maybeLost} ⇒
\text{chosenLost}' ∪ \{pdThis\} \subseteq \text{maybeLost}

This follows by:

\text{chosenLost}' ∪ \{pdThis\}
   \subseteq \text{maybeLost}' ∪ \{pdThis\} \hspace{1cm} \text{[hypothesis]}
   \subseteq \text{maybeLost} \hspace{1cm} \text{[previous argument for case 1]}

■ 14.8.4
Case 2: old transaction not in limbo

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

\[ pdThis \notin \text{maybeLost} \land \text{status} = \text{eaFrom} \Rightarrow \text{chosenLost}' \subseteq \text{maybeLost} \]

This follows by

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

\[ \blacksquare \text{14.8.4} \]

Case 3: no transaction to abort

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

\[ \text{status} = \text{eaFrom} \Rightarrow \text{chosenLost}' \subseteq \text{maybeLost} \]

This follows by

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

\[ \blacksquare \text{14.8.4} \]

\[ \blacksquare \text{14.8.4} \]

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`, `Read-ExceptionLog`, `ExceptionLogClear`).
**Lemma 14.5 (abort backward)** Where a concrete operation is written as a composition of \textit{AbortPurseOkay} and a simpler operation starting from \textit{eaFrom}, it is sufficient to prove that the promotion of the simpler operation alone refines the relevant abstract operation.

$$\exists \Delta \text{ConPurse} \land \Phi \land (\text{AbortPurseOkay} \land \text{BOpPurseEaFromOkay})\land \text{RabOut}$$

$$\exists \Delta \text{AbWorld} \land \text{AIN} \land \text{RabIn} \land \text{AOp}$$

**Proof**

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

  $$(\text{AbortOkay} \land \text{BOpEaFromOkay}) \land \text{RabOut}$$

  $$\exists \Delta \text{AbWorld} \land \text{AIN} \land \text{RabIn} \land \text{AOp}$$

- If \text{BOp1} refines \text{AOp1} and \text{BOp2} refines \text{AOp2}, then \text{BOp1} \land \text{BOp2} refines \text{AOp1} \land \text{AOp2} (invoke lemma ‘compose backward’, section C.9).

- Take \text{BOp1} = \text{AbortOkay}, \text{AOp1} = \text{AbIgnore}, and invoke lemma ‘Abort refines AbIgnore’ (section 14.8), to discharge this proof.

- Take \text{BOp2} = \text{BOpEaFromOkay}, \text{AOp2} = \text{AOp}, and note that we have that \text{BOp} refines \text{AOp} in the hypothesis.

- Note that \text{AbIgnore} \land \text{AOp} = \text{AOp}, to reduce this expression in the consequent.

**14.10 Summary of lemmas**

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

$$\text{BOp, Rab'}; \text{RabOut} \Rightarrow \exists \text{AbWorld}; \text{AIN} \land \text{RabIn} \land \text{AOp}$$

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

\text{AOp} and \text{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 \text{Ignore} or \text{Abort} branches of \text{BOp} need be proved once only for all \text{BOps}. These two branches are proved in lemmas later on, after further simplification for a general disjunct (\text{Ignore}, \text{Abort} or \text{Okay}) of \text{BOp}. This simplification starts with lemma ‘deterministic’ (section 14.4) which removes the \exists \text{AbWorld} in the consequent of the correctness obligation. In doing so, it requires us to prove three side-lemmas \text{exists-pd}, \text{exists-chosenLost}, check-operation. Lemma ‘lost unchanged’ (section 14.5) allows the side-lemmas \text{exists-pd} and \text{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 \text{AOp} is \text{AbIgnore}.

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

Where the \text{Okay} branch of an operation is composed of \text{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 \text{Ignore} or \text{Abort} branches of a \text{BOp} operation are dealt with by the lemmas. This leaves the \text{Okay} branch (if this contains an initial \text{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 \text{BOp}.
Correctness of \textit{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 \textit{Ignore} (in section 14.7), leaving the \textit{Okay} branch to be proven here.
- We use lemma ‘deterministic’ (section C.1) to reduce the proof obligation to the three cases $\text{exists-pd}$, $\text{exists-chosenLost}$, and \textit{check-operation}.
- Since this operation leaves the sets $\text{maybeLost}$ and $\text{definitelyLost}$ unchanged, we use lemma ‘lost unchanged’ (section C.2) to discharge the $\text{exists pd and exists chosenLost-obligations}$ automatically.
- Since this operation refines $\text{AbIgnore}$, we use lemma ‘$\text{AbIgnore}$’ (from section C.3) to simplify \textit{check-operation} to \textit{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 \textit{status} nor the exception log of the purse. \textit{Increase} includes $\Xi_{\text{ConPurseIncrease}}$, which says exactly that. We can therefore invoke lemma ‘Lost unchanged’.
15.3 **check-operation-ignore**

\* BOp; IncreasePurseOkay; RabOut; RabCIPD\{pdThis\}/pdThis';
\* AbWorld; RabCIPD; Rabin |
\* chosenLost' = chosenLost
\* \& maybeLost = maybeLost
\* \& definitelyLost = definitelyLost

\* ∃ n : dom.abAuthPurse •
\* (abAuthPurse' n).balance = (abAuthPurse n).balance
\* \& (abAuthPurse' n).lost = (abAuthPurse n).lost

**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 A operation into one for each individual B operation.

This chapter proves the 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.

\[ P \iff pdThis = (\text{conAuthPurse’ name}).pdAuth \]

\[ Q \iff chosenLost = chosenLost’ \]

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

\[ (\text{conAuthPurse name}).pdAuth.fromSeqNo > pdThis’.fromSeqNo \]

but we know these two pdAuths 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 toInEpa and toLogged can’t change. Hence

\[ toInEpa’ = toInEpa \]

\[ toLogged’ = toLogged \]

\[ fromInEpa’ = fromInEpa \]

\[ fromLogged’ = fromLogged \]

It follows that maybeLost is unchanged:

\[ maybeLost’ = toInEpa’ \cup (\text{fromInEpa’} \cup \text{fromLogged’}) \]

\[ = toInEpa \cup (\text{fromInEpa} \cup \text{fromLogged}) \]

\[ = maybeLost \]

16.4 exists-pd

Also, definitelyLost is unchanged:

\[ \text{definitelyLost’} \]

\[ = toLogged’ \cap (\text{fromInEpa’} \cup \text{fromLogged’}) \]

\[ = toLogged \cap (\text{fromInEpa} \cup \text{fromLogged}) \]

\[ = \text{definitelyLost} \]

16.5 exists-chosenLost

\[ \Phi BOp; \text{StartFromPurseEafromOkay}; \text{RabOut}; \text{RabCl’}; \text{RabIn} \]

\[ \exists \text{pdThis} : \text{PayDetails} \bullet \text{pdThis} = (\text{conAuthPurse’ name}).pdAuth \]

Proof:

We use the [one point] rule with the expression for pdThis in the quantifier.

\[ 16.4 \]

Proof:

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

\[ \Phi BOp; \text{StartFromPurseEafromOkay}; \text{RabOut}; \text{RabCl’’} [\text{pdThis’/pdThis’}’]; \text{RabIn} \]

\[ \exists \text{chosenLost’} : \text{PayDetails} \bullet \text{chosenLost’} = \text{chosenLost’} \]

\[ \bullet \text{chosenLost’} \subseteq \text{maybeLost’} \]

We then have

\[ \text{chosenLost’} \subseteq \text{maybeLost’} [\text{RabCl’’}] \]

\[ \subseteq \text{maybeLost [unchanging maybeLost]} \]
16.6 check-operation

\[ \Phi \text{Bop}, \text{StartFromPurseEafromOkay}; \text{RabCIPd}'[pdThis|pdThis']; \text{AbWorld}; \text{RabCIPd} | \]
\[ \text{pdThis} = \{\text{conAuthPurse'} name?, pdAuth} \]
\[ \land \text{chosenLost} = \text{chosenLost}' \]
\[ \vdash \forall n : \text{dom abAuthPurse} \]
\[ \text{(abAuthPurse')n}.\text{balance} = (\text{abAuthPurse}n).\text{balance} \]
\[ \land (\text{abAuthPurse')n}.\text{lost} = (\text{abAuthPurse}n).\text{lost} \]

**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 \( A \) operation into one for each individual \( B \) operation.

This chapter proves the \( B \) 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.

\[
P \iff pdThis = (\text{conAuthPurse' name?}).pdAuth
\]

\[
Q \iff chosenLost = chosenLost'
\]

17.3 Behaviour of \( \text{maybeLost} \) and \( \text{definitelyLost} \)

We argue that \( pdThis \) is not in any of the before sets \( \text{fromlnEpa} \), \( \text{fromLogged} \), \( \text{toInEpv} \), or \( \text{toLogged} \), where we have:

\( pdThis = (\text{conAuthPurse' name?}).pdAuth \).

\[
\text{(conAuthPurse' name?).nextSeqNo \quad \text{[defn. StartTo]}}
\]

\[
\Rightarrow (\text{conAuthPurse' name?).pdAuth.toSeqNo)
\]

\[
\Rightarrow \text{req pdThis } \not\in \text{ether} \quad \text{[BetweenWorld constraint B–2]}
\]

\[
\Rightarrow \text{pdThis } \not\in \text{fromlnEpa } \cup \text{fromLogged} \quad \text{[BetweenWorld constraint B–12]}
\]

\[
\land \text{pdThis } \not\in \text{toInEpv } \cup \text{toLogged} \quad \text{[BetweenWorld constraint B–10]}
\]

The operation moves one purse from \( eaFrom \) into \( epv \); no logs are written. Hence \( pdThis \) is in \( \text{toInEpv' \cup toInEpv} \), but not newly added to any of the other after sets.

So

\[
\text{toInEpv'} = \text{toInEpv } \cup \text{pdThis}
\]

\[
\text{toLogged'} = \text{toLogged}
\]

\[
\text{fromInEpa'} = \text{fromInEpa}
\]

\[
\text{fromLogged'} = \text{fromLogged}
\]

It follows that \( \text{maybeLost} \) is unchanged:

\[
\text{maybeLost'} = \text{maybeLost } \cup \text{pdThis} \quad \text{[unchanging maybeLost]}
\]

17.4 EXISTS-PD

Also, \( \text{definitelyLost} \) is unchanged:

\[
\begin{align*}
\text{definitelyLost'} &= \text{toLogged'} \cap (\text{fromlnEpa' } \cup \text{fromLogged'}) \\
&= \text{toLogged } \cap (\text{fromlnEpa } \cup \text{fromLogged}) \\
&= \text{definitelyLost}
\end{align*}
\]

17.5 EXISTS-chosenLost

\[
\Phi BOp; \text{StartToPurseEafromOkay}; \text{RabOut}; \text{RabCIP'} \{pdThis' / pdThis\}; \text{RabIn} \quad \vdash \exists \text{chosenLost} : P \text{PayDetails} \quad \text{chosenLost' } = \text{chosenLost' } \land \text{chosenLost } \subseteq \text{maybeLost}
\]

Proof:

We apply the \([\text{one point}]\) rule for \( \text{chosenLost} \) in the consequent to give

\[
\Phi BOp; \text{StartToPurseEafromOkay}; \text{RabOut}; \text{RabCIP'} \{pdThis' / pdThis\}; \text{RabIn} \quad \vdash \text{chosenLost' } = \text{maybeLost'} \quad \text{[RabCIP']} \]

\[
\subseteq \text{maybeLost } \subseteq \text{maybeLost} \quad \text{[unchanging maybeLost]}
\]

\[
\text{\[17.5\]}
\]
17.6 check-operation

ΦBOp; StartToParseEqfromOkay; RabClPd'[pdThis/pdThis'];
AbWorld; RabClPd 

pdThis = (comAuthParse' name?), pdAuth
∧ chosenLost = chosenLost'

⊢ ∀ n: dom abAuthPurse

(abAuthPurse' n).balance = (abAuthPurse n).balance
∧ (abAuthPurse' n).lost = (abAuthPurse n).lost

Proof:
From Rab, we have that lost is a function of definitelyLost ∪ chosenLost, which is unchanging, and that balance is a function of maybeLost \ 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 A operation into one for each individual B operation.

This chapter proves the 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

P ⇔ req m? = pdThis
Q ⇔ chosenLost = chosenLost' \ {pdThis}
18.3 Discussion

The correctness proof for \( \text{Req} \) is summarised in figure 18.1. There are three cases:

- **The to purse for the transaction is in \( \text{epv} \), and we choose that the transfer will succeed.**
  Before the operation, \( \text{pdThis} \notin \text{maybeLost} \cup \text{definitelyLost} \), and the appropriate retrieve is \( \text{RabEnd} \).
  After the operation, \( \text{pdThis} \in \text{maybeLost} \), and the appropriate retrieve is \( \text{RabOkay} \); the abstract operation is \( \text{AbTransferOkay} \).

- **The to purse is in \( \text{epv} \), and we choose the transfer will fail (the to purse will move out of \( \text{epv} \) before receiving the val).**
  Before, \( \text{pdThis} \notin \text{maybeLost} \cup \text{definitelyLost} \), and the appropriate retrieve is \( \text{RabEnd} \).
  After, \( \text{pdThis} \in \text{chosenLost} \), and the appropriate retrieve is \( \text{RabWillBeLost} \); the abstract operation is \( \text{AbTransferLost} \).

- **The to purse has already moved out of \( \text{epv} \), so will not receive the val: the transfer has failed.**
  Before, \( \text{pdThis} \notin \text{maybeLost} \cup \text{definitelyLost} \), and the appropriate retrieve is \( \text{RabEnd} \).
  After, \( \text{pdThis} \in \text{definitelyLost} \), and the appropriate retrieve is \( \text{RabHasBeenLost} \); the abstract operation is \( \text{AbTransferLost} \).

The following proof establishes that these are indeed the only cases, and that \( \text{ReqOkay} \) correctly refines \( \text{AbTransfer} \) in each case.

18.4. EXISTSD-PD

\[ \Phi \text{BOp}; \text{ReqPurseOkay}; \text{RabOut}; \text{RabCl}; \text{RabIn} \]
\[ \exists \text{pdThis} : \text{PayDetails} \cdot \text{req} \not= \text{pdThis} \]

**Proof:**
We discharge this by removing the existential for \( \text{pdThis} \) because we have an explicit equation for it, using the [one point] rule.

18.5. EXISTSC-CHOOSELOST

\[ \Phi \text{BOp}; \text{ReqPurseOkay}; \text{RabOut}; \text{RabCl}\{\text{pdThis}/\text{pdThis}\}; \text{RabIn} \mid \text{req} \not= \text{pdThis} \]

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

\[ \text{chosenLost} \subseteq \text{maybeLost} \]
\[ \text{chosenLost} \setminus \{\text{pdThis}\} \subseteq \text{maybeLost} \setminus \{\text{pdThis}\} \]

**Lemma 'not lost before', section C.14**

18.6. CHECKOPERATION

\[ \Phi \text{BOp}; \text{ReqPurseOkay}; \text{RabOut}; \text{RabCl}\{\text{pdThis}/\text{pdThis}\}; \]
\[ \text{AbWorld}; \text{RabCl}; \text{RabIn} \]
\[ \text{req} \not= \text{pdThis} \]
\[ \text{chosenLost} = \text{chosenLost} \setminus \{\text{pdThis}\} \]

**Proof:**
We discharge this by removing the existential for \( \text{pdThis} \) because we have an explicit equation for it, using the [one point] rule.

18.5. EXISTSC-CHOOSELOST

\[ \Phi \text{BOp}; \text{ReqPurseOkay}; \text{RabOut}; \text{RabCl}\{\text{pdThis}/\text{pdThis}\}; \text{RabIn} \mid \text{req} \not= \text{pdThis} \]

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

\[ \text{chosenLost} \subseteq \text{maybeLost} \]
\[ \text{chosenLost} \setminus \{\text{pdThis}\} \subseteq \text{maybeLost} \setminus \{\text{pdThis}\} \]

**Lemma 'not lost before', section C.14**

18.6. CHECKOPERATION

\[ \Phi \text{BOp}; \text{ReqPurseOkay}; \text{RabOut}; \text{RabCl}\{\text{pdThis}/\text{pdThis}\}; \]
\[ \text{AbWorld}; \text{RabCl}; \text{RabIn} \]
\[ \text{req} \not= \text{pdThis} \]
\[ \text{chosenLost} = \text{chosenLost} \setminus \{\text{pdThis}\} \]

**Proof:**
We discharge this by removing the existential for \( \text{pdThis} \) because we have an explicit equation for it, using the [one point] rule.

18.5. EXISTSC-CHOOSELOST

\[ \Phi \text{BOp}; \text{ReqPurseOkay}; \text{RabOut}; \text{RabCl}\{\text{pdThis}/\text{pdThis}\}; \text{RabIn} \mid \text{req} \not= \text{pdThis} \]

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

\[ \text{chosenLost} \subseteq \text{maybeLost} \]
\[ \text{chosenLost} \setminus \{\text{pdThis}\} \subseteq \text{maybeLost} \setminus \{\text{pdThis}\} \]

**Lemma 'not lost before', section C.14**

18.6. CHECKOPERATION

\[ \Phi \text{BOp}; \text{ReqPurseOkay}; \text{RabOut}; \text{RabCl}\{\text{pdThis}/\text{pdThis}\}; \]
\[ \text{AbWorld}; \text{RabCl}; \text{RabIn} \]
\[ \text{req} \not= \text{pdThis} \]
\[ \text{chosenLost} = \text{chosenLost} \setminus \{\text{pdThis}\} \]

**Proof:**
We discharge this by removing the existential for \( \text{pdThis} \) because we have an explicit equation for it, using the [one point] rule.
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*.

### Proof:

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

\[
\begin{align*}
\Phi &\; \BoxOp; \Req{\ReqOkay}; \RabOut; \RabOkayClPd' [\pdThis/\pdThis'] ; \\
&\quad \AbWorld; \RabEndClPd; \RabIn \mid \\
&\quad \req m? = \pdThis \\
&\quad \land \; \text{chosenLost = chosenLost' \ \{} \pdThis \} \\
&\quad \land \; \text{maybeLost = maybeLost' \ \{} \pdThis \} \\
&\quad \land \; \text{definitelyLost = definitelyLost' \ \{} \pdThis \} \\
\end{align*}
\]

\[\vdash \AbTransferOkay \]

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

\[
\begin{align*}
\Phi &\; \BoxOp; \Req{\ReqOkay}; \RabOut; \RabWillBeLostClPd'[\pdThis/\pdThis'] ; \\
&\quad \AbWorld; \RabEndClPd; \RabIn \mid \\
&\quad \req m? = \pdThis \\
&\quad \land \; \text{chosenLost = chosenLost' \ \{} \pdThis \} \\
&\quad \land \; \text{maybeLost = maybeLost' \ \{} \pdThis \} \\
&\quad \land \; \text{definitelyLost = definitelyLost' \ \{} \pdThis \} \\
\end{align*}
\]

\[\vdash \AbTransferLost \]

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

\[
\begin{align*}
\Phi &\; \BoxOp; \Req{\ReqOkay}; \RabOut; \RabHasBeenLostClPd'[\pdThis/\pdThis'] ; \\
&\quad \AbWorld; \RabEndClPd; \RabIn \mid \\
&\quad \req m? = \pdThis \\
&\quad \land \; \text{chosenLost = chosenLost' \ \{} \pdThis \} \\
&\quad \land \; \text{maybeLost = maybeLost' \ \{} \pdThis \} \\
&\quad \land \; \text{definitelyLost = definitelyLost' \ \{} \pdThis \} \\
\end{align*}
\]

\[\vdash \AbTransferLost \]

**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.

\[
\begin{align*}
\Phi &\; \BoxOp; \Req{\ReqOkay}; \RabOut; \RabEndClPd'[\pdThis/\pdThis'] ; \\
&\quad \AbWorld; \RabEndClPd; \RabIn \mid \\
&\quad \req m? = \pdThis \\
&\quad \land \; \text{chosenLost = chosenLost' \ \{} \pdThis \} \\
&\quad \land \; \text{maybeLost = maybeLost' \ \{} \pdThis \} \\
&\quad \land \; \text{definitelyLost = definitelyLost' \ \{} \pdThis \} \\
\end{align*}
\]

\[\vdash \AbTransfer \]

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

### 18.7. CASE 1: *ReqOkay* and *RabOkayClPd'*

**Case 1:** We prove one case.

\[
\begin{align*}
\Phi &\; \BoxOp; \Req{\ReqOkay}; \RabOut; \RabOkayClPd'[\pdThis/\pdThis'] ; \\
&\quad \AbWorld; \RabEndClPd; \RabIn \mid \\
&\quad \req m? = \pdThis \\
&\quad \land \; \text{chosenLost = chosenLost' \ \{} \pdThis \} \\
&\quad \land \; \text{maybeLost = maybeLost' \ \{} \pdThis \} \\
&\quad \land \; \text{definitelyLost = definitelyLost' \ \{} \pdThis \} \\
\end{align*}
\]

\[\vdash \AbTransferOkay \]
18.7.1 The behaviour of \textit{maybeLost} and \textit{definitelyLost}

We argue that the transaction \textit{pdThis} is initially not in \textit{maybeLost} or \textit{definitelyLost}, and is moved into \textit{maybeLost} \textbackslash \ \textit{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 \textit{RabOkayCIPd} that

\[ pdThis \in \text{maybeLost} \ \textbackslash \ \textit{chosenLost} \]

Therefore \textit{pdThis} = \textit{chosenLost} (by the definition of set minus) and \textit{pdThis} \not\in \textit{definitelyLost} (by lemma 'lost'). So we have

\[
\text{definitelyLost} = \text{definitelyLost}' \]
\[
\text{maybeLost} = \text{maybeLost}' \ {\text{pdThis}} \]
\[
\text{chosenLost} = \text{chosenLost}'
\]

18.7.2 \textit{AbTransferOkay}

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

\[ \Phi_{\text{Rab}}; \text{ReqPurseOkay}; \text{RabOut}; \text{RabOkayCIPd}'[\textit{pdThis}|\textit{pdThis}'] \]
\[ \text{AbWorld}; \text{RabEndCIPd}; \text{RabIn} \]
\[ \text{req} \quad m_0 = \textit{pdThis} \]
\[ \wedge \text{chosenLost} = \text{chosenLost}' \ {\text{pdThis}} \]
\[ \wedge \text{maybeLost} = \text{maybeLost}' \ {\text{pdThis}} \]
\[ \wedge \text{definitelyLost} = \text{definitelyLost}' \ {\text{pdThis}} \]
\[ \Rightarrow \text{AbTransferOkay} \]

Each element of \textit{AbWorld} is defined by an explicit equation in \textit{RabEndCIPd}, and we show that this value satisfies \textit{AbTransferOkay} by showing each predicate holds.

A-1 \textit{AbOp}: This trivial: \textit{AbOp} imposes no constraints.

A-2 \textit{AbWorldSecureOp}

\begin{itemize}
\item \textit{at} \in \textit{ran} \textit{transfer}
\end{itemize}

true by construction of \textit{at} from \textit{m_0} in \textit{RabIn}. 

18.7. CASE 1: \textit{REQOKAY AND RABOKAYCIPD'}

\begin{itemize}
\item no purses other than \textit{from} and \textit{to} change
\end{itemize}

For \textit{balance} and \textit{lost} we show that \textit{RabEndCIPd} and

\[ \text{RabOkayCIPd}'[\textit{pdThis}|\textit{pdThis}'] \]

are essentially the same. This is immediate because in both cases the relevant predicates are captured in the same schema \textit{OtherPursesRab}.

A-3 \textit{Authentic}([\textit{from}/\textit{name}'], \textit{Authentic}([\textit{to}/\textit{name}'])

We have \textit{pdThis} \not\in \textit{maybeLost}', hence it is in both \textit{authenticFrom}' and in \textit{authenticTo}'. Hence, by \textit{\PhiOp} and \textit{AbstractBetween}, it is also in both \textit{authenticFrom} and in \textit{authenticTo}.

A-4 \textit{SufficientFundsProperty}

true from \textit{ConPurse} constraint \textit{P-2b}

A-5 \textit{to} = \textit{from}?

true because \textit{pdThis} is a \textit{PayDetails}.

A-6 \textit{abAuthPurse' from} = \ldots, \textit{abAuthPurse' to} = \ldots

Each of the four elements (\textit{from} and \textit{to} purses, each with \textit{balance} and \textit{lost}) are handled below, followed by all the other elements in one section.

The \textit{from} \textit{purse}’s balance component

\[ (\text{abAuthPurse pdThis}.from).balance \]
\[ = (\text{conAuthPurse pdThis}.from).balance \]
\[ + \text{sumValue}((\text{maybeLost} \ \text{chosenLost}) \]
\[ \wedge \text{req} \quad m_0 = \textit{pdThis} \]
\[ \wedge \text{maybeLost} = \text{maybeLost}' \ {\text{pdThis}} \]
\[ \wedge \text{definitelyLost} = \text{definitelyLost}' \ {\text{pdThis}} \]
\[ \Rightarrow \text{AbTransferOkay} \]

section 18.7.1]
CHAPTER 18. REQ.

\[ (\text{conAuthPurse}' \ \text{pdThis}.from).\text{lost} = \text{sumValue}((\text{definitelyLost} \cup \text{chosenLost}) \cap \{ p : \text{PayDetails} \ | \ p.d. = \text{pdThis}.from \}) \cup \{ \text{pdThis} \}) \]

\[ \text{RabOkayClPd}[…] \]

So

\[ (\text{conAuthPurse}' \ \text{from}?).\text{balance} = (\text{conAuthPurse from}?).\text{balance} - \text{value}? \]

The from purse's lost component

\[ (\text{abAuthPurse pdThis}.from).\text{lost} \]

\[ = \text{sumValue}((\text{definitelyLost} \cup \text{chosenLost}) \cap \{ p : \text{PayDetails} \ | \ p.d. = \text{pdThis}.from \}) \]

\[ \text{[RabEndClPd]} \]

\[ = \text{sumValue}((\text{definitelyLost} \cup \text{chosenLost}) \cap \{ p : \text{PayDetails} \ | \ p.d. = \text{pdThis}.from \}) \]

\[ \text{[section 18.7.1]} \]

\[ = (\text{abAuthPurse pdThis}.from).\text{lost} \text{[RabOkayClPd][…]} \]

The to purse's balance component

\[ (\text{abAuthPurse pdThis.to}).\text{balance} = \text{conAuthPurse pdThis}.to).\text{balance} \]

\[ + \text{sumValue}((\text{maybeLost} \ \text{chosenLost}) \cap \{ p : \text{PayDetails} \ | \ p.d. = \text{pdThis}.to \}) \]

\[ \text{[RabEndClPd]} \]

\[ = (\text{conAuthPurse pdThis}.to).\text{balance} \]

\[ + \text{sumValue}((\text{maybeLost} \ \text{chosenLost}) \cap \{ p : \text{PayDetails} \ | \ p.d. = \text{pdThis}.to \}) \]

\[ \text{[rearranging]} \]

\[ = (\text{conAuthPurse pdThis}.to).\text{balance} \]

\[ + \text{sumValue}((\text{maybeLost} \ \text{chosenLost}) \cap \{ p : \text{PayDetails} \ | \ p.d. = \text{pdThis}.to \}) \]

\[ \text{[RabEndClPd]} \]

\[ = (\text{abAuthPurse pdThis}.to).\text{balance} + \text{pdThis}.value \text{[RabOkayClPd][…]} \]

\[ \text{From the form of (abAuthPurse pdThis.to).balance = pdThis.value + n in Ab-TransferOkay, we see that this last subtraction gives a positive result. So}\]

\[ (\text{abAuthPurse}' to?).\text{balance} = (\text{abAuthPurse to}?).\text{balance} + \text{value}? \]

The to purse's lost component

\[ (\text{abAuthPurse pdThis.to}).\text{lost} \]

\[ = \text{sumValue}((\text{definitelyLost} \cup \text{chosenLost}) \cap \{ p : \text{PayDetails} \ | \ p.d. = \text{pdThis}.to \}) \]

\[ \text{[RabEndClPd]} \]

\[ = \text{sumValue}((\text{definitelyLost} \cup \text{chosenLost}) \cap \{ p : \text{PayDetails} \ | \ p.d. = \text{pdThis}.to \}) \]

\[ \text{[section 18.7.1]} \]

\[ = (\text{abAuthPurse pdThis.to}).\text{lost} \text{[RabOkayClPd][…]} \]

The remaining from and to purse components

These are unchanging, by \(\exists\text{ConParseReq}\), and that the retrieves each define a unique abstract world.

- 18.7.2
- 18.7
18.8 case 2: ReqOkay and RabWillBeLostPd

ΦBOp, ReqPurseOkay, RabOut; RabWillBeLostClPd'[pdThis/pdThis']
  ∨ AbWorld; RabEndClPd; RabIn |
  req' m? = pdThis
∧ chosenLost = chosenLost' \{pdThis\}
∧ maybeLost = maybeLost' \{pdThis\}
∧ definitelyLost = definitelyLost' \{pdThis\}

⊢ AbTransferLost

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 ∈ chosenLost'

Therefore

pdThis ∈ maybeLost'

because chosenLost' ⊆ maybeLost’. But we can say that pdThis ∉ definitelyLost’ (by lemma ‘lost’). So we have

definitelyLost = definitelyLost'
maybeLost = maybeLost' \{pdThis\}
chosenLost = chosenLost' \{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

ΦBOp, ReqPurseOkay, RabOut; RabWillBeLostClPd'[pdThis/pdThis']
  ∨ AbWorld; RabEndClPd; RabIn |
  req' m? = pdThis
∧ chosenLost = chosenLost' \{pdThis\}
∧ maybeLost = maybeLost' \{pdThis\}
∧ definitelyLost = definitelyLost' \{pdThis\}

⊢ AbTransferLost

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? ∈ ran transfer
  true by construction of a?
• no purses other than from? and to? change
  For balance and lost we show that RabEndClPd and RabWillBeLostClPd'[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 ∈ 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? = from?

true because pdThis is a PayDetails.

A–6 abAuthPurse' from? = . . . , abAuthPurse' to? = . . .

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

\[
(ab \text{AuthPurse pdThis.from}).balance = \begin{align*}
&= (\text{conAuthPurse pdThis.from}).balance \\
&\quad + \text{sumValue}(\{\text{maybeLost} \cup \text{chosenLost}\} \\
&\quad \cap \{pd : \text{PayDetails} \mid pd.to = pdThis.from\}) \\
&\quad \setminus \{pdThis\} \tag{RabEndCIPD} \\
&= (\text{conAuthPurse pdThis.from}).balance \\
&\quad + \text{sumValue}(\{\text{maybeLost} \cap \text{chosenLost}\} \\
&\quad \cap \{pd : \text{PayDetails} \mid pd.to = pdThis.from\}) \\
&\quad \setminus \{pdThis\} \tag{section 18.8.1} \\
&= \text{pdThis.value} + (\text{conAuthPurse’ pdThis.from}).balance \\
&\quad + \text{sumValue}(\{\text{maybeLost} \cap \text{chosenLost}\} \\
&\quad \cap \{pd : \text{PayDetails} \mid pd.to = pdThis.from\}) \\
&\quad \setminus \{pdThis\} \tag{RabWillBeLostCIPD’[…]} \\
&= \text{pdThis.value} + (\text{abAuthPurse’ pdThis.from}).balance \\
&\quad + \text{sumValue}(\{\text{maybeLost} \cup \text{chosenLost}\} \\
&\quad \cap \{pd : \text{PayDetails} \mid pd.to = pdThis.from\}) \\
&\quad \setminus \{pdThis\} \tag{rearranging} \\
&= (\text{abAuthPurse’ pdThis.from}).lost \setminus \text{pdThis.value} \tag{RabWillBeLostCIPD’[…]} \\
\end{align*}
\]

So

\[
(ab \text{AuthPurse’ from?}).balance = (ab \text{AuthPurse from?}).balance – value?
\]

The from purse’s lost component

\[
(ab \text{AuthPurse pdThis.from}).lost = \begin{align*}
&= \text{sumValue}(\{\text{definitelyLost} \cup \text{chosenLost}\} \\
&\quad \cap \{pd : \text{PayDetails} \mid pd.from = pdThis.from\}) \\
&\quad \setminus \{pdThis\} \tag{RabEndCIPD} \\
&= \text{sumValue}(\{\text{definitelyLost} \cup \text{chosenLost}\} \setminus \{pdThis\}) \\
&\quad \cap \{pd : \text{PayDetails} \mid pd.from = pdThis.from\} \tag{section 18.8.1} \\
&= \text{sumValue}(\{\text{definitelyLost} \cup \text{chosenLost}\} \\
&\quad \cap \{pd : \text{PayDetails} \mid pd.from = pdThis.to\}) \\
&\quad \setminus \{pdThis\} \tag{Φ-BOp} \\
&= (ab \text{AuthPurse pdThis.to}).balance \tag{RabWillBeLostCIPD’[…]} \\
\end{align*}
\]

The to purse’s balance component

\[
(ab \text{AuthPurse pdThis.to}).balance = \begin{align*}
&= (\text{conAuthPurse pdThis.to}).balance \\
&\quad + \text{sumValue}(\{\text{maybeLost} \cup \text{chosenLost}\} \\
&\quad \cap \{pd : \text{PayDetails} \mid pd.to = pdThis.to\}) \\
&\quad \setminus \{pdThis\} \tag{RabEndCIPD} \\
&= (\text{conAuthPurse pdThis.to}).balance \\
&\quad + \text{sumValue}(\{\text{maybeLost} \cap \text{chosenLost}\} \\
&\quad \cap \{pd : \text{PayDetails} \mid pd.to = pdThis.to\}) \\
&\quad \setminus \{pdThis\} \tag{section 18.8.1} \\
&= (\text{conAuthPurse pdThis.to}).balance \\
&\quad + \text{sumValue}(\{\text{maybeLost} \cap \text{chosenLost}\} \\
&\quad \cap \{pd : \text{PayDetails} \mid pd.to = pdThis.to\}) \\
&\quad \setminus \{pdThis\} \tag{rearranging} \\
&= (\text{conAuthPurse’ pdThis.to}).balance \\
&\quad + \text{sumValue}(\{\text{maybeLost} \cup \text{chosenLost}\} \\
&\quad \cap \{pd : \text{PayDetails} \mid pd.to = pdThis.to\}) \\
&\quad \setminus \{pdThis\} \tag{Φ-BOp} \\
&= (ab \text{AuthPurse’ pdThis.to}).balance \tag{RabWillBeLostCIPD’[…]} \\
\end{align*}
\]

The to purse’s lost component

\[
(ab \text{AuthPurse pdThis.to}).lost = \begin{align*}
&= \text{sumValue}(\{\text{definitelyLost} \cup \text{chosenLost}\} \\
&\quad \cap \{pd : \text{PayDetails} \mid pd.from = pdThis.to\}) \\
&\quad \setminus \{pdThis\} \tag{RabEndCIPD} \\
&= \text{sumValue}(\{\text{definitelyLost} \cup \text{chosenLost}\} \setminus \{pdThis\}) \\
&\quad \cap \{pd : \text{PayDetails} \mid pd.from = pdThis.to\} \tag{section 18.8.1} \\
&= \text{sumValue}(\{\text{definitelyLost} \cup \text{chosenLost}\} \\
&\quad \cap \{pd : \text{PayDetails} \mid pd.from = pdThis.to\}) \\
&\quad \setminus \{pdThis\} \tag{Φ-BOp} \\
&= (ab \text{AuthPurse pdThis.to}).lost \tag{RabWillBeLostCIPD’[…]} \\
\end{align*}
\]
These are unchanging, by \( \Xi \) from the remaining unique abstract world.

18.9 Case 3: ReqOkay

\[ \star \]

Therefore \( pd \) 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

\[ \Phi BoP; ReqParseOkay; RabOut; RabHas BeenLostCl Pd'[pdThis/pdThis']; AbWorld; RabEndClPd; Rabln \mid \]

\( req \cdot m? = pdThis \)

\( \land \ chosenLost = chosenLost' \setminus \{pdThis\} \)

\( \land \ maybeLost = maybeLost' \setminus \{pdThis\} \)

\( \land \ definitelyLost = definitelyLost' \setminus \{pdThis\} \)

\[ \vdash AbTransferLost \]

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

- \( a? \in \) ran \( transfer \)
- true by construction of \( a? \)
- no purses other than \( from \) and \( to \) change

For balance and lost we show that \( RabEndClPd \) and \( RabHas BeenLostClPd'[pdThis/ pdThis'] \) 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 \( \Phi BoP \) and AbstractBetween, it is also in both \( authenticFrom \) and in \( authenticTo \).

A-4 SufficientFundsProperty

true from \( ConPurse \) constraint P-2b
\textbf{A-5} to? = from?

true because pdThis is a PayDetails.

\textbf{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.

\textbf{The from purse's balance component}

\[
(ab\text{AuthPurse pdThis.from}).balance \\
= (\text{conAuthPurse pdThis.from}).balance \\
+ \text{sumValue}((\text{maybeLost} \setminus \text{chosenLost}) \\
\cap \{ pd : \text{PayDetails} \mid pd.to = pdThis.from \}) \\
\setminus \{ pdThis\}) \quad [\text{RabEndClPd}]
\]

\[
= (\text{comAuthPurse pdThis.from}).balance \\
+ \text{sumValue}((\text{maybeLost} \setminus \text{chosenLost}) \\
\cap \{ pd : \text{PayDetails} \mid pd.to = pdThis.from \}) \\
\setminus \{ pdThis\}) \quad [\text{section 18.9.1}]
\]

\[
= \text{pdThis.value} + (\text{abAuthPurse' pdThis.from}).balance \\
+ \text{sumValue}((\text{maybeLost} \setminus \text{chosenLost}) \\
\cap \{ pd : \text{PayDetails} \mid pd.to = pdThis.from \}) \\
\setminus \{ pdThis\}) \quad [\text{ReqPurseOkay}]
\]

\[
= \text{pdThis.value} + (ab\text{AuthPurse' pdThis.from}).balance \\
[\text{RabHas BeenLostClPd'}[\ldots]]
\]

\textbf{So}

\[
(ab\text{AuthPurse' from?}).balance = (ab\text{AuthPurse from?}).balance - \text{value?}
\]

\textbf{The from purse's lost component}

\[
(ab\text{AuthPurse pdThis.from}).lost \\
= \text{sumValue}((\text{definitelyLost} \cup \text{chosenLost}) \\
\cap \{ pd : \text{PayDetails} \mid pd.from = pdThis.from \}) \\
\setminus \{ pdThis\}) \quad [\text{RabEndClPd}]
\]

\[
= \text{sumValue}((\text{definitelyLost} \cup \text{chosenLost}) \\
\cap \{ pd : \text{PayDetails} \mid pd.from = pdThis.from \}) \\
\setminus \{ pdThis\}) \quad [\text{section 18.9.1}]
\]

18.9. CASE 3: REQOKAY AND RABHASBEENLOSTPD'

\[
= \text{sumValue}((\text{definitelyLost} \cup \text{chosenLost}) \\
\cap \{ pd : \text{PayDetails} \mid pd.from = pdThis.from \}) \\
\setminus \{ pdThis\}) \quad [\text{rearrange}]
\]

\[
= (ab\text{AuthPurse' pdThis.from}).lost - pdThis.value \\
[\text{RabHas BeenLostClPd'}[\ldots]]
\]

\textbf{The to purse's balance component}

\[
(ab\text{AuthPurse pdThis.to}).balance \\
= (\text{conAuthPurse pdThis.to}).balance \\
+ \text{sumValue}((\text{maybeLost} \setminus \text{chosenLost}) \\
\cap \{ pd : \text{PayDetails} \mid pd.to = pdThis.to \}) \\
\setminus \{ pdThis\}) \quad [\text{RabEndClPd}]
\]

\[
= (\text{comAuthPurse pdThis.to}).balance \\
+ \text{sumValue}((\text{maybeLost} \setminus \text{chosenLost}) \\
\cap \{ pd : \text{PayDetails} \mid pd.to = pdThis.to \}) \\
\setminus \{ pdThis\}) \quad [\text{section 18.9.1}]
\]

\[
= (\text{conAuthPurse pdThis.to}).balance \\
+ \text{sumValue}((\text{maybeLost} \setminus \text{chosenLost}) \\
\cap \{ pd : \text{PayDetails} \mid pd.to = pdThis.to \}) \\
\setminus \{ pdThis\}) \quad [\Phi\text{BoP}]
\]

\[
= (ab\text{AuthPurse' pdThis.to}).balance \\
[\text{RabHas BeenLostClPd'}[\ldots]]
\]

\textbf{The to purse's lost component}

\[
(ab\text{AuthPurse pdThis.to}).lost \\
= \text{sumValue}((\text{definitelyLost} \cup \text{chosenLost}) \\
\cap \{ pd : \text{PayDetails} \mid pd.from = pdThis.to \}) \\
\setminus \{ pdThis\}) \quad [\text{RabEndClPd}]
\]

\[
= \text{sumValue}((\text{definitelyLost} \setminus \text{chosenLost}) \\
\cap \{ pd : \text{PayDetails} \mid pd.from = pdThis.to \}) \\
\setminus \{ pdThis\}) \quad [\text{section 18.9.1}]
\]

\[
= \text{sumValue}((\text{definitelyLost} \setminus \text{chosenLost}) \\
\cap \{ pd : \text{PayDetails} \mid pd.from = pdThis.to \}) \\
\setminus \{ pdThis\}) \quad [\text{section 18.9.1}]
\]
= \textit{sumValue}((\{pd : \text{PayDetails} \mid \text{pd.from} = \text{pdThis.to}\})
\setminus \{\text{pdThis}\}) \hspace{1cm} \text{[rearrange]}
= (ab\text{AuthPurse'} \text{pdThis}.to).\text{lost} \hspace{1cm} \text{[RabHasBeenLostCIPd'[...]]}

The remaining \textit{from} and \textit{to} purse components
These are unchanging, by $\exists \text{ConPurseReq}$, and that the retrieves each define a unique abstract world.

$\blacksquare$

18.10 case 4: \textit{ReqOkay} and \textit{RabEndPd}'

$\Phi\text{BoP}; \text{ReqPurseOkay}; \text{RabOut}; \text{RabEndCIPd'}[\text{pdThis}/\text{pdThis'}];$
\hspace{0.5cm} $\text{AbWorld'}; \text{RabEndCIPd'; RabIn} |$
\hspace{0.5cm} $\text{req'} m! = \text{pdThis}$
\hspace{0.5cm} $\land \text{chosenLost} = \text{chosenLost'} \setminus \{\text{pdThis}\}$
\hspace{0.5cm} $\land \text{maybeLost} = \text{maybeLost'} \setminus \{\text{pdThis}\}$
\hspace{0.5cm} $\land \text{definitelyLost} = \text{definitelyLost'} \setminus \{\text{pdThis}\}$
\hspace{0.5cm} $\Rightarrow$
\hspace{0.5cm} $\text{AbTransfer}$

We show that $\text{RabEndCIPd'}[...]$ is false under $\text{ReqOkay}$, and then proceed by $\text{[contradiction]}$, because this shows the antecedent of the theorem is false, and hence the theorem is true.

$\Phi\text{BoP}; \text{ReqPurgeOkay}; \text{RabOut}; \text{AbWorld'};$
\hspace{0.5cm} $\text{pdThis : PayDetails; chosenLost'} : \exists \text{PayDetails} |$
\hspace{0.5cm} $\text{req'} m! = \text{pdThis}$
\hspace{0.5cm} $\Rightarrow$
\hspace{0.5cm} $\neg \text{RabEndCIPd'}[\text{pdThis}/\text{pdThis'}]$

It suffices to show that $\text{pdThis} \in \text{definitelyLost'} \cup \text{maybeLost'}$. We have
\hspace{0.5cm} $\text{definitelyLost'} \cup \text{maybeLost'}$
\hspace{0.5cm} $= (\text{fromInEpa'} \cup \text{fromLogged'}) \cap (\text{toInEpv'} \cup \text{toLogged'})$

$\text{ReqPurseOkay}$ gives us that the after state of the purse is $\text{epa'}$; $\text{pdThis}$ is in
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 A operation into one for each individual A operation.

This chapter proves the 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

\[ P \iff pdThis = (\text{conAuthPurse name?}) \cdot pdAuth \]
\[ Q \iff chosenLost = chosenLost' \]
19.3 exists-pd

\[ B \circ \text{Op}; \text{ValPurseOkay}; \text{RabOut}; \text{RabCl}; \text{RabIn} \]

\( \vdash \exists \text{pdThis : PayDetails} \bullet \text{pdThis} = (\text{conAuthPursename} \circ \text{pdAuth}) \]

Proof:
This is immediate by the [one point] rule, as we have an explicit definition of \text{pdThis}.
\( \blacksquare \)

19.4 exists-chosenlost

\[ B \circ \text{Op}; \text{ValPurseOkay}; \text{RabOut}; \text{RabClPd}^\prime[\text{pdThis}/\text{pdThis}^\prime]; \text{RabIn} \]

\( \vdash \exists \text{chosenLost : P PayDetails} \bullet \]

\( \text{chosenLost} = \text{chosenLost}^\prime \)
\( \wedge \text{chosenLost} \subseteq \text{maybeLost} \)

Proof:
We can [one point] away the quantification because we have an explicit definition of \text{chosenLost} (as \text{chosenLost}'). We show that the constraint holds by
\[
\begin{align*}
\text{chosenLost} &= \text{chosenLost}' & \text{[defn.]} \\
\subseteq &\text{maybeLost}' & \text{[RabClPd'[...]]} \\
\subseteq &\text{maybeLost} \setminus \{\text{pdThis}\} & \text{[see 19.6.7]} \\
\subseteq &\text{maybeLost} & \text{[defn.]}
\end{align*}
\]

\( \blacksquare \)

19.5 check-operation

\[ B \circ \text{Op}; \text{ValPurseOkay}; \text{RabClPd}^\prime[\text{pdThis}/\text{pdThis}^\prime]; \text{AbWorld}; \text{RabClPd} \]

\( \vdash \forall n : \text{dom abAuthPurse} \bullet \]

\( (\text{abAuthPurse'} n).\text{balance} = (\text{abAuthPurse} n).\text{balance} \)
\( \wedge (\text{abAuthPurse'} n).\text{lost} = (\text{abAuthPurse} n).\text{lost} \)

19.6 Behaviour of maybeLost and definitelyLost

We prove this first by investigating the way in which the key sets \text{definitelyLost} and \text{maybeLost} are modified by the operation. Having got equations for these changes, we then look at the equations for the components \text{balance} and \text{lost} for two types of purses: the to purse in the transaction \text{pdThis}, and all other purses.

19.6.1 fromLogged

No logs change, so
\[
\text{fromLogged}' = \text{fromLogged}
\]

19.6.2 toLogged

No logs change, so
\[
\text{toLogged}' = \text{toLogged}
\]

After the operation the purse is in \text{aTo}, and \text{pdThis} is in \text{authenticTo}, from \( B \circ \text{Op} \), hence \text{pdThis} \in \text{toInEpayee}'. Lemma 'notLoggedAndIn' (section C.12) gives us:

\( \text{pdThis} \notin \text{toLogged}' \)

19.6.3 toInEpv

From the precondition of \text{ValPurseOkay} we know the purse is in \text{epv}, and we know that the name of this purse is equal to \text{pdThis}. To. After the operation, this purse is in \text{aTo} (that is, not in \text{epv}). No other purses change.

\[
\text{toInEpv}' = \text{toInEpv} \setminus \{\text{pdThis}\}
\]
\[
\text{toInEpv} = \text{toInEpv}' \cup \{\text{pdThis}\}
\]
19.6.6 chosenLost

Only the to purse changes.

\[ \text{fromInEpa} = \text{fromInEpa} \]

19.6.5 definitelyLost

\[
\begin{align*}
\text{definitelyLost}' & = \text{toLogged}' \land (\text{fromLogged}' \lor \text{fromInEpa}') \quad \text{[defn]} \\
\text{definitelyLost} & = \text{toLogged} \land (\text{fromLogged} \lor \text{fromInEpa}) \quad \text{[above]} \\
\end{align*}
\]

19.6.6 chosenLost

\[ \text{chosenLost}' = \text{chosenLost} \]

by choice. So

\[ \text{definitelyLost} \lor \text{chosenLost} = \text{definitelyLost}' \lor \text{chosenLost}' \]

19.6.7 maybeLost

\[
\begin{align*}
\text{maybeLost}' & = (\text{fromInEpa}' \lor \text{fromLogged}') \land \text{toInEpa}' \quad \text{[defn]} \\
& = (\text{fromInEpa} \lor \text{fromLogged}) \land (\text{toInEpa} \lor \text{pdThis}') \quad \text{[above]} \\
& = (\text{fromInEpa} \lor \text{fromLogged} \land \text{toInEpa}) \land (\text{pdThis}) \quad \text{[Spivey]} \\
& = \text{maybeLost} \land (\text{pdThis}') \quad \text{[defn]} \\
\end{align*}
\]

\[ \text{val} \in \text{ether} \land \text{to.status} = \text{epv} \quad \text{[precondition ValParseOkay]} \]

\[ \Rightarrow \text{pdThis} \in \text{fromInEpa} \lor \text{fromLogged} \quad \text{[B-11]} \]

\[ \Rightarrow \text{pdThis} \in \text{maybeLost} \land (\text{toInEpa}, \text{defn maybeLost}) \]

\[ \text{pdThis} \in \text{maybeLost} \land \text{pdThis} \notin \text{chosenLost} \]

\[ \Rightarrow \text{pdThis} \in \text{maybeLost} \land \text{pdThis} \notin \text{chosenLost} \]

Also

\[ \text{maybeLost} \land \text{chosenLost} = (\text{maybeLost} \land \text{chosenLost}) \cup \{\text{pdThis}\} \]

19.7. CLARIFYING THE HYPOTHESIS

We can show that the hypothesis is actually stronger than it looks, in that we can replace \(\text{RabCIPD} \lor \text{RabOkayCIPD}\) with \(\text{RabClPd} \lor \text{RabOkayClPd}\). This is because \(\text{pdThis} \in \text{maybeLost} \land \text{chosenLost}\), implying that \(\text{RabOkayCIPD}\) holds.

\[ \text{pdThis} \notin \text{maybeLost}' \text{ (see construction of maybeLost') and so it cannot be in chosenLost'}. \text{ pdThis} \notin \text{maybeLost}' \text{ and so it cannot be in maybeLost'} \land \text{chosenLost}'. \text{ pdThis} \notin \text{definitelyLost'} because it is not in toLogged'. \]

This implies that \(\text{RabEndClPd}'[\ldots]\) holds. So we have to prove

\[ \Phi[n]: \text{ValParseOkay, RabEndClPd'[pdThis]/pdThis'}; \]

\[ \text{AbWorld, RabOkayClPd | pdThis} = (\text{conAuthPurse name}), \text{pdAuth} \land \text{chosenLost} = \text{chosenLost'} \]

\[ \forall \text{ n : dom abAuthPurse} \bullet \]

\[ (\text{abAuthPurse' n).balance} = (\text{abAuthPurse n).balance} \land \text{abAuthPurse' n).lost} = (\text{abAuthPurse n).lost} \]

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

\[ \forall \text{ n : dom abAuthPurse | n \in pdThis.to} \bullet \]

\[ (\text{abAuthPurse' n).balance} = (\text{conAuthPurse' n).balance} + \text{sumValue}((\text{maybeLost} \land \text{chosenLost})' \land \text{pdThis'}) \land (\text{pd : PayDetails | pd.to = n}) \land (\text{pdThis}) \]

\[ \Rightarrow \text{RabEndClPd'[pdThis]/pdThis'} \]

\[ \text{[union and subtraction cancel]} \]
\[ \text{conAuthPurse}' n \].balance + \text{sumValue}((\text{maybeLost}' \cap \text{chosenLost}') \cap \{ \text{pdThis} \}) \]

\[ \text{abAuthPurse}' \text{pdThis.to} \].balance
\[ \text{RabOkayClPd} \]

\[ \text{conAuthPurse}' \text{pdThis.to} \].balance + \text{sumValue}((\text{maybeLost}' \cap \text{chosenLost}') \cup \{ \text{pdThis} \}) \]

\[ \text{abAuthPurse}' \text{pdThis.to} \].balance
\[ \text{RabOkayClPd} \]

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 \cup chosenLost, which we derived as unchanging earlier. PhD of does not change the concrete values, so the abstract values do not change either.

19.7.3 Case balance component for pdThis.to purse

\[ \text{abAuthPurse}' \text{pdThis.to} \].balance
\[ \text{RabOkayClPd} \]

\[ \text{abAuthPurse}' \text{pdThis.to} \].balance
\[ \text{RabOkayClPd} \]

19.7.4 Case lost component for pdThis.to purse

In this case the defining equations in the retrieve depend upon definitelyLost \cup 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
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 A operation into one for each individual B operation.

This chapter proves the 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

\[ P \iff pdThis = (\text{conAuthPurse name?}).pdAuth \]

\[ Q \iff \text{chosenLost} = \text{chosenLost}' \]

20.3 exists-pd

\[ \Phi \Box \text{op}; \text{AckPurseOkay}; \text{RabOut}; \text{RabCl}'; \text{RabIn} \]

\[ \exists pdThis : \text{PayDetails} \bullet pdThis = (\text{conAuthPurse name?}).pdAuth \]

Proof:
This is immediate by \[ \text{one point} \] rule, as we have an explicit definition of \( pdThis \).

20.4 exists-chosenlost

\[ \Phi \Box \text{op}; \text{AckPurseOkay}; \text{RabOut}; \text{RabCIPd}' [pdThis/pdThis']; \text{RabIn} | pdThis = (\text{conAuthPurse name?}).pdAuth \]

\[ \exists \text{chosenLost} : P \text{PayDetails} \bullet \text{chosenLost} = \text{chosenLost}' \]

\[ \land \text{chosenLost} \subseteq \text{maybeLost} \]

Proof:
We can \[ \text{one point} \] away the quantification because we have an explicit definition of \( \text{chosenLost} \) (as \( \text{chosenLost}' \)). We show that the constraint holds by

\[ \text{chosenLost} = \text{chosenLost}' \quad \text{[def]} \]

\[ \subseteq \text{maybeLost}' \quad \text{[RabCIPd]' [...]} \]

\[ \subseteq \text{maybeLost} \quad \text{[see 20.6.6]} \]

20.5 check-operation

\[ \Phi \Box \text{op}; \text{AckPurseOkay}; \text{RabCIPd}' [pdThis/pdThis']; \text{AbWorld}; \text{RabCIPd} | pdThis = (\text{conAuthPurse name?}).pdAuth \]

\[ \land \text{chosenLost} = \text{chosenLost}' \]

\[ \land \forall n : \text{dom abAuthPurse} \bullet (abAuthPurse n).balance = (abAuthPurse n').balance \]

\[ \land (abAuthPurse n).lost = (abAuthPurse n').lost \]

Proof:
We prove this by investigating the way in which the key sets \text{definitelyLost} and \text{maybeLost} are modified by the operation.

20.6 Behaviour of \text{maybeLost} and \text{definitelyLost}

We argue that the transaction \( pd \) is initially in neither \text{maybeLost} nor \text{definitelyLost}, and is not moved into either of them by the \text{AckOkay} operation. The transaction was initially far enough along to have already succeeded.

20.6.1 Behaviour of fromLogged

From \( \Phi \Box \text{op} \), which says that only the purse name? changes, and then only according to \text{AckPurseOkay}, and from the definition of \text{AckPurseOkay}, in which \text{exLog}' = \text{exLog}, we can see that

\[ \text{fromLogged}' = \text{fromLogged} \]

20.6.2 Behaviour of toLogged

Exactly as we argued for \text{fromLogged},

\[ \text{toLogged}' = \text{toLogged} \]

20.6.3 Behaviour of toInEpv

If \( \text{toInEpv}' = \text{toInEpv} \), there must be some \( pd \) in one and not in the other. From the definition of \text{toInEpv}, this means that for some purse that changes, either before or after the operation its status must equal \text{epv}. That is,

\[ (\text{conAuthPurse pd.to}).status = \text{epv} \]

\[ \lor (\text{conAuthPurse'} pd.to)).status = \text{epv} \]
From $ΦBOp$ we have that the only purse that changes is name?. From AckPurseOkay we have that

$(\text{conAuthPurse name?}).\text{status} = \text{epa}$
$(\text{conAuthPurse' name?}).\text{status} = \text{eaFrom}$

(neither equal to $\text{epv}$). Therefore, no such pd exists, and we have

toInEv$′ = \text{toInEv}$

**20.6.4 Behaviour of fromInEpa**

If $\text{fromInEpa}′ ≠ \text{fromInEpa}$, 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,

$(\text{conAuthPurse pd from}).\text{status} = \text{epa}$
\lor
$(\text{conAuthPurse' pd from}).\text{status} = \text{epa}$

The only name that changes is name?, and from AckPurseOkay we have that

$(\text{conAuthPurse name?}).\text{status} = \text{epa}$
$(\text{conAuthPurse' name?}).\text{status} = \text{eaFrom}$

Therefore, we have

$\text{fromInEpa}′ = \text{fromInEpa} \setminus \{pd : \text{PayDetails} | pd, \text{from} = \text{name}\}$
\land
$(\text{conAuthPurse name?}).\text{status} = \text{epa}$
\land
$(\text{conAuthPurse' name?}).\text{pdAuth} = pd$

In fact, the last predicate in this set limits the pd to a single value, equal to $pdThis$, so we have

$\text{fromInEpa}′ = \text{fromInEpa} \setminus \{pdThis\}$

We now build up the two sets definitelyLost and maybeLost.

**20.6.5 Behaviour of definitelyLost**

$\text{definitelyLost}′ = \text{toLogged}′ \cap (\text{fromLogged}′ \cup \text{fromInEpa}′)$ [defn.]

$= \text{toLogged}′ \cap (\text{fromLogged} \cup (\text{fromInEpa} \setminus \{pdThis\}))$ [above identities]

$= \text{toLogged} \cap (\text{fromLogged} \cup (\text{fromInEpa} \setminus \{pdThis\}))$ [pdThis $∈ \text{fromLogged}$, see below]

$= (\text{fromLogged} \cup \text{fromInEpa}) \setminus \{pdThis\}$ [algebra]

$= (\text{fromLogged} \cup \text{fromInEpa}) \setminus \text{toLoggedThis} \notin \text{toLogged}$, see below

$= \text{definitelyLost}$ [defn.]

We have $pdThis \notin \text{fromLogged}$, from the fact that $pdThis \in \text{fromInEpa}$ (because the before purse state is epa, and $ΦBOp$ gives $pdThis \in \text{authenticFrom}$), and using lemma ‘notLoggedAndIn’.

We have $pd \notin \text{toLogged}$.

ack pd $∈ \text{either}$ [precondition AckPurseOkay]

$⇒ pd \notin \text{toInEpa} \cup \text{toLogged}$ [BetweenWorld constraint B-10]

$⇒ pd \notin \text{toLogged}$ [law]

Thus we have

$\text{definitelyLost}′ = \text{definitelyLost}$

**20.6.6 Behaviour of maybeLost**

$\text{maybeLost}′ = (\text{fromInEpa}′ \cup \text{fromLogged}′) \cap \text{toInEpa}$ [defn.]

$= (\text{fromInEpa} \cup (\text{fromLogged} \setminus \{pdThis\})) \cap \text{toInEpa}$ [above identities]

$= (\text{fromInEpa} \cup \text{fromLogged}) \setminus \{pdThis\}$ [pdThis $\notin \text{fromLogged}$, as above]

$= (\text{fromInEpa} \cup \text{fromLogged}) \setminus \text{toInEpa} \setminus \{pdThis\}$ [algebra]

$= (\text{fromInEpa} \cup \text{fromLogged}) \setminus \text{toInEpa} \setminus \{pdThis\}$ [pdThis $\notin \text{toInEpa}$, see below]

$= \text{maybeLost}$ [defn.]
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.

\[ \text{maybeLost}' = \text{maybeLost} \]

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 \( A \) operation into one for each individual \( B \) operation.

This chapter proves the \( B \) 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 \( E \) fromPurseOkay 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 \( \exists \text{pd}, \exists \text{chosenLost} \), and \( \text{check-operation} \).
- Since this operation leaves the sets \( \text{maybeLost} \) and \( \text{definitelyLost} \) unchanged, we use lemma 'lost unchanged' (section C.2) to discharge the \( \exists \text{pd and exists chosenLost obligations} \) automatically.
- Since this operation refines \( \text{AbortOkay} \), we use lemma 'AbortOkay' (from section C.3) to simplify \( \text{check-operation} \) to \( \text{check-operation-ignore} \).
21.2 Invoking lemma 'lost unchanged'

We have the constraint $\Xi_{\text{ConPurse}}$ in the definition of $\text{ReadExceptionLogPurse-EafromOkay}$. From $\Phi_{\text{BOp}}$ and $\Xi_{\text{ConPurse}}$, 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

$\forall n : \text{dom abAuthPurse} \Rightarrow
\begin{array}{l}
(\text{abAuthPurse}' n).\text{balance} = (\text{abAuthPurse} n).\text{balance} \\
\wedge
definitelyLost' = definitelyLost
\end{array}$

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.

22 Correctness of $\text{ClearExceptionLog}$

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 $A$ operation into one for each individual $B$ operation.

This chapter proves the $B$ operation.

- We use lemma 'ignore' (see section 14.3) to simplify the proof obligation by proving the correctness of $\text{Ignore}$ (in section 14.7), and $\text{Abort}$ (in section 14.8), leaving the $\text{Okay}$ branch to be proven here.
- Since the $\text{Okay}$ branch of this operation is expressed as a promotion of $\text{AbortPurseOkay}$ composed with a simpler $\text{EafromPurseOkay}$ 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 $\text{exists-pd, exists-chosenLost, and check-operation}$. 
- Since this operation leaves the sets maybeLost and definitelyLost unchanged, we use lemma 'lost unchanged' (section C.2) to discharge the $\text{exists pd and exists chosenLost-obligations}$ automatically.
- Since this operation refines $\text{AbIgnore}$, we use lemma 'AbIgnore' (from section C.3) to simplify $\text{check-operation}$ to $\text{check-operation-ignore}$. 

21
CHAPTER 22. CLEAREXCEPTIONLOG

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

\(\Phi BOp; \text{ClearExceptionLogPurseEa fromOkay};\)
\(\text{RabOut}; \text{RabCPD}[\text{pdThis/ pdThis'}];\)
\(\text{AbWorld}; \text{RabCPD}; \text{Rabn}\)
\(\text{chosenLost'} = \text{chosenLost}\)
\(\wedge \text{maybeLost'} = \text{maybeLost}\)
\(\wedge \text{definitelyLost'} = \text{definitelyLost}\)
\(\vdash \forall n : \text{dom.abAuthPurse} \bullet \)
\(\text{\textcolor{red}{(\text{abAuthPurse'} n).balance} = (abAuthPurse n).balance}\)
\(\wedge (\text{abAuthPurse'} n).lost = (abAuthPurse n).lost\)

\textbf{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.

\(\Box 22.3\)
\(\Box 22\)

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 \(A\) operation into one for each individual \(B\) operation.

This chapter proves the \(B\) 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

\(\text{AuthoriseExLogClearOkay; Rab'}; \text{RabOut}\)
\(-\)
\(\exists \text{AbWorld}; a? : \text{AIN} \bullet \text{Rab} \wedge \text{Rabn} \wedge \text{AbIgnore}\)

23.2 Proof

First we choose an input. We argue exactly as in section 14.4.1 to reduce the obligation to:

\(\text{AuthoriseExLogClearOkay; Rab'}; \text{RabOut}; \text{Rabn}\)
\(-\)
\(\exists \text{AbWorld} \bullet \text{Rab} \wedge \text{AbIgnore}\)
CHAPTER 23. AUTHORISEEXLOGCLEAR

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.

\[ \text{AuthoriseExLogClearOkay; Rab'; RabOut; RabIn; AbWorld} \]
\[ \vdash \delta \text{AbWorld} = \delta \text{AbWorld'} \]
\[ \vdash Rab \land \text{AbIgnore} \]

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'.

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 A operation into one for each individual B operation.

This chapter proves the 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

\[ \text{Archive; Rab'; RabOut} \vdash \exists \text{AbWorld}; \exists \delta? : \text{AIN} \land Rab \land RabIn \land AbIgnore \]

24.2 Proof

First we choose an input. We argue exactly as in section 14.4.1 to reduce the obligation to:

\[ \text{Archive; Rab'; RabOut; RabIn} \vdash \exists \text{AbWorld} \land Rab \land AbIgnore \]

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.

\[ \text{Archive; Rab'; RabOut; RabIn; AbWorld} \]
\[ \vdash \delta \text{AbWorld} = \delta \text{AbWorld'} \]
\[ \vdash Rab \land AbIgnore \]
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'.

Part III

Second Refinement: B to 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 $B$ 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:

- $\text{StartTo} \sqsubseteq \text{CStartTo}$
- $\text{StartFrom} \sqsubseteq \text{CStartFrom}$
- $\text{Req} \sqsubseteq \text{CReq}$
- $\text{Val} \sqsubseteq \text{CVal}$
- $\text{Ack} \sqsubseteq \text{CAck}$
- $\text{ReadExceptionLog} \sqsubseteq \text{CReadExceptionLog}$
- $\text{ClearExceptionLog} \sqsubseteq \text{CClearExceptionLog}$
- $\text{AuthoriseExLogClear} \sqsubseteq \text{CAuthoriseExLogClear}$
- $\text{Archive} \sqsubseteq \text{CArchive}$
- $\text{Abort} \sqsubseteq \text{CAbort}$
- $\text{Increase} \sqsubseteq \text{CIncrease}$
- $\text{Ignore} \sqsubseteq \text{CIgnore}$
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 $B$ 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

\[ CInit \vdash \exists B' \cdot BInit \land R' \]

25.2.3 Finalisation

\[ R; CFin \vdash BFin \]

25.2.4 Applicability

\[ R; BIn \vdash \text{pre BOp} \dashv \text{pre COp} \]

25.2.5 Correctness

\[ R; COp \vdash \exists B' \cdot R' \land BOp \]

We can simplify the correctness condition because we know that all the between operations are total, i.e.

\[ \text{pre BOp} = \text{true} \]

This was proved earlier, in section 8.3.2.

We can therefore simplify the correctness condition to

\[ R; COp \vdash \exists B' \cdot R' \land BOp \]
Chapter 26

**B to C retrieve relation**

### 26.1 Retrieve state

The $B$ and $C$ worlds are identical, except that the $C$ world can 'lose' ether messages.

\[
\text{Rbc} \\
\begin{array}{ll}
\text{BetweenWorld} \\
\text{ConWorld}_0
\end{array}
\]

\[
\text{conAuthPurse}_0 = \text{conAuthPurse} \\
\text{ether}_0 = \text{ether} \\
\text{archive}_0 = \text{archive}
\]

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

\[ \text{ConInitState} \vdash \exists \text{World} \cdot \text{BetweenInitState} \land Rbc \]

Proof:
We expand ConInitState in the hypothesis according to its definition.

\[ \text{ConWorld}^0 \mid (\exists \text{World} \cdot \text{BetweenInitState} \land \text{conAuthPurse}^0 = \text{conAuthPurse} \land \text{archive}^0 = \text{archive} \land \bot \subseteq \text{ether}^0 \subseteq \text{ether}^0) \]

\[ \vdash \exists \text{World} \cdot \text{BetweenInitState} \land Rbc \]

From the definition of \( Rbc \), we can see that the consequent follows directly from the hypothesis.

\( \blacksquare \) 27.1

27.2 Finalisation proof

\[ \text{Rbc} \land \text{ConFinState} \vdash \text{BetwFinState} \]

Proof:
We have defined ConFinState and BetwFinState to have the same mathematical form.
**CHAPTER 27. INITIALISATION, FINALISATION, AND APPLICABILITY**

In the hypothesis, $Rbc$, the concrete and between purse states and archives must 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, $gAuth\text{-Purse}$ for between finalisation is identical to that for concrete finalisation.

### 27.3 Applicability proofs

Applicability follows automatically from the totality of the concrete operations as shown in section 8.4.

---

**Chapter 28**

**Lemmas for the $B$ to $C$ correctness proofs**

#### 28.1 Specialising the proof rules

For each concrete operation $COp$ and corresponding between operation $BOp$ we have to show

$$Rbc; COp \vdash \exists \text{BetweenWorld}' \cdot Rbc' \land BOp$$

Many operations are defined as the disjunction of other operations. A $COp$ will have the same branches as a corresponding $BOp$: a $Clgnore$ branch, and either a $CAbort$ or $COpOkay$ branch, or both. We split the proof obligation into $Clgnore$, $CAbort$ 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 $COp$:

- $Rbc; Clgnore \vdash \exists \text{BetweenWorld}' \cdot Rbc' \land Ignore$
- $Rbc; CAbort \vdash \exists \text{BetweenWorld}' \cdot Rbc' \land Abort$
- $Rbc; COpOkay \vdash \exists \text{BetweenWorld}' \cdot Rbc' \land BOpOkay$

The correctness of the $Clgnore$ 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

\[ \theta_{\text{BetweenWorld}'} = \theta_{\text{BetweenWorld}} \]

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 conAuthPurse0 and archive0 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').

\[
\begin{align*}
\text{conAuthPurse'} &= \text{conAuthPurse}_0 \\
\text{archive'} &= \text{archive}_0
\end{align*}
\]

mt 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 ĈOp or explicitly as in Archive).

\[
\begin{align*}
\text{ether'} &= \text{ether} \cup \{\text{mt}\}
\end{align*}
\]

We need to show that the chosen BetweenWorld' and mt 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 ĈOpOkay. Let us call this new purse value \( p \). This gives

\[
\begin{align*}
\text{conAuthPurse}_0 &= \text{conAuthPurse}_0 \ominus \{p\} \\
\text{conAuthPurse}_0 &= \text{conAuthPurse}_0 \ominus \{p\} [\text{Rbc}] \\
\text{conAuthPurse'} &= \text{conAuthPurse}_0 \ominus \{p\} [\text{choice of conAuthPurse'}]
\end{align*}
\]

This satisfies the constraint on conAuthPurse' in BOpOkay (where at most one purse changes in an identical manner to ĈOpOkay).

archive' is a function of archive and mt, defined in BOpOkay. Call this function \( f \):

\[
\begin{align*}
f : \text{Logbook} \times \text{MESSAGE} &\rightarrow \text{Logbook}
\end{align*}
\]

Because ĈOpOkay is defined in an analogous way, \( f \) also relates archive0 to archive0 and mt.

From the hypothesis we have ĈOpOkay and Rbc, and with our choice of archive' we have, respectively

\[
\begin{align*}
\text{archive}_0 &= f(\text{archive}_0, \text{mt}) \\
\Rightarrow \text{archive} &= \text{archive}_0 \\
\Rightarrow \text{archive}' &= \text{archive}_0
\end{align*}
\]

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'.

28.3.3 Case Rbc'

Both the conAuthPurse' and archive' components of BetweenWorld' satisfy Rbc' from the choice of BetweenWorld'.

All ĈOpOkay operations constrain ether' as

\[
\begin{align*}
\text{ether}_0 &= \text{ether}_0 \cup \{\text{mt}\} \\
\Rightarrow \text{ether}' &= \text{ether}_0 \cup \{\text{mt}\} [\text{Rbc}] \\
\Rightarrow \text{ether}' &= \text{ether}_0 [\text{ĈOpOkay}]
\end{align*}
\]

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 allLogs. 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.

28.3.5 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.

28.5 Correctness of CAabort

Lemma 'constraint' is not applicable, because CAabort 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 \( \bot \) message is emitted, and (possibly) one log is recorded.

B-1 req \( \Rightarrow \) authentic to purse. No new req messages.

B-2 No future reqs. No new req messages.

B-3 No future vals. No new val messages.

B-4 No future acks. 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-77, 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-77, 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 \( \{ epv, epv \} \), so no future from logs. The purse moves into eaFrom, so no new purses in \( epv \) 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 \( epv \Rightarrow \neg val \land \neg ack \). The purse moves into eaFrom, and so does not move into \( epv \).

B-10 \( req \land \neg ack \Rightarrow toInEpv \lor toLogged \).

• case =:
  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 <=:
  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 \( epv \land val \Rightarrow fromInEpa \lor fromLogged \). 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 \lor 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 exceptionLogResults 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.

\[
\text{BOpOkay } | \text{archive' = archive} \\
\land \text{req} \succ \text{ether'} = \text{req} \succ \text{ether} \\
\land \forall n \colon \text{dom conAuthPurse} \bullet (\text{conAuthPurse n}.exLog = (\text{conAuthPurse n}.exLog) \land \text{allLogs' = allLogs} \\
\land \text{toLogged' = toLogged} \\
\land \text{fromLogged' = fromLogged}
\]

[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, Clear-ExceptionLog, 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 B operation is structured analogously, it is sufficient to prove that the simpler C operation refines the corresponding B operation.

\[(C\text{Abort} \circ C\text{OpEafrom}); \text{Rbc};
\text{\begin{array}{l}
(\forall C\text{Abort}; \text{Rbc} \rightarrow \exists BetweenWorld' \cdot Rbc' \wedge \text{Abort});
(\forall C\text{OpEafrom}; \text{Rbc} \rightarrow \exists BetweenWorld' \cdot Rbc' \wedge B\text{OpEafrom})
\end{array})
\]
\[\vdash \exists BetweenWorld' \cdot Rbc' \wedge (\text{Abort} \not\circ B\text{OpEafrom})\]

\[\]
29.1 Introduction

Many of the following arguments are about constraints of the form
\[ \text{antecedent} \Rightarrow \text{consequent} \]

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 \texttt{CStartFrom}

\texttt{StartFromOkay} comprises \texttt{AbortPurse} followed by \texttt{StartFromEafromPurseOkay} at the unpromoted level. As a result, we can apply lemma ‘abort forward’ (section C.8), leaving us to prove the correctness of \texttt{StartFromEafromPurseOkay}. 
Lemma ‘constraint’ is not applicable, because $\text{StartFromEaFromPurseOkay}$ changes status: it moves the purse from $\text{eaFrom}$ into $\text{epr}$. Therefore we have to show that our chosen $\text{BetweenWorld}'$ 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 $\text{allLogs}$, $\text{fromLogged}$ and $\text{toLogged}$ remain unchanged.

B-1 $\text{req} \Rightarrow \text{authentic to purse. No new req messages.}$

B-2 No future reqs. No new req messages.

B-3 No future vals. No new val messages.

B-4 No future acks. 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 $\{\text{epr, epa}\} \Rightarrow \text{no future from logs. There are no new logs, but the purse moves into epr, so we must prove that the constraint for this purse holds (for all other purses in epv, the constraint holds beforehand, and so holds afterwards). In StartFrom, the post-state pdAuth fromSeqNo is equal to pre-state nextSeqNo. Coupling this with constraint B-5 we have}$

$$\forall pd:\text{fromLogged} | \text{pd.from = name? \bullet pd.fromSeqNo < (\text{conAuthPurse pd from}).pdAuth fromSeqNo}$$

Since the logs don’t change we have

$$\forall pd:\text{fromLogged} | \text{pd.from = name? \bullet pd.fromSeqNo < (\text{conAuthPurse pd from}).pdAuth fromSeqNo}$$

which proves the constraint for purse name?.

B-8 to in $\{\text{epv, eaTo}\} \Rightarrow \text{no future to logs. No new logs, and the purse moves into epr.}$

B-9 $\text{epr} \Rightarrow \neg \text{val} \land \neg \text{ack}$. 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

$$pd = (\text{conAuthPurse name?}).pdAuth$$

Noting that $pd . \text{from} = \text{name?}$, the definition of $\text{StartFrom}$ then gives us that

$$\left( \text{conAuthPurse name?}.nextSeqNo = \text{pdAuth fromSeqNo} \Rightarrow \text{pdAuth fromSeqNo} \right)$$

$\Rightarrow \text{val pd} \notin \text{ether}$ $\Rightarrow \text{ack pd} \notin \text{ether}$ $\Rightarrow \text{fromInEpv}$ $\Rightarrow \text{fromLogged}$ $\Rightarrow \text{no future in epv}$ $\Rightarrow \text{no new val messages. The purse did not move out of epv}$

B-10 $\text{req} \land \neg \text{ack} \Rightarrow \text{toInEpv v toLogged,}$

• case $\Rightarrow$

No new req messages. The purse moved from $\text{eaFrom}$ to $\text{epr}$ without generating new logs. Hence, true before implies true after.

• case $\Leftarrow$

No purses newly in $\text{epv}$ and no new logs. No acks added to the $\text{ether}$.

B-11 $\text{epv} \land \text{val} \Rightarrow \text{fromInEpv v fromLogged}$. No purses newly in $\text{epv}$; no new val messages. The purse did not move out of $\text{epv}$.

B-12 fromInEpv v fromLogged $\Rightarrow$ req. No purses newly in $\text{epv}$; no new logs.

B-13 toLogged finite. No new logs.

B-14 exceptionLogResults in $\text{allLogs}$. 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 $\text{StartToEaFromPurseOkay}$ moves one purse into $\text{epv}$, and it was not in this state before. Therefore we have to show that our chosen $\text{BetweenWorld}'$ 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.
29.3. CORRECTNESS OF CSTARTTO

B-1 req ⇒ authentic to purse. One new req, which refers to the name?, purge as the to purse. ΦBOp states that this purge is authentic.

B-2 No future reqs. 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 (itsself) has a larger nextSeqNo after the operation.

B-3 No future vals. No new val messages.

B-4 No future acks. 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. There are no new logs and the purse moves into epv, so this constraint does not apply to this purge.

B-8 to in \{epr, eaTo\} ⇒ 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 epr, 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

\[ \forall pd: \text{toLogged} \mid pd.to = \text{name}? \quad \text{pd.toSeqNo} < (\text{conAuthPurse}(\text{pd.to}), \text{pdAuth.toSeqNo} \]

Since the logs don’t change, we have

\[ \forall pd: \text{toLogged} \mid pd.to = \text{name}? \quad \text{pd.toSeqNo} < (\text{conAuthPurse}(\text{pd.to}), \text{pdAuth.toSeqNo} \]

which proves the constraint for purse name?.

B-9 epr ⇒ ¬ val ∧ ¬ ack. No purses newly in epr; no new vals or acks.

B-10 req ∧ ¬ ack ⇒ toInEpv ∨ 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 =:

We must prove toInEpv ∨ toLogged. The purse moves into epv, thus establishing the consequent.

• case <=:

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

\[ pd = (\text{conAuthPurse}(\text{name}?)). \text{pdAuth} \]

Noting that pd.to = name?, the definition of StartTo gives us that

\[ (\text{conAuthPurse}(\text{name}?)). \text{nextSeqNo} = (\text{conAuthPurse}(\text{name}?)). \text{pdAuth.toSeqNo} \]

⇒ (conAuthPurse pd.to).nextSeqNo = pd.toSeqNo

⇒ ack pd ≠ ether \quad \text{[BetweenWorld constraint B-4]}

Hence, we have the corresponding req but no ack.

B-11 epv ∧ val ⇒ fromInEpa ∨ 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

\[ pd = (\text{conAuthPurse}(\text{name}?)). \text{pdAuth} \]

Noting that pd.to = name?, the definition of StartTo gives us that

\[ (\text{conAuthPurse}(\text{name}?)). \text{nextSeqNo} = (\text{conAuthPurse}(\text{name}?)). \text{pdAuth.toSeqNo} \]

⇒ (conAuthPurse pd.to).nextSeqNo = pd.toSeqNo

⇒ val pd ≠ ether \quad \text{[BetweenWorld constraint B-3]}

Hence, there is no val before, and no val is emitted by this operation.

B-12 fromInEpa ∨ fromLogged ⇒ req. No purses newly in epv; 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.
29.4 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 reqs. No new req messages.
B-3 No future vals. Req puts a val in the ether’. Let pd be the pay details of the val. Hence,

\[
\begin{align*}
    pd &= (\text{ConPurse name?}), pd\text{Auth} \\
    m? &= \text{req pd} \\
    m &= \text{val pd}
\end{align*}
\]

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?:

\[
\begin{align*}
    pd\text{toSeqNo} &< (\text{ConPurse pd.to}), next\text{SeqNo} \\
    pd\text{fromSeqNo} &< (\text{ConPurse pd.from}), next\text{SeqNo}
\end{align*}
\]

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

\[
    pd\text{fromSeqNo} < next\text{SeqNo}
\]

Since Req does not alter nextSeqNo, we thus have

\[
    pd\text{fromSeqNo} < (\text{ConPurse pd.from}), next\text{SeqNo}
\]

B-4 No future acks. 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.

29.5 Correctness of CVal

Lemma ‘constraint’ is not applicable, because a purse moves from epv to ea-Payee 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 reqs. Val emits no new req messages.
B-3 No future vals. Val emits no new val messages.
29.6. CORRECTNESS OF CACK

Lemma `constraint` is not applicable, because a purse moves from *epa* to *ea*-Payer. Therefore we have to show that our chosen *BetweenWorld* obeys the constraints.

It emits a `\{ \}` message. We can invoke lemma `logs unchanged`, section C.7, because no new req messages are produced, no new purge 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 reqs. No new req messages.
B-3 No future vals. No new val messages.
B-4 No future acks. 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; no purses newly in *epr* or *epa*.
B-8 to in \{ *epv*, *eaTo* \} ⇒ no future to logs. No new logs.

The to purse moves from *epv* into *eaTo*. *BetweenWorld* constraint B-8 held on *epv*.

B-9 *epv* ⇒ ~ *val* ∧ ~ *ack*. No purses newly in *epv*.

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.

\[ \text{B-10 req } \land \neg \text{ack} \Rightarrow \text{toInEpv} \lor \text{toLogged}. \]

- 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.

\[ \text{B-11 epv } \land \text{val} \Rightarrow \text{fromInEpa } \lor \text{fromLogged}. \] No purses newly in *epv*; no new *val* messages; no purses leaving *epa*, no changing logs.

\[ \text{B-12 fromInEpa } \lor \text{fromLogged} \Rightarrow \text{req}. \] 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.

\[ \text{■ 29.5} \]

29.6 Correctness of CACK

Lemma `constraint` is not applicable, because a purse moves from *epa* to *ea*-Payer. Therefore we have to show that our chosen *BetweenWorld* obeys the constraints.

It emits a `\{ \}` message. We can invoke lemma `logs unchanged`, section C.7, because no new req messages are produced, no new purge logs are produced, and the archive does not change. Therefore, the sets allLogs, fromLogged and toLogged remain unchanged.
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 ⇒ authentic to purse. No new req messages.
B–2 No future reqs. No new req messages.
B–3 No future vals. No new val messages.
B–4 No future acks. 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 purses newly in epr or epa.
B–8 to in {epv, eaTo} ⇒ no future to logs. No purses newly in epv or eaTo.
B–9 epr ⇒ ¬ val ∧ ¬ ack. No purses newly in epr; no new vals or acks.
B–10 req ∧ ¬ ack ⇒ toInEpv v toLogged.
  • case ⇒: No new reqs; no new acks; no purses moving out of epv; no logs lost.
  • case ⇒: No purses newly in epv; no new logs.
B–11 epv ∧ val ⇒ fromInEpa v fromLogged. No purses newly in epv; no new vals; no purse moves out of epv; no logs lost.
B–12 fromInEpa v fromLogged ⇒ req. No purses newly in epv; 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

\[ m \in \{\} \cup \{ld : exLog \in \text{exceptionLogResult}(name, ld)\} \]

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 ⊥ 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 ⇒ authentic to purse. No new req messages.
B–2 No future reqs. No new req messages.
B–3 No future vals. No new val messages.
B–4 No future acks. No new ack messages.
B–5 No future from logs. No new logs.

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 ⊥ 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 ⇒ authentic to purse. No new req messages.
B–2 No future reqs. No new req messages.
B–3 No future vals. No new val messages.
B–4 No future acks. No new ack messages.
B–5 No future from logs. No new logs.
202  CHAPTER 29. CORRECTNESS PROOFS

B–6 No future to logs. No new logs.
B–7 from in \{epv, epa\} ⇒ no future from logs. No purses newly in epv or epa.
B–8 to in \{epv, eaTo\} ⇒ no future to logs. No purses newly in epv or eaTo.
B–9 epv ⇒ ¬val ∧ ¬ack. No purses newly in epv; no new vals or acks.
B–10 req ∧ ¬ack ⇒ tolnEpv ∨ toLogged.
   • case ⇒: No new reqs; no new acks; no purses moving out of epv; no logs lost.
   • case ⇐: No purses newly in epv; no new logs.
B–11 epv ∧ val ⇒ fromlnEpa ∨ fromLogged. No purses newly in epv; no new vals; no purse moves out of epv; no logs lost.
B–12 fromlnEpa ∨ fromLogged ⇒ req. No purses newly in epv; 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 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.
CHAPTER 30. SUMMARY

System secure by definition
1. Abstract preserves security properties by definition
1.1. Abstract Ignored upholds properties
2.4
1.2. AbTransfer upholds properties
2.4
1.2.1. SP 1 (Okay
2.4
1.2.1.2. Lost
2.4
1.2.2. SP 2.1 (Okay
2.4
1.2.2.2. Lost
2.4
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
11.2
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
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
9 and 14.3
4.3.2.1. AbTransfer 14.7
4.3.2.1.2. Okay 14.7
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 (Okay
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 14.8
4.3.2.2.1.3.2. Epayer operation 14.7
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 14.8
4.3.2.2.2.3.1. StartTo 14.3
4.3.2.2.2.3.2. Epayer operation 14.7
4.3.2.2.2.3.2.1. Ignore 14.7
4.3.2.2.2.3.2.2. Abort 14.8
4.3.2.2.2.3.2.3. Okay 14.8
4.3.2.2.2.3.2.3.1. Epayer operation 14.7
4.3.2.2.2.3.2.3.2. Ignore 14.7
4.3.2.2.2.3.2.3.2.1. Okay 14.8
4.3.2.2.2.3.2.3.2.2. Epayer operation 14.7
4.3.2.2.2.3.2.3.2.2.1. Ignore 14.7
4.3.2.2.2.3.2.3.2.2.2. Abort 14.8
4.3.2.2.2.3.2.3.2.2.3. Okay 14.8
4.3.2.2.2.3.2.3.2.2.3.1. Epayer operation 14.7
4.3.2.2.2.3.2.3.2.2.3.1.1. Ignore 14.7
4.3.2.2.2.3.2.3.2.2.3.1.1.1. Okay 14.8
4.3.2.2.2.3.2.3.2.2.3.1.1.1.1. Epayer operation 14.7
4.3.2.2.2.3.2.3.2.2.3.1.1.1.1.1. Ignore 14.7
4.3.2.2.2.3.2.3.2.2.3.1.1.1.1.1.1. Okay 14.8
CHAPTER 30. SUMMARY

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 C.5
4.3.2.2.7.3. Abort 14.8
4.3.2.2.7.4. 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. Ea from branch 29.3
5.3.1.1.1.2. Abort branch 28.5
5.3.1.1.2. C ignore branch 28.2
5.3.1.1.3. C abort 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. Ea from branch 29.2
5.3.1.2.1.2. Abort branch 28.5
5.3.1.2.2. C ignore branch 28.2
5.3.1.2.3. C abort 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. C ignore 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. C ignore 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. C ignore 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. Ea from branch 29.7
5.3.1.6.1.2. Abort branch 28.5
5.3.1.6.2. C ignore 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. Ea from branch 29.8
5.3.1.7.1.2. Abort branch 28.5
5.3.1.7.2. C ignore branch 28.2
5.3.1.7.3. C abort 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. C ignore 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
A.1 Notation

The notation

\[ \text{Abs } \sqsubseteq \text{Conc} \]

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

\[ \forall D \mid P \cdot Q \]

is always true is expressed as the theorem

\[ \vdash \forall D \mid P \cdot Q \]

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.
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

\[
\frac{P_1 \quad P_2 \quad \ldots \quad P_n}{C} \quad [\text{erule name}]
\]

says that conclusion \( C \) can be inferred if every premise \( P_i \) can be proved. (The rule name is used for labelling proof steps.)

The inference rule

\[
\frac{P_1, P_2, \ldots, P_n}{C} \quad [\text{erule name}]
\]

says that conclusion \( C \) can be inferred if any premise \( P_i \) can be proved.

B.1 Universal quantifier becomes hypothesis

\[
\frac{S \vdash P}{\vdash \forall S \cdot P} \quad [\text{uni hyp}]
\]
APPENDIX B. INFERENCE RULES

B.2 Disjunction in the hypothesis

Given an hypothesis containing a disjunct, it is sufficient to prove the theorem for each case.

\[
\frac{R \vdash P \quad S \vdash P}{R \lor S \vdash P} \quad [\text{hyp disj}]
\]

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).

\[
\frac{R \vdash P, R \vdash Q}{R \vdash P \lor Q} \quad [\text{consq disj}]
\]

B.4 Conjunction in the consequent

Given a consequent containing a conjunct, it is sufficient to prove the theorem for each case separately.

\[
\frac{R \vdash P \quad R \vdash Q}{R \vdash P \land Q} \quad [\text{consq conj}]
\]

We can add conjuncts to the consequent (since this is a harder thing to prove).

\[
\frac{R \vdash P \land Q}{R \vdash P} \quad [\text{strengthen consq}]
\]

B.5 Cut for lemmas

Cut is a way to introduce new hypotheses, and discharge them as lemmas.

\[
\frac{R, D \mid Q \vdash P \quad R \vdash \exists D \bullet Q}{R \vdash P} \quad [\text{cut}]
\]

B.6 Thin

We can remove assumptions.

\[
\frac{\vdash R}{P \vdash R} \quad [\text{thin}]
\]

B.7 Universal Quantification

Universals can be replaced by a particular choice in the hypothesis

\[
\frac{x \in X \Rightarrow P(x) \vdash R}{\forall x : X \bullet P(x) \vdash R} \quad [\text{hyp uni}]
\]

B.8 Negation

In order to prove something, you can assume its negation.

\[
\frac{\neg P \vdash}{
 \frac{\vdash R}{P \vdash}} \quad [\text{negation}]
\]

B.9 Contradiction

If \( R \) can be proved, assuming its negation allows you to prove anything (because \( false \Rightarrow anything \)).

\[
\frac{\vdash R}{\neg R \vdash anything} \quad [\text{contradiction}]
\]

B.10 One Point Rule

In order to prove there exists a value with a property, it is enough to exhibit such a value.

\[
\frac{\vdash P[t/x]}{\vdash \exists x \bullet P \land x = t} \quad [\text{one point}]
\]

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**

\[
\frac{P \vdash Q}{P \vdash \exists P \cdot Q} \quad \text{[consq exists]}
\]

and very similarly

\[
\frac{P \vdash Q}{P \vdash (\exists P) \land Q} \quad \text{[consq exists]}
\]

**B.11.2 Existential in the hypothesis**

\[
\frac{x : X; D \mid P \vdash \exists \left(\exists x : X \mid P\right)}{D \mid \exists x : X \cdot P \vdash \exists \left(\exists x : X \mid P\right)} \quad \text{[cut in \(x : X\)]}
\]

\[
\frac{D \mid x : X \mid P \vdash (\exists x : X \cdot P)}{D \mid x : X \mid P \vdash \exists x : X \cdot P} \quad \text{[discharge side lemma from hyp]}
\]

\[
\frac{D \mid x : X \mid P \vdash \exists x : X \cdot P}{D \mid P \vdash \exists x : X \cdot P} \quad \text{[thin]}
\]
as required.

**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 \(p\).

\[
\frac{D \mid p \vdash \exists D \cdot p \land q}{D \mid p \vdash \exists D \cdot p \land q} \quad \text{[starting point]}
\]

\[
\frac{D \mid p \vdash \exists D' \cdot p[D'/D] \land q[D'/D]}{D \mid p \vdash \exists D' \cdot p[D'/D] \land q[D'/D]} \quad \text{[rename bound declaration]}
\]

\[
\frac{D \mid p \vdash \exists D' \cdot p[D'/D] \land q[D'/D] \land D' = D}{D \mid p \vdash \exists D' \cdot p[D'/D] \land q[D'/D] \land D' = D} \quad \text{[strengthen consequent]}
\]

\[
\frac{D \mid p \vdash p[D'/D'] \land q[D'/D][D/D']}{D \mid p \vdash p[D'/D'] \land q[D'/D][D/D']} \quad \text{[one point rule]}
\]

\[
\frac{D \mid p \vdash p \land q}{D \mid p \vdash p \land q} \quad \text{[simplify renaming]}
\]

\[
\frac{D \mid p \vdash q}{D \mid p \vdash q} \quad \text{[discharge \(p\) from hyp]}
\]

The second one-point-cut rule follows exactly the same way, except that \(q\) is not bound by the existential, and so none of the renamings alters it.
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:

$$\Phi \BoxOp; \BoxOpPurseOkay; \BoxOpOut; \BoxOpCT; \BoxOpIn \vdash \exists pd \text{This} : \text{PayDetails} \land P$$

exists-chosenLost:

$$\Phi \BoxOp; \BoxOpPurseOkay; \BoxOpOut; \BoxOpCIpd[pdThis/pdThis']; \BoxOpIn \vdash P \land \exists chosenLost : \text{PayDetails} \land Q \land chosenLost \subseteq maybeLost$$

cHECK-operation:

$$\Phi \BoxOp; \BoxOpPurseOkay; \BoxOpOut; \BoxOpCIpd[pdThis/pdThis']; \AbWorld; \BoxOpCIpd; \BoxOpIn \vdash P \land Q \vdash AOp$$

---

C.2 Lemma ‘lost unchanged’

Lemma 2 (lost unchanged) For \( BOp = Lost \) operations, where we have that \( maybeLost = maybeLost \) and \( definitelyLost = definitelyLost \), the proof obligations \( \exists psd \) and \( \exists chosenLost \) are satisfied automatically by the instantiation of the predicates \( P \) and \( Q \) as: 2

\[
\begin{align*}
P & \iff true \\
Q & \iff chosenLost = chosenLost'
\end{align*}
\]

Proof:
See section 14.4.5.

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

\[
\begin{align*}
\Phi & BOp; BOpIgPurse; RabCIPd'[\text{pdThis}/\text{pdThis}']; AbWorld; RabCIPd | P \land Q \\
\vdash & \forall n : \text{dom.abAuthPurse} \times (\text{abAuthPurse}\ n).\text{lost} = (\text{abAuthPurse}\ n).\text{lost} \\
& \land (\text{abAuthPurse}\ n).\text{balance} = (\text{abAuthPurse}\ n).\text{balance}
\end{align*}
\]

Proof:
See section 14.5

C.4 Lemma ‘Abort refines AbIgnore’

Lemma 4 (Abort refines AbIgnore) Concrete \( Abort \) refines abstract \( AbIgnore \).

\[
\text{Abort; Rab; RabOut} \vdash \exists \text{AbWorld; ai : AIN} \land \text{RabIn} \land \text{AbIgnore}
\]

Proof:
See section 14.6

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

\footnote{Used in: ‘Ignore’, section 14.7; lemma ‘Abort refines AbIgnore’, section 14.8; used to simplify every \( A \to B \) operation proof that refines \( AbIgnore \).}

\footnote{Used in: lemma ‘abort backward’, section C.5}
the relevant abstract operation. 1
( ∃ ∆ ConPurse • BOp ∧ (AbortPurseOkay ∩ BOpPurseOkay));
Rab′; RabOut;
( ∃ ConPurse′; Rab′; RabOut •
∃ AbWorld; a′: AIN • Rab ∧ Rab′ ∧ AOp)

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
PayDetails representing all the logs does not change either. 7

BOpOkay | archive′ = archive
∧ (ran req) ∩ ether′ = (ran req) ∩ ether •
∧ ∃ n : dom conAuthPurse •
(conAuthPurse n).exLog = (conAuthPurse n).exLog

⊢ allLogs′ = allLogs
∧ toLogged′ = toLogged
∧ fromLogged′ = fromLogged

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
B operation is structured similarly, it is sufficient to prove that the simpler C
operation refines corresponding B operation 8.

(CAbort ∪ COpEafrom); Rbc;
( ∃ BetweenWorld′ • Rbc ∧ BOpEafrom)

⊢
( ∃ BetweenWorld′ • Rbc ∧ (Abort ∪ BOpEafrom))

Proof:
See section 28.7.
■ C.8

1Used in: StartFrom, section 16; StartTo, section 17; ClearExceptionLog, section 22; ReadExceptionLog, section 21
2Used in: Increase, section 28.4; CClearExceptionLog, section 29.8; CArchive, section 29.10.
C.9 Lemma 'compose backward'

Lemma C.1 (compose backward) If, under the backwards refinement rules, a concrete operation \( \text{COp}_1 \) is a refinement of abstract operation \( \text{AOp}_1 \), and \( \text{COp}_2 \) is a refinement of \( \text{AOp}_2 \), then their composition is a refinement of the abstract composition.

\[
\text{(COp}_1 \triangleleft \text{COp}_2) \land R' \land \text{ROut};
\]

\[
\text{(V COp}_2, R', \text{ROut} \land (\exists A; \text{AIn} \land R \land \text{RIn} \land \text{AOp}_1));
\]

\[
\text{(V COp}_2, R', \text{ROut} \land (\exists A; \text{AIn} \land R \land \text{RIn} \land \text{AOp}_2));
\]

\[
\models \exists A; \text{AIn} \land R \land \text{RIn} \land (\exists A, A \land \text{AOp}_1 \land \text{AOp}_2)
\]

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.

\[
\exists c \land \text{COp}_1(c/c) \land \text{COp}_2(c/a);
\]

\[
\text{R}' \land \text{ROut};
\]

\[
(\forall \text{COp}_1(c/c) \land R_0 \land \text{ROut} \land (\exists A; \text{AIn} \land R \land \text{RIn} \land \text{AOp}_1(a/a')));
\]

\[
(\forall \text{COp}_2(c/a) \land R_0 \land \text{ROut} \land (\exists A; \text{AIn} \land R \land \text{RIn} \land \text{AOp}_2(a/a')));
\]

\[
\models \exists A; \text{AIn} \land R \land \text{RIn} \land (\exists A_0 \land \text{AOp}_1(a_0/a') \land \text{AOp}_2(a_0/a))
\]

Use \([\text{hypo exists}]\) to drop the \( \exists \) in the hypothesis, then simplify.

C.10 Lemma 'compose forward'

Lemma C.2 (compose forward) If, under the forwards refinement rules, a concrete operation \( \text{COp}_1 \) is a refinement of abstract operation \( \text{AOp}_1 \), and \( \text{COp}_2 \) is a refinement of \( \text{AOp}_2 \), then their composition is a refinement of the abstract composition.

\[
\text{Use } D \land (\forall D \land P) \Rightarrow P \text{ to simplify the second universal quantifier in the hypothesis.}
\]

\[
\text{COp}_1(a/a'); \text{COp}_2(a/a); R'; \text{ROut};
\]

\[
(\exists A; \text{AIn} \land R \land \text{RIn} \land \text{AOp}_1(a_0/a'))\)

\[
(\exists A; \text{AIn} \land R \land \text{RIn} \land \text{AOp}_2(a_0/a))
\]

\[
\models \exists A; \text{AIn} \land R \land \text{RIn} \land (\exists A_0 \land \text{AOp}_1(a/a) \land \text{AOp}_2(a/a))
\]

Repeat the previous three steps to simplify the remaining quantifier in the hypothesis.

\[
\text{AOp}_1(a_0/a') \land \text{AOp}_2(a_0/a)
\]

\[
\models \exists A; \text{AIn} \land R \land \text{RIn} \land (\exists A_0 \land \text{AOp}_1(a/a') \land \text{AOp}_2(a_0/a))
\]

Move the inner \( \exists \) in the consequent outsides.

\[
\text{AOp}_1(a_0/a') \land \text{AOp}_2(a_0/a)
\]

\[
\models \exists A; \text{AIn} \land R \land \text{RIn} \land (\exists A_0 \land \text{AOp}_1(a_0/a') \land \text{AOp}_2(a_0/a))
\]

All the terms are in the hypothesis.

C.9
composition \(^{10}\).

\[
(COp_1 \circ COp_2); \quad R; \\
(\forall COp_1; \quad R \bullet (\exists A' \bullet R' \land AOp_1)); \\
(\forall COp_2; \quad R \bullet (\exists A' \bullet R' \land AOp_2)) \\
\vdash \\
\exists A' \bullet R' \land (AOp_1 \circ AOp_2)
\]

\[\square\]

**Proof:**
Follows as for lemma ‘compose backward’, above.
\[\square\]

**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, \(\Phi\)

\[
\begin{align*}
\text{Local} \\
x: X
\end{align*}
\]

\[
\begin{align*}
\text{Global} \\
\text{locals}: \text{NAME} \rightarrow \text{Local}
\end{align*}
\]

\[
\begin{align*}
\Phi \\
\Delta\text{Global} \\
\Delta\text{Local} \\
n? : \text{NAME} \\
n? \in \text{dom locals} \\
\text{locals} n? = 0\text{Local} \\
\text{locals}' = \text{locals} \oplus \{n? \mapsto 0\text{Local}'\}
\end{align*}
\]

\[\Phi: OOp_1; OOp_2; \\
(\exists \Delta\text{Local} \bullet \Phi \land (OOp_1 \circ OOp_2)) \\
= (\exists \Delta\text{Local} \bullet \Phi \land OOp_1) \circ (\exists \Delta\text{Local} \bullet \Phi \land OOp_2)
\]

\[\square\]

**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 \(\Phi\).

\[
(\exists \Delta\text{Local} \bullet \Phi \land OOp_1) \circ (\exists \Delta\text{Local} \bullet \Phi \land OOp_2) \\
= (\exists \text{Global}_0 \bullet (\exists \Delta\text{Local} \bullet \Phi(\text{locals}_0/\text{locals}_0') \land OOp_1) \\
\land (\exists \Delta\text{Local} \bullet \Phi(\text{locals}_0/\text{locals}) \land OOp_2)) \\
= (\exists \Delta\text{Local} \bullet \Phi \\
\{\text{locals}; \text{locals}_0 : \text{NAME} \rightarrow \text{Local} | \\
n? \in \text{dom locals} \\
\land \text{locals} n? = 0\text{Local} \\
\land \text{locals}_0 = \text{locals} \oplus \{n? \mapsto 0\text{Local}'\}\} \\
\land OOp_1) \\
\land (\exists \Delta\text{Local} \bullet \Phi \\
\{\text{locals}_0; \text{locals}' : \text{NAME} \rightarrow \text{Local} | \\
n? \in \text{dom locals}_0 \\
\land \text{locals}_0 n? = 0\text{Local} \\
\land \text{locals}' = \text{locals}_0 \oplus \{n? \mapsto 0\text{Local}'\}\} \\
\land OOp_2)
\]

Rename the after state in the first operation to \(\text{Locals}_0\) and the before state in the second operation to \(\text{Locals}'\). Choosing different names makes it easier to

\[\square\]

\(^{10}\)Used in: lemma ‘abort forward’, section 28.7.
\(^{11}\)Used in: lemma ‘abort backward’, section C.5
combine the schemas across the quantifiers.

= ∃ Global, •
    ( ∃ Local; Locals •
        ( locals; locals*: NAME → Local |
        n? ∈ dom locals
        ∧ locals n? = 0Local
        ∧ locals* = locals ⋈ (n? → 0Local*) )
        ∧ Op1[xn/x]* )
    ∧ ( ∃ Locals; Locals*
        ( locals0; locals*: NAME → Local |
        n? ∈ dom locals
        ∧ locals0 n? = 0Local
        ∧ locals* = locals0 ⋈ (n? → 0Local*) )
        ∧ Op2[z0/x] )

Combine all these as a single schema, putting the quantifications into the predicate.

= ( locals; locals*: NAME → Local |
    ∃ Locals; Local; Local*; Locals0; Locals0 •
    n? ∈ dom locals
    ∧ locals n? = 0Local
    ∧ locals0 n? = 0Local
    ∧ locals0 n? = 0Local
    ∧ locals* = locals0 ⋈ (n? → 0Local*)
    ∧ Op1[xn/x]*
    ∧ Op2[z0/x] )

We can remove the quantification of local0 because we have a full definition of it in terms of other variables. This leaves the following equations relating the remaining variables.

= ( locals; locals*: NAME → Local |
    ∃ Locals; Local*; Locals0; Locals0 •
    n? ∈ dom locals
    ∧ locals n? = 0Local
    ∧ 0Locals = 0Local
    ∧ locals* = locals n? (n? → 0Local*)
    ∧ Op1[xn/x]*
    ∧ Op2[z0/x] )

C.12. Lemma ‘notLoggedAndIn’

Using the equation that ∂Local0 = ∂Localv, rename Localu and Local0 both to Localv.

= ( locals; locals*: NAME → Local |
    ∃ Locals; Local*; Locals0 •
    n? ∈ dom locals
    ∧ locals n? = 0Local
    ∧ locals* = locals ⋈ (n? → 0Local*)
    ∧ Op1[xn/x]*
    ∧ Op2[z0/x] )

Redistribute the quantifications

= ( locals; Local* •
    ( locals; locals*: NAME → Local |
    n? ∈ dom locals
    ∧ locals n? = 0Local
    ∧ locals* = locals ⋈ (n? → 0Local*)
    ∧ (∃ Locals • Op1[xn/x]* ⋈ Op2[z0/x] ) )

and rewrite in terms of composition

= ( Locals; Local* • ⋆ ⋆ (Op1 ⋈ Op2)
= ( ∃ ∆Local; ⋆ ⋆ (Op1 ⋈ Op2)

This is the left hand side of the equation, and hence the proof is complete. ■

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.

BetweenWorld

⇒

(fromInEpr ∪ fromInEpa) ∩ fromLogged = □

(toInEpr ∪ toInEapayee) ∩ toLogged = □

12Used in: Yal, behaviour of tolagged, section 19.6.2; Ack, behaviour of definitelyLost, section 29.6.5; CVal, §10, section 29.3; lemma ‘lost’, section C.13; lemma ‘not lost before’, section C.14.
APPENDIX C. LEMMAS

Proof: Consider the to purse case. We consider the pd stored in the to purse, so

\[ pd \in \text{toLogged} \Rightarrow pd \in \text{toSeqNo} \]

We have, from BetweenWorld constraint B-8, that

\[ pd \in \text{toLogged} \Rightarrow pd \in \text{toSeqNo} \]

Hence there can be no pd in both sets.

The arguments for the from cases follow similarly, from BetweenWorld constraint B-7.

C.12 Lemma 'lost'

Lemma C.5 (lost) The sets definitelyLost and maybeLost are disjoint: a pd can never be in both. 13

\[ \text{BetweenWorld} \vdash \text{definitelyLost} \cap \text{maybeLost} = \emptyset \]

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:

\[ \text{fromLogged} = \text{fromLogged}' \]
\[ \text{toLogged} = \text{toLogged}' \]
\[ \text{toInEpv} = \text{toInEpv}' \]

For the set fromInEpa, we know from ReqOkay that beforehand this pdThis was not in the set and afterward it was. So

\[ pdThis \in \text{fromInEpa} \]
\[ \text{fromInEpa} = \text{fromInEpa}' \]

From Lemma ‘notLoggedAndIn’ (section C.12), we have:

\[ pdThis \in \text{fromInEpa}' \Rightarrow pdThis \notin \text{fromLogged} \]

Reminding ourselves of the definitions of definitelyLost and using the identities above, we have

\[ \text{definitelyLost} = \text{toLogged} \cap \text{fromLogged} \]

\[ \text{toLogged}' \cap (\text{fromLogged} \cup \text{fromInEpa}) \]

\[ \text{fromInEpa}' \]

\[ \text{pdThis} \in \text{fromLogged}' \]

\[ \text{definitelyLost}' \]

\[ \text{Spivey} \]

C.14 Lemma 'not lost before'

Lemma C.6 (not lost before) pdThis is not lost before the Req operation, although it may be lost after. 24

\[ \Phi; \text{ReqPurseOkay}; \text{pdThis : PayDetails} \mid (\text{req} \not= \text{mi}) = \text{pdThis} \]
\[ \text{definitelyLost} = \text{definitelyLost}' \setminus \{ \text{pdThis} \} \]
\[ \text{maybeLost} = \text{maybeLost}' \setminus \{ \text{pdThis} \} \]

Proof: From the definition of the way the state changes in ReqOkay we can say that

\[ \text{fromLogged} = \text{fromLogged}' \]
\[ \text{toLogged} = \text{toLogged}' \]
\[ \text{toInEpv} = \text{toInEpv}' \]

For the set fromInEpa, we know from ReqOkay that beforehand this pdThis was not in the set and afterward it was. So

\[ pdThis \in \text{fromInEpa} \]
\[ \text{fromInEpa} = \text{fromInEpa}' \]

From Lemma ‘notLoggedAndIn’ (section C.12), we have:

\[ pdThis \in \text{fromInEpa}' \Rightarrow pdThis \notin \text{fromLogged}' \]

\[ \text{definitelyLost} = \text{toLogged} \cap (\text{fromLogged} \cup \text{fromInEpa}) \]

\[ \text{fromInEpa}' \]

\[ \text{definitelyLost}' \]

\[ \text{Spivey} \]

13Used in: Req, case 1, section 18.7.1; Req, case 2, section 18.8.1; Req, case 3, section 18.9.1.

24Used in: Req, exists-chosenLost, section 18.5; Req, check-operation, section 18.6.
Similarly for maybeLost:

\[
\text{maybeLost} = \left( (\text{fromInEpa} \cup \text{fromLogged}) \cap \text{toInEpv} \right) \cap \text{toInEpv}' \quad \text{[defn]}
\]

\[
\text{maybeLost} = \left( (\text{fromInEpa}' \cup \text{fromLogged}') \cap \text{toInEpv}' \right) \cap \text{toInEpv}' \quad \text{[above]}
\]

\[
\text{maybeLost} = \left( (\text{fromInEpa}' \cup \text{fromLogged}') \cap \text{toInEpv}' \right) \cap \text{toInEpv}' \quad \{\text{pdThis} \notin \text{fromLogged}'\} \quad \text{[prop]}
\]

\[
\text{maybeLost} = \left( (\text{fromInEpa}' \cup \text{fromLogged}') \cap \text{toInEpv}' \right) \cap \text{toInEpv}' \quad \{\text{pdThis}\} \quad \{\text{def}\}
\]

\[\text{C.14}\]

**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.\(^\text{15}\)

\[
\text{BetweenWorld; chosenLost : PayDetails; pdThis : PayDetails} | \text{chosenLost} = \text{maybeLost} \\
\exists_1 \text{AbWorld} \land \text{RabClPd}
\]

**Proof:**

Each element of AbWorld has an explicit equation in Rab defining it uniquely in terms of BetweenWorld 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:

- \text{abAuthPurse : NAME} \rightarrow \text{AbPurse}
  \text{conAuthPurse} is a finite function. From the retrieve AbstractBetween the domain of abAuthPurse equals the domain of conAuthPurse, and so is finite, too.

\[\text{C.15} \text{ C}\]

\[\text{D.1 Total abstract balance}\]

The function totalAbBalance returns the total value held in a finite collection of purses.

\[
\text{totalAbBalance : (NAME} \rightarrow \text{AbPurse)} \rightarrow \mathbb{N}
\]

\[
\text{totalAbBalance} \varnothing = 0
\]

\[
\forall w : \text{NAME} \rightarrow \text{AbPurse}; n : \text{NAME}; \text{AbPurse} | n \notin \text{dom w} \cdot \text{totalAbBalance}((n \to \varnothing \text{AbPurse}) \cup w) = \text{balance} + \text{totalAbBalance w}
\]

This recursive definition is valid, because it is finite, and hence bounded.

\[\text{D.2 Total lost value}\]

The function totalLost returns the total value lost by a finite collection of purses.

\[
\text{totalLost : (NAME} \rightarrow \text{AbPurse)} \rightarrow \mathbb{N}
\]

\[
\text{totalLost} \varnothing = 0
\]

\[
\forall w : \text{NAME} \rightarrow \text{AbPurse}; n : \text{NAME}; \text{AbPurse} | n \notin \text{dom w} \cdot \text{totalLost}((n \to \varnothing \text{AbPurse}) \cup w) = \text{lost} + \text{totalLost w}
\]

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.

\[
\text{sumValue} : \text{PayDetails} \rightarrow \mathbb{N}
\]
\[
\text{sumValue} \emptyset = 0
\]
\[
\forall \text{pds} : \text{PayDetails}; \text{PayDetails} \cup \theta \text{PayDetails} \notin \text{pds} \cdot
\text{sumValue}(\{\theta \text{PayDetails}\} \cup \text{pds}) = \text{value} + \text{sumValue pds}
\]

Bibliography

[Barden et al. 1994]

[Flynn et al. 1990]

[Spivey 1992a]

[Spivey 1992b]

[Stepney]

[Woodcock & Davies 1996]
Index

ΦBOp, 31
ΦCOp, 37
⊥, 20

abAuthPurse; AbWorld, 16
AbFinOut, 18
AbFinState, 18
AbIgnore, 16
AbInitIn, 18
AbInitState, 17
AbOp, 16
Abort, 32
AbortPurseOkay, 22
AbPurse, 15
AbPurseTransfer, 16
AbstractBetween, 45
AbTransfer, 17
AbTransferLost, 17
AbTransferLostTD, 17
AbTransferOkay, 17
AbWorld, 16
AbWorldSecureOp, 16
Ack, 33
ack, 20
AckPurseOkay, 24
AIN, 16

allLogs; AuxWorld, 28
AllValueAccounted, 13
aNullIn, 16
aNullOut, 16
AOUT, 16
Archive, 27
Authentic, 14
AuthenticAckMessage, 24
authenticFrom AuxWorld, 28
AuthenticReqMessage, 24
authenticTo AuxWorld, 28
AuthenticValMessage, 24
AuthoriseExLogClearOkay, 33
AuxWorld, 28
balance, AbPurse, 15
balance, ConPurse, 20
BetweenInitState, 35
BetweenWorld, 30
BetwFinOut, 36
BetwFinState, 36
BetwInitIn, 36
CAbort, 38
CAck, 38
CArchive, 39

D.3. SUMMING VALUES

CAuthoriseExLogClear, 39
CClearExceptionLog, 38
check-operation, 58
CIgnore, 38
CIncrease, 38
CLEAR, 19
ClearExceptionLog, 33
ClearExceptionLogEapayerOkay, 33
ClearExceptionLogPurseEapayerOkay, 26
ClearExceptionLogPurseOkay, 26
ConFinState, 39
ConInitState, 39
ConPurse, 20
ConPurseAbort, 22
ConPurseAck, 22
ConPurseClear, 26
ConPurseIncrease, 21
ConPurseReq, 22
ConPurseStart, 22
ConPurseVal, 22
consequent, 114
consqconj, 116
consqdisj, 116
consqexists, 117
contradiction, 117
ConWorld, 27
CounterPartyDetails, 19
cpd; StartFromPurseEapayerOkay, 22
cpd; StartToPurseEapayerOkay, 23
cpd; ValidStartFrom, 22
cpd; ValidStartTo, 23
CReadExceptionLog, 38
CReq, 38
CStartFrom, 38
CStartTo, 38
cut, 116
CVal, 38
definitelyLost; AuxWorld, 28
eapayer, 18
epaPayer, 18
epa, 18
epr, 18
epx, 18
ether; ConWorld, 27
exceptionLogClear, 20
exceptionLogResult, 20
exists-chosenLost, 58
exists-pd, 58
exLog, ConPurse, 20
from; TransferDetails, 16
fromInEpa; AuxWorld, 28
fromInEpr; AuxWorld, 28
fromLogged; AuxWorld, 28
fromSeqNo; PayDetails, 19
GlobalWorld, 18
hypothesis, 116
hypothesis, 116
hypothesis, 114
Ignore, 32, 55
image, 19
Increase, 32
IncreasePurseOkay, 21

GlobalWorld, 18
hypothesis, 116
hypothesis, 116
hypothesis, 114
Ignore, 32, 55
image, 19
Increase, 32
IncreasePurseOkay, 21

lemma 'Abort refines AbIgnore', 61
lemma 'AbIgnore', 119
lemma 'abort backward', 65, 119
lemma 'Abort refines AbIgnore', 119
lemma ' AllocWorld unique', 125
lemma 'compose backward', 121
lemma 'Compose forward', 121
lemma 'constraint', 100, 120
lemma 'deterministic', 58, 118
lemma 'ignore', 55
lemma 'logs unchanged', 120
lemma 'lost unchanged', 59, 119
lemma 'lost', 124
lemma 'not lost before', 124
lemma 'notLoggedAndIn', 124
lemma 'promoted composition', 122
Logbook, 27
LogIfNecessary, 13
lost; AbPurse, 15
maybeLost; AuxWorld, 28
MESSAGE, 20
NAME, 15
name; ConPurse, 20
name; CounterPartyDetails, 19
negation, 116
nextSeqNo; ConPurse, 20
nextSeqNo; CounterPartyDetails, 19
NoValueCreation, 12
onepoint, 117
OtherPursesRab, 46
PayDetails, 19
pdAuth; ConPurse, 20
purse; ConWorld, 27
Rab, 46
RabCl, 45
RabCIPd, 46
RabEnd, 49
RabEndCIPd, 48
RabHasBeenLost, 49
RabHasBeenLostCIPd, 48
Rabh, 50
RabOkay, 49
RabOkayCIPd, 47
RabOut, 50
RabWillBeLost, 49
RabWillBeLostCIPd, 47
Rbc, 96
ReadExceptionLog, 33
readExceptionLog, 20
ReadExceptionLogEapayerOkay, 33
ReadExceptionLogPurseEapayerOkay, 25
ReadExceptionLogPurseOkay, 25
Req, 33
req, 20
ReqPurseOkay, 24
RetryAck, 25
RetryReq, 25
RetryVal, 25
StartFrom, 32
startFrom, 20
StartFromEapayerOkay, 33
StartFromPurseEapayerOkay, 22
StartFromPurseOkay, 23
StartTo, 32
startTo, 20
StartToEapayerOkay, 33
StartToPurseEapayerOkay, 23
StartToPurseOkay, 24
STATUS, 18
status; ConPurse, 20
strengthenconsq, 116
SufficientFundsProperty, 13
sumValue, 127
thin, 116
to; TransferDetails, 16
tolnEapayer; AuxWorld, 28
tolnEpv; AuxWorld, 28
toLogged; AuxWorld, 28
toSeqNo; PayDetails, 19
totalAbBalance, 126
totalLost, 126
transfer, 16
TransferDetails, 16
unihyp, 115
Val, 33
val, 20
ValidStartFrom, 22
ValidStartTo, 23
ValPurseOkay, 24
value; CounterPartyDetails, 19
value; TransferDetails, 16