

University of South Australia

Formal Specification and Verification of the Internet Open Trading Protocol using Coloured Petri Nets

CHUN OUYANG
B.Eng. (Hons) and M.Eng.

A thesis submitted in total fulfilment
of the requirements for the degree of
Doctor of Philosophy
in Computer Systems Engineering

Computer Systems Engineering Centre
School of Electrical and Information Engineering
University of South Australia

June 2004

UNIVERSITY OF
SOUTH AUSTRALIA
1 8 Oct 2004
EIE LIBRARY

Contents

1 Introduction 1

1.1 Background . . . 1
1.2 Problem Statement . . . 1
1.3 Research Aims and Scope . . . 2
1.4 Structure of the Thesis . . . 3

2 Electronic Commerce 5

2.1 Hierarchical Framework . . . 5
2.1.1 Technological Infrastructure . . . 5
2.1.2 Business Infrastructure . . . 6
2.1.3 Business Services . . . 7
2.1.4 E-Market Services . . . 7
2.2 Standards and Specifications . . . 7
2.2.1 System Architecture . . . 7
2.2.2 Payment and Trading Mechanisms . . . 9
2.2.3 Electronic Data Interchange . . . 14
2.3 Formal Analysis of E-Commerce Protocols . . . 14

3 Internet Open Trading Protocol 17

3.1 Structure of the IOTP Specification . . . 17
3.2 IOTP Definition . . . 18
3.2.1 Trading Roles . . . 18
3.2.2 IOTP Transactions . . . 19
3.2.3 IOTP Messages . . . 21
3.2.4 Document Exchanges . . . 24
3.2.5 Trading Exchanges . . . 26
3.2.6 Transaction Cancellation . . . 28
3.2.7 Error Handling . . . 29
3.3 Scope of Baseline IOTP . . . 33

4 Initial Assessment of the Internet Open Trading Protocol 34

4.1 Discussion of the IOTP Conceptual Framework . . . 34
4.1.1 Trading and Document Exchanges . . . 34
4.1.2 Trading Transaction and Transaction Mode . . . 36
4.1.3 Authentication Document Exchange and Transaction . . . 38

4.2 IOTP Architecture 39
4.3 Re-structuring the IOTP Specification 40

5 Protocol Engineering **42**
5.1 Layered Protocol Architectures 42
5.1.1 The OSI Reference Model 43
5.1.2 Concepts of Layering 44
5.2 Service and Protocol Specifications 44
5.2.1 Service Specification 44
5.2.2 Protocol Specification 46
5.3 Protocol Engineering Methodology 47
5.3.1 Protocol Engineering Process 47
5.3.2 Protocol Engineering Activities 48
5.4 Protocol Verification Methodology 49
5.4.1 Verification of Properties 49
5.4.2 Verification against Service 50
5.4.3 Service Validation 51

6 Formal Techniques and Tools **52**
6.1 Coloured Petri Nets 53
6.1.1 Introduction to CPNs 53
6.1.2 Example: IOTP Payment Exchange 54
6.1.3 CPN Model of Payment Exchange 54
6.1.4 Dynamic Behaviour 57
6.2 Simulation and State Space Analysis 58
6.2.1 Simulation 58
6.2.2 State Space Analysis 59
6.3 Design/CPN 61
6.3.1 CPN Editor 62
6.3.2 CPN Simulator 63
6.3.3 State Space Tool 63
6.4 Finite State Automata 64
6.4.1 Automata and Language Definitions 64
6.4.2 Language Generation 66
6.4.3 Language Comparison 66
6.5 FSM, LexTools and GraphViz 66

7 Internet Open Trading Service Definition **68**
7.1 IOTS Model 68
7.2 IOTS Primitives and Parameters 69
7.2.1 Offer Phase 71
7.2.2 Payment Phase 73
7.2.3 Delivery Phase 75
7.2.4 Authentication Phase 76
7.2.5 Cancel Phase 78

7.3 IOTS Local Service Languages . . . 79
7.3.1 Authentication Transaction . . . 79
7.3.2 Non-Authentication Trading Transactions . . . 81
7.3.3 Authentication-combined Trading Transactions . . . 86

8 Internet Open Trading Service CPN **93**
8.1 General Conditions . . . 93
8.2 Modelling Decisions . . . 94
8.3 Overview of the IOTS CPN . . . 95
8.3.1 Net Structure . . . 95
8.3.2 The Top Level . . . 96
8.3.3 Global Declarations . . . 96
8.4 Trading Phase Pages . . . 99
8.4.1 General Page Structure . . . 99
8.4.2 Offer Phase Page . . . 101
8.4.3 Payment Phase Page . . . 103
8.4.4 Delivery Phase Page . . . 107
8.4.5 Authentication Phase Pages . . . 110
8.4.6 Cancel Phase Pages . . . 117

9 Validation of the Internet Open Trading Service **125**
9.1 Validation of the IOTS . . . 125
9.1.1 State Space Analysis . . . 125
9.1.2 Language Generation . . . 127
9.1.3 Validating the Global IOTS Language . . . 127
9.2 Revised IOTS CPN model and its Analysis . . . 129
9.2.1 Revised IOTS CPN Model . . . 129
9.2.2 Validation of the Revised IOTS CPN Model . . . 130
9.2.3 Language Analysis . . . 131

10 Internet Open Trading Protocol CPN **133**
10.1 IOTP Structure . . . 134
10.2 Scope and Assumptions of the IOTP CPN . . . 134
10.2.1 Transactions and Exchanges . . . 136
10.2.2 Service Primitives . . . 138
10.2.3 Security Considerations . . . 138
10.2.4 Transaction Cancellation . . . 139
10.2.5 Error Handling . . . 139
10.2.6 IOTP Messages . . . 140
10.2.7 Transport Service Provider . . . 141
10.3 Hierarchy Page . . . 142
10.4 Overview Page . . . 143
10.5 IOTP Messages and Communication Queue . . . 144
10.6 Trading Role Pages . . . 146
10.6.1 Consumer Trading Role Page . . . 146

10.6.2 Merchant Trading Role Page . . . 149
10.6.3 Payment Handler Trading Role Page . . . 150
10.6.4 Delivery Handler Trading Role Page . . . 150
10.7 Transaction Pages . . . 151
10.7.1 Authentication Transaction . . . 151
10.7.2 Purchase Transaction . . . 151
10.7.3 Other Payment-related Transactions . . . 152
10.8 Exchange Level Pages . . . 153
10.8.1 General Page Structure . . . 153
10.8.2 Authentication Document Exchange . . . 156
10.8.3 Offer Document Exchanges . . . 159
10.8.4 Payment Document Exchanges . . . 163
10.8.5 Payment-and-Delivery Document Exchange . . . 166
10.8.6 Delivery Document Exchange . . . 166
10.8.7 Transaction Cancellation . . . 169
10.8.8 Error Handling . . . 171

11 Analysis of Internet Open Trading Protocol Properties 177
11.1 Analysis Approach . . . 177
11.1.1 Transaction Analysis Simplification . . . 177
11.1.2 Parameters and Configurations of IOTP . . . 178
11.2 Desired Properties of Transactions . . . 178
11.3 Transaction Termination . . . 181
11.3.1 Inspection of Terminal States . . . 181
11.3.2 Problem 1: Merchant Not Informed of Cancelled Transaction . . . 187
11.3.3 Problem 2: Solely Cancelled Transaction by Consumer . . . 188
11.4 Livelocks . . . 192
11.5 Dead Transitions . . . 192

12 Verification Against the Service Specification 193
12.1 Verification Approach . . . 193
12.2 Language Comparison for Successful Transactions . . . 194
12.2.1 Authentication Transaction . . . 194
12.2.2 Deposit Transaction . . . 194
12.2.3 Purchase Transaction . . . 197
12.2.4 Value Exchange Transaction . . . 200
12.3 Valid Implementation of Successful Transaction Service . . . 201
12.4 Verification of Successful Transactions . . . 204
12.5 Verification of Transactions with Cancellation . . . 204
12.5.1 Authentication Transaction . . . 205
12.5.2 Deposit Transaction . . . 206
12.5.3 Purchase Transaction . . . 214
12.5.4 Value Exchange Transaction . . . 217
12.5.5 Discussion of the Results . . . 218

13 A Revised Internet Open Trading Protocol and its Verification **219**
13.1 Summary of Problems in IOTP 219
13.2 Solutions to Problem 1 and Discussion 219
13.2.1 Solution 1: Use of the Inquiry Transaction 220
13.2.2 Solution 2: Notification of the Transaction Status 220
13.2.3 Solution 3: "Back-End" System Operations 220
13.2.4 Discussion of Solutions 220
13.3 Solution to Problem 2 and Discussion 221
13.4 Solutions to Problem 3 and Discussion 222
13.4.1 Solution 1: Use of the Cancel Message 222
13.4.2 Solution 2: Setting Timer for User Response 222
13.4.3 Solution 3: Inclusion of Time Value with Message 222
13.4.4 Discussion of Solutions 222
13.5 Solutions Chosen 223
13.6 Changes to the IOTP CPN 223
13.6.1 Keeping the Merchant Informed of the Transaction Status 224
13.6.2 Avoiding Cancellation at the Consumer Only 226
13.6.3 Synchronizing Transaction Cancellation upon Time-Out 227
13.7 State Space and Language Analysis 228
13.7.1 Desired Terminal States and Deadlocks 228
13.7.2 Livelocks 228
13.7.3 Dead Transitions 230
13.7.4 Language Comparison 232

14 Conclusions **234**
14.1 Contributions 234
14.1.1 An Initial Assessment of IOTP 234
14.1.2 IOTS Specification and Validation 235
14.1.3 IOTP Formal Specification 235
14.1.4 Analysis of IOTP and Verification against IOTS 236
14.1.5 Revised IOTP and its Verification 236
14.2 Future Work 237
14.2.1 Extending the Current IOTP CPN 237
14.2.2 Extending the Results for Larger Values of Parameters 237
14.2.3 Enhancing the Methodology to Synthesize the Global Service Language from Local Service Languages 238
14.2.4 Verification of Other Properties of IOTP 238

References **239**

A A Revised Table of Contents for RFC 2801 **257**

B IOTS Parameter Description **262**
B.1 Transaction Parameters 262
B.2 Offer-related Parameters 263

B.3 Payment-related Parameters . . . 263
B.4 Delivery-related Parameters . . . 266
B.5 Authentication-related Parameters . . . 267
B.6 Cancel-related Parameters . . . 267

C IOTS State Space Analysis Results **269**
C.1 State Space Report for the Original IOTS CPN . . . 269
C.2 State Space Report for the Revised IOTS CPN . . . 270
C.3 Validation of Expected Dead Markings . . . 270

D CPN State Space to Minimised FSA **272**
D.1 Service Primitives to Integers . . . 272
D.2 State Space to FSM Input . . . 273
D.2.1 Mapping State Space to FSA . . . 273
D.2.2 Writing the State Space to FSM Input . . . 275
D.3 FSA Minimisation and Comparison . . . 276
D.4 FSA Graphics and Language Text File . . . 276

E IOTP CPN Declarations **277**

F IOTP State Space Analysis Results **279**
F.1 State Space Reports . . . 279
F.1.1 Authentication Transaction (*A-1-1*) . . . 279
F.1.2 Purchase Transaction (*P-1-1*) . . . 280
F.1.3 Deposit, Withdrawal or Refund Transaction (*D/W/R-1-1*) . . . 282
F.1.4 Value Exchange Transaction (*V-1-1*) . . . 286
F.2 Dead Markings . . . 289
F.3 Terminal SCCs . . . 289

G Verification Results of IOTP **293**
G.1 IOTP CPN Binding Element Map Specification . . . 293
G.2 Modified IOTS CPN for Verification of IOTP . . . 296
G.3 Minimal FSA for Service Languages . . . 296
G.3.1 Successful Transactions . . . 296
G.3.2 Transactions with Cancellation . . . 298
G.4 Minimal FSA for Protocol Languages . . . 299
G.4.1 Successful Transactions . . . 299
G.4.2 Transactions with Cancellation . . . 300
G.5 Language Comparison Results . . . 301
G.5.1 Successful Transactions . . . 301
G.5.2 Transactions with Cancellation . . . 302

H Revised IOTP CPN and its Verification Results **305**
H.1 Revised IOTP CPN Model . . . 305
H.1.1 Global Declarations . . . 305
H.1.2 The Revised CPN Pages . . . 306

H.2 State Space Analysis Results of the Revised IOTP 313
H.2.1 State Space Reports 313
H.2.2 Dead Markings 322
H.2.3 Dead Transitions 327
H.3 Verification Results of the Revised IOTP 327
H.3.1 Revised IOTP CPN Binding Element Map Specification 327
H.3.2 Minimal FSA for the Revised IOTP Languages 330

I Publications **331**

List of Figures

2.1 The hierarchical framework of e-commerce. 6

3.1 IOTP conceptual trading model. 19
3.2 A possible sequence of messages in a Purchase transaction. 20
3.3 The XML definition of (a) an IOTP message and (b) a Transaction Reference Block. 22
3.4 An example of the XML for an Offer Response Message. 23
3.5 The six document exchanges defined in IOTP. 25
3.6 Mapping document exchanges to trading exchanges. 27
3.7 Five typical scenarios of cancelling the Purchase transaction shown in Figure 3.2 at five stages: (a) during the Offer document exchange; (b) during the Payment document exchange; (c) during the Delivery document exchange; (d) between the Offer and the Payment document exchange; and (e) between the Payment and the Delivery document exchange. 30
3.8 A Purchase transaction involving messages reporting errors with different severities. . 31

4.1 Five valid transaction modes of Purchase with (a) payment before delivery; (b) payment after delivery; (c) payment during delivery; (d) payment in advance; (e) payment per unit of time. 37

5.1 The OSI Reference Model. 43
5.2 Structure of (N)-layer in a layered architecture. 44
5.3 (N)-service model in a layered architecture. 45
5.4 Time sequence diagrams for (a) a confirmed service and (b) a provider-initiated service. 46
5.5 (N)-protocol model in a layered architecture. 47
5.6 Protocol engineering process and activities. 48
5.7 Steps of protocol verification methodology. 49
5.8 Details of service validation step. 51

6.1 Hierarchy page for the CPN model. 54
6.2 Top level page: PayExchange. 55
6.3 Two trading role pages. 56
6.4 State space of the Payment exchange CPN model. 60
6.5 SCC graph of the state space shown in Figure 6.4. 62
6.6 Four possible sequences of message exchanges in an IOTP Payment exchange which is: (a) complete; (b) cancelled by the Payment Handler; (c) cancelled by the Consumer but completed at the Payment Handler side; and (d) cancelled by both Consumer and Payment Handler. 64

6.7 Graphical representation of two FSA. 65
6.8 Procedure of FSA reduction, determinisation and minimisation. 67
6.9 Format of a text file as the input to FSM. 67

7.1 The abstract model of IOTS. 68
7.2 Two valid primitive sequences in the Offer phase. 73
7.3 Two valid primitive sequences in the Payment phase. 75
7.4 Two valid primitive sequences in the Delivery phase. 76
7.5 Two valid primitive sequences in the Authentication phase. 78
7.6 Four valid primitive sequences of transaction cancellation in the Offer phase. 79
7.7 FSA defining the local service language for an Authentication transaction at (a) the Authenticator and (b) the Authenticatee trading role. 80
7.8 FSA for the local service language at the C-SAP for non-Authentication trading transactions without cancellation. 82
7.9 FSA for the local service language for non-Authentication trading transactions at the M-SAP. 84
7.10 FSA for the local service language at the P-SAP for non-Authentication trading transactions without cancellation. 84
7.11 FSA for the local service language for non-Authentication trading transactions at the D-SAP. 86
7.12 FSA for the local service language at the M-SAP for Authentication-combined trading transactions without cancellation. 88
7.13 FSA for the local service language at the P-SAP for Authentication-combined trading transactions without cancellation. 89
7.14 FSA for the local service language at the D-SAP for Authentication-combined trading transactions without cancellation. 90
7.15 FSA for the local service language at the C-SAP for Authentication-combined trading transactions without cancellation. 91

8.1 Hierarchy page for IOTS CPN. 95
8.2 IOTS_Global page in the IOTS CPN model. 96
8.3 Offer page in the IOTS CPN model. 100
8.4 Payment page in the IOTS CPN model. 104
8.5 Delivery page in the IOTS CPN model. 107
8.6 Authentication page in the IOTS CPN model. 110
8.7 AuthCP page in the IOTS CPN model. 110
8.8 AuthPC page in the IOTS CPN model. 113
8.9 AuthCM page in the IOTS CPN model. 115
8.10 AuthMC page in the IOTS CPN model. 115
8.11 AuthCD page in the IOTS CPN model. 116
8.12 AuthDC page in the IOTS CPN model. 116
8.13 Cancel page in the IOTS CPN model. 117
8.14 Two valid primitive sequences of cancellations at the beginning of an Offer phase with *TradingOptions.req* followed by (a) *Cancel.req* or (b) *Cancel.ind* at the Merchant entity only. 118

8.15 Three scenarios of cancellations between two trading phases, where (a) the Consumer initiates cancellation between Offer and Payment, (b) the Consumer initiates cancellation between Payment and Delivery, and (c) the IOTS Provider initiates cancellation between Offer and Payment. 119
8.16 Four scenarios of cancellations where (a) the Consumer submits a *Cancel.req* before a *PayInvoke.ind* and the corresponding *Cancel.ind* occurs at the Merchant only, (b) the Consumer submits a *Cancel.req* after a *PayInvoke.ind* and the *Cancel.ind* occurs at both the Merchant and the Payment Handler, (c) the Payment Handler submits a *Cancel.req* and the *Cancel.ind* occurs at the Consumer and then the Payment Handler, and (d) the IOTS Provider issues *Cancel.ind* to the Consumer, the Merchant and the Payment Handler, respectively. 120
8.17 CnlCM page in the IOTS CPN model. 121
8.18 CnlCP page in the IOTS CPN model. 121
8.19 CnlCD page in the IOTS CPN model. 121

9.1 The FSA specifying primitive sequences that are defined in the local P-SAP FSA but missing in the P-SAP projected-FSA. 128
9.2 A valid primitive sequence at the P-SAP not preserved in the P-SAP projected-FSA. 129
9.3 Corrected Delivery page in the IOTS CPN model. 130
9.4 P-SAP projected-FSA obtained from the revised IOTS CPN model. 131
9.5 The FSA for the global service language of Purchase without delivery and authentication. 132

10.1 Structure of IOTP constituted by four parts: (a) the part between Consumer_PE and Merchant_PE in Offer document exchanges; (b) the part between Consumer_PE and PHandler_PE in Payment and Delivery document exchanges; (c) the part between Consumer and DHandler_PE in Delivery document exchanges; and (d) the part between a Consumer_PE and a non-Consumer_PE in Authentication document exchanges. 135
10.2 Hierarchy page for IOTP CPN. 142
10.3 IOTP_TopLevel page in the IOTP CPN model. 143
10.4 Consumer page in the IOTP CPN model. 146
10.5 Merchant page in the IOTP CPN model. 149
10.6 PHandler page in the IOTP CPN model. 150
10.7 DHandler page in the IOTP CPN model. 150
10.8 AuthTr_C page. 151
10.9 AuthTr_M page. 151
10.10 Purchase_C page. 152
10.11 Purchase_P page. 152
10.12 Purchase_D page. 152
10.13 Purchase_M/Deposit_M/Withdrawal_M/Refund_M/ValExTr_M page. 152
10.14 Deposit_C/Withdrawal_C/Refund_C page. 153
10.15 Deposit_P/Withdrawal_P/Refund_P page. 153
10.16 ValExTr_C page. 153
10.17 ValExTr_P page. 153
10.18 Authenticatee page in the IOTP CPN model. 154
10.19 Authenticator page in the IOTP CPN model. 154

10.20 A completed Authentication document exchange. 156
10.21 Two sequences of message exchanges in (a) a Brand Dependent Offer and (b) a Brand Independent Offer document exchange. 159
10.22 Two pages modelling a Brand Dependent Offer document exchange. 161
10.23 Two pages modelling a Brand Independent Offer document exchange. 162
10.24 A possible sequence of message exchanges in a Payment document exchange. 163
10.25 Two pages modelling the Payment document exchange(s). 164
10.26 A sequence of message exchanges in a Payment-and-Delivery document exchange. . . 166
10.27 Two pages modelling the Payment-and-Delivery document exchange. 167
10.28 The message exchanges in a Delivery document exchange. 168
10.29 Two pages modelling the Delivery document exchange. 168
10.30 Cancellation_C page in the IOTP CPN model. 169
10.31 Cancellation_NC page in the IOTP CPN model. 170
10.32 Error Handling Procedure at a TRole_PE. 172
10.33 ErrHandling_C page in the IOTP CPN model. 173
10.34 ErrHandling_NC page in the IOTP CPN model. 174

11.1 The four dead markings of an Authentication transaction in *Config A-1-1*. 182
11.2 Path in the Authentication transaction (*Config A-1-1*) state space leading to M_{163}. . 182
11.3 Time sequence diagram of the path shown in Figure 11.2. 183
11.4 Path in the Purchase transaction (*Config P-1-1*) state space leading to M_7. 185
11.5 Time sequence diagram of the path shown in Figure 11.4. 185
11.6 Path in the Purchase transaction (*Config P-1-1*) state space leading to M_{472}. 186
11.7 Time sequence diagram of the path shown in Figure 11.6. 186
11.8 Two paths in the Purchase transaction (*Config P-1-1*) state space leading to M_{619}. . 189
11.9 Time sequence diagrams of the two paths shown in Figure 11.8: (a) the left path involving M_{110} followed by M_{193} and (b) the right path involving M_{110} followed by M_{194}. 190
11.10 Path in the Purchase transaction (*Config P-1-1*) state space leading to M_{748}. 191
11.11 Time sequence diagram of the path shown in Figure 11.10. 191

12.1 The FSA for the service language of a successful Authentication transaction. 194
12.2 Three FSA for a successful Deposit transaction specifying (a) the service language, (b) the protocol language and (c) the missing sequences in the protocol. 195
12.3 Two scenarios in the Offer phase using (a) a combined TPO & Offer Response Message and (b) two separate TPO and Offer Response Messages to implement the *TradingOptions* and *TradingResult* services. 196
12.4 Two FSA for a successful Purchase transaction specifying (a) the service language and (b) the protocol language. 198
12.5 Different scenarios in the combined Payment and Delivery phases using (a) a Payment Request & Delivery Request Message and a Payment Response & Delivery Response Message and (b) the separate Payment Request, Delivery Request, Payment Response and Delivery Response Messages. 199
12.6 Three FSA for a successful Value Exchange transaction specifying (a) the service language, (b) protocol language and (c) the missing sequences in the protocol. 200

12.7 Two FSA specifying (a) $MASS^1_{Ofr}$ and (b) $MASS^2_{Ofr}$. 201
12.8 Five FSA specifying (a) $MASS^1_{AuthOfr}$, (b) $MASS^2_{AuthOfr}$, (c) $MASS^3_{AuthOfr}$, (d) $MASS^4_{AuthOfr}$ and (e) $MASS^5_{AuthOfr}$. 202
12.9 Four FSA of (a) $MASS^1_{PDlv}$, (b) $MASS^2_{PDlv}$, (c) $MASS^3_{PDlv}$ and (d) $MASS^4_{PDlv}$. . 203
12.10 Two FSA specifying (a) $MASS_{Pay}$ and (b) $MASS_{Dlv}$. 203
12.11 The FSA specifying the missing sequences in the protocol for an Authentication transaction with cancellation. 206
12.12 Four time sequence diagrams of the Authentication transaction with cancellation showing four missing sequences in the protocol. 207
12.13 The FSA specifying the illegal sequences in the protocol for a Deposit transaction with cancellation. 209
12.14 The FSA for the missing sequences in the protocol for a Deposit transaction with cancellation excluding those defined by FSA 12.11. 210
12.15 Time sequence diagram showing the primitive sequence (*TOreq, TRreq*) that is implemented by IOTP but is not in IOTS. 211
12.16 Six time sequence diagrams of a Deposit transaction with cancellation showing six missing primitive sequences in the protocol. 213
12.17 The FSA specifying the subsequences of a standalone Offer phase and an Authentication phase followed by an Offer phase in any successful Payment-related transaction. 215
12.18 The FSA specifying the illegal subsequences for payment and delivery services in the protocol for a Purchase transaction with delivery and cancellation. 216
12.19 The FSA specifying the missing subsequences for payment and delivery services in the protocol for a Purchase transaction with delivery and cancellation. 216
12.20 The FSA specifying the illegal subsequences for the payment service in the protocol for a Value Exchange transaction service that is cancelled during the second Payment phase. 217
12.21 The FSA specifying the missing subsequences for the payment service in the protocol for a Value Exchange transaction service that is cancelled during the second Payment phase. 218

C.1 ML code for the InValidTerminal function. 271
C.2 Result of checking the unexpected dead markings. 271

F.4 Result of calculating terminal SCCs in (a) the Authentication, (b) the Purchase, (c) the Deposit and (d) the Value Exchange transactions state spaces in Configuration Group 1-1. 289
F.1 Dead markings in the state space of the Purchase transaction with *Config P-1-1.* . . 290
F.2 Dead markings in the state space of the Deposit transaction with *Config D-1-1.* . . . 291
F.3 Dead markings in the state space of the Value Exchange transaction with *Config V-1-1.* 292

G.1 Three modified IOTS CPN pages: (a) the Offer page, (b) the Payment page and (c) the AuthMC page. 297

H.1 BrdDepOffer_C page in the revised IOTP CPN. 307
H.2 BrdDepOffer_M page in the revised IOTP CPN. 307

H.3 BrdIndOffer_C page in the revised IOTP CPN. 307
H.4 BrdIndOffer_M page in the revised IOTP CPN. 308
H.5 Payment_C page in the revised IOTP CPN. 308
H.6 Payment_P page in the revised IOTP CPN. 309
H.7 PayDelivery_P page in the revised IOTP CPN. 309
H.8 Delivery_D page in the revised IOTP CPN. 309
H.9 Cancellation_C page in the revised IOTP CPN. 310
H.10 Cancellation_NC page in the revised IOTP CPN. 310
H.11 ErrHandling_C page in the revised IOTP CPN. 311
H.12 ErrHandling_NC page in the revised IOTP CPN. 312
H.13 Result of checking for undesired terminal states in the Authentication transaction state space with *Config A-1-1*. 323
H.14 Result of checking for undesired terminal states in the Purchase transaction state space with *Config P-1-1*. 324
H.15 Result of checking for undesired terminal states in the Deposit transaction state space with *Config D-1-1*. 325
H.16 Result of checking for undesired terminal states in the Value Exchange transaction state space with *Config V-1-1*. 327

List of Tables

3.1 The 21 trading components used to construct all trading blocks except a Transaction Reference Block in IOTP. 22

5.1 Four service primitive types. 45
5.2 Service parameter table with parameter usage notation. 45

6.1 Global declarations. 55

7.1 IOTS primitives and parameters. 70
7.2 IOTS primitives and parameters in Offer phase. 71
7.3 IOTS primitives and parameters in the Payment phase. 73
7.4 IOTS primitives and parameters in the Delivery phase. 75
7.5 IOTS primitives and parameters in the Authentication phase. 77
7.6 IOTS primitives and parameters in the Cancel phase. 78
7.7 Meaning of states defined in the FSA in Figure 7.7. 80
7.8 Meaning of states defined in the FSA in Figure 7.8. 82
7.9 Meaning of states defined in the FSA in Figure 7.9. 84
7.10 Meaning of states defined in the FSA in Figure 7.10. 85
7.11 Meaning of states defined in the FSA in Figure 7.11. 86

8.1 Colour sets and variables for modelling states of the interface at each trading role SAP. 97
8.2 Colour sets and variables for modelling significant parameters and SDUs. 98
8.3 Meaning of SDUs in the IOTS CPN model. 99
8.4 Meaning of primitive transitions on the Offer page. 101
8.5 Meaning of primitive transitions on the Payment page. 104
8.6 Mapping of payment-related states between the IOTS CPN model and the local C-SAP or P-SAP FSA. 104
8.7 Meaning of primitive transitions on the Delivery page. 108
8.8 Meaning of primitive transitions on the AuthCP page. 111
8.9 Meaning of Authentication-related primitive transitions on page AuthPC. 112
8.10 Meaning of *Cancel* primitive transitions on CnlCM, CnlCP and CnlCD pages. 120
8.11 Definition of the function IsInOfr. 122
8.12 Definitions of two functions IsInPay and IsInDlv. 123

9.1 State space statistics of the IOTS CPN model. 125
9.2 Forms of the desired dead markings for successful transactions in the IOTS state space. 126
9.3 Forms of the desired dead markings for cancelled transactions in the IOTS state space. 126

9.4 State space statistics of the revised IOTS CPN model. 130

10.1 Colour sets for modelling IOTP messages and trading role communication queue. . . 144
10.2 Colour sets for modelling states of TRole_PEs. 147
10.3 Function used to determine the initial state of a TRole_PE. 148
10.4 Function used to check a message reporting a hard error or a Cancel Message. 176

11.1 Expected dead transitions under certain parameter values in the IOTP CPN. 180
11.2 State space size and dead markings for Configuration Group 1-1. 181
11.3 Dead markings for Payment-related transactions in Configuration Group 1-1. 184
11.4 Statistics of SCC graphs for Configuration Group 1-1. 192
11.5 Dead transitions in the activated CPN page instances for Configuration Group 1-1. . 192

12.1 Statistics of the three FSA in Figure 12.2. 197
12.2 Statistics of FSA in Figure 12.4 and the FSA for the missing sequences in the protocol. 199
12.3 Statistics of the three FSA in Figure 12.6. 200
12.4 Statistics of three FSA for an Authentication transaction with cancellation. 205
12.5 Statistics of four FSA for a Deposit transaction with cancellation. 208
12.6 Statistics of four FSA for a Purchase transaction with cancellation. 214
12.7 Statistics of four FSA for a Value Exchange transaction with cancellation. 217

13.1 Definition of function Uni_Multicast in the revised IOTP CPN. 224
13.2 Definition of colour set DelivExch in the revised IOTP CPN. 225
13.3 Functions IsPayDlvRsp and NotLastRsp in the revised IOTP CPN. 225
13.4 Terminal states for revised IOTP updated from those in Group 5 in Table 11.3. . . . 226
13.5 Terminal states for revised IOTP updated from those in Group 6 in Table 11.3. . . . 227
13.6 Terminal states for revised IOTP updated from those in Group 7 in Table 11.3. . . . 227
13.7 Statistics for the state space size and dead markings for selected configurations of the revised IOTP CPN model. 229
13.8 Statistics for the SCC graph and livelocks for selected configurations of the revised IOTP. 230
13.9 Expected dead transitions under certain parameter values in the revised IOTP CPN. 230
13.10 Dead transitions for selected configurations of the revised IOTP. 231
13.11 Statistics of the FSA specifying the service languages used for the verification of IOTP. 232
13.12 Statistics on the size of the FSA for selected configurations of the revised IOTP. . . . 232

B.1 IOTS parameter groups. 262

D.1 Abbreviations and integers used for IOTS primitives to generate language. 272

G.1 Definition of function Is_trdresult(). 296

H.1 Dead transitions in the activated CPN page instances in the revised IOTP CPN model with configurations in Group 1-1. 327

Acronyms and Abbreviations

ASC	Accredited Standards Committee
ANSI	American National Standards Institute
API	Application Programming Interface
b-b	business-to-business
b-c	business-to-consumer
CA	Certificate Authority
CBL	Common Business Library
CEPS	Common Electronic Purse Specification
CSP	Communicating Sequential Processes
CORBA	Common Object Request Broker Architecture
CPN	Coloured Petri Nets
DCOM	Distributed Component Object Model
DFSA	Deterministic FSA
DTD	Data Type Definitions
EBES	European Board for EDI Electronic Commerce Standardization
ECDTF	Electronic Commerce Domain Task Force
ECML	Electronic Commerce Modelling Language
ebXML	electronic business XML
e-commerce	electronic commerce
ECTSG	Electronic Commerce Technical Steering Group
EDI	Electronic Data Interchange
EDIFACT	Electronic Data Interchange for Administration, Commerce and Transport
EDIINT	Electronic Data Interchange-Internet INTegration
EFT	Electronic Funds Transfer
EHPT	Ericsson Hewlett-Packard Telecommunications
EMV	Europay, Mastercard and Visa
FIFO	First-In First-Out
FSA	Finite State Automaton / Finite State Automata
HTTP	Hypertext Transfer Protocol
IATA	International Air Transport Association
IETF	Internet Engineering Task Force
IOTP	Internet Open Trading Protocol

IOTS	Internet Open Trading Service
IP	Internet Protocol
ISO	International Organization for Standardization
ITU	International Telecommunications Union
ITU-T	ITU Telecommunications Standardization Sector
JECF	Java Electronic Commerce Framework
JOTP	Open Trading Protocol Toolkit for Java
MDA	Model Driven Architecture
MIME	Multipurpose Internet Mail Extensions
ML	Meta Language
MSC	Message Sequence Chart
NDFSA	Non-Deterministic FSA
OBI	Open Buying on the Internet
OGSA	Open Grid Services Architecture
OMG	Open Management Group
OSI	Open System Interconnection
OTP	Open Trading Protocol
PDU	Protocol Data Unit
POS	Point-Of-Sale
QoS	Quality-of-Service
RFC	Request For Comments
SAP	Service Access Point
SCC	Strongly Connected Component
SDL	Specification and Description Language
SDU	Service Data Unit
SEMPER	Secure Electronic Market Place for EuRope
SET	Secure Electronic Transaction
SGML	Standard Generalised Markup Language
SMILEs	Standard SMart Card Integrated SettLEment system
SML	Standard ML
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
SOM-EC	Smart Organization Models for Electronic Commerce
SSL	Secure Socket Layer
SWIFT	Society for Worldwide Interbank Financial Telecommunications
TCP	Transmission Control Protocol
TDI	Trade Data Interchange
TLS	Transport Layer Security
TPO	Trading Protocol Options
TSD	Time Sequence Diagram
UCAF	Universal Cardholder Authentication Field

VAN	Value Added Network
VTP	Value Transfer Protocol
WAP	Wireless Application Protocol
WTP	Wireless Transaction Protocol
WWW	World Wide Web
W3C	World Wide Web Consortium
XBRL	eXtensible Business Reporting Language
XML	eXtensible Markup Language

Summary

The Internet Open Trading Protocol (IOTP) is being developed by the Internet Engineering Task Force (IETF) to support electronic commerce (e-commerce) over the Internet. It aims to encapsulate different payment systems such as the Secure Electronic Transaction (SET) protocol for credit card payments and Mondex for smart card-based electronic cash payments. The protocol provides reliable common transaction services to four trading roles: Consumer, Merchant, Payment Handler and Delivery Handler. IOTP defines transactions for authentication, deposit, withdrawal, purchase, refund and value exchange. The specification of IOTP, published as Request For Comments (RFC) 2801, is the largest RFC developed by IETF so far, spanning 290 pages. The RFC provides an informal narrative description of IOTP and is still at an early stage of development as, so far, no complete implementation of IOTP exists.

An important issue in developing a protocol is the verification of its functional correctness. For protocols involving financial transactions, such as IOTP, it is vitally important that they work correctly. This thesis applies a protocol engineering methodology using a formal technique called Coloured Petri Nets (CPNs) to develop models of IOTP at different levels of abstraction. These models constitute formal specifications of IOTP and the service provided to the users of IOTP (i.e., the trading roles), which we refer to as the protocol and service specifications of IOTP, respectively. The verification of IOTP is concerned with proving that the protocol specification provides the requirements stated in its service specification.

RFC 2801 does not define the service provided by IOTP to its users. The service specification of IOTP is therefore derived from the RFC and formalised using CPNs. State space generation and automata reduction are used to produce the service language, which defines the set of possible global sequences of service primitives. Language comparison is also conducted to check consistency between the global sequences of primitives and the local sequences that are pre-defined in the service specification.

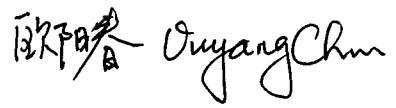
In the RFC, the dynamic behaviour of IOTP is described in a narrative way and without any use of formal techniques. An IOTP CPN model is created as a formal specification of the protocol. Without loss of generality, the transactions are modelled as independent of each other. The underlying service (provided by the Application Layer and the layers below in the Internet protocol stack) is error free (no corruption, no loss and no duplication), but allows for message transmission delay.

State space analysis is used to investigate a set of desired properties of IOTP (e.g., correct termination). Language analysis is used to check if IOTP is a faithful refinement of its required service, in terms of sequences of primitives. The analysis reveals errors in the current design of IOTP. A set of changes are proposed, leading to a revised IOTP. Further analysis proves that the set of desired properties hold for the revised IOTP and the protocol presents a valid implementation of its service.

Declaration

I declare that this thesis does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge it does not contain any material previously published or written by another person except where due reference is made in the text.

Chun Ouyang



Acknowledgements

With a deep sense of gratitude, the first person I would like to thank is my principal supervisor, Professor Jonathan Billington. His endless enthusiasm and tireless devotion to research inspired my own efforts. His invaluable guidance, substantial feedback, stimulating suggestions and encouragement helped me all the time of research for and writing of this thesis. He could not even realize how much I have learned from him. Also, I am very grateful to Professor Billington for his support for all other matters related to my PhD study, such as funding and employment opportunities.

I would like to thank my associate supervisor, Dr. Lars Michael Kristensen, for our many discussions, providing me advice and tips that helped me stay on the right track, cooperation in a number of publications, and the assistance with modelling skills and the computer tool, in the second and third years of my PhD study. Also, thanks go to Lars for providing valuable and detailed comments on a draft of this thesis.

The Computer Systems Engineering Centre (CSEC) contributed to the development of this research. I would like to thank all members of CSEC for attending my group seminars and providing helpful comments. In particular, I would like to thank several people: Steven Gordon and Maria Villapol, two former members, for their help with background knowledge and the computer tool when I started my research; Lin Liu with whom I had discussions about concepts and methodologies in the early stage of this research; and Guy Gallasch for discussions on applying other analysis techniques in the further work of this research. I am also thankful to the Telecommunication Networks and Services (TNS) Group of the Institute for Telecommunication Research (ITR) in advancing my general knowledge and communication skills during the first year of my PhD study.

The work for this thesis could not have been realized without the vital financial support from the following sources: an International Postgraduate Research Scholarship; an University of South Australia Postgraduate Research Award; a combined University of South Australia and Division of Information Technology, Engineering and the Environment Research Grant; and the School of Electrical and Information Engineering and CSEC for funding to attend conferences and general support.

I am very grateful to my husband Zhikai, for his loving support, patience and understanding that enabled me to complete this work. I also want to thank my parents, who taught me the value of hard work by their own example.

Finally, I would like to thank all whose direct and indirect support helped me completing my PhD study successfully.

Chapter 1

Introduction

1.1 Background

The explosive growth of the Internet [54] and its applications strongly influence the transition from traditional commerce to *electronic commerce* (e-commerce) [249]. Increasingly people are making use of e-commerce for buying goods and services (such as books and airline tickets) via the World Wide Web (WWW) [18]. E-commerce is made possible by the presence of *communication protocols* that make up the worldwide communication infrastructure and enable geographically distributed users to communicate via the Internet. The increased popularity of e-commerce means that new communication protocols are currently in the process of being developed by a number of organisations.

An important example of a communication protocol related to e-commerce is the Internet Open Trading Protocol (IOTP) [36]. It is currently being developed by the Internet Engineering Task Force (IETF) [108], the international forum responsible for the standardization of the information infrastructure that comprises the Internet. The core of IOTP is a set of business transactions that reflect the common trading activities in the real world. IOTP aims to specify components and rules necessary for trading parties (which have no prior association) to complete an electronic business transaction. It is also referred to as a *shopping protocol* [27, 139] that focuses on payment aspects of e-commerce. IOTP is expected to evolve into one of the central building blocks of next generation e-commerce on the Internet.

A key issue in developing a protocol is the verification of its functional correctness [21]. For e-commerce protocols involving financial transactions, such as IOTP, it is vitally important to ensure that they work correctly and do not contain errors (e.g., deadlocks), which may cause disruption of the service or worse situations such as incorrect financial transactions. Formal methods [51], which encompass mathematically based modelling languages, techniques and computer tools for specification, verification, implementation, and testing of systems, are necessary for the construction of unambiguous and precise models which can be analysed by computer tools to identify errors and verify correctness before implementation. Application of formal methods will lead to more reliable and trustworthy e-commerce protocols [30, 185].

1.2 Problem Statement

The specification of IOTP, published as *Request For Comments* (RFC) 2801 [36], is the largest RFC developed by IETF so far, spanning 290 pages. The RFC only provides a narrative description of IOTP, and some parts of the RFC are ambiguous, difficult to understand and

imprecise. The literature review on IOTP (see Chapter 2), including IOTP design and standard documents, shows that few formal studies have been carried out on this protocol. Given the complexity of IOTP, implementations based on such an informal protocol specification are likely to contain errors. As a consequence, the cost for fixing errors in the protocol during and/or after implementation can be high. Hence, one of the open research issues concerning IOTP is the verification of its functional correctness.

1.3 Research Aims and Scope

The primary aim of this thesis is to analyse and verify IOTP. One approach to ensuring the correctness of the IOTP design is to create formal models of IOTP at different levels of abstraction. These models constitute formal specifications of IOTP and the service provided to the users of IOTP, which we refer to as the protocol and service specifications of IOTP, respectively. The verification of IOTP is concerned with proving that the protocol specification provides the requirements stated in its service specification. Coloured Petri Nets (CPNs) [123] are a suitable modelling language for this verification task, as they can conveniently express non-determinism, concurrency and different levels of abstraction that are inherent in protocols. CPNs are also supported by a mature computer tool called Design/CPN [176]. In order to achieve the above primary aim, four objectives must be reached in this thesis.

1. In RFC 2801 [36], the informal nature of the specification and its complexity make it difficult to maintain consistency and clarity. Defining a simpler conceptual framework for IOTP is necessary for clarifying the internal mechanism of protocol operation. Apart from this, IOTP, like other e-commerce protocols, does not work in isolation. To provide the required service, IOTP needs to interact with other protocols. Such interaction is usually specified as the protocol architecture and is not defined in the RFC. Hence, the first step towards modelling IOTP is to obtain a simpler but much clearer conceptual framework as well as the definition of a protocol architecture for IOTP.

2. The verification of IOTP involves comparing the protocol to the service provided by IOTP. RFC 2801 [36] does not include a service specification. The second objective, therefore, is to develop a service specification for IOTP. This consists of two tasks: 1) taking the protocol specification given in the RFC and identifying the service required; and 2) creating the formal model of the service using CPNs.

3. A formal model of IOTP will provide an unambiguous protocol specification and is needed for protocol verification. Thus, the third objective is to develop a CPN model of IOTP that is consistent with its definition given in RFC 2801 [36]. To achieve the goal of verifying IOTP, the model must be optimized for verification.

4. Once the IOTP CPN has been developed and debugged, the model needs to be verified. The verification task is divided into verification of desired properties (e.g., the IOTP transaction termination property) and verification against the service specification. The fourth objective is then to show that IOTP satisfies a set of desired properties (e.g., the desired termination of transactions), and that IOTP presents a faithful refinement of the service defined in the service specification.

The core of IOTP is a set of business transactions which support common trading activities over the Internet. This thesis focuses on the verification of logical correctness of this set of transactions defined in the current version of IOTP (version 1.0) [36].

1.4 Structure of the Thesis

To meet our research objectives, the remainder of this thesis is structured as follows:

Chapter 2 (Electronic Commerce) gives a state-of-art introduction to and provides a literature review for e-commerce. This mainly includes a general hierarchical framework of e-commerce, the existing and emerging standards and specifications concerning the information infrastructure of e-commerce, and related work on applying formal methods to analyse and verify e-commerce protocols.

Chapter 3 (Internet Open Trading Protocol) summarizes the specification of IOTP given in RFC 2801 [36]. A detailed description of IOTP is given, focusing on its basic concepts as well as main protocol procedures.

Chapter 4 (Initial Assessment of the Internet Open Trading Protocol) assesses IOTP and includes a proposal for a simpler conceptual framework and structure for IOTP.

Chapter 5 (Protocol Engineering) describes a general protocol engineering methodology [21] and outlines the approach we have used in verifying IOTP.

Chapter 6 (Formal Techniques and Tools) introduces Coloured Petri Nets (CPNs), the formal technique used to model IOTP, the state space method used for analysing the IOTP model, and finite state automata (FSA), the formal technique used for language analysis. The supporting computer tools are also introduced.

Chapter 7 (Internet Open Trading Service Definition) defines the service provided by IOTP to its users. At first, a complete set of service primitives and their parameters are defined. Then, finite state automata are used to define the local service language, specifying all valid primitive sequences that can occur at each local user interface. These form a basis for developing a formal service specification of IOTP in the next chapter.

Chapter 8 (Internet Open Trading Service CPN) presents a CPN model of IOTS, which is developed to synthesize the previous local automata into a specification of the global service capturing the sequences of primitive occurrences at different users. Chapters 7 and 8 provide further development of the service specification for IOTP published in [182].

Chapter 9 (Validation of the Internet Open Trading Service) describes the validation of the IOTS CPN model by ensuring consistency between the global service language generated from the CPN model and the local service languages defined in IOTS. Language comparison is used to check if the global service language matches the local service language for each IOTS user. Any detected inconsistencies are removed by modifying the IOTS CPN model. The above approach to the validation of IOTS was firstly published in [182].

Chapter 10 (Internet Open Trading Protocol CPN) gives a detailed description of the IOTP CPN model. This includes discussion of the modelling scope and assumptions that are made to tailor the CPN model for verification while still correctly representing IOTP as defined in [36]. Earlier models of IOTP were published in [179–181].

Chapter 11 (Analysis of Internet Open Trading Protocol Properties) consists of investigating and formally reasoning about a set of desired properties of IOTP using the state space method. This reveals errors in the design of IOTP given in [36]. Earlier work on the analysis of the transaction termination property was presented in [181].

Chapter 12 (Verification Against the Service Specification) consists of comparing the IOTP specification against the IOTS specification to check for language equivalence. This is firstly applied to the IOTP transactions that are successful and error-free. It is then applied to the transactions with error-handling and transaction cancellations, which also reveals errors in the design of IOTP in [36]. The initial verification of successful and error-free transactions was published in [177].

Chapter 13 (A Revised Internet Open Trading Protocol and its Verification) presents solutions to the errors revealed from the analysis of IOTP in Chapters 11 and 12. Changes reflecting the suggested solutions are made to IOTP, leading to a revised protocol specification. State space and language analysis of the revised IOTP CPN shows that the protocol does not only have a set of desired properties, but also provides a faithful refinement of the IOTS.

Chapter 14 (Conclusions) concludes the thesis by presenting a summary of the contributions of the research as well as suggesting areas for future work.

Chapter 2

Electronic Commerce

Electronic commerce (e-commerce) [7], traditionally conducted over proprietary Value Added Networks (VANs) [135], is now used widely on the Internet. With the growth of the Internet, e-commerce may lead to the creation of entirely new businesses, reduce cost and improve services for consumers. A global marketplace is predicated on an open or interoperable e-commerce infrastructure, where a multitude of technology standards and specifications have been developed offering approaches for e-commerce solutions. Since failures in e-commerce applications can lead to financial loss, it is important that the underlying e-commerce protocols are properly designed and analysed.

This chapter presents a state-of-art literature review of e-commerce, with a focus on its underlying standards and specifications. Section 2.1 introduces a hierarchical framework for e-commerce and briefly describes the e-commerce infrastructure and applications within this framework. Section 2.2 provides an overview of the technology standards and specifications developed to establish the required information infrastructure for e-commerce, including the Internet Open Trading Protocol which we investigate in this thesis. Section 6.2 concludes this chapter with a survey of the research work into the formal analysis of e-commerce protocols.

2.1 Hierarchical Framework

E-commerce can be defined as the entire collection of actions that support commercial activities via electronic networks [7]. It is the combination of networked applications, commerce and security functions to facilitate business transactions as well as information to support strategic business models [46]. Zwass [255, 256] introduces a hierarchical framework for e-commerce that extends from the required networking infrastructure to global electronic marketplaces. Figure 2.1 depicts this framework which includes four levels with each level delivering well-defined functional support to upper levels. Below we discuss each of the levels in more detail.

2.1.1 Technological Infrastructure

The Internet is the basic infrastructure that enables today's e-commerce applications [5]. Based on its Transmission Control Protocol / Internet Protocol (TCP/IP) reference model [53, 219], several leading technologies enhance the performance, reliability, interoperability and security of the Internet to accommodate mission-critical business requirements. Examples include Quality-of-Service (QoS) management [74], IP multicast [87], delivery receipts, authenticated packets, smart firewalls and middleware technologies [149] such as the Common Object Request Broker Architecture (CORBA) [200] and Distributed Component Object Model (DCOM) [160].

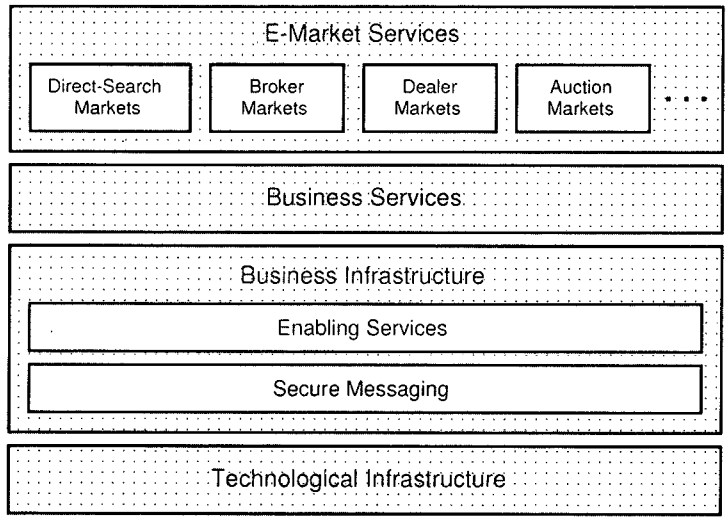


Figure 2.1: The hierarchical framework of e-commerce.

2.1.2 Business Infrastructure

This level consists of *secure messaging* and *enabling services* for e-commerce.

Secure Messaging

In contrast to traditional e-commerce which relies on secured private VANs, Internet-based e-commerce offers public communication but has fundamental security flaws in the Internet infrastructure, such as eavesdropping, absence of authentication and altering contents of packets. Therefore, provision of secure messaging is necessary for safely conducting business transactions over the Internet. It has to feature the following attributes: confidentiality, which can be accomplished through encryption; message integrity, which can be achieved with hash values accompanying the message; authentication of two communicating traders via a digital signature or possession of a private key; and non-repudiation by either trader using a combination of the methods mentioned above. Due to the limitations of the TCP/IP protocol suite, a number of e-commerce protocols are developed to secure business communications over the Internet. An example is the Secure Electronic Transaction (SET) protocol [237] for secure credit card transactions over the Internet [73]. A further description of SET is given in Section 2.2.2. Other principal messaging services range from traditional Electronic Data Interchange (EDI) [141] and Electronic Funds Transfer (EFT) [65] to today's Open-EDI [81], Extensible Markup Language (XML) Messaging [33] and XML/EDI [246]. Section 2.2.3 provides an overview of EDI and XML related standards.

Enabling Services

These are basic commerce-oriented service facilities that allow individuals and companies to authenticate their identities, make payments, locate vendors, collaborate and participate in business activities. Examples include various payment mechanisms (see Section 2.2.2) such as electronic money (e-money) [226] and smart-card systems [72]; agent-based services [61,97,186, 217,241] such as intelligent agents helping to search for desired goods or services; electronic authentication services; application billing and accounting; and electronic delivery services such as digital-content delivery.

2.1.3 Business Services

Generic business processes and applications common to multiple electronic markets (e-markets) are present in this level. It covers three categories of e-commerce applications, known as business-to-consumer (b-c) applications, business-to-business (b-b) applications, and intra-organizational business applications.

B-c e-commerce applications are consumer-oriented. Two typical examples are: remote customer services, such as online retailing, home banking and virtual stock brokerage, which are often accompanied by Web advertising; and the infotainment on-demand services such as remote educational offerings and fee-based contents viewing. Moreover, the services can be expanded in many ways, such as to electronic benefit systems.

B-b e-commerce applications are also referred to as supply-chain transactions [14]. It involves electronic procurement, order entry, inventory and supplier-consumer linkage management, and logistics [223]. These b-b activities are carried out between trading partners with well-established relationships.

Intra-organizational business applications include both intranet- and extranet-based information sharing and collaboration services within an organisation or between organisations [95]. The emerging technologies to support these applications are known as inter-organisational workflow management [92, 145, 205, 230] and business processing management [45, 229, 231].

2.1.4 E-Market Services

On this level, a number of markets with closely aligned businesses, collectively called virtual markets [242], are created to facilitate business transactions over networks between multiple buyers and multiple suppliers. The long-lasting supplier-customer relationships between them are maintained with networks and coordinated by management. Zwass [255, 256] divides them into four categories, which are direct-search markets, broker markets, dealer markets and auction markets [147]. Other examples of e-market based services are real estate, cooperative manufacturing, or any vertical supply chain.

2.2 Standards and Specifications

Based on the TCP/IP protocol stack that governs the Internet, e-commerce employs a range of its own technology standards and specifications to build up the required technological and business infrastructure. To support business processes and sub-processes and to meet specific requirements, these standards and specifications with different playing roles can be divided into three categories: *system architecture*; *payment and trading mechanisms*; and *EDI*. The following subsections provide an overview of the main contents in each of the three categories. A summary of these and the relevant standards and specifications can be found in [40].

2.2.1 System Architecture

The standards and specifications covered in this category provide information on architectural approaches which aim to address the requirements of e-commerce in a systematic way. As e-commerce is rapidly moving onto the Internet, a myriad of Internet commerce applications

has emerged and the interoperability therefore becomes a key issue in developing robust Internet commerce. To achieve interoperability, many technical standards and specifications have been developed to establish the frameworks for e-commerce. Well-known examples include (in alphabetical order): *BizTalk Framework* [161], *Building Blocks for E-Commerce* (or *Building Blocks* for short) [78], *eCo Architecture* [55], *Java Electronic Commerce Framework* (JECF) [52], *Open Buying on the Internet* (OBI) [173] and *Secure Electronic Marketplace for Europe* (SEMPER) [209]. Below, we briefly introduce these standards and specifications. A comparison between them is beyond the scope of this thesis and can be found in [212].

- The BizTalk Framework [161], developed by Microsoft since 1999, defines a conceptual architecture and exchange mechanism for general b-b electronic exchanges. The documentation [161] provides a detailed specification for the construction of documents and messages, and their secure transport over a number of Internet transport and transfer protocols. The main protocols and schemes supported are the *HyperText Transfer Protocol* (HTTP) [79], XML [33], *Multipurpose Internet Mail Extensions* (MIME) [63] and *Simple·Object Access Protocol* (SOAP) [208]. The framework has now been implemented into Microsoft's suite of products under the brand name $BizTalk^{TM}$.

- Building Blocks [78], initiated by the European Board for EDI Electronic Commerce Standardization / Electronic Commerce Technical Steering Group (EBES/ECTSG) in 1997, has been undertaken by the CEN/ISSS Electronic Commerce Workshop [250] since 2000. The concept of building blocks is derived from the sub-processes that comprise the commercial activities of e-commerce, and is defined as e-commerce specific functions that can be implemented as a discrete e-commerce product or service. Three categories of building blocks are identified in relation to where the service or product belongs, i.e., *buyer*, *seller* or *third party*. The set of main building blocks are defined [78] for the marketing process, the contract process, the logistics process, the settlement process, and for the interface with the administration process.

- The eCo Architecture [55], proposed by the CommerceNet Consortium in 1996, began as an architectural vision for open Internet commerce. Conceived originally as a *Common Object Request Broker Architecture* (CORBA)-based interoperability framework, the eCo system architecture was recast in 1997 on an XML foundation. Today's eCo System aims to transform the WWW into an agent-based infrastructure for Internet commerce and enable companies to communicate over the Internet using self-defining XML business documents [86]. It proposes "a framework of frameworks" [222] that models key business processes and services for building Internet Markets and provides a set of specifications to support interoperability among electronic business implementations.

- JECF [52], initiated by Sun Microsystems in 1996, aims to provide an open platform which can support all standards and payment protocols running concurrently in the same environment. The JECF currently comprises the related technologies of *Java Wallet* and *Commerce JavaBeans*. The Java Wallet [162] is intended to be a platform-independent client-side architecture which integrates "plug-in" commerce components for secure online transactions, and also to be extensible to support different operations using a variety of value transfer instruments and protocols. The Commerce JavaBeans [94] are modular

components of Java code that are re-usable for implementing specific online transaction protocols, e.g., credit card payments and electronic cheques. Both Java Wallet and Commerce JavaBeans are now under the umbrella heading of Java Commerce [224].

- OBI [173], originally developed by the OBI Consortium, has been managed by CommerceNet since 1998. Its objective is to provide a standard framework for secure and interoperable b-b Internet commerce with an initial focus on automating high-volume, low-dollar transactions between trading partners. The OBI architecture comprises four entities, namely *Requisitioner*, *Buying Organisation*, *Selling Organisation* and *Payment Authority*. OBI uses the EDI standard [141] for data formats for order-related information that is exchanged between trading partners. The transport is based on HTTP and the security service is provided by the *Secure Sockets Layer* (SSL) [68].

- SEMPER [209], a project led by IBM since 1996 and concluded in 1999, was intended to provide the first open and comprehensive solutions for secure commerce over the Internet and other public information networks. It defines an open and system independent architecture for e-commerce. This SEMPER architecture specifies four layers [143]: the *business applications layer* acting as the environment where the SEMPER library is used; the *commerce layer* providing services that directly implement protocols of business sessions; the *transfer & fair exchange layer* offering transfer of goods as well as fair exchange services between two parties with given security attributes; and the *supporting services* providing support to the above three layers. Apart from the four layers, the SEMPER architecture also includes standard buyer/seller scenarios.

2.2.2 Payment and Trading Mechanisms

Most of the standards and specifications for e-commerce architecture apply to b-b e-commerce. Nevertheless, the most popular applications of Internet commerce are consumer-oriented and comprise b-c applications, such as online shopping and Internet banking. Since payment plays a key role in e-commerce, various electronic payment systems have emerged [10], dominating b-c e-commerce. However, the b-c trading service like buying online is more than just paying [27], where ordering and delivery are other essential components of a trading transaction. A universal trading protocol is then required to support the entire process of b-c transactions over the Internet. In the following, we consider a set of main standards and specifications for payment and trading mechanisms, and provide a brief description of each of them. There are several surveys [6, 10, 148, 215, 233] on the various electronic payment systems.

Secure Electronic Transaction Protocol

SET [237] is the most well-known protocol designed to *secure* credit card transactions over open networks such as the Internet. It is developed jointly by MasterCard and the VISA International Association (incorporated as SETCo [210]) since 1997. An introduction to SET is given in [107, 195], and its application in a wireless computing environment can be found in [82, 151]. Section 6.2 includes a survey of research into the analysis of SET.

The SET protocol defines the following payment system participants in an e-commerce environment.

- A *Cardholder* uses a payment card that has been issued by an *Issuer*.
- The Issuer is a financial institution (e.g., a bank) that establishes an account for the Cardholder and issues the payment card.
- A *Merchant* offers goods or services in exchange for payment. The Merchant that accepts payment cards from the Cardholder must have a relationship with an *Acquirer*.
- The Acquirer is the financial institution that establishes an account with the Merchant and processes payment card authorisations and payments.
- Finally, a trusted third party called the *Certificate Authority* (CA) is responsible for issuing certificates to any of the above participants.

In a SET-based transaction, the electronic processing of the transaction begins with the Cardholder. Cardholders can visit Web pages, selecting items to be included in an order. When the Cardholder finishes shopping, the Merchant's Web server sends a completed order form for the Cardholder to review and approve. Once the Cardholder approves the order and chooses to use a credit card for payment, the SET protocol is initiated and provides the mechanisms for the Cardholder to securely transmit payment instructions as well as for the Merchant to obtain authorisation and receive payment for the order.

In SET, message data is initially encrypted with the key generated on the Cardholder's side. This key is then sent with the message encrypted with the recipient's public key. SET uses a distinct public/private key pair to create a *digital signature* for exchanged information. For example, the signature can be used to link an order message sent to the Merchant with the payment instructions containing the Cardholder's account information sent to the Acquirer. Authentication is further strengthened by the use of certificates issued by a CA. Cardholders must register with the CA before they can send SET messages to Merchants. More details can be found in the SET specifications [237].

Due to its complicated security mechanisms, acceptance of SET has been low. To boost acceptance, VISA has introduced *3-D(omain) Secure* (also called *3-D SET*) [207] and MasterCard has introduced *Universal Cardholder Authentication Field* (UCAF) [227]. These are independent verification methods supporting username/password that rely on the use of *Transport Layer Security* (TLS, previously *Secure Socket Layer*, SSL) [68]. VISA has implemented 3-D SET in Europe and Latin America, and the 3-D SET implementation and pilots are also underway in the US, EU, Canada, and Asia Pacific [207].

Smart Card Payment

Smart cards are credit-card-size plastic cards embedded with microchips that can store and process data. They are therefore designed to provide and record information about the card user, e.g., the user's account. Smart card technology has been in existence for many years, and a number of procedures have been developed for smart-card-based electronic cash payments. Three typical examples are: *Mondex* [166], *EMV Integrated Circuit Card Specifications for Payments* [80] and *Common Electronic Purse Specifications* (CEPS) [42]. Literature on smart card-based payment systems can be found in [29, 159, 188, 211].

- Mondex [166], originally developed by a consortium led by National Westminster Bank, Midland Bank and British Telecom since 1997, is now owned by MasterCard. In a payment transaction using Mondex, the electronic value flows directly from one Mondex chip to another Mondex chip. No third-party intermediary clears or processes the transaction, and only the holders of the two Mondex chips are involved. For example, when consumers at home or at the ATM withdraw electronic cash from their bank, only two chips are involved, i.e., the bank's Mondex chip and the chip on the consumer's card. Likewise, consumers can purchase items over the Internet from their home PC by inserting the Mondex card into a smart card reader and sending payment directly to the merchant's Mondex chip. Every Mondex transaction uses the *Mondex Value Transfer Protocol* (VTP) [239]. The VTP controls message transmission between two Mondex cards and enables a secure and legitimate transfer from one card to another by protecting the Mondex value using encryption. Due to its simplicity, Mondex has become the most widely used smart card system in the US [166].

- EMV Integrated Circuit Card Specifications for Payments [80] was developed and is maintained by Europay, Mastercard and VISA (EMV) since 1999. It defines the terminal and integrated circuit card procedures necessary to effect a payment system transaction in an international interchange environment. In particular, it covers electronic payment using cards with embedded integrated circuits, and the integrated circuit card application for payment systems. Source documents for EMV Specifications can be obtained from [183].

- CEPS [42] was initiated by CEPSCO[1] in 1999. The objective of CEPS is to define the requirements for all components needed by an organisation to implement a globally interoperable electronic purse program, while maintaining full accountability and auditability. Specifically, it covers security, card/terminal applications, load/unload transactions, currency exchange transactions, point-of-sale (POS) transactions, settlement and reconciliation, and certificate procedures. CEPS requires compatibility with EMV. The documentation [42] outlines overall system requirements, certification and migration.

Micropayment

Micropayment is the term [58] used to describe payments that are very small. There is no precise definition of the amount, but it is typically below 10 cents, often between 0.1 and 1 cent or even smaller. Until recently, there was no mechanism to make payments smaller than 1 cent. The launch of the Internet and WWW enables an environment and infrastructure of micropayment so that any party has the ability to sell anything of *any value* with no physical limitations. In particular, micropayment supports a number of newly emerging low-value services, such as a focused news service, small online utilities and programs, one day passes for monthly subscription sites, and pay per play games. Three typical examples of the standards and specifications for micropayment transactions are: *Millicent* [83], *Minipay* (or Mpay) [102] and *Jalda* [93]. Literature on micropayment systems can be found in [64, 70, 84, 206].

[1]CEPSCO [214] is a limited liability company owned by CEPSCO Espanola A.I.E., EURO Kartensysteme, Europay International/VISA International, Groupement des Cartes Bancaires and Proton World.

- Millicent [83] is a lightweight and secure protocol for small scale commercial transactions over the Internet. It is designed to support purchases costing less than a cent. It is based on decentralized validation of electronic cash at the vendor's server without any additional communication, expensive encryption, or off-line processing. The key innovations of Millicent are its use of *brokers* and of *scrip*. Brokers take care of account management, billing, connection maintenance and establishing accounts with vendors. Scrip is digital cash that is only valid for a specific vendor. The vendor locally validates the scrip to prevent customer fraud, such as double spending. Millicent was originally developed by Digital in 1995 [83], and then taken over by Compaq.

- MiniPay [102], developed by IBM Haifa Research Lab in 1997, is another micropayment system. It features low cost, negligible delay, a natural user interface, scalable design, support for multiple currencies and high security including non-repudiation, overspending prevention and protection against denial of service. A typical MiniPay system would consist of five parties: a *buyer*; a *seller*; the *buyer's billing system*; the *seller's billing system*; and an *exchange* connecting the buyer's billing system to the seller's billing system. There are also a set of protocols that are part of MiniPay, known as registration and routing protocols, daily buyer protocol, purchase protocol, extra-spending protocol, and deposit and clearing protocol.

- Jalda [93] provides a non-proprietary secure Internet payment method marketed as a mobile payment service, but also applicable to other technologies. It is designed as an account-based system, which associates every user with a specific account and enables customers to be charged on the basis of a wide range of parameters the e-commerce content providers desire, especially for small scale transactions. For example, charges can accrue through elapsed time, mouse click, searches, levels or other specified parameters and in any amount. Jalda was originally developed by Ericsson Hewlett-Packard Telecommunications (EHPT), and has been subsumed under Ericsson within the "Ericsson Mobility World" (Mobile Commerce Platform 1.0) [251] since August 2001.

Internet Open Trading Protocol

IOTP [36, 37], also referred to as a *shopping protocol* [27, 139], specifies the communication components and rules necessary for two parties, which have no prior association, to complete an electronic business transaction. It defines the various roles of the parties that may be involved in trading over the Internet, and the different types of transactions that can occur between these parties. IOTP uses XML as the document format for the content of messages and sets out the sequence of messages that are exchanged between trading parties. A main design goal of IOTP is the encapsulation of payment systems such as SET and Mondex. Being payment system-independent, IOTP aims to provide an interoperable framework for b-c Internet commerce.

IOTP was originated as the Open Trading Protocol (OTP) in January 1998, by a number of organisations including Hewlett Packard, IBM, MasterCard International, SmartCard Integrations, Sun Microsystems, and others, working cooperatively as the OTP Consortium. The original specification, namely *OTP version 0.9.9* [35], consisted of two parts: the Business Description; and the Technical Specification. In June 1998, the IETF Trade Working Group and

Network Working Group took over the development of OTP and changed the name to IOTP. The IOTP specification, namely *IOTP version 1.0*, evolved from Internet Draft to Request For Comments (RFC) 2801 [36] in April 2000. Together with IOTP version 1.0, a number of IOTP-related specifications have been developed, comprising a protocol suite for IOTP. These are:

- *IOTP Version 1.0* (RFC 2801) [36] provides a main specification of IOTP version 1.0 and can be used to design and implement systems which support electronic trading on the Internet.
- *IOTP HTTP Supplement* (RFC 2935) [77] describes the mapping to HTTP. The goal is to ensure that the IOTP Messages, which are XML documents, are carried successfully between the various trading parties.
- *Digital Signatures for IOTP* (RFC 2802) [137] describes the syntax and procedures for the computation and verification of digital signatures within IOTP.
- *Payment API for IOTP* [99] focuses on the Payment *Application Programming Interface* (API) which is essential for payment transactions in IOTP.
- *IOTP SET Supplement* (RFC 3538) [136] describes detailed Input/Output parameters for the IOTP Payment API and a procedure for using SET as the payment protocol in IOTP.
- *IOTP Version 2.0 Requirements* (RFC 3354) [75] describes design principles and scope, and divides features into those that will or will not be included in IOTP version 2.0.
- *IOTP Version 1.0 Errata* (RFC 3504) [76] lists the errors that have been noted in IOTP version 1.0 and provides corrections for them.

Concurrently with the specification work being conducted within IETF, a number of research groups and companies are planning and working on the first trial implementations of selected parts of IOTP. One of these is the production IOTP implementation of Withdrawal and Deposit of Mondex value by Royal Bank of Canada [174]. The developers have built an IOTP server and a Windows client which is in use by customers of the bank. Another is the Standard SMart Card Integrated SettLEment system (SMILEs) project [194] developed jointly by Hitachi [111] and Fujitsu [85]. It incorporates IOTP version 1.0 as the base transaction protocol to provide a transaction service over the Internet. The third one is the Open Trading Protocol Toolkit for Java (JOTP) [120] proposed by Xenosys Corporation. It intends to implement IOTP on both the client and server sides. The JOTP client will be implemented as a protocol commerce bean for the Java Wallet. The JOTP server framework will be based on the Enterprise JavaBean and Servlet component models, and designed for seamless integration with the merchants' existing information systems. Other examples include CyberSource, Differential, PayByCheck and Brokat, and InterPay, who are implementing IOTP version 1.0 for selected services.

So far, there is no complete implementation of IOTP, and an interoperability test between independent implementations has not yet been conducted. The development of IOTP is still in an early stage. In Chapter 3, we shall examine IOTP in detail.

2.2.3 Electronic Data Interchange

EDI [141], the most traditional standard for e-commerce messaging, is a straightforward electronic version of paper forms, and is intended for business messaging over VANs. The two main standards, *American National Standards Institute Accredited Standards Committee X12 standards* (ANSI ASC X12) [67] and *United Nations Electronic Data Interchange for Administration, Commerce and Transport* (UN/EDIFACT) [69], are both primarily data format standards. ANSI ASC X12 covers EDI messaging for control and foundation sectors, and UN/EDIFACT for business and administration sectors. Other EDI standards include: *Trade Data Interchange* (TDI) covering a series of EDI messaging standards for retail and distribution; *Healthcare EDI standards* concerning exchange of messages related to clinical, financial and administrative information in healthcare; *International Air Transport Association (IATA) EDI standards* concerning interchange of information related to air transport, and *Society for Worldwide Interbank Financial Telecommunications (SWIFT) EDI standards* specifying a set of EDI messages for interbank data exchange.

Recently, traditional EDI has been moving to the far more elaborate, validated and customisable business scenarios of *Open-EDI* [81]. Open-EDI is defined as the electronic data interchange among multiple organizations to accomplish an explicit shared business goal according to Open-EDI standards [81]. There are also standards developed for moving EDI onto the Internet. One typical example is *Electronic Data Interchange-Internet Integration* (EDIINT), which is developed by IETF as RFC 1767 [62] for interoperability between EDI transaction packages over the Internet.

XML [33], developed by World Wide Web Consortium (W3C) [59], is considered to be the native language for the next generation of most of the popular Web browsers, due to its power in enabling interoperability and simplifying the sharing and reuse of information between different domains. This makes it the new trend in e-commerce messaging to integrate XML with EDI. Three main standards were produced: *X12-XML* [252] which moves X12 into XML syntax; *XML/EDI* [246] which combines XML and EDIFACT to generate EDIFACT messages through XML-coded electronic forms; and *electronic business XML* (ebXML) [172] which applies XML to EDI messaging and aims to provide an open XML-based infrastructure enabling the global use of electronic business information in an interoperable, secure and consistent manner.

2.3 Formal Analysis of E-Commerce Protocols

E-commerce applications are susceptible to failures if the underlying protocols are not properly designed and analysed. These failures could result in financial loss to any participant, e.g., commercial traders, financial institutions or consumers. However, ensuring the correctness of complex e-commerce protocols is a difficult task and informal methods are inadequate. Formal methods [51], which are those based on mathematics (see Chapter 6), have proven to be powerful in investigating the correctness of communication protocols, including e-commerce protocols. The main advantage of using formal methods is that they can be used to create unambiguous protocol specifications amenable to computer-aided verification. In this subsection, we review related work on applying formal methods to the modelling and analysis of e-commerce protocols.

Several different approaches based on different techniques to the design and analysis of e-commerce protocols can be found in [96, 185, 196, 247]. Gurgens et al. [96] provides a search method that relies on automatic theorem proving [187] tools for detecting potential security flaws in key distribution and authentication protocols used for e-commerce transactions. Ray and Ray [196] apply model checking [49] to obtain an assurance about the existence of several properties (e.g., the confidentiality and integrity of information exchanged) in an e-commerce protocol given the presence of site or communication failures. Papa et al. [185] present an approach for the analysis of e-commerce protocols based on a formalism that integrates logics [165] and process calculi [163]. Wei et al. [247] describe their research on the synthesis of the extended Unified Modelling Language (UML) [146] and their formal verification approach for automatic development of reliable e-commerce protocols.

In most commercial transactions, *accountability* refers to the ability to hold individuals or organizations accountable for transactions. Kessler et al. [140] present an authentication logic which is capable of analysing the accountability of e-commerce protocols. Kailar [133] defines a framework for the formal analysis of e-commerce protocols that require accountability. Zhou et al. [254] propose a new framework with three major improvements to [133] by taking into account the reliability of communication channels. Other properties of interest are *fairness* and *atomicity* in e-commerce. A *fair* e-commerce system must ensure that any participant does not gain any advantage over a correctly behaved participant. Asokan [9] introduces and analyses *fairness* in e-commerce protocols. Atomicity is concerned with logically linking multiple operations so that either all of them or none of them are executed in transaction processing. Tygar [225] introduces atomicity in e-commerce, and describes how to analyse atomic or non-atomic e-commerce protocols.

Formal analysis of the security of electronic payment protocols is addressed in [17, 30, 101, 184]. Heintze et al. [101] develop model checking techniques to analyse that two example payment protocols (NetBill [213] and Digicash [43]) retain crucial security constraints. Bolignano [30] proposes a generalized approach to expressing and verifying security properties in electronic payment protocols. Benerecetti et al. [17] apply multi-agent model checking, an extension of traditional model checking to include time and beliefs, to verify electronic payment protocols. Panti et al. [184] present a methodology for verifying security requirements of electronic payment protocols by means of a symbolic model checker.

Work presented in [16, 44, 157, 201, 234] focuses on the analysis of the SET protocol. Chen et al. [44] analyse two key problems associated with the payment process of SET. Sakurada et al. [201] specifies SET as a collection of processes where each process expresses the role of a participant, and then analyse SET over this role-based specification. Bella et al. [16] presents the work on modelling and verification of the Cardholder's registration process in SET. Both [157] and [234] apply model checking techniques to the analysis of SET. Lu et al. [157] establish five essential correctness properties of SET and then prove that SET satisfies these properties, and Verdicchio et al. [234] verifies that the message creation and processing steps prescribed by SET guarantee data integrity and confidentiality.

However, none of the above work relates to IOTP. As far as we are aware, the only work on the formal analysis of IOTP is Wang's [244, 245] and ours [177, 179–182]. In [244, 245], Wang et al. present their progressive work on the development of an approach for modelling e-commerce protocols in a language that combines Communicating Sequential Processes (CSP) [103] and

Z [218]. The approach concentrates on how to formally specify the XML documents exchanged in an e-commerce protocol. With automatic tools, their formal specification can be used to detect errors in the definition of XML documents. The approach is illustrated using IOTP, which uses XML as the document format for the content of messages. As a result, an error was revealed in the definition of an XML element used in constructing IOTP messages.

In contrast to Wang's work, we attempt to verify the dynamic behaviours of IOTP transactions. We apply a protocol engineering methodology [21] using Coloured Petri nets [123]. Our work started with the modelling and analysis of just the IOTP Deposit transaction [180]. We then proposed a simplified protocol architecture for IOTP [179]. In [181] we presented a formal specification of the complete set of IOTP transactions with an analysis of transaction termination properties. In [178] we presented an improved formal specification of IOTP by refining the structure of our previous specifications to better reflect IOTP's definition and to improve modularity and re-use. The IOTP's procedures for error handling and arbitrary transaction cancellation are also included. In parallel with the development of the protocol specification, we have also created a formal service specification for IOTP [182]. In [177], we presented the first attempt to verify IOTP against its service. This thesis consolidates and extends the above work presented in [177–182].

Chapter 3

Internet Open Trading Protocol

As described in Chapter 2, the Internet Open Trading Protocol (IOTP) is an e-commerce protocol being developed by the Internet Engineering Task Force (IETF), and is published as Request For Comments number 2801 (RFC 2801) [36]. The core of IOTP is a set of electronic transactions that capture common trading activities. Our objective is to investigate the functional behaviour of IOTP and verify certain properties. To do this we need a very good understanding of IOTP. The main purpose of this chapter is to provide an introduction to IOTP. Section 3.1 outlines the structure of RFC 2801, Section 3.2 introduces IOTP according to its definition in the RFC, and the scope of the current version of IOTP is given in Section 3.3. The introduction of IOTP focuses on the set of basic concepts as well as its major protocol operations. Parts of the RFC that do not reflect the functional behaviour of IOTP are not included.

3.1 Structure of the IOTP Specification

RFC 2801 [36] specifies IOTP version 1.0, and can be used to design and implement systems that support electronic trading transactions using IOTP. The specification is the largest RFC ever developed by IETF, comprising 17 sections and totalling 290 pages. Below, we summarize each of these sections.

Section 1 (Background)[1] gives a brief background on e-commerce and the benefits of IOTP.

Section 2 (Introduction) introduces five *trading roles* that participate in electronic trading, and describes four *trading exchanges* that involve the exchange of data between trading roles. The scope of IOTP version 1.0 is also given in this section.

Section 3 (Protocol Structure) defines the structure of *IOTP messages* and briefly describes the set of *IOTP transactions*. Each IOTP message is constructed from *trading blocks* and *trading components*. A detailed definition of the *Transaction Reference Block* and its trading components is given. This section also defines transaction cancellation.

Section 4 (IOTP Error Handling) provides a generic overview of how trading roles handle errors and exceptions in processing IOTP messages during an IOTP transaction.

Section 5 (Security Considerations) and 6 (Digital Signatures and IOTP) discusses how security issues, such as the use of digital signatures, are addressed in IOTP.

Section 7 (Trading Components) and 8 (Trading Blocks) provides a complete definition of all trading components and trading blocks that are used to construct IOTP messages.

[1]When referring to sections from RFC 2801, we will include the section name in parentheses to distinguish references to sections in this thesis (which only include the number.)

Section 9 (Internet Open Trading Protocol Transactions) introduces six *document exchanges* and specifies how these document exchanges are used to implement IOTP transactions. During each transaction procedure defined in this section, transaction cancellation is mentioned, but error handling is not considered.

Section 10 (Retrieving Logos) describes how to retrieve IOTP specific logos.

Section 11 (Brands) provides an overview of payment brand definition and selection.

Section 12 (IANA Considerations) describes how the new codes used in IOTP are coordinated by the Internet Assigned Numbers Authority (IANA).

Section 13 (Internet Open Trading Protocol Data Type Definition) contains the XML Data Type Definitions (DTD) for IOTP messages.

Section 14 (Glossary) describes all major terminology used by IOTP.

The last three sections are: Section 15 (References) containing a list of normative and informative references, Section 16 (Author's Address) and Section 17 (Full Copyright Statement).

3.2 IOTP Definition

In this section, we introduce IOTP according to its definition given in RFC 2801 [36]. Our aim is to accurately summarize the essential parts of IOTP. No attempt is made to discuss or justify any of the decisions made in the design of IOTP in this section.

Five interrelated concepts defined in IOTP are: *trading roles*, *IOTP transactions*, *IOTP messages*, *document exchanges* and *trading exchanges*. A trading role is used to identify a certain role that an organisation can assume while trading, e.g., a consumer who buys or a merchant who sells. An IOTP transaction comprises a sequence of IOTP message exchanges between trading roles. Document and trading exchanges are designed as building blocks for implementing IOTP transactions.

Two important procedures related to IOTP transactions are: *transaction cancellation* and *error handling*. An IOTP transaction may be cancelled by any trading role engaged in that transaction. Rules are defined which determine when a transaction can be cancelled by a trading role. IOTP error handling concerns how trading roles handle errors and exceptions when processing each IOTP message. The errors are categorised and specific handling measures are applied to different error categories.

In the following, we describe each of the above five concepts, transaction cancellation and error handling, in detail.

3.2.1 Trading Roles

IOTP defines five trading roles, known as *Consumer*, *Merchant*, *Payment Handler*, *Delivery Handler* and *Merchant Customer Care Provider*. Figure 3.1 (based on Figure 1 in Section 2.1 of [36]) depicts the conceptual trading model for IOTP, where the trading roles communicate across the Internet. The Consumer requests, receives and pays for goods or services. The Merchant supplies goods or services and receives payment via the Payment Handler. The Payment Handler (intuitively a bank) receives money from the Consumer on behalf of the Merchant. The Delivery Handler (intuitively a courier firm) delivers the goods to the Consumer on behalf of the Merchant. Finally, the Merchant Customer Care Provider is responsible for

resolving disputes and problems between the Consumer and Merchant. The details of this trading role is however not given in the RFC.

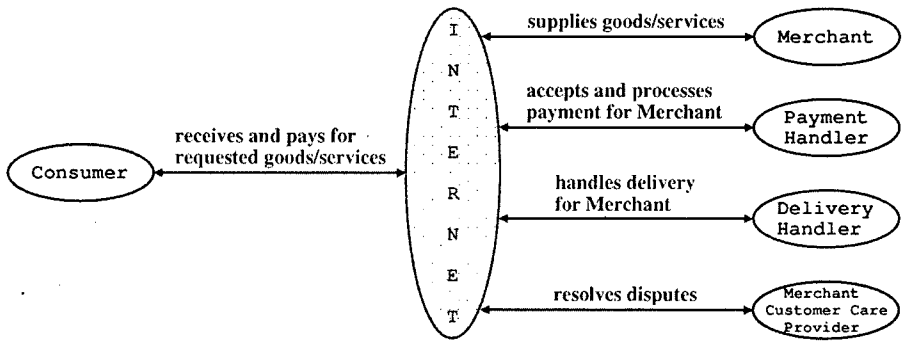


Figure 3.1: IOTP conceptual trading model.

3.2.2 IOTP Transactions

IOTP defines two transaction categories: *Authentication and Payment-related transactions*, and *Infrastructure transactions*. An Authentication or Payment-related transaction (in Section 9.1 (Authentication and Payment Related Transactions) of [36]) is constructed by combining document exchanges, whereas an Infrastructure transaction (in Section 9.2 (Infrastructure Transactions) of [36]) is designed without reference to document exchanges.

Payment-related Transactions

The core of IOTP is a set of Payment-related transactions named *Purchase*, *Refund*, *Deposit*, *Withdrawal* and *Value Exchange*. The Purchase transaction is used for buying goods or services electronically. The Refund occurs when the refund of a payment is required, usually as a result of returning an earlier Purchase. The Deposit and Withdrawal transactions are used for the deposit or withdrawal of *electronic cash*. Finally, the *Value Exchange* transaction supports currency conversion.

We shall use the Purchase transaction to illustrate the nature of Payment-related transactions. Figure 3.2 shows a possible sequence of IOTP message exchanges between trading roles during a Purchase transaction, in the form of a message sequence chart [118]. Each column of the message sequence chart corresponds to a trading role. In Figure 3.2, the sequence of message exchanges is divided into three subsequences, indicating that the Purchase transaction is conducted in three trading phases called *Offer*, *Payment* and *Delivery*.

The Offer phase (events 1-4) is concerned with negotiations between the Consumer and Merchant, regarding payment options and their associated payment protocols. The payment options comprise a list of *payment brands*, e.g, Visa, MasterCard, and Mondex. Each payment brand identifies a certain type of *payment instrument* like a credit card or electronic cash. The Consumer and Merchant need to agree on making the payment with one of the payment options, such as using a Visa credit card or Mondex electronic cash. Each payment option has its associated payment protocol(s). For example, when using a Visa credit card for payment, the SET protocol [237] can be used, while Mondex electronic cash requires the use of the Mondex Value Transfer Protocol [166]. In Figure 3.2, the Consumer decides to buy goods and sends an Offer Request (event 1) to the Merchant to purchase them. This event is currently

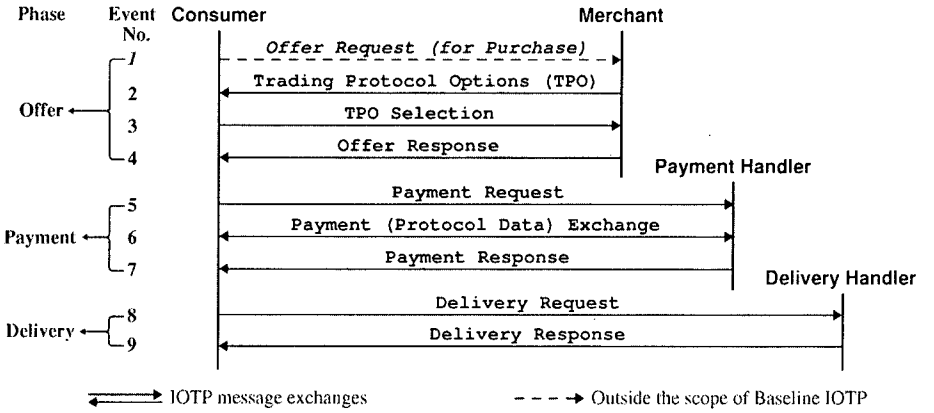


Figure 3.2: A possible sequence of messages in a Purchase transaction.

outside the scope of IOTP. In practice, it could be an HTTP request or an e-mail conveying the information that initiates a transaction. Upon receiving the Offer Request, the Merchant starts the transaction by offering the Consumer a list of *Trading Protocol Options* (TPO) (event 2). This includes the available payment options and associated payment protocols. The Consumer makes a choice, and sends it back as a *TPO Selection* (event 3) by, e.g., clicking on a button in a Web browser. The Merchant uses the selection to create and send back an *Offer Response* (event 4), which contains details of the goods to be purchased together with payment and delivery instructions.

The second phase concerns Payment interactions between the Consumer and Payment Handler (events 5-7). After checking the Offer Response for purchase details, the Consumer sends the Payment Handler a *Payment Request* (event 5). The Payment Handler checks the Payment Request, and if valid, the payment is made by exchanging *Payment (Protocol Data) Exchange* IOTP messages (event 6) that encapsulate the selected payment protocol messages (e.g., SET messages). At the end of the Payment phase, the Payment Handler sends a *Payment Response* (event 7) containing the payment result (e.g., the payment receipt).

The third phase is concerned with delivering goods to the Consumer (events 8-9). After checking the Payment Response for the payment made, the Consumer sends the Delivery Handler a *Delivery Request* (event 8). The Delivery Handler schedules the delivery and sends the Consumer a *Delivery Response* (event 9) containing details of the delivery such as when and where the delivery is to be made, and possibly the actual delivery if the goods are electronic (e.g., an e-journal).

Authentication Transaction

An Authentication transaction allows one trading role (the *Authenticator*) to verify the *bona fides* of the other trading role (the *Authenticatee*). It consists of only an authentication phase comprising the exchange of three IOTP messages. At first, an *Authentication Request Message* is issued by the Authenticator, specifying the authentication algorithm (e.g., the Secure Hash Algorithm (SHA-1) [175]) to be used. As a result, the Authenticatee replies with an *Authentication Response Message* containing the authentication data generated by using the above algorithm. Finally, the Authenticator provides an *Authentication Status Message* conveying the results of the Authenticator's verification of the Authenticatee's response, i.e., whether the authentication succeeded or failed.

The Authentication transaction may occur at any time, e.g., it could be carried out in the process of another transaction, or independently of any other transaction. Combining the Authentication transaction with an ongoing transaction is typically used by the Consumer to authenticate the Merchant, Payment Handler or Delivery Handler, or by the Payment Handler or Delivery Handler to authenticate the Consumer. The trading role that decides to authenticate another peer trading role is required to suspend the current transaction, so that a stand-alone Authentication transaction can be carried out. Whether or not to continue with the suspended transaction is determined by the authentication result.

Infrastructure Transactions

IOTP defines two infrastructure transactions: *Inquiry* and *Ping*. The Inquiry transaction provides information on the status of another transaction. For example, whether the transaction succeeded, failed, or is suspended. The Ping transaction supports a simple query on the basic connectivity between any two trading roles within an ongoing transaction. Both Inquiry and Ping can be conducted between any two trading roles. Each of them consists of one round-trip of request and response messages, namely *Inquiry Request*, *Inquiry Response*, *Ping Request* and *Ping Response*.

3.2.3 IOTP Messages

IOTP messages are formatted as Extensible Markup Language (XML) [33] documents and transfer transaction information between trading roles. IOTP messages are introduced in Section 3 (Protocol Structure) excluding 3.10 (Cancelled Transactions) of RFC 2801 [36].

Trading Blocks

An IOTP message consists of a number of *trading blocks*, as given by its XML definition in Figure 3.3 (a) (taken from Section 3.2 (IOTP Message) of [36]). IOTP defines 19 trading blocks, shown in Figure 3.3 (a), to construct IOTP messages. Section 3.3 (Transaction Reference Block) and 8 (Trading Blocks) of RFC 2801 [36] provide a full definition of these 19 trading blocks. Except for the *Transaction Reference Block*, *Signature Block* and *Error Block*, each of the other 16 trading blocks corresponds to each of the 16 different IOTP messages with the same name. For example, a *TPO Block* is only used in a *TPO Message*. The Transaction Reference Block contains information that identifies an IOTP transaction and an IOTP message, and is therefore required in each IOTP message. The Signature Block conveys the digital signatures and certificates that may be needed in an IOTP transaction. The use of Signature Blocks is optional in an IOTP message, and will not be considered any further as it is beyond the scope of this thesis. The Error Block is used to convey error information within an IOTP message. The details of an Error Block are closely related to IOTP error handling, and will be described in Section 3.2.7.

Trading Components

Each trading block comprises a pre-defined set of *trading components*. For example, Figure 3.3 (b) (taken from Section 3.3 (Transaction Reference Block) of [36]) depicts the structure

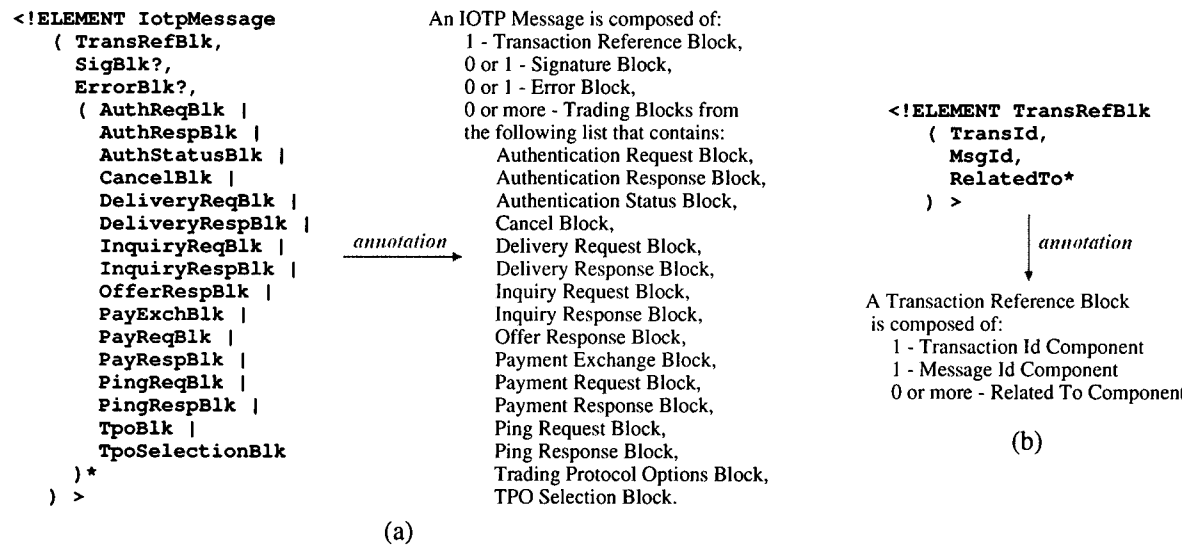


Figure 3.3: The XML definition of (a) an IOTP message and (b) a Transaction Reference Block.

of a Transaction Reference Block comprising three trading components. The *Transaction Id Component* uniquely identifies an IOTP transaction, the *Message Id Component* uniquely identifies an IOTP message, and the *Related To Component* links the current IOTP transaction to other IOTP transactions. The Related To Component is needed when two IOTP transactions interrelate with each other. For instance, a Refund transaction might occur as a result of an earlier Purchase transaction, or an Authentication transaction could be carried out by the Consumer to authenticate the Merchant within an ongoing Payment-related transaction.

Apart from the above three trading components that are only used in a Transaction Reference Block, IOTP defines another 21 trading components, as listed in Table 3.1, to construct the other 18 trading blocks. Section 7 (Trading Components) of RFC 2801 [36] contains a detailed definition of each of these 21 trading components, taking 70 pages in the RFC.

No.	Trading Component
1	Protocol Options
2	Authentication Request
3	Authentication Response
4	Trading Role Information
5	Order
6	Organisation
7	Brand List
8	Brand Selection
9	Payment
10	Payment Scheme
11	Payment Receipt
12	Payment Note
13	Delivery Component
14	Consumer Delivery Data
15	Delivery Note
16	Status
17	Trading Role Data
18	Inquiry Type
19	Signature
20	Certificate
21	Error

Table 3.1: The 21 trading components used to construct all trading blocks except a Transaction Reference Block in IOTP.

In further detail, each trading block and trading component have a set of *attributes*, and each trading component may also contain a number of *XML elements*. It is the attributes and XML elements of a trading component that convey the concrete transaction information. These are illustrated with the example shown next.

Example of Offer Response Message

Figure 3.4 depicts the XML definition of an Offer Response Message. The message (defined from line 1 to 34) consists of a Transaction Reference Block (line 2 to 14) and an Offer Response Block (line 15 to 33). The Transaction Reference Block comprises a Transaction Id Component (line 3 to 8) and a Message Id Component (line 9 to 13). The Offer Response Block comprises a Status Component (line 16 to 19), an Order Component (line 20 to 23), a Payment Component (line 24 to 28) and a Delivery Component (line 29 to 32).

```
1   <IotpMessage>
2     <TransRefBlk ID='M2.1'>
3       <TransId ID='M2.2'
4         Version='1.0'
5         IotpTransId='abc123'
6         IotpTransType='BaselinePurchase'
7         TransTimeStamp='2003-01-01T11:30:02Z+0930'>
8       </TransId>
9       <MsgId ID='M2'
10        RespIotpMsg='C1'
11        xml:lang='english'
12        TimeStamp='2003-01-01T11:31:28Z+0930'>
13      </MsgId>
14    </TransRefBlk>
15    <OfferRespBlk ID='M2.3'>
16      <Status ID='M2.4'
17        StatusType='Offer'
18        ProcessState='CompletedOk'
19      </Status>
20      <Order ID='M2.5'
21        OrderIdentifier='Bk658.054678E13'
22        ShortDesc='1 IOTP BK by Burdett'
23      </Order>
24      <Payment ID='M2.6'
25        OkFrom='2003-01-01T11:32:00Z+0930'
26        OkTo='2003-01-07T11:32:00Z+0930'
27        SignedPayReceipt='True'
28      </Payment>
29      <Delivery ID='M2.7'
30        DelivExch='True'
31        DelivAndPayResp='False'
32      </Delivery>
33    </OfferRespBlk>
34  </IotpMessage>
```

Figure 3.4: An example of the XML for an Offer Response Message.

Each trading block or trading component, except the Message Id Component, has an *ID* attribute uniquely identifying that trading block or trading component within the transaction.

The ID attribute of the Message Id Component uniquely identifies the *IOTP message* within the transaction. It consists of a representative letter and one or more digits. The letter indicates which trading role sends the message and is given by the initial letter of that trading role name. For example, each IOTP message sent by the Consumer has a message identifier starting with the letter "*C*". The digits, which have the same definition as the digits used in XML [33], constitute the number of the message identifier. As a guideline recommended by RFC 2801, the message identifier of the first message sent by a trading role is given the digit "*1*", and subsequent (non-duplicate) messages sent by the same trading role have their message

identifier incremented by one for each message sent. For example, the Offer Response Message shown in Figure 3.4 is identified as M2 (line 9), indicating it is the 2nd IOTP message sent by the Merchant in the transaction. The identifiers of the trading blocks and trading components are specified at the same level in one IOTP message. For example, the Transaction Reference Block shown in Figure 3.4 has an identifier of M2.1 and the Transaction Id Component has M2.2, indicating that they are the 1st and 2nd trading block or trading component in the message identified as M2.

Apart from the ID attribute, each trading component also has a set of other attributes.

In the Transaction Reference Block, the Transaction Id Component has four other attributes: *Version* (line 4) specifying the version of IOTP, *IotpTransId* (line 5) containing the data value of a transaction identifier, *IotpTransType* (line 6) indicating the type of the transaction being carried out, and *TransTimeStamp* (line 7) specifying the time at which the transaction started. The Message Id Component has a *RespIotpMsg* attribute (line 9) carrying the identifier of the IOTP message to which this IOTP message is a response. For example, the Offer Response Message in Figure 3.4 is a response to the message with an identifier of C1, i.e., the 1st IOTP message sent by the Consumer. The other two attributes are: *xml:lang* (line 11) specifying the language used within the message, and *TimeStamp* (line 12) set to the time at which the message was created.

In the Offer Response Block, the Status Component has two other attributes: *StatusType* (line 17) indicating the type of the phase this component is reporting on, such as *Offer*, *Payment*, *Delivery* or *Authentication*, and *ProcessState* (line 18) containing a *state code* that indicates the current state of the phase being carried out, e.g., the state code CompletedOk is used when the processing in the phase has completed successfully. The Order Component has an *OrderIdentifier* attribute (line 21) conveying a code, reference number or other identifier that can be used to identify the order made by the Consumer, and a *ShortDesc* attribute (line 22) which provides a short description of the order and is used to facilitate selecting an individual order from a list of orders. The Payment Component has two attributes, *OkFrom* (line 25) and *OkTo* (line 26), specifying the earliest time (OkFrom) and the latest time (OkTo) when the Payment Handler may accept for processing a Payment Request Block, and also a *SignedPayReceipt* attribute with a value of True or False, indicating whether or not a Payment Response Block generated by the Payment Handler must be digitally signed. Finally, the Delivery Component has a *DelivExch* attribute (line 30) specifying if the transaction involves a delivery and a *DelivAndPayResp* attribute (line 31) indicating if the Delivery Response Block and the Payment Response Block is combined into one IOTP message (see Section 3.2.4). Both attributes have two valid values: True and False. For example, in the IOTP transaction that uses the Offer Response Message shown in Figure 3.4, a delivery is involved, and the Delivery Response Block and the Payment Response Block will be used to construct different IOTP messages.

3.2.4 Document Exchanges

The three phases, Offer, Payment and Delivery of the Purchase transaction (see Figure 3.2), are called document exchanges. IOTP defines six document exchanges (from Section 9.1.1 (Authentication Document Exchange) to 9.1.5 (Payment and Delivery Document Exchange) of [36]) named *Brand Dependent Offer*, *Brand Independent Offer*, *Payment*, *Delivery*, *Payment*

and Delivery, and *Authentication*. Note that the term *brand* refers to the *payment brand* defined in Section 3.2.2. Figure 3.5 depicts the sequences of messages for each of these six document exchanges.

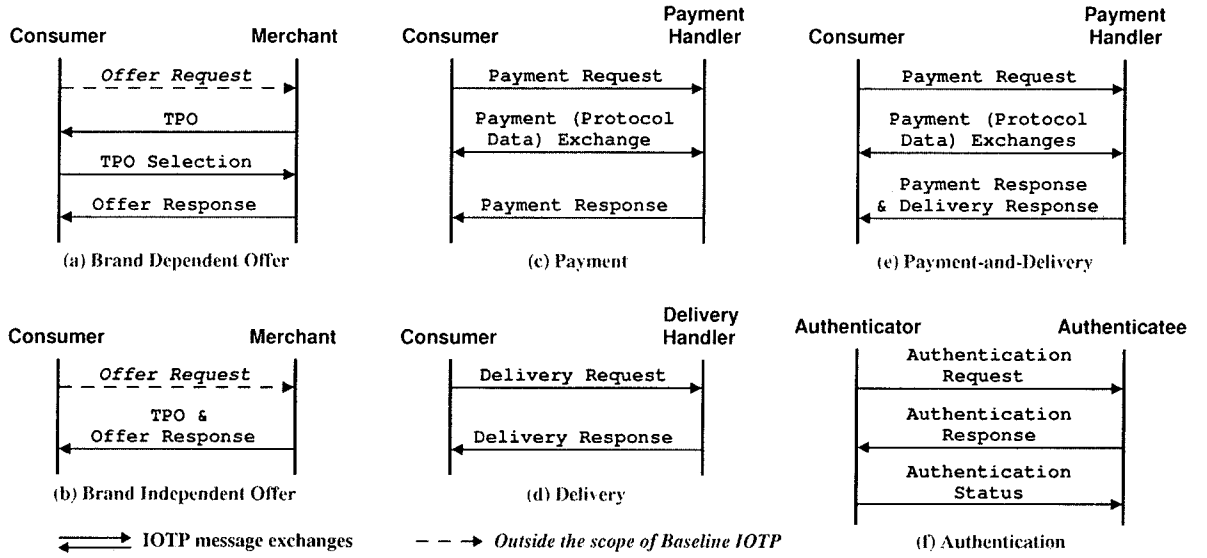


Figure 3.5: The six document exchanges defined in IOTP.

As shown in Figure 3.5 (a) and (b), the distinction between a Brand Dependent Offer and a Brand Independent Offer concerns TPO selection. The Brand Dependent Offer occurs when the Merchant offers some additional benefits to the Consumer depending on the specific payment brand being selected, e.g., a 5% discount when the Consumer pays for an air ticket with British Airways Mastercard. As a result, the exact payment amount provided in the Offer Response *depends* on the *payment brand* in the TPO Selection. However, the Merchant may not offer such additional benefits, in which case, the payment amount is the same no matter which payment brand is selected by the Consumer. This is defined in IOTP as a Brand Independent Offer, in that the content of the Offer Response is *independent* of the *payment brand* chosen by the Consumer. In this case, the Merchant does not require the Consumer's TPO Selection before the Offer Response can be generated, and therefore sends a combined *TPO & Offer Response Message* to the Consumer.

The Payment and the Delivery document exchanges shown in Figure 3.5 (c) and (d) have already been encountered in the Purchase transaction of Figure 3.2. In addition, two things need to be mentioned. Firstly, the Payment (Protocol Data) Exchanges that are required by a particular payment system (such as SET) can occur any number of times, catering for various payment systems to be encapsulated by IOTP. From this point of view, we consider IOTP as payment-system independent. Secondly, the delivery may also occur between the Consumer and Payment Handler during the payment. This results in a Payment-and-Delivery document exchange as shown in Figure 3.5 (e), where the last message transmitted is a combined *Payment Response & Delivery Response Message*.

Finally in Figure 3.5 (f), an Authentication document exchange comprises three Authentication specific messages exchanged between the Authenticator and Authenticatee. The Authenticator and Authenticatee may be any two distinct trading roles.

The above six document exchanges constitute a common set of sub-transactions used to implement Authentication and Payment-related transactions. An Authentication transaction

consists of an Authentication document exchange. A Purchase transaction may start with either a Brand Dependent Offer or a Brand Independent Offer, followed by a Payment and optionally a Delivery document exchange. It may also contain either of the two Offer document exchanges followed by a Payment-and-Delivery document exchange. For example, the Purchase transaction scenario shown in Figure 3.2 comprises three sequential document exchanges: a Brand Dependent Offer, a Payment and a Delivery. The Refund, Deposit and Withdrawal transactions start with an Offer document exchange which is followed by a Payment document exchange. In a Value Exchange transaction, an Offer document exchange is followed by two Payment document exchanges that occur in sequence. Within each Payment-related transaction, an optional Authentication document exchange can occur but only for the Merchant authenticating the Consumer at the beginning of a transaction, as specified in Section 9.1.12 (Valid Combinations of Document Exchanges) of RFC 2801 [36]. For authentication required at any other time during a transaction, an Authentication transaction is required. This is specified in Section 9.1.13 (Combining Authentication Transactions with other Transactions) of RFC 2801 [36] with no reasons provided.

3.2.5 Trading Exchanges

In Section 2.2 (Trading Exchanges) of RFC 2801 [36], IOTP defines four trading exchanges as an alternative to document exchanges for constructing IOTP transactions. These are *Offer*, *Payment*, *Delivery* and *Authentication*. Each trading exchange involves exchanges of basic trading information between trading roles, in terms of a set of *trading components*. As described in Section 3.2.3, trading components are used to construct trading blocks, which are concatenated to form IOTP messages. For example, a *Brand List Component* is one of the trading components used to construct a *TPO Block*. The TPO Block, together with a Transaction Reference Block, constitutes a TPO Message.

Relationship between document exchanges and trading exchanges. IOTP states the relationship between trading exchanges and document exchanges as follows (page 185, [36]): "There is a close, but not exact, correspondence between a document exchange and a trading exchange", where "the main difference is that some document exchanges implement part or all of two trading exchanges" by combining trading components into IOTP messages. Figure 3.6 depicts three document exchanges (Brand Dependent Offer, Payment and Delivery) on the left, and their relationship with the corresponding trading exchanges (Offer, Payment and Delivery) on the right. Each dotted line connecting an event in a document exchange with an event in a trading exchange indicates a correspondence between these two events. For example, the event of sending a TPO Message in an Offer document exchange corresponds to the event of transferring a Brand List Component from the Merchant to Consumer in a Payment trading exchange. We show three typical examples of the correspondence between trading exchanges and document exchanges, indicated by the dotted areas shown in Figure 3.6. Firstly, a Brand Dependent Offer document exchange is the combination of an Offer trading exchange and the first part of a Payment trading exchange. Secondly, a Payment document exchange corresponds to the second part of a Payment trading exchange. Finally, a Delivery document exchange corresponds to the second part of a Delivery trading exchange. An Authentication trading exchange is directly implemented by an Authentication document exchange.

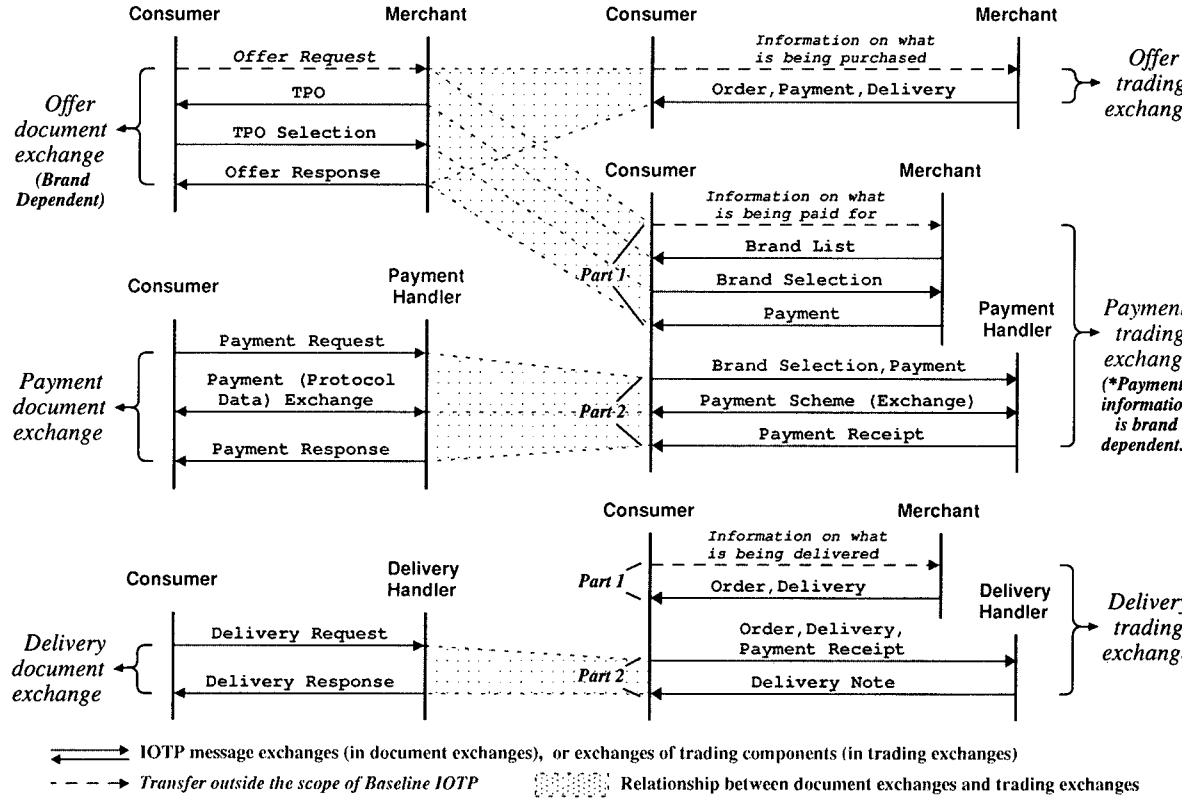


Figure 3.6: Mapping document exchanges to trading exchanges.

Three trading exchanges. Below, we describe in detail the three trading exchanges involving the set of trading components depicted in Figure 3.6. For brevity, we do not introduce other trading components that are not shown in Figure 3.6. The reader interested is referred to Section 2.2 (Trading Exchanges) of RFC 2801 [36].

At first, to initiate an Offer trading exchange, the Consumer sends information on what to purchase to the Merchant. The Merchant then replies with the required information that is conveyed by a group of trading components being *Order*, *Payment* and *Delivery*. The Order Component provides descriptions of the goods or services being offered by the Merchant. The Payment Component contains details of how much to pay, the currency to be used and who receives the payment. The Delivery Component conveys information about how delivery is to occur, e.g., either online or physically.

A Payment trading exchange is more complicated and is conducted in two phases. In the first phase, communication is between the Consumer and Merchant. Upon receiving the information on what is being paid for, the Merchant replies to the Consumer with a list of available payment brands and associated payment protocols via a Brand List component. The Consumer makes a selection and sends it back via a *Brand Selection* component. The Merchant uses the selection to create and send back a Payment component. Entering the second phase, the Consumer switches to communicate with the Payment Handler by sending the information conveyed by the above Payment component together with the previous Brand Selection component. The Payment Handler checks the information, and if valid, the payment is made by exchanging *Payment Scheme* components that encapsulate the selected payment protocol messages. When the payment completes, the Payment Handler provides the Consumer with a record of the payment via a *Payment Receipt* component. Payment trading exchanges

may occur in different ways. The above Payment trading exchange shown in Figure 3.6 is a general case where the payment amount conveyed by the Payment component is *dependent* on the payment brand given in the Brand Selection component. In the other case, the Consumer is not required to return a trading selection to the Merchant, and thus the payment amount is *independent* of the payment brand selected.

In a Delivery trading exchange, the Consumer firstly communicates with the Merchant to obtain the information that is required for delivery. Such information is conveyed by an Order and a Delivery components. Afterwards, the Consumer communicates with the Delivery Handler by sending this information via the above Order and Delivery components together with the Payment Receipt obtained from the preceding Payment trading exchange. The Delivery Handler schedules the delivery and replies with details of the delivery, such as when and where the delivery is to be made, via a *Delivery Note* component.

3.2.6 Transaction Cancellation

Any trading role involved in an IOTP transaction may cancel that transaction when it is in progress. Transaction cancellation, as briefly described in Section 3.10 (Cancelled Transactions) in RFC 2801 [36], is separate from the definition of IOTP transactions in Section 9 (Internet Open Trading Transactions) of the RFC.

IOTP defines a *Cancel Message* for transaction cancellation. The Cancel Message is composed of a Transaction Reference Block and a *Cancel Block* (Section 8.18 (Cancel Block) of [36]). The Cancel Block contains a *Status Component* (Section 7.16 (Status Component) of [36]) which has a *CompletionCode* attribute that indicates the reason for cancelling the transaction. The Cancel Message can be sent at any time within an ongoing transaction. Any trading role engaged in a transaction should stop any further action or processing associated with that transaction upon sending or receiving a Cancel Message.

A Payment-related transaction involves at least three trading roles and comprises at least two different document exchanges. IOTP has specific consideration with respect to the cancellation issued by different trading roles at different points within a Payment-related transaction. We summarize this into the following four cases based on our understanding of IOTP specification in Section 3.10 of [36].

1. When the Consumer cancels a transaction *during* a document exchange, the Cancel Message should be sent to the peer non-Consumer trading role that is engaged in that document exchange. For example, the Consumer sends a Cancel Message to the Payment Handler to cancel a Purchase transaction during a Payment document exchange.

2. Suppose the Consumer cancels a transaction *after* a document exchange is complete but *before* the transaction completes. In this case, the Cancel Message should be sent to the Merchant that is engaged in the transaction. This is, for instance, applicable when the Consumer cancels an ongoing Purchase transaction (that involves the delivery of goods or services), after the Payment document exchange has completed but before the Delivery document exchange starts.

3. The Merchant should only cancel a transaction after sending the first IOTP message (e.g., the TPO or the Authentication Request Message) but before transmitting the last one (e.g., the Offer Response or the Authentication Status Message).

4. The other two non-Consumer trading roles, the Payment Handler and the Delivery Handler, should only cancel a transaction after having received the request message but before sending the response message within the document exchange in which the trading role is engaged. For example, to cancel an ongoing Purchase transaction, the Payment Handler should only send the Cancel Message to the Consumer, after the Payment Request Message has been received but before the Payment Response Message is sent.

Below, we describe how to apply the above restrictions to the cancellation of the Purchase transaction shown in Figure 3.2.

At first, we consider the cancellation initiated by different trading roles. The Consumer can cancel the transaction at any time after the TPO Message has been received from the Merchant (event 2) but before the Delivery Response Message is received from the Delivery Handler (event 9). The Merchant can only cancel the transaction after the TPO Message has been sent (event 2) but before the Offer Response Message is transmitted (event 4). The Payment Handler can only cancel the transaction after receiving the Payment Request Message (event 5) but before sending the Payment Response Message (event 7). Finally, the Delivery Handler can only cancel the transaction after receiving the Delivery Request Message (event 8) but before sending the Delivery Response message (event 9).

Further, we consider cancellation at different stages within an ongoing transaction. Figure 3.7 depicts five typical scenarios of cancellation initiated at five different points within the Purchase transaction shown in Figure 3.2. Firstly, if the cancellation occurs *during* a document exchange, the Cancel Message is transmitted between the two peer trading roles engaged in that document exchange. The corresponding scenarios are illustrated in Figure 3.7 (a)-(c). Secondly, a transaction may be cancelled *between* two document exchanges, i.e. after one exchange has completed and before the other starts. In this case, only the Consumer can cancel the transaction, and only the Merchant is notified about the cancellation. The corresponding scenarios are illustrated in Figure 3.7 (d) and (e). For brevity, we have not shown in Figure 3.7 (b) and (c) the Offer and Payment document exchanges that have completed.

3.2.7 Error Handling

Section 4 (IOTP Error Handling) of RFC 2801 [36] describes how trading roles handle errors and exceptions when processing each IOTP message exchanged in a transaction. The errors that may occur are categorized by IOTP according to their *error type* and *error depth*. The two error types are *technical errors* and *business errors*, and the error depth indicates whether the error occurs at the *transport level*, the *message level*, or the (trading) *block/component level*. Each error type has its associated error depth, and different error handling procedures are defined in IOTP with respect to the different types of errors being detected. In the following, we describe the IOTP error handling procedures that are applied to the technical errors and business errors, respectively. It should also be mentioned that the IOTP error handling specified in the RFC is mainly concerned with the handling of technical errors.

Technical Errors

Technical errors are those that concern the technical aspects of an IOTP message exchange regardless of any business information conveyed by the message. These may occur at all three

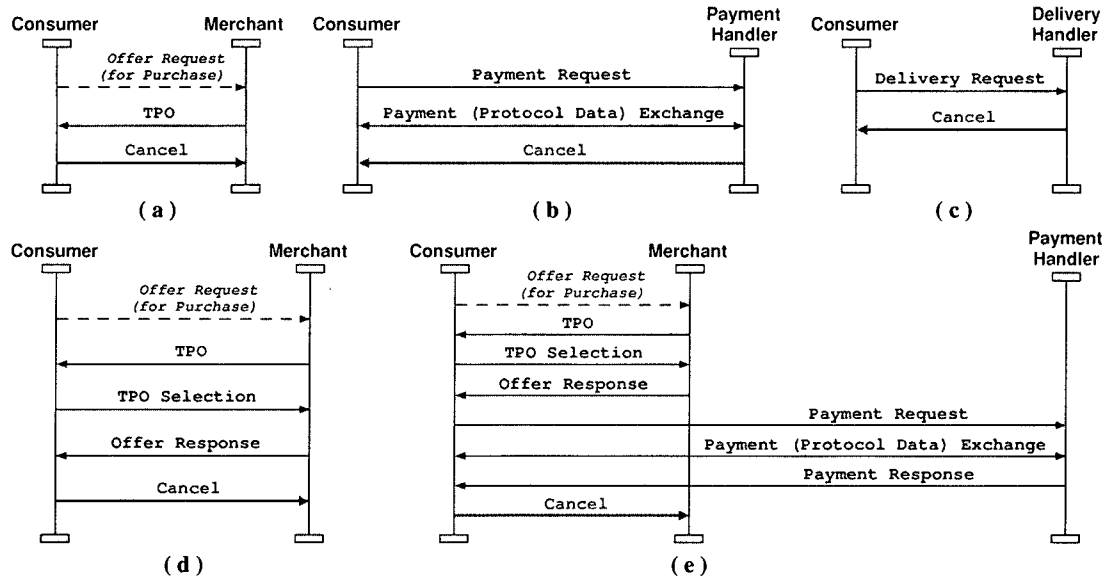


Figure 3.7: Five typical scenarios of cancelling the Purchase transaction shown in Figure 3.2 at five stages: (a) during the Offer document exchange; (b) during the Payment document exchange; (c) during the Delivery document exchange; (d) between the Offer and the Payment document exchange; and (e) between the Payment and the Delivery document exchange.

levels of error depth. Firstly, any problem associated with the transport mechanism e.g., a time out caused by delay in HTTP, is considered a technical error occurring at the transport level. Secondly, any problem related to an entire IOTP message is specified as a technical error at the message level. For example, the structure of the message being received does not conforms to the XML definition of an IOTP message (Figure 3.3 (a)). Thirdly, any technical problem concerned with a trading block or one of its components in an IOTP message is called a block level technical error. For example, the attribute of a trading component has a value that is not specified in the Data Type Definition (DTD) of that trading component. For clarity, we make a distinction between a *message in error* and a *message reporting an error* (or, an *error reporting message* for short). A message in error refers to an IOTP message which contains or causes a technical error, while an error reporting message is an IOTP message which provides a technical error indication that results from a message in error.

Information for reporting a technical error is encoded in an *Error Component* of an *Error Block*. Section 7.21 (Error Component) of RFC 2801 provides a complete definition for each Error Component. Two attributes of an Error Component, namely *ErrorCode* and *Severity*, contain the information about the technical error being reported. The ErrorCode attribute contains an error code which indicates the nature of the error that is detected in the message in error. Section 7.21.2 (Error Code) of [36] specifies 22 valid values for this attribute. For example, the value, *XmlNotWellFrmd*, is used when the message structure does not conform to the XML definition. The Severity attribute indicates how serious the error is. It has three valid values: *Warning*, *TransientError* and *HardError*. Section 7.21.1 (Error Processing Guidelines) of [36] specifies the appropriate actions that need to be taken according to the severity of the error. These are illustrated with a Purchase transaction scenario shown in Figure 3.8, which involves three messages reporting the errors of three different severities.

- A Warning Error can be ignored because it does not affect the operation of the transaction (event 2). IOTP applies the following restriction regarding the construction of an IOTP

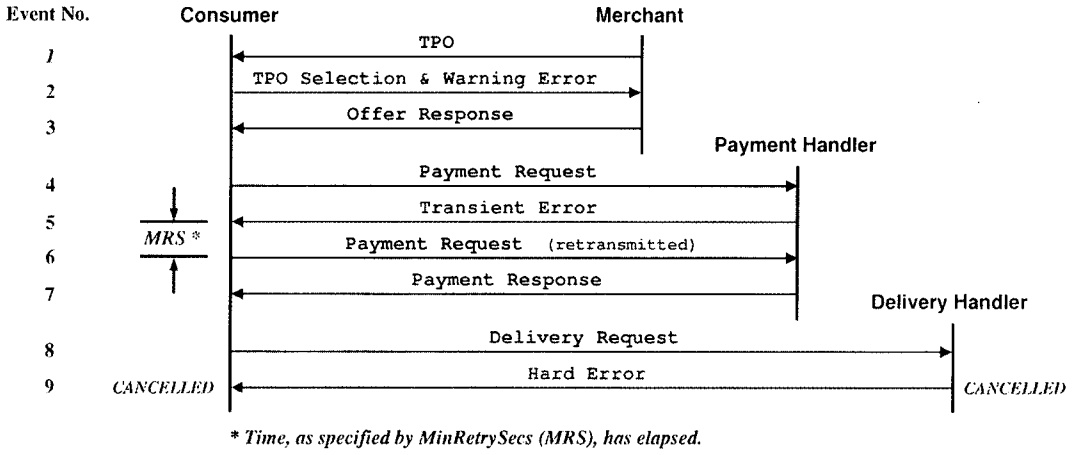


Figure 3.8: A Purchase transaction involving messages reporting errors with different severities.

message reporting a Warning Error. An Error Block comprising an Error Component with its Severity attribute set to Warning must be used together with the Transaction Reference Block and another trading block (e.g., a TPO Selection Block) to construct the corresponding IOTP message (e.g., a TPO Selection Message).

- A Transient Error cannot be ignored but can be recovered by retransmitting the previous message when the time set by the *MinRetrySecs* attribute of the Error Component expires (events 5 and 6). An Error Block comprising an Error Component with its Severity attribute set to TransientError must be used together with only the Transaction Reference Block to construct an *Error Message*. This means that the occurrence of any Transient Error has an effect on the operation of the transaction.

- A Hard Error indicates a fatal problem and results in failure and thereby the cancellation of the IOTP transaction (event 9). Similarly to a Transient error, an Error Message must be used for reporting a Hard Error. Upon sending or receiving an Error Message reporting a Hard Error, the trading role should stop any further processing.

When implementing IOTP, the Consumer trading role is realised by an IOTP client, and the non-Consumer roles are realised by IOTP servers. The technical error handling procedures are defined respectively for IOTP client and server, as specified in Sections 4.5 (Server Role Processing Sequence) and 4.6 (Client Role Processing Sequence) of RFC 2801 [36]. The two procedures are very similar, and can be outlined without making any distinction between client and server operations.

1. In brief, on receipt of a message (in error), the trading role should check 1) the message structure and the transaction identifier (in IotpTransId attribute of the Transaction Id Component), 2) whether the message has been received before (i.e., duplicate message), and 3) any other errors at the message or the block/component level. If an error is detected or processing the previous message has not yet completed when a duplicate is received due to message re-transmission (see below), the trading role generates a message reporting the error and returns it to the sender of the original message. The message reporting the error has an Error Block that consists of an Error Component with its ErrorCode and Severity attributes set to appropriate values. For example, if the message

structure does not conform to the XML definition, the ErrorCode attribute is set to XmlNotWellFrmd and the Severity attribute to HardError.

2. Next, upon receiving a message reporting an error, the trading role should take appropriate actions according to the severity of the error.

3. Finally, a time-out problem in a message exchange (e.g., due to a transport level error) is also considered by IOTP. It is handled by retransmitting the previous message if the response message has not been received within a pre-defined amount of time. The RFC states that "this should occur up to a maximum number of times dependent on the reliability of the transport mechanism being used" (page 70, [36]). The RFC however does not specify what the maximum number of re-transmissions should be.

Business Errors

Business errors are those closely related to the content of the business information conveyed by an IOTP message. For example, an Authentication Status Message may be used for an authentication failure which is due to the invalid password provided by the Authenticatee, or a Payment Response Message may indicate that there are insufficient funds available in the Consumer's account. A business error is specific to a particular trading procedure, e.g., the above "invalid password" indication is normally an authentication error associated with an authentication exchange, while the "insufficient funds" indication is considered a payment error and always occurs in a payment exchange. As mentioned in Section 3.2.3, the concrete trading information is contained in the attributes and XML elements of a trading component. Therefore, all business errors are block level errors.

Information for reporting a business error can be encoded in a *Status Component* of a response block (e.g., an *Authentication Status Block* or a *Payment Response Block*) in the corresponding response message (e.g., an *Authentication Status Message* or a *Payment Response Message*). Section 7.16 (Status Component) of RFC 2801 [36] defines a Status Component. It has a *ProcessState* attribute indicating whether a trading process has completed successfully or failed, and a *CompletionCode* attribute used to report the business error if the trading process has failed. Also, we recall the definition of a *Cancel Message* at the beginning of Section 3.2.6, where the Status Component is used to form a *Cancel Block* in the Cancel Message and the CompletionCode attribute of the Status Component provides the reason for cancelling the transaction. All valid values for the CompletionCode attribute are further divided into six groups, each related to a particular trading process. For example, the group of *Payment Completion Code* values are used in a payment exchange, such as the value *InsuffFunds* indicating insufficient funds available.

Handling business errors requires user interaction. For example, if there are insufficient funds available, the payment cannot be processed until the Consumer solves the problem by, e.g., selecting another account that has enough money for the payment. Therefore, business error information may have to be presented to the (human) users so that they can decide, e.g., whether to retry the payment or to cancel the transaction. IOTP is intended to support the recovery of business errors associated with payment and authentication, by allowing the trading role user to retry the payment or authentication upon an error indication. However, definition of the corresponding error recovery procedures is not given in [36].

3.3 Scope of Baseline IOTP

We have introduced the basic operation of IOTP version 1.0. RFC 2801 [36] calls IOTP version 1.0 *Baseline IOTP*. This forms a baseline specification for developing an extended version, known as IOTP version 2.0, the requirements of which are currently given in [75]. Below, we list a number of issues that are beyond the scope of Baseline IOTP. Part of them are included in [75].

- No *Offer Request Message* is defined. An Offer Request Message is expected to convey the information used by the Consumer trading role to initiate a Payment-related transaction, e.g., information on what is being purchased. Such information is currently transferred via an HTTP request, as specified in the IOTP HTTP Supplement in RFC 2935 [77].

- IOTP does not contain its own payment protocol but aims to be applicable to any existing electronic payment scheme. When a payment exchange occurs, IOTP encapsulates an existing payment protocol (e.g., SET) using *Payment Scheme Components*. Any payment scheme-specific operations are contained in a set of payment scheme supplements to IOTP, e.g., the SET supplement for IOTP in [99].

- IOTP specifies the encoding of business error information in messages, but does not define the procedures for business error recovery. For example, the Authenticator and Authenticatee stop any further actions upon sending or receiving an Authentication Status Message indicating an authentication error. Also, a Cancel Message is used instead of a Payment Response Message for reporting a payment error, according to the message processing guidelines of a Payment document exchange in RFC 2801 (page 205, [36]).

- There exist various ways of trading over the Internet (see Section 4.1.2). However, IOTP currently supports only a limited number of them via a set of Payment-related transactions. For example, purchasing online goods may occur in different ways where the payment can be made before, after, or during the delivery [56]. The Baseline IOTP Purchase transaction does not support purchasing with payment after delivery.

- Communication between non-Consumer trading roles is currently not supported by IOTP. As previously mentioned, the Consumer trading role is realised by an IOTP client, and the non-Consumer roles are realised by IOTP servers. Apart from the client-to-server communication between a Consumer and a non-Consumer, any two non-Consumer trading roles also need to be coordinated via server-to-server communications. For example, during a Purchase transaction, the Payment Handler may *only* put through a credit card charge *after* the purchased goods are delivered by the Delivery Handler [37]. IOTP currently assumes such server-to-server communication will be handled by "back-end" systems which are beyond the scope of Baseline IOTP.

- The Merchant Customer Care Provider is so far not specified in any IOTP transaction.

Chapter 4

Initial Assessment of the Internet Open Trading Protocol

RFC 2801 [36] contains an informal narrative description of IOTP. The informal nature of the specification and its complexity make it difficult to maintain consistency and clarity. We believe there are several areas in which IOTP can be improved. In this chapter, we present our initial assessment of IOTP focusing on the basic concepts of IOTP described in Chapter 3. Section 4.1 proposes a simpler conceptual framework for IOTP and Section 4.2 discusses the components of a protocol architecture for IOTP. In Section 4.3, we conclude this chapter with comments and suggested changes to the structure of the IOTP specification given in the RFC.

4.1 Discussion of the IOTP Conceptual Framework

We consider a simpler conceptual framework for IOTP, which helps to clarify the protocol. With this simplified conceptual framework, further investigation of IOTP using formal specification and verification techniques, is the main task of this thesis (in Chapters 7 to 12).

In our experience, the introduction of two basic IOTP building blocks, namely, *trading exchanges* and *document exchanges*, is not helpful nor necessary and can easily be confusing. We therefore attempt to clarify the relationship between these two concepts (see Figure 3.6), something that is only very briefly discussed in RFC 2801 [36]. Further analysis of the two concepts based on a detailed understanding of IOTP transaction mechanisms has shown that there is no need for both trading exchanges and document exchanges. This is discussed in Section 4.1.1 where we advocate the use of document exchanges only. To further demonstrate the merits of our solution, in Section 4.1.2 we focus on the construction of five different ways of carrying out a Purchase transaction based on the set of document exchanges. The last issue we discuss concerns *Authentication* and its implementation as both a document exchange and a transaction in IOTP.

4.1.1 Trading and Document Exchanges

As discussed in Sections 3.2.4 and 3.2.5, IOTP defines two conceptual building blocks called trading exchanges and document exchanges. The relationship between these two sets of exchanges has been given in Section 3.2.5, from which we can observe three differences:

1. The protocol data units (PDUs, see Section 5.2.2) used to transfer information between trading roles are defined at different levels of granularity. "Trading exchanges

consists of trading components that are transmitted between the various trading roles" (page 18, [36]). This indicates that *trading components* are used as PDUs in trading exchanges. Whereas in document exchanges, the PDUs are a set of *IOTP messages* comprising a number of trading blocks that are constructed from trading components. For example, a TPO Message used in an Offer document exchange is composed of a Transaction Reference Block and a TPO Block. The TPO Block consists of a Protocol Options Component, a Brand List Component, and several Organization Components. In contrast, the Brand List Component is the only trading component transferred from the Merchant to Consumer at the beginning of a Payment trading exchange.

2. Different numbers of trading roles may be involved. Document exchanges always occur between the same *two* trading roles, whereas trading exchanges may involve *three* trading roles. For example, a Payment document exchange always occurs between the Consumer and Payment Handler, whereas a Payment trading exchange involves not only the Consumer and Payment Handler but also the Merchant.

3. A trading exchange involving three trading roles has more complexity than the corresponding document exchange. For example, a Payment trading exchange is more complicated than a Payment document exchange in that it should consider the communication between three trading roles. Furthermore, the information transferred within the different trading exchanges could overlap and thereby cause redundancy when combining trading exchanges into transactions. We shall discuss it shortly.

Regarding the granularity of the PDUs used in the two sets of exchanges, it would seem natural if a document exchange is defined as a combination of one or more trading exchanges, since IOTP messages are defined as being constructed from trading components. However, this does not hold in that a trading exchange may involve three trading roles while a document exchange always involves only two of them.

Now, we discuss what trading exchanges and document exchanges are used for. In IOTP, each Authentication or Payment-related transaction is constructed as a direct combination of document exchanges. The set of trading exchanges are also meant to be composed to form a transaction. This is however conducted in a rather indirect procedure, where the trading exchanges are required to be converted to their corresponding document exchanges by combining trading components into IOTP messages (see Section 3.2.5). From this point of view, both sets of exchanges are used to construct transactions, and the set of document exchanges evolve from trading exchanges. The purpose for doing so is to minimize the number of PDUs that are needed to be exchanged over the Internet, as stated in the RFC (see page 185 of [36]).

For example, in a Payment trading exchange, the Consumer has to communicate with the Merchant to get the available payment methods before initiating the payment exchange with the Payment Handler, and similarly, in a Delivery trading exchange, the Consumer needs to communicate with the Merchant to get the delivery instructions before requesting the delivery of goods from the Delivery Handler. To form a Purchase transaction, the three trading exchanges: Offer, Payment and Delivery, will be converted to the three document exchanges with the same names, where the Consumer communicates with the Merchant to get all the required information in the Offer document exchange only.

Based on the above, we start to consider whether both sets of exchanges are needed. If not, which one should be kept and which one should be deleted and why? The answer can be obtained from the following assessment of trading exchanges and document exchanges.

1. Interoperability is a key issue when developing an international standard such as an IETF RFC. A major concern is the interoperability between implementations that are developed with trading exchanges and those using document exchanges. As the two sets of exchanges use PDUs at different levels of granularity, it can be inferred that the corresponding transactions constructed from each of them will not be able to work together.

2. There is no need to define two conceptual build blocks (i.e., trading exchanges and document exchanges) that are essentially at the same level of detail. Moreover, the use of both concepts makes it difficult to understand the basic transaction mechanisms of IOTP.

3. Trading exchanges are not designed to be modular, as they are not clearly separated. For example, the Order and Delivery Components transferred during the first part of a Delivery trading exchange overlaps with those transferred at the end of an Offer trading exchange. Hence, composing trading exchanges to obtain transactions will lead to the redundant exchange of information, which is inefficient and conceptually inelegant.

4. The simplicity of a document exchange, where only two trading roles are involved, is conceptually much more elegant than a trading exchange that may involve three trading roles. Also, the use of one level of detail for message exchange (i.e., IOTP message) in both document exchanges and the transactions being constructed, is to be preferred to the approach applied in trading exchanges where trading components are used.

Accordingly, we believe that the concept of document exchange is sufficient. We therefore propose that the set of document exchanges be the only conceptual build blocks for transactions, which suffices in that it can implement all Authentication and Payment-related transactions defined by IOTP. The concept of trading exchange can be deleted.

Finally, we consider the term *document exchange* to be too generic for the message exchanges used by IOTP for trading. Hence, we suggest the more precise term, *trading-message exchange* (or *trading-data exchange*), to replace *document exchange* in future versions of IOTP [75].

4.1.2 Trading Transaction and Transaction Mode

The core of IOTP is a set of Payment-related transactions that capture common b-c trading activities. In the following, we will refer to these as *trading transactions*. As previously mentioned, all trading transactions defined by IOTP can be expressed as combinations of document exchanges. Furthermore, by combining document exchanges in different ways, we can obtain different ways of carrying out a trading transaction. We will refer to these as *transaction modes*. For example, a Purchase transaction may occur in five different modes [56], given that payment and delivery involved in the transaction can take place at different times.

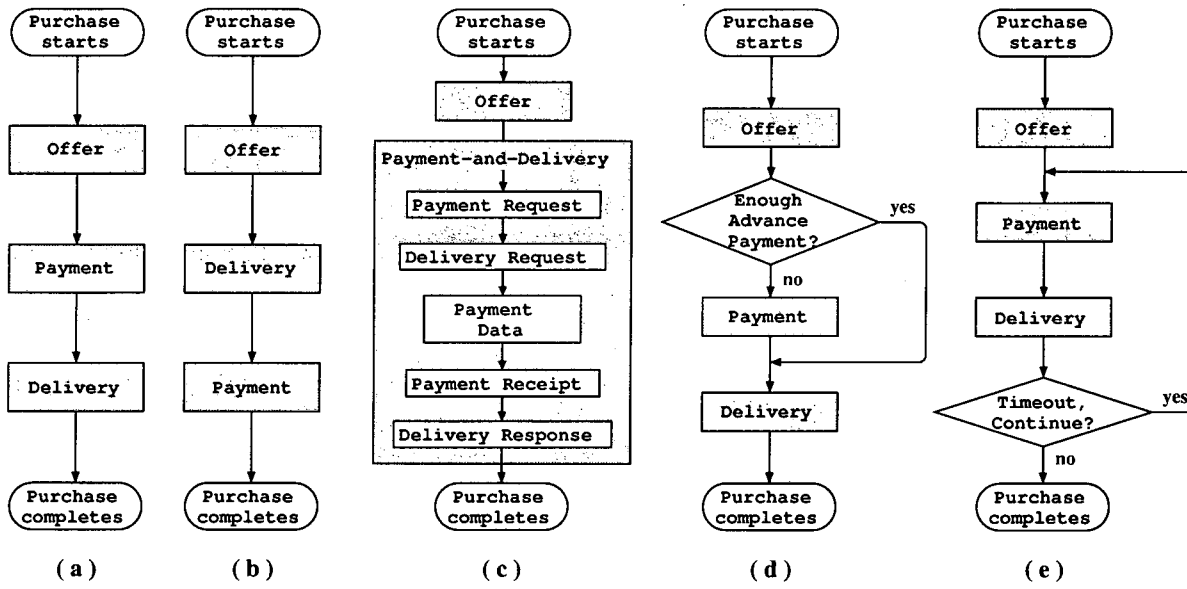


Figure 4.1: Five valid transaction modes of Purchase with (a) payment before delivery; (b) payment after delivery; (c) payment during delivery; (d) payment in advance; (e) payment per unit of time.

The Purchase transaction for each of these five modes can be constructed by using document exchanges. Figure 4.1 depicts the five Purchase transaction modes using flow charts.

In Figure 4.1 (a), a Purchase with *payment before delivery* is the case where the Consumer cannot initiate the delivery until a payment has been made. From the Merchant's point of view, this minimizes the chance of a Consumer acquiring the goods without paying for them. In contrast, the Consumer is likely to take the risk of paying for the goods without receiving them if the Merchant is well known. Purchasing with *payment after delivery* in Figure 4.1 (b), may be the approach the Consumer would prefer since payment is made on receipt of the goods. In Figure 4.1 (c), purchasing with *payment during delivery* is the situation where payment occurs in parallel with delivery. It can be implemented using a Payment-and-Delivery document exchange defined by IOTP. In this case, the Consumer need not wait for payment to complete before delivery commences, but has to wait for payment to complete as indicated by a payment receipt before the delivery of goods or services can occur. The fourth instance is purchasing with *payment in advance* as Figure 4.1 (d) shows. In this case, the Merchant asks for an advance payment, then, whenever the cost of the goods or services being purchased exceed the total amount paid, the Merchant asks for another advance payment. An example of this is prepaid phone cards. Finally, Figure 4.1 (e) illustrates *payment per unit of time* where the Consumer pays a fixed amount every so often, e.g., to take part in an online game with payment every five minutes, instead of paying just once.

According to the scope of Baseline IOTP (see Section 3.3), only two of these five Purchase transaction modes are currently supported by IOTP. They are purchasing with payment before delivery (Figure 4.1 (a)) and purchasing with payment during delivery (Figure 4.1 (c)). Nevertheless, as depicted in Figure 4.1 (b), (d) and (e), a Purchase transaction for each of the other three modes can also be expressed based on the set of document exchanges.

From the above, we can see that it is not difficult to extend Baseline IOTP to support a trading transaction with more transaction modes in the future, by using document exchanges. This illustrates that the concept of document exchanges is adequate to enhance a trading transaction with transaction modes that are beyond the scope of Baseline IOTP.

4.1.3 Authentication Document Exchange and Transaction

Authentication may occur during a Payment-related transaction. Below we give three examples of authentication within the Purchase transaction shown in Figure 3.2. The first is when the Merchant wants to authenticate the Consumer at the beginning of the transaction, or more specifically, after the Offer Request Message has been received (event 1) but before the TPO Message is sent (event 2). The second is when the Consumer wants to authenticate the Payment Handler after the Offer exchange has completed (event 4) but before the Payment exchange starts (event 5). The last one is when the Consumer wants to authenticate the Delivery Handler after the Payment exchange (event 7) but before the Delivery exchange starts (event 8).

IOTP defines both an *Authentication document exchange* and an *Authentication transaction*. An Authentication document exchange is used directly to implement an Authentication transaction which is carried out as a stand-alone transaction. When authentication is required during a Payment-related transaction, either an Authentication document exchange or an Authentication transaction can be used. There are two distinctions that can be made between conducting an Authentication document exchange and conducting an Authentication transaction within a Payment-related transaction.

1. As mentioned in Section 3.2.3, each IOTP transaction is uniquely identified by a *Transaction Identifier* (that is indicated by the *IotpTransId* attribute of the *Transaction ID Component*). All the document exchanges being carried out within a trading transaction have the same Transaction Identifier. Accordingly, conducting an Authentication document exchange within a Payment-related transaction does not require a new Transaction Identifier. In contrast, conducting an Authentication transaction does result in the initiation of a new transaction with a new Transaction Identifier.

2. An Authentication transaction can be carried out at any time when authentication is required in a Payment-related transaction, whereas it is only valid to conduct an Authentication document exchange for the Merchant authenticating the Consumer at the beginning of a Payment-related transaction. RFC 2801 specifies this restriction without explanation (see Section 9.1.12 (Valid Combinations of Document Exchanges) and Section 9.1.13 (Combining Authentication Transactions with other Transactions) of [36]).

However, the specification of the two concepts (i.e., Authentication document exchange and Authentication transaction) unnecessarily complicates implementing authentication within a Payment-related transaction. Conducting an Authentication transaction during a Payment-related transaction introduces the complexity of relating the Authentication transaction to the original transaction (by using the *Related To Component*). Also, the reason for the restriction on conducting an Authentication document exchange within a Payment-related transaction is not clear.

Therefore, we suggest the following modification for Authentication. Firstly, we remove the restriction that an Authentication document exchange can only occur at the beginning of a Payment-related transaction, allowing an Authentication document exchange to occur at any time when authentication is required. Secondly, we add two restrictions. One is that an Authentication transaction always occurs as a stand-alone transaction. The other is that every

authentication required within a Payment-related transaction is conducted as an Authentication document exchange only. We believe that this approach is conceptually cleaner and will lead to simpler and hence more maintainable implementations.

4.2 IOTP Architecture

IOTP currently lacks a specification of its protocol architecture, i.e., the environment in which it operates. We have presented our initial ideas on a layered protocol hierarchy for IOTP in [179] and have recently refined this architecture. However, defining a sound architecture for IOTP is still one of the open issues in developing IOTP and requires a broad range of background knowledge.

In summary, the IOTP architecture comprises *four* major components. At the top of the architecture are the applications that will use IOTP (*IOTP Applications*). This is followed by IOTP's core functionality (*IOTP Core*) which uses a range of payment systems (*IOTP-encapsulated Payment Systems*). These mechanisms operate over a communications infrastructure (*IOTP Transport Mechanisms*) which interconnects the distributed trading roles. We provide more details on these components in the following paragraphs.

IOTP Applications. These comprise mainly consumer-oriented commercial services, such as home shopping, remote banking and online retailing. A major design goal of IOTP is to establish an interoperable e-commerce environment suitable for users to build applications and services that can reach a wide variety of different e-commerce platforms.

IOTP Core. This comprises all IOTP mechanisms, such as its conceptual building blocks and transaction operations, and is implemented by each of the five trading role entities. The core service provided by IOTP is currently the set of common trading transactions, which are Purchase, Refund, Deposit, Withdrawal, Value Exchange and Authentication. In addition, there are also two infrastructure transactions: Inquiry and Ping.

IOTP-encapsulated Payment Systems. A broad range of payment services can be provided by the various payment protocols. IOTP is designed to encapsulate various payment systems. To enable the interoperability between IOTP and the payment systems to be encapsulated, a common interface called the *Payment Application Programming Interface for IOTP* (IOTP Payment API) is specified in [99]. However, an architecture of how the existing payment systems is embedded within IOTP has not yet been devised.

IOTP Transport Mechanisms. This provides the transport service over which IOTP (trading role) entities communicate. It could be the Internet's Transmission Control Protocol (TCP) [193], and an application protocol such as the HyperText Transfer Protocol (HTTP) [79] or the Simple Mail Transfer Protocol (SMTP) [192]. It might also be the Wireless Application Protocol (WAP) [110] to support mobile commerce [232] applications. IOTP is currently designed to operate over HTTP. RFC 2935 [77] specifies the mapping from IOTP to HTTP that allows a trading role to be embedded in a HTTP client (for a Consumer) or a HTTP server (for a non-Consumer trading role). The HTTP client performs full *HTTP POST requests* to send IOTP messages from the Consumer to the peer non-Consumer trading role.

4.3 Re-structuring the IOTP Specification

In Section 3.1, we have provided an overview of the structure of RFC 2801 [36], from which it follows that the overall specification of IOTP is not well organised. Below, we present our comments on the structure of the RFC and also suggest ways to improve the organisation of the IOTP specification. A revised table of contents for the RFC is given in Appendix A.

Definition of Basic Concepts

Specifications of some basic concepts of IOTP are spread over several sections in the RFC. To understand the concept, one needs to read carefully all the sections in which it is addressed and synthesize all related information. For example, to obtain an overall understanding of transaction cancellation in IOTP, one needs to read all of the following sections of the RFC:

- Section 3.10 (Cancelled Transactions) for cancelling a transaction and dealing with a cancelled transaction;
- Section 4.5.3 (Cancelling a Transaction) for a server role to cancel a transaction;
- Section 4.6.3 (Cancelling a Transaction) for a client role to cancel a transaction;
- Section 7.16 (Status Component) for reasons to cancel a transaction;
- Section 8.18 (Cancel Block) for definition of the Cancel Block; and
- Section 9 (Internet Open Trading Protocol Transactions) for how transaction cancellation is applied in each IOTP transaction.

Accordingly, we suggest to group specifications related to one concept into one section or subsection, or if this is not possible, provide references to where these related specifications reside in the RFC.

Specification of Protocol Operations

Sections concerning major operations of the protocol, such as basic transaction procedure, error handling and transaction cancellation, are not organised in an logical order in the RFC. Here are two typical examples.

- Section 9 specifies IOTP transactions without taking into account any errors that may occur during the transaction. However, this comes after the definition of error handling in Section 4 (IOTP Error Handling) in the RFC.
- Section 3 (Protocol Structure) is mainly concerned with the definition of IOTP messages. However, Section 3.10 specifies the procedure of transaction cancellation.

Accordingly, we suggest to: 1) move the definition of transaction cancellation from Section 3 and the related descriptions in the specification of IOTP transactions from Section 9, into a new section named "Transaction Cancellation"; and 2) move the rest of Section 9 (for basic transaction procedures) before Section 4 (for error handling) and the above newly constructed section (for transaction cancellation). Also, we suggest the more precise title, "IOTP Messages", to replace the current title of Section 3, "Protocol Structure", in the RFC.

Proposed Conceptual Framework and Protocol Architecture

The re-structuring of the IOTP specification in the RFC needs to take into account our proposed simpler conceptual framework in Section 4.1 and the IOTP architecture presented in Section 4.2. This involves:

- removal of Section 2.2 (Trading Exchange) from the RFC based on our discussion in Section 4.1.1;
- inclusion of a section on IOTP architecture in Section 2 (Introduction) in the RFC;
- a much earlier introduction of the document exchange material (renamed as trading-message exchange) in the RFC; and
- revision of Section 9.1 (Authentication and Payment-related IOTP transactions) in the RFC to simplify the authentication procedures as proposed in Section 4.1.3.

Chapter 5

Protocol Engineering

Communication protocols are the rules that define the interaction between communicating entities residing in distributed systems. E-commerce protocols, such as electronic trading and payment protocols, concern business interactions between distributed trading parties, and are therefore a subset of communication protocols. Protocol engineering [191] is a well-established systems engineering approach for the rigorous design and maintenance of communication protocols using formal methods [51].

This chapter describes a general protocol engineering methodology [21] and outlines the approach we have used in verifying IOTP. Section 5.1 introduces a layered architecture as the basis for designing complex communication systems. A set of essential concepts of the layering approach is described in Section 5.2, with a focus on service and protocol definitions. Section 5.3 outlines the general protocol engineering process and points out the major activities in the process. Finally, Section 5.4 details the protocol engineering process and activities used to verify that the protocol does provide its required service. These are illustrated with the approach taken to verify IOTP.

It should be mentioned that this section provides only a brief introduction to protocol engineering. For further information the reader is referred to survey papers [21, 154, 191], articles in special issues of journals [2, 155, 202] and proceedings of conferences [1].

5.1 Layered Protocol Architectures

Modern communication systems are designed in a highly structured way to cope with the complexity of advanced information services. A basic structuring technique applied to complex systems is known as *layering*. According to [104, 221], the layering approach allows the whole system to be partitioned into smaller sub-systems that are more comprehensible and subject to independent implementation and maintenance. Following the principles of layering, a complex communication system can be logically decomposed into levels of abstraction called *layers*. Each layer uses the service provided by the underlying layers to perform functions, and these functions further provide a service to the next upper layer. The functions of a layer are defined by a protocol (or a class of protocols). As consequence, a *layered architecture* is created.

The Open Systems Interconnection (OSI) Reference Model [115] is a well-known seven-layer architecture, developed jointly by the International Organization for Standardization (ISO) and the Telecommunication Standardization Sector of the International Telecommunication Union (ITU-T). The main contributions of OSI are not only the definition of a common framework

for intersystem communication, but also the recommendations regarding terminologies, conventions and concepts to be used in constructing a layered architecture. We briefly present the OSI Reference Model in Section 5.1.1 and describe the essential concepts of layering in Section 5.1.2.

5.1.1 The OSI Reference Model

Figure 5.1 (which is based on Figure 2 and partly on Figure 11 of [115]) illustrates the seven layers of the OSI Reference Model above the physical medium. Starting from the bottom of the hierarchy, the *physical layer* deals with transmitting raw bits over a physical communication medium. The *data link layer* transmits data frames across a single link and provides error recovery mechanisms. The *network layer* controls the routing of packets and prevents congestion in the network. The *transport layer* provides multiplexing services and ensures end-to-end delivery of packets from source to destination. The *session layer* establishes sessions between systems for ordinary data transport as well as enhanced services including dialogue and synchronization control. The *presentation layer* concerns the syntax and semantics of the information transmitted, such as the encoding of data in a standard format. Finally, at the top of the hierarchy, the *application layer* directly provides services to end users, e.g., file transfer between terminals.

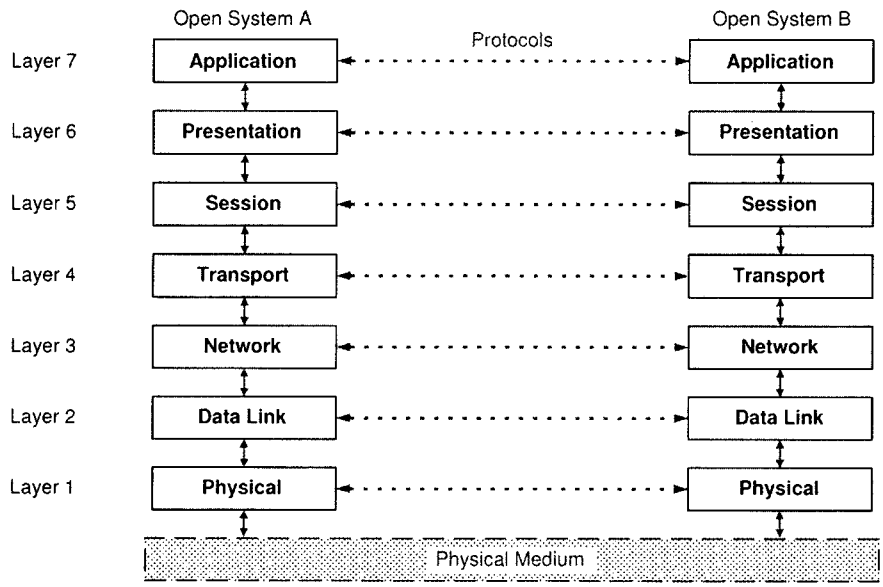


Figure 5.1: The OSI Reference Model.

Referring to Figure 5.1, the flow of information between two communicating systems moves down the hierarchy at one end, uses the physical transmission medium, and then moves upwards to reach its destination at the other end. Through the services provided by the layers beneath, each system can consider its communication with the other system in the same layer as being horizontal. In Figure 5.1, virtual communication via protocols is shown by dotted lines and physical communication by solid lines.

5.1.2 Concepts of Layering

Figure 5.2 (which is based partly on Figures 5 and 7 of [115]) shows in general one layer (represented by the (N)-layer) and its interaction with the layers immediately above and below (respectively represented by the (N+1)-layer and the (N-1)-layer). In a layered architecture, each layer comprises a collection of *entities*, which are considered *peer entities* since they reside in the same layer. The *(N)-service* refers to the capabilities of the (N)-layer (and all layers below) provided to the (N+1)-layer, which is available at a set of *(N)-service-access-points* (or (N)-SAPs for short) on the boundary between the (N)-layer and (N+1)-layer. Entities in the (N)-layer are denoted *(N)-entities* and those in the (N+1)-layer *(N+1)-entities*. Each (N)-SAP is where an (N)-entity and an (N+1)-entity interact. The *(N)-protocol* defines the rules and formats the (N)-entities should obey when they communicate using the (N-1)-service.

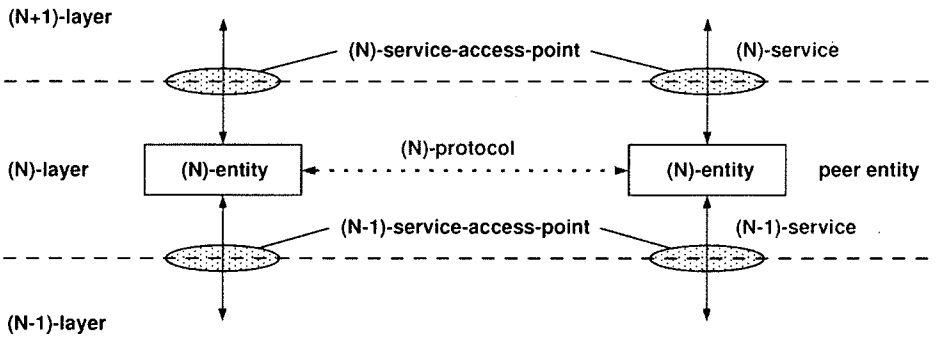


Figure 5.2: Structure of (N)-layer in a layered architecture.

5.2 Service and Protocol Specifications

Service and protocol are two crucial concepts in the design of layered communication architectures. Services represent the logical interactions between adjacent layers, whereas protocols represent the operations performed inside layers. Accordingly, the service specification and the protocol specification describe the system behaviour at two different levels of abstraction.

5.2.1 Service Specification

The conventions for defining OSI services are prescribed in [114]. Figure 5.3 (which is based partly on Figure 1 of [114]) illustrates an abstract model of the (N)-service, where *(N)-service-users* are entities in the (N+1)-layer and *(N)-service-provider* represents all entities in the (N)-layer and all the layers below that provide the required capabilities to (N)-service-users. The interactions between user and provider are described using *service primitives* that are either *submitt*ed (from user to provider) or *deliver*ed (from provider to user) at the corresponding SAP. The service definition specifies the external behaviour of a service provider (as seen by its users) in terms of the possible sequencing of service primitives at all SAPs. Once the (N)-service is defined, the *(N)-service-users* can make use of it, without any knowledge of the details of the (N)-service-provider.

Service primitive. Service primitives are abstract representations of the logical exchange of information between a service provider and its users. Each service primitive is given by its name and the list of its parameters. The name of a service primitive is structured as

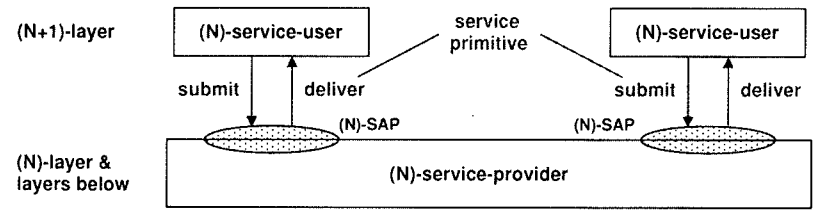


Figure 5.3: (N)-service model in a layered architecture.

<service name>-<primitive name>.<primitive type>. The service name indicates the overall service offered by the service provider, e.g., *IOTS* stands for the Internet Open Trading Service provided by IOTP. The primitive name identifies the service facility the primitive belongs to, e.g. *Delivery* can be used for the delivery of the goods in a Purchase transaction and *Cancel* for the cancellation of an ongoing transaction. There are four primitive types named *request*, *indication*, *response* and *confirmation*. Table 5.1 lists the definition and abbreviation (abbr.) of the four primitive types. In addition, the primitive parameters convey user data or control information, and may be given as a tuple following the name of the service primitive when necessary.

Type	Abbr.	Definition
request	req	submitted by a user to initiate the service facility identified by the primitive name.
indication	ind	delivered to the peer user, 1) conveying information from the request primitive; or 2) indicating events initiated by the service provider.
response	res	submitted by the peer user to respond to the indication primitive.
confirm	cnf	delivered to the initiating user, conveying information from the response primitive.

Table 5.1: Four service primitive types.

In [109, 116], service primitives are defined using *service parameter tables*. A service parameter table indicates how the associated primitive parameters are used with the different primitive types. For example, an IOTS primitive (denoted *IOTS-primitive* in general) with four types is defined in Table 5.2. Below the service primitive table is the notation for parameter usage in each primitive type. With this notation, Parameter 1 MUST be present in *IOTS-primitive.req* and the corresponding *IOTS-primitive.ind*. *Parameter 2* MAY be specified in *IOTS-primitive.res* and in that case it MUST be present and also have the equivalent value in the corresponding *IOTS-primitive.cnf*; otherwise, it will not be used.

Primitive / ***Parameter***	**IOTS-primitive**			
	req	***ind***	***res***	***cnf***
Parameter 1	*M*	*M*(=)		
Parameter 2			*U*	*C*(=)

M	The parameter is mandatory, i.e., it MUST be present.
U	The use of the parameter is a service user option, i.e., it MAY be omitted.
C	The parameter is conditional, depending on presence of other parameters.
(=)	The value of the parameter is identical to the corresponding parameter in the preceding corresponding service primitive.
blank	The parameter is not used in the type of the service primitive being defined.

Table 5.2: Service parameter table with parameter usage notation.

Primitive sequences. A confirmed service is represented by the sequence of four primitive types [*request, indication, response, confirm*] that are associated with the same primitive name, and a non-confirmed service may only use the sequence [*request, indication*]. There are also special circumstances when a service is initiated by the provider instead of the user. In this case, only an indication type is present in the corresponding primitive sequence.

Time sequence diagrams (TSDs) may be used to illustrate primitive sequences and how they are related in time. Conventions for TSDs are defined in [114]. Using an example from IOTS, Figure 5.4 (a) shows a TSD for a confirmed service (denoted by *IOTS-primitive1* in general), and Figure 5.4 (a) shows a TSD for a service initiated by the provider (*IOTS-primitive2*). In each TSD, the vertical line represents the interface between the IOTS-User and the IOTS-Provider, and also indicates the passage of time increasing from top to bottom. Submission or delivery of an IOTS primitive is represented by an arrow that is labelled with the name of that primitive. An arrow towards the vertical line indicates that the primitive is submitted by the user, and an arrow away from the vertical line indicates that the primitive is delivered to the user. In Figure 5.4 (a), a dotted line in the IOTS-Provider is used to correlate the submission of a request or response primitive by one IOTS-User with the delivery of an indication or confirm primitive to the other. In Figure 5.4 (b), a dotted wave between two indication primitives delivered to peer users indicates there is no time relationship between them.

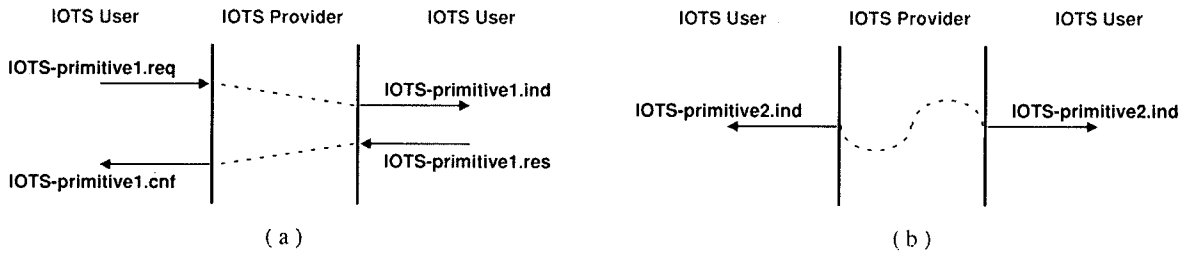


Figure 5.4: Time sequence diagrams for (a) a confirmed service and (b) a provider-initiated service.

Service specification. Based on the above, a *local view* of a service is defined by the valid sequences of primitives that the service-user and the service-provider are allowed to exchange at a single SAP. For example, the sequence [*IOTS-primitive1.req, IOTS-primitive1.cnf*] occurring at one interface (left side) in Figure 5.4 (a) is a valid *local* primitive sequence of IOTS at that interface. A *global view* of the service is the composition of the correlated local views at all SAPs. Therefore, a well-defined service specification comprises [114]:

1. Definition of the abstract service model introduced in Figure 5.3, including the service-users, the service-provider, and the set of service primitives (and primitive parameters);
2. Definition of all valid *local* primitive sequences constituting the local view at each SAP;
3. Definition of all possible *global* primitive sequences (as illustrated in TSDs) that constitute the global view at all SAPs.

5.2.2 Protocol Specification

Figure 5.5 (which is based partly on Figures 4, 5 and 7 in [115]) illustrates an abstract model of the (N)-protocol, where the (N)-service-provider shown in Figure 5.3 is decomposed into the (N)-layer and (N-1)-service-provider. The (N)-entities (referred to as (N)-protocol-entities) can

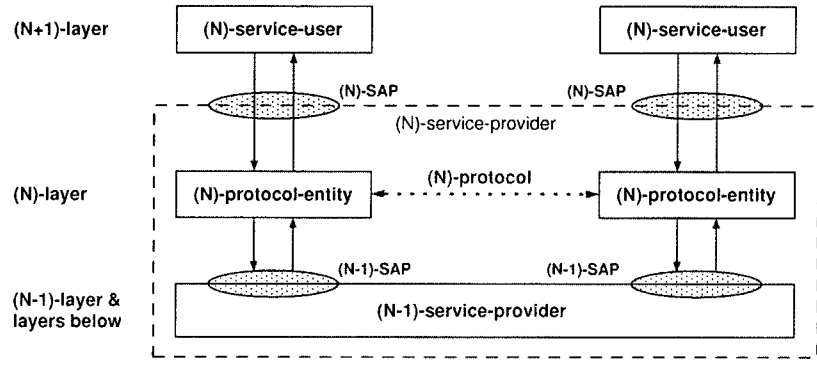


Figure 5.5: (N)-protocol model in a layered architecture.

communicate only by using the (N-1)-service, and their communication behaviour is controlled by the (N)-protocol offering the intended service to the (N+1)-layer (or (N)-service-users).

Protocol data units (PDUs) are used to convey the information between peer protocol entities. Each PDU is encapsulated in a *service data unit* (SDU) that is transferred but not interpreted by the service provider. In Figure 5.5, the (N)-protocol-entities communicate with each other by sending and receiving PDUs (shown by the dotted bi-directional arrow).

The definition of a protocol must specify the encoding of PDUs, the rules for exchanging PDUs, and the use of the underlying medium. A PDU comprises protocol control information and the user data. The rules define the procedures for providing the desired service of the protocol. The underlying medium represents the service provider over which the protocol entities communicate. From the abstract model shown in Figure 5.5, the (N)-protocol can be regarded as the logical implementation of the (N)-service, given that the (N-1)-service is available. Accordingly, the specification of the (N)-protocol can be defined in terms of the (N)-service provided to the (N+1)-layer and the (N-1)-service available from the (N-1)-layer and all layers below. When designing a protocol, one must ensure that the protocol does provide the specified service to its users. Checking that the protocol does provide its required service is known as protocol verification and is part of the overall process of protocol engineering.

5.3 Protocol Engineering Methodology

Protocol engineering is part of systems engineering and concerns the application of a mathematically based methodology to protocol development. Figure 5.6 (based on Figure 1 in [21]) sketches the protocol engineering process, which consists of five steps covering major design and development activities in the protocol life cycle.

5.3.1 Protocol Engineering Process

The *first* step is to gather and document user requirements. The design and implementation of a communication system is driven by the requirements of the users of distributed applications. Rules and procedures for defining user requirements from the domain of software engineering [216] can be applied to this step.

The *second* step is to design a high-level protocol architecture to meet the user requirements. In many cases, the existing architectures or at least their design principles can be re-used, if they satisfy the user requirements that have been identified in the *first* step. For example,

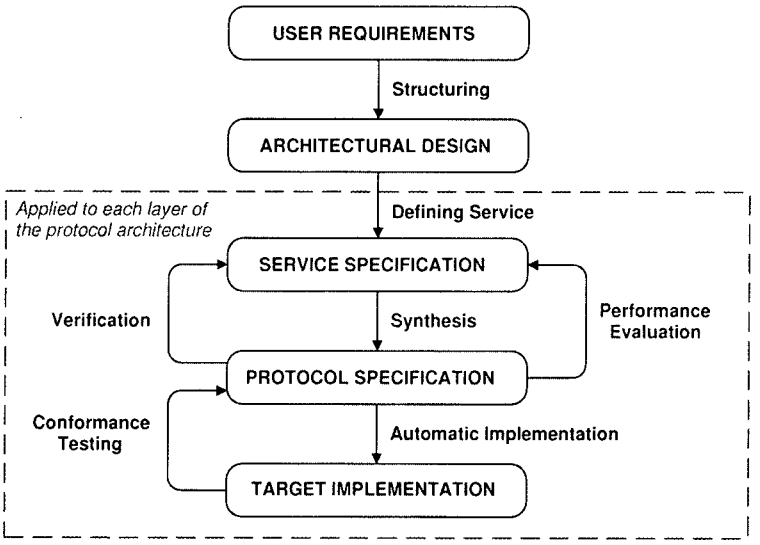


Figure 5.6: Protocol engineering process and activities.

the OSI Reference Model [115] presented in Section 5.1 provides a common framework as well as terms and conventions for architectural design of a protocol hierarchy. Once a layered architecture has been defined, each layer in the protocol hierarchy is then designed separately as defined in the next three steps.

The *third* step is defining the service to be provided at each layer and the *fourth* step is specifying the protocol (or a class of protocols) that will provide the service at that layer. The procedures for defining the service specification and the protocol specification are given in Section 5.2. Both specifications need to be formally defined in protocol engineering.

The *last* step is to produce the target implementation from the service and protocol specifications.

5.3.2 Protocol Engineering Activities

As shown in Figure 5.6, there are five major protocol design activities associated with the last three steps in a protocol engineering process.

In protocol design and development, *protocol synthesis* and *protocol verification* are two inherently different but complementary approaches to ensuring the functional correctness of protocols [253]. Protocol synthesis is the discipline of deriving a conformant protocol specification from the adjacent service specifications. Another [203] surveys different protocol synthesis techniques, and examples of their application can be found in [132, 134, 169]. Protocol verification [104] is the process of proving the protocol specification does realize the service specification, and is usually applied to an already designed but not yet analysed protocol. It also involves verifying general properties (such as absence of deadlocks and livelocks), which are essential to the correctness of protocols. Some literature on protocol verification techniques and their application are [8, 19, 26, 104, 106, 168, 199]. Section 5.4 details a protocol verification procedure.

Apart from investigating the functional behaviour of protocols, *performance evaluation* is necessary in determining whether the protocol meets a set of performance criteria such as throughput, delay and reliability. Analytical, experimental or simulation techniques can be applied to evaluate the performance of a protocol [100, 198].

When implementing protocols, *automatic implementation* intends to complement *conformance testing* to make sure that the implementation does conform to the protocol specification. It is expected that a protocol can be automatically implemented from its protocol specification, once the specification has been verified and its performance evaluated. Conformance testing is the process of checking a protocol implementation against its specification, using test suites that may be generated from the formal protocol specification [131, 171, 240]. With today's techniques, automatic protocol implementation is however still an open research topic.

5.4 Protocol Verification Methodology

For a newly designed protocol, such as the Internet Open Trading Protocol (IOTP), it is highly desirable to verify its functional correctness before implementation, so that the costs of protocol failures can be avoided. This subsection presents a previously developed protocol verification methodology [26]. It is applied to verifying the functional behaviour of IOTP, and is enhanced by a newly added step for validating the correctness of the service specification of IOTP. The initial idea of applying this service validation step to the development of a service specification was suggested by Dr. Lars Michael Kristensen.

Figure 5.7 sketches the protocol verification methodology for IOTP. As introduced in Section 5.3, protocol verification consists of two parts: verification of general properties (such as absence of deadlocks and livelocks); and verification of the protocol against its service specification, to determine whether the protocol has been designed to provide the services required in the service definition. In Figure 5.7, the two dashed boxes set out the procedures corresponding to these two parts.

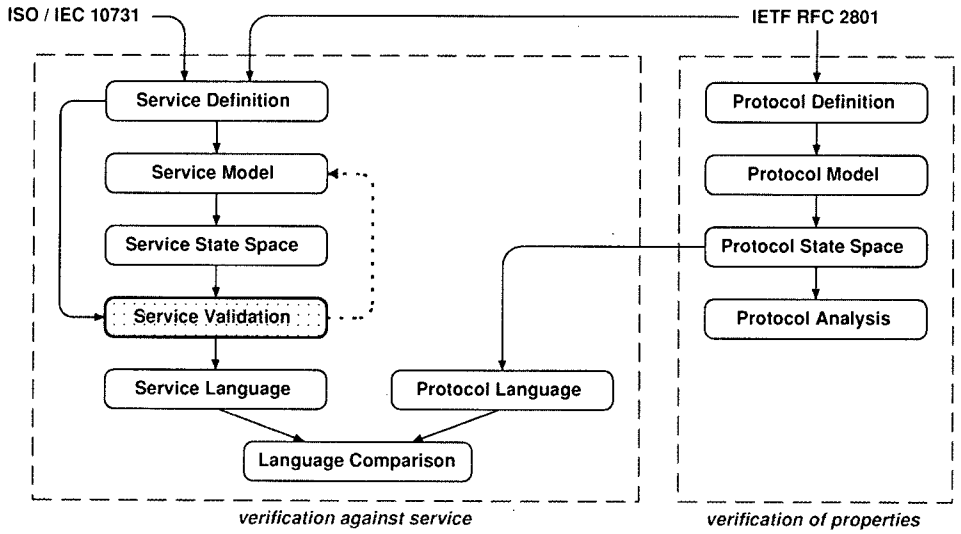


Figure 5.7: Steps of protocol verification methodology.

5.4.1 Verification of Properties

The dashed box on the right of Figure 5.7 shows the procedure for verifying properties of a protocol. It starts with the Protocol Definition, which is often given by an international standard. For example, RFC 2801 [36] provides the definition of IOTP.

The Protocol Definition is used to create the *Protocol Models*. There are two types of Protocol Model: one for specification and one for analysis, which we call the *specification model* and the *analysis model*, respectively. The specification model specifies the complete protocol and can be used to generate the protocol implementation, but often contains information that will increase the complexity of the analysis. For verification purposes, the analysis model is created by employing a set of restrictions, simplifications and abstractions regarding the Protocol Definition. The Protocol Model in Figure 5.7 refers to the analysis model.

The *Protocol State Space* is calculated from the (analysis) Protocol Model. A state space comprises all system states and state transitions. The Protocol State Space can be analysed to check if the protocol satisfies a set of general properties. For example, IOTP's transaction termination property (i.e., whether each IOTP transaction terminates properly) can be investigated from its state space. *Protocol Analysis* may detect errors in the Protocol Model, or more interestingly, reveal deficiencies in the Protocol Definition. If errors are detected in the Protocol Model it is modified to remove them. If deficiencies in the Protocol Definition are revealed, the Protocol Model can be used to incorporate the suggested fixes and then re-analysed to prove that they do remove the undesired behaviour, while not introducing any new problems.

5.4.2 Verification against Service

In Figure 5.7, the dashed box on the left illustrates a first step to verify a protocol against its service specification. As an important first step, we do this by comparing the *Protocol Language* with the *Service Language*. The Protocol Language comprises the sequences of service primitives occurring as a result of the protocol's operation, whereas the Service Language prescribes what these sequences of primitives should be.

At first, the services provided by the protocol needs to be defined, if no Service Definition is available. Following the conventions for the definition of OSI services [114], we define the services provided by IOTP with respect to the existing information given in the RFC [36]. The Service Definition is used to develop the *Service Model* from which the *Service State Space* can be calculated. As mentioned at the beginning of this subsection, the next step, *Service Validation*, is performed to check the correctness of the service specification, i.e., the Service Model. We shall detail the procedure for this step shortly. Once the Service Model has been validated, the Service Language can be generated from the Service State Space, and forms the basis against which the protocol's sequences of service primitives are verified.

In parallel with the service process, we obtain the Protocol Language from the Protocol State Space that has been analysed to see if the protocol satisfies the set of general properties. Comparing the Protocol Language with the Service Language determines whether the protocol implements the possible sequences of service primitives defined by the service. The protocol is a faithful refinement of the service, if the Protocol and Service Languages are *identical* (i.e., they comprise the same set of primitive sequences), or if the Protocol Language is an *acceptable* subset (comprising those primitive sequences that are *sufficient* to implement the service) of the Service Language.

5.4.3 Service Validation

Figure 5.8 shows in detail the Service Validation step (enclosed in the dashed box) where two *Local Service Languages* are compared. The Local Service Language comprises the sequences of service primitives occurring at the *local* interface (service access point) between each service user and the provider.

The Service Definition defines all valid primitive sequences that can be observed at each local interface. We call these sequences the *Defined*-Local Service Language. The Service Model synthesizes the local service languages into a specification of the *global* (distributed) service, which captures the sequencing relationships between the service primitives occurring at different interfaces. The Service State Space (generated from the Service Model) includes the possible sequences of primitive occurrences at all interfaces, thus comprising the *Global Service Language*. We project the Global Service Language onto the set of primitives that can *only* be viewed at the interface under consideration, and thus obtain what we call the *Projected*-Local Service Language at that interface.

The Projected- and Defined-Local Service Languages (at each interface) are compared to see if the two languages are the same. If these two languages are identical at each interface, then the Global Service Language is said to be *consistent* with the Local Service Languages. Any differences detected must be removed to ensure the Global Service Language is consistent with the local service language at each interface. This is achieved by debugging the Service Model or the Service Definition (drawn as a dotted line from Language Comparison to Service Model or Service Definition in Figure 5.8) and iterating the above steps until no difference exists between the Projected- and Defined-Local Service Languages. The Service Language shown in Figure 5.7 refers to the Global Service Language obtained from the validated Service Model after Service Validation.

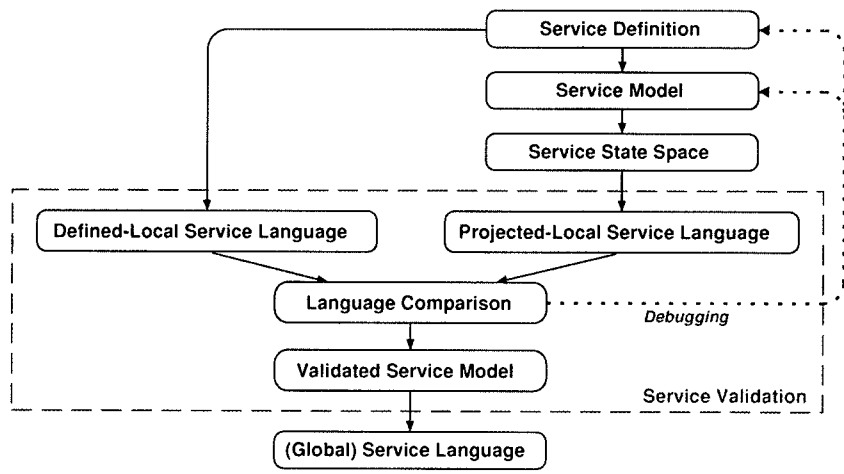


Figure 5.8: Details of service validation step.

When applying the above protocol verification methodology to IOTP, we have used Coloured Petri Nets (CPNs) [123] for modelling and analysing both IOTP and its service. Finite state automata (FSA) [138] are used for language definition and comparison. Introductions to these two formal techniques are given in Chapter 6.

Chapter 6

Formal Techniques and Tools

Formal methods [39,51,238] encompass mathematically based modelling languages, techniques and computer tools for specification, verification, implementation and testing of systems. Protocol engineering, as introduced in Chapter 5, applies formal methods to the design, implementation, testing and maintenance of protocols. At the design stage, using formal methods is essential in that they not only provide unambiguous specifications, but also allow specifications to be formally refined and analysed. Formal modelling languages, such as Petri Nets [190] and their derivatives [123,158,243], can be used to construct models amenable to analysis. Formal analysis techniques, such as reachability analysis [248], invariants analysis [124], language theory [105] and program proofs [38], provide the ability to formally reason about the properties of a protocol being modelled, including verifying the protocol against its service. Formal methods also assist in the other protocol engineering activities described in Chapter 5.

A range of formal methods have been applied to protocol engineering. These include: the Specification and Description Language (SDL) [119, 204], Estelle [34, 112], Language of Temporal Ordering Specification (LOTOS) [31, 113], net theory [22, 32, 71, 197], finite state automata [28,144], temporal logic [50, 91], high-level programming languages [170], set theory (e.g., Z [218]) and process algebras [103,163]. A comparison of these and other formal methods is beyond the scope of this thesis and can be found elsewhere [1,57,220].

For the modelling and analysis of IOTP and its service, we use Coloured Petri Nets [123–125], a derivative of Petri Nets [190]. Five reasons for choosing Coloured Petri Nets are:

1. They are suitable for intuitive modelling of concurrency and non-determinism, and have the ability to specify system concepts at different levels of abstraction (Section 6.1);
2. They support reachability analysis for verifying properties of the protocol (known as the state space method of Coloured Petri Nets, Section 6.2);
3. They have adequate computer tool support (i.e., Design/CPN, Section 6.3);
4. They have been successfully applied to the modelling and analysis of a wide range of communication protocols [24];
5. Our research group [41] has considerable experience in the application of Coloured Petri Nets to the specification and verification of communication protocols [25,88,98,153,235].

For the verification of IOTP against its service, we also apply language analysis based on finite state automata (Sections 6.4 and 6.5).

6.1 Coloured Petri Nets

Petri Nets [167, 189, 190, 197] are a formal method that is appropriate for modelling and analysing distributed systems which exhibit concurrency, synchronization and non-determinism. Coloured Petri Nets (CPNs) [123] are a type of high-level nets [129] that extend the features of basic Petri Nets, and have previously been successfully applied to the modelling and analysis of a wide range of communication protocols [24]. The literature on CPNs is widely available, including introductory publications [122, 142], books [123–125], articles in special issues of journals [126, 127] and in the proceedings of international conferences and workshops [3, 4].

This section provides a tutorial style introduction to CPNs. Section 6.1.1 gives an informal description of the basic concepts and dynamic behaviour of CPNs. The formal definition of CPNs can be found in [121, 123]. For illustration purposes, we take the IOTP Payment document exchange described in Section 6.1.2 as an example, present its CPN model in Section 6.1.3 and examine the dynamic behaviour of the CPN model in Section 6.1.4.

6.1.1 Introduction to CPNs

CPNs [123] are a class of high-level nets that enhance Petri Nets with data types. A CPN is a labelled bipartite graph comprising two types of nodes, *places* (drawn as ellipses) and *transitions* (drawn as rectangles), and directed edges known as *arcs*. A place is typed by a *colour set* and contains collections (*multi-sets*) of data items called *tokens* of the same type as the place. A transition represents an event and may have a *guard* associated with it. The guard is a boolean expression enclosed in square brackets. Arcs connect places to transitions and transitions to places, and have associated inscriptions comprising variables, constants and functions. Variables are typed. Data values belonging to the type of a variable can be assigned (*bound*) to the variable. CPNs use a variant of Standard ML [164, 228] for net inscriptions and declarations of variables and types.

For a transition, its *input places* are places having arcs going to the transition, and *output places* are places having arcs coming from the transition. A transition is *enabled* if: 1) sufficient tokens exist in each input place to match each respective input arc expression when evaluated for a particular binding of its variables; and 2) the transition guard evaluates to true for the binding. If a transition is enabled, it can *occur* (or be *fired*). The *occurrence* of an enabled transition removes the tokens specified by the respective arc expressions from its input places, and deposits tokens specified by the expressions on the output arcs into the connected output places. The state of a CPN is called a *marking*. It consists of tokens distributed on each place of the CPN. Occurrences of transitions represent state changes.

CPNs support hierarchical net construction, which is particularly important when dealing with models of large systems. The basic idea behind hierarchical CPNs is to facilitate the construction of a large model by using a number of small CPNs called *pages*. These pages are then related to each other in a well-defined way. By means of the hierarchical structuring mechanism it is possible to capture different abstraction levels of the modelled system in the same CPN model.

6.1.2 Example: IOTP Payment Exchange

We consider an IOTP Payment document exchange (or Payment exchange for short) introduced in Section 3.2.4. The Payment exchange allows a payment using some payment brand to be made by the Consumer to the Payment Handler. RFC 2801 [36] defines a set of message processing guidelines, which are used as the basis for modelling the Payment exchange.

Revisiting Figure 3.5, the messages used in a *successful* Payment exchange consist of:

- A *Payment Request Message* sent from the Consumer to the Payment Handler to start the payment;
- Optionally, a set of *Payment Exchange Messages* exchanged between the Consumer and Payment Handler to process the payment system-specific data; and
- A *Payment Response Message* (with a successful result) sent to the Consumer from the Payment Handler to complete the payment.

Also, during the Payment exchange, the Consumer and Payment Handler may cancel the payment by sending a *Cancel Message* to the other side (see Section 3.2.6). In this case, the payment should be cancelled at both sides resulting in a *cancelled* Payment exchange.

6.1.3 CPN Model of Payment Exchange

The IOTP Payment exchange is modelled by a hierarchical CPN. Figure 6.1 shows the *hierarchy page* of the CPN model, which provides an overview of the pages (modules) constituting the CPN model and their interrelationship. Each node represents a page labelled with a *page name* and a *page number*. For example, the page node at the top of Figure 6.1 is named PayExchange and has number 2. An arc between two nodes indicates that the destination page is a *subpage* (submodule) of the source page. The *arc expression* (label) specifies the name of the *substitution transition*, which represents the corresponding subpage at the source page.

The PayExchange page is the top level page of the CPN model which provides an abstract view of the IOTP Payment exchange. It has has two subpages: Consumer and PHandler. These two pages model the IOTP's procedures at each of the corresponding trading roles, and are hereafter referred to as *trading role pages*. There is also a page named Declarations, which defines the colour sets, variables, constants and functions used in the net inscription of the CPN model. Below, we describe each of these pages in detail.

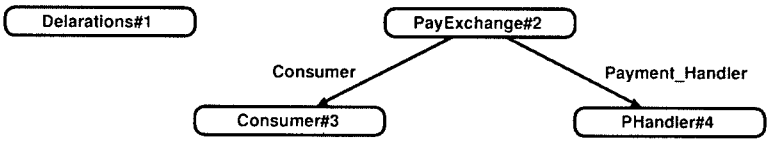


Figure 6.1: Hierarchy page for the CPN model.

Global Declarations

Table 6.1 lists the global declarations for the CPN model. The colour set State (line 1) is an *enumeration type* containing a set of simple values, used to represent the states of the two trading roles (Consumer and Payment Handler) throughout the Payment exchange. Based on the XML definition of IOTP messages (Section 3.2.3), the colour set IotpMsg (line 3) is

defined as a *list* of TradingBlks denoting that each message contains one or more trading blocks. The colour set TradingBlk (line 2) specifies five trading blocks used for IOTP Messages in the Payment exchange. A list of TradingBlks is written in square brackets. For example, a Payment Request Message contains a Transaction Reference Block and a Payment Request Block, and is written as [TransRefBlk,PayReqBlk]. The colour set IotpMQ (line 4), defined as a list of IotpMsgs, is used to represent the message queue between the Consumer and Payment Handler. Finally, the three variables s, m and q (line 5) are defined to represent tokens of the respective colour sets State, IotpMsg and IotpMQ.

```
1  color State = with IDLE | WAIT | PROCESS | COMPLETED | CANCELLED;
2  color TradingBlk = with TransRefBlk | PayReqBlk | PayExchBlk | PayRespBlk | CancelBlk;
3  color IotpMsg = list TradingBlk;
4  color IotpMQ = list IotpMsg;
5  var s: State;    var m: IotpMsg;    var q: IotpMQ;
```

Table 6.1: Global declarations.

The Top Level Page

Figure 6.2 depicts the top level page PayExchange. It has two *substitution transitions* (transitions with an *HS-tag*, where HS stands for Hierarchy and Substitution) named Consumer and Payment_Handler. A substitution transition is a transition created to be linked with another CPN page (subpage). The subpage contains a more precise and detailed description of the activity represented by the substitution transition. For example, the above two substitution transitions are linked to their respective trading role pages Consumer and PHandler.

There are two places in Figure 6.2, all typed by the colour set IotpMQ (positioned next to each place). The two places model the transport medium over which the two trading roles communicate. ConToPH holds the message queue from the Consumer to Payment Handler, and PHToCon is used for the message queue in the opposite direction. Each place is a *socket place* used to link the CPN modules of the substitution transitions with the top level.

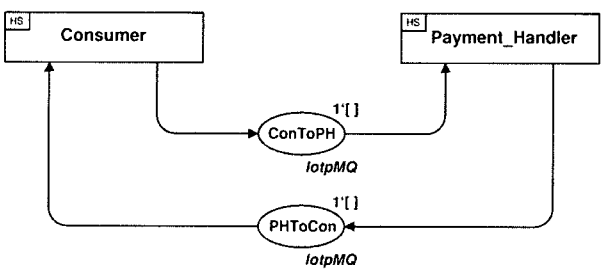
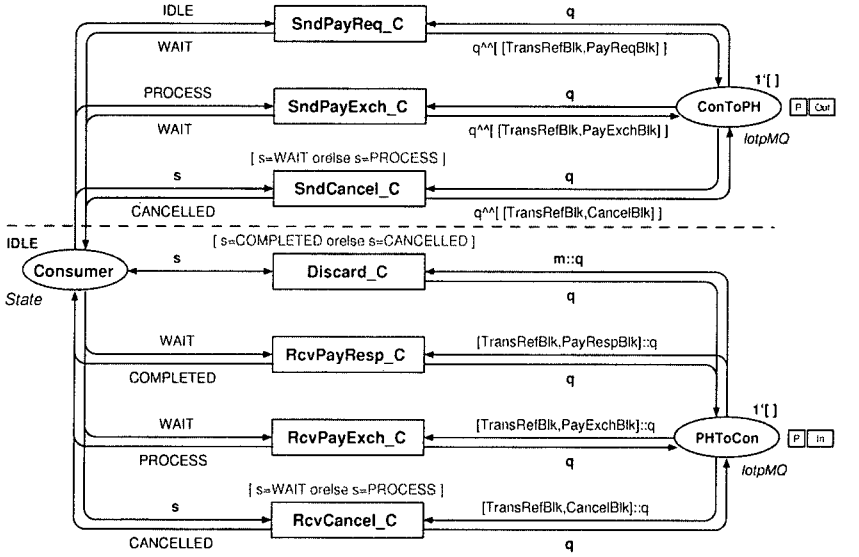


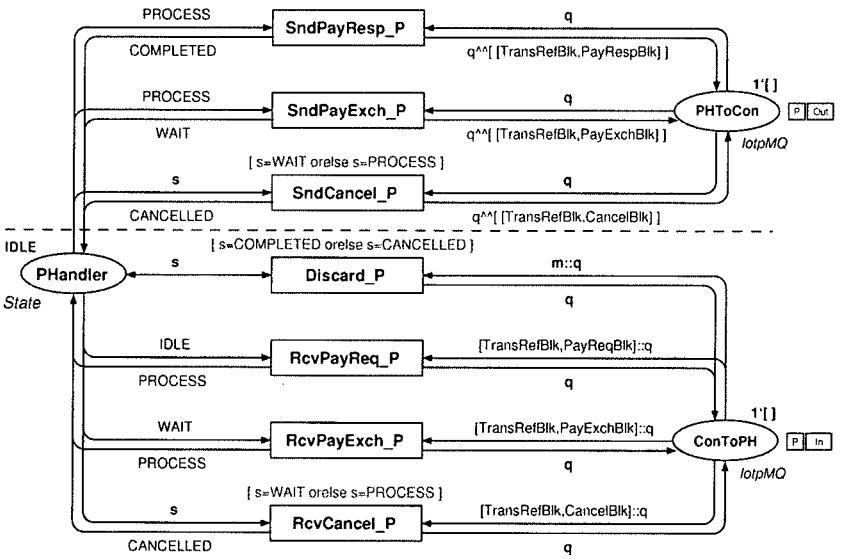
Figure 6.2: Top level page: PayExchange.

Trading Role Pages

Details of the Payment exchange are specified by the Consumer and PHandler pages shown in Figure 6.3. These two pages are associated with the accordingly named substitution transition on page PayExchange. Both trading role pages have a similar structure and comprise two parts. The top part (above the horizontal dashed line) models the sending of IOTP messages by the trading role, and the bottom part models the receipt of IOTP messages from the peer trading role.



(a) Consumer page



(b) PHandler page

Figure 6.3: Two trading role pages.

In Figure 6.3 (a), the place Consumer, typed by the colour set State, models the state of the Consumer. The two places, ConToPH and PHToCon are *port places* as indicated by the attached *Out-tag* and *In-tag*. Port places are used to link a CPN page with the corresponding substitution transition. The two port places ConSnd and ConRcv on page Consumer are linked to the ConToPH and PHToCon socket places in Figure 6.2, respectively. Similarly in Figure 6.3 (b), the place PHandler models the state of the Payment Handler, and the two port places PHSnd and PHRcv are linked to the PHToCon and ConToPH socket places in Figure 6.2.

All transitions except Discard_C and Discard_P in Figure 6.3 model the actions of sending and receiving messages. For example, SndPayReq_C on page Consumer models the sending of a Payment Request Message by the Consumer, and RcvPayReq_P on page PHandler models the receipt of this message by the Payment Handler. Also, the SndCancel_C and RcvCancel_C transitions on page Consumer have a guard, specifying they are only used when the Consumer is in state WAIT or PROCESS (i.e., not IDLE, COMPLETED or CANCELLED). The SndCancel_P and

RcvCancel_P transitions on page PHandler are defined in a similarly way. With a single Payment document exchange, the Consumer and Payment Handler will discard any message (m) being received after the exchange has COMPLETED or has been CANCELLED. This is modelled by the Discard_C and Discard_P transitions in Figure 6.3.

Arcs between the Consumer or PHandler place and transitions model the trading role state changes. Each arc from such a place to a transition has an inscription specifying the required state of the trading role for the transition to occur. The expression on the arc in the opposite direction specifies the state of the trading role after the transition occurs. For example, the occurrence of SndPayReq_C on page Consumer changes the state of the Consumer from IDLE to WAIT. If a Discard transition occurs, it will not have any impact on the state of a trading role, and therefore the trading role will remain in the previous state, i.e., COMPLETED or CANCELLED.

Arcs between the ConToPH or PHToCon place and transitions model how IOTP messages are delivered via the transport medium. Each arc expression has a variable q which represents the current message queue from the Consumer to Payment Handler (in ConToPH) or in the opposite direction (in PHToCon). A message sent by a trading role is modelled as the concatenation of a single token value of that message to the message queue, using the infix append operator ^^ of Standard ML and the list constructor [ ]. For example, the occurrence of SndPayReq_C on page Consumer adds a Payment Request Message ([TransRefBlk,PayReqBlk]) to the end of the queue (q) in ConToPH. A message received by a trading role is modelled by removing the first element from the non-empty message queue, using the infix cons operator ::. For example, when RcvPayReq_P on page PHandler occurs, the first message ([TransRefBlk,PayReqBlk]) in the queue will be removed and the rest of the queue (q) is put back on place ConToPH. Since IOTP is designed to operate over HTTP, SMTP or TCP which all provide reliable delivery of messages (i.e., no loss, no duplicate and no overtaking), the above modelling mechanism is applied to ensure that all IOTP messages are transferred in First-In-First-Out (FIFO) order.

6.1.4 Dynamic Behaviour

We now consider executing the CPN model presented in Section 6.1.3, and illustrate how it captures a Payment exchange. At the beginning, both Consumer and Payment Handler are in their initial state IDLE, as represented by the *initial marking* of the Consumer and PHandler places in Figure 6.3. The other two places ConToPH and PHToCon has an initial marking 1`[ ], indicating no messages are in transit between the two trading roles.

Transition SndPayReq_C on page Consumer is enabled. On its occurrence (with variable q bound to the empty list [ ]), the IDLE token is replaced by a WAIT token in place Consumer, and a [TransRefBlk,PayReqBlk] is appended to the end of the empty queue in place ConToPH. This models the Consumer sending a Payment Request Message to initiate a Payment exchange.

Next, two transitions are enabled. One is SndCancel_C on page Consumer (with variable s bound to WAIT), the other is RcvPayReq_P on page PHandler. The two transitions can occur *concurrently*, i.e., occurrence of one transition does not affect occurrence of the other.

If RcvPayReq_P occurs, the [TransRefBlk,PayReqBlk] is removed from the queue in ConToPH, and the IDLE token is replaced by a PROCESS token in place PHandler. This models the Payment Handler initiating the Payment exchange on receipt of the Payment Request Message. Apart from SndCancel_C on page Consumer, another three transitions SndPayExch_P, SndPayResp_P and SndCancel_P on page PHandler are now enabled.

However, occurrences of SndPayExch_P and SndPayResp_P are *in conflict*, i.e., occurrence of one transition disables the other. If SndPayExch_P occurs, the Payment Handler will send a Payment Exchange Message ([TransRefBlk,PayExchBlk]) and enter state WAIT. In this case, a payment system-specific data exchange is initiated, and SndPayResp_P will not be enabled until the state of the Payment Handler changes to PROCESS. If SndPayResp_P occurs, the Payment Handler will complete the Payment exchange by sending a Payment Response Message ([TransRefBlk,PayRespBlk]). In this case, no payment system-specific data exchange is needed.

On receipt of the Payment Exchange Message, transitions RcvPayExch_C and then SndPayExch_C can occur on page Consumer. As a result, transition RcvPayExch_P on page PHandler becomes enabled. On its occurrence, the Payment Handler enters PROCESS, and SndPayExch_P and SndPayResp_P are both enabled again. Similarly, occurrence of SndPayExch_P will initiate another payment system-specific data exchange, while occurrence of SndPayResp_P will complete the Payment exchange at the Payment Handler side.

When the Consumer or Payment Handler is in WAIT or PROCESS, the respective SndCancel transition (SndCancel_C or SndCancel_P) can always occur. If there is also a Cancel Message ([TransRefBlk,CancelBlk]) at the front of the message queue, the respective RcvCancel transition (RcvCancel_C or RcvCancel_P) becomes enabled as well. The occurrence of either SndCancel or RcvCancel transition aborts the Payment exchange at the corresponding trading role side.

Finally, if the Consumer or Payment Handler is in COMPLETED or CANCELLED and still receives message(s), the respective Discard transition will occur to remove the message (m) from the message queue (m::q) until the queue is empty.

6.2 Simulation and State Space Analysis

CPN models are executable. This implies that we can investigate the behaviour of the system by *simulating* (or executing) the CPN model. More importantly, the *state space method* of CPNs makes it possible to validate and verify the functional correctness of systems.

6.2.1 Simulation

One major goal of conducting simulation is to validate that the CPN model conforms with the system design specification. This can be done by demonstrating that the CPN model can generate the same set of sequences of event occurrences as defined in the system design specification. A CPN model can be executed from its initial marking to a destination marking after firing a sequence of transitions, which represents a sequence of event occurrences. For example, by simulating the CPN model in Section 6.1.3, we obtain the same set of message exchanges between the Consumer and Payment Handler for a successful Payment exchange as specified in RFC 2801 [36]. The RFC defines the message exchanges between trading roles using *message sequence charts* (MSCs) [118]. The computer tool supporting CPNs has a facility to generate MSCs (see Section 6.3.2). To make it easy to compare the message exchanges generated from the CPN model of Payment exchange with those specified by the RFC, the CPN model is instrumented so that MSCs displaying the message exchanges between the Consumer and Payment Handler can be automatically produced during a simulation.

Another purpose of doing simulation is to investigate specific sequences of event occurrences. By simulating these sequences, we can examine the system behaviour under specific

circumstances. For example, in a Payment exchange, the sequence of message exchanges involving the cancellation by the Consumer may be of particular interest, and can be generated by firing a sequence of transitions including SndCancel_C with s bound to either WAIT or PROCESS on page Consumer.

6.2.2 State Space Analysis

The basic idea behind the state space method of CPNs is to compute all reachable states and state changes of the system based on the CPN model. State space analysis can be used to formally reason about certain properties of the model.

A state space is represented as a directed graph which has a node for each reachable marking (or state) of the model and an arc for each occurring *binding element* (or state change). A *binding element* is a pair consisting of a transition and an assignment of data values to the variables appearing in the arc expressions of its surrounding arcs and associated guard expression. For example, for the CPN model shown in Figure 6.3, (RcvCancel_C: {s=WAIT, q=[ ]}) and (RcvCancel_C: {s=PROCESS, q=[ ]}) are the two binding elements for which the transition SndCancel_C is enabled. From a given initial marking, the state space can be generated by executing enabled binding elements until all reachable markings have been calculated. State spaces are also called *occurrence graphs* or *reachability graphs*.

Figure 6.4 depicts the state space for the CPN model of Payment exchange. There are *31* nodes and *56* arcs, representing *31* different states during a Payment exchange. The text inside the node tells us the node number as well as the number of predecessors and successors (separate by colon :) of the node. A solid box next to a node lists the markings of the individual places, and a dashed box next to an arc lists the occurring binding element. We use M_n to denote the marking corresponding to node number n. In Figure 6.4, M_1 represents the initial marking, where both the Consumer and Payment Handler are IDLE, and the message queues are empty. It has 0 predecessor and 1 successor. The arc (with an arc number *1*) between M_1 and its successor M_2 shows that in M_1 an occurrence of SndPayReq_C with q bound to the empty list [ ] will lead to M_2, representing the transmission of a Payment Request Message from the Consumer. In M_2, RcvPayReq_P with q bound to [ ] may occur and lead to M_4, indicating the receipt of the Payment Request Message by the Payment Handler; or SndCancel_C with q bound to [ ] may occur and lead to M_4, indicating the Consumer sending a Cancel Message to cancel the Payment exchange. Analogously, we can reach all the markings in the state space shown in Figure 6.4.

A set of standard properties of the model can be proved from its state space. The formal definitions of these properties can be found in [123]. Below, we consider three of them that are important in verifying the correctness of IOTP in Chapter 11. They are illustrated with the state space of the Payment exchange CPN model in Figure 6.4.

Absence of deadlocks. A marking with no successors means that no transition is enabled in that marking. It is referred to as a *dead marking*, i.e., a state in which the system being modelled has terminated. A dead marking representing an undesirable terminal state is called a *deadlock*. Deadlocks result from errors either in the model or in the system being modelled. Hence, a functionally correct system must be free from deadlocks. In Figure 6.4, there are three dead markings: M_{11}, M_{15} and M_{26} (drawn with thicker border). Inspection of the solid

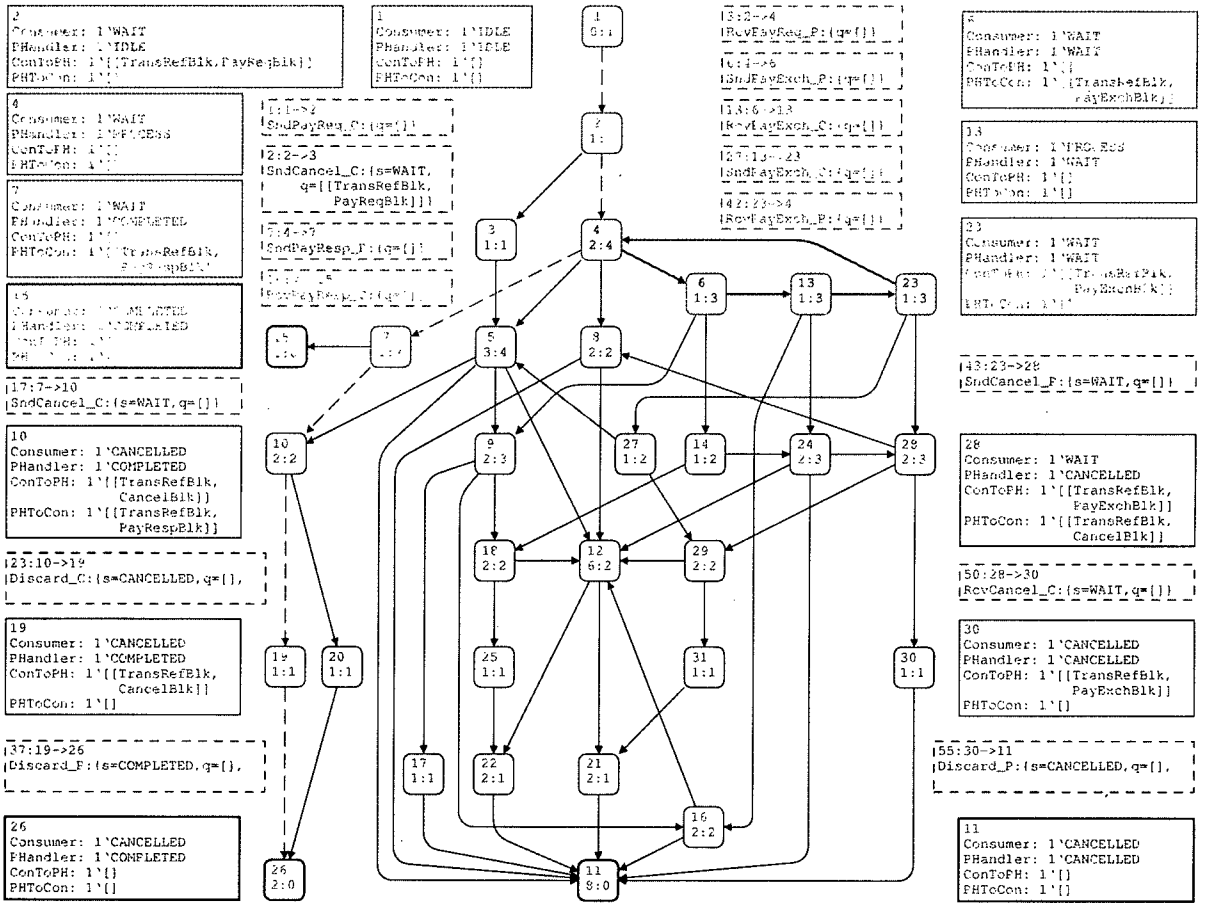


Figure 6.4: State space of the Payment exchange CPN model.

boxes next to each of these three markings reveals that M_{15} and M_{11} are desired terminal states of the IOTP Payment exchange, whereas M_{26} is undesired.

- M_{15} and M_{11} represent the successful completion (M_{15}) or cancellation (M_{11}) of the Payment exchange, where both the Consumer and Payment Handler are in `COMPLETED` or `CANCELLED` state, and the message queues are empty.
- M_{26} indicates that, the Consumer and Payment Handler enter inconsistent terminal states, i.e., the Consumer in `CANCELLED` while the Payment Handler in `COMPLETED`, upon transaction cancellation. In Figure 6.4, a sequence of event occurrences via nodes (1,2,4,7, 10,19,26) connected by dashed arcs can be used to identify the problem causing the inconsistency in M_{26}. Arc 16, 17, 23 and 37 reveal that, such inconsistency arises when the Consumer cancels a Payment exchange (arc 17) after the Payment Handler has completed the Payment exchange (arc 16), and both trading roles discard the messages received afterwards (i.e., the Payment Response Message (arc 23) and Cancel Message (arc 37)). Therefore, M_{26} represents a deadlock in the state space which results from the design of the Payment exchange in IOTP. To remove the deadlock IOTP needs improvement. We shall explain this in detail when analysing IOTP in Chapter 11.

Absence of livelocks. If a state space contains a cycle that leads to no markings outside the cycle, then once the cycle is entered it will repeat for ever. This is called a *livelock*. A system, which provides meaningful operations and terminates properly, should be free from livelocks.

Here we introduce a new concept, a *strongly connected component* (SCC) of the state space, which is a maximal sub-graph whose nodes are mutually reachable from each other [123, 124]. A *SCC graph* has a node for each SCC and arcs which connect two different SCCs. The *initial SCC* is a SCC without incoming arcs, and the terminal SCC is a SCC without outgoing arcs. Since each node in the state space belongs to only one SCC, the SCC graph is always smaller than or equal to the corresponding state space. The state space and its SCC graph help to determine whether the system has livelocks. There are two circumstances.

- Firstly, if the state space and its SCC graph are *isomorphic*, i.e., the SCC graph is equal to the corresponding state space, and also there are no self-loops, then the system is free of livelocks.
- Secondly, if the state space contains self-loops, or if there exists at least one SCC consisting of more than one node, then we need to examine if all terminal SCCs are *trivial* (i.e., consists of a single node and no arcs). Only if all terminal SCCs are trivial, will the system *always be able to* terminate in one of the dead markings in the state space, demonstrating the absence of livelocks. Otherwise, a non-trivial terminal SCC represents a livelock in the system.

Figure 6.5 shows the SCC graph of the state space in Figure 6.4. It has *28* nodes and *52* arcs. Node 1 is the initial SCC, and nodes 15, 25 and 28 are terminal SCCs (drawn with solid border). A SCC node has information about the number of nodes (markings) and arcs (in the state space) which have been grouped in that SCC. The number of arcs connecting one SCC to others, are indicated next to the arcs. For example, SCC node 28 (i.e., a terminal SCC) has one marking, one input arc from each of the SCC node 26 and 27, and no output arcs.

In Figure 6.5, each SCC except node 4 corresponds to one marking in Figure 6.4. SCC node 4 comprises *4* markings and *4* arcs in the state space, which are nodes 4, 6, 13, and 23 connected by thicker arcs in Figure 6.4. It represents payment system-specific data exchanges that are only required by a particular payment system (such as SET). This can occur any number of times, catering for various payment systems to be encapsulated by IOTP. Since SCC node 4 is not a terminal SCC, we can prove the IOTP Payment exchange has no livelocks.

Absence of unexpected dead transitions. If a transition is never enabled in any reachable marking then this transition is a *dead transition*. An unexpected dead transition may indicate the event being modelled is redundant or that unexpected behaviour has occurred, resulting in that event not being utilized. There are no dead transitions in the CPN model of IOTP Payment exchange, because all transitions can occur at least once when executing the model.

6.3 Design/CPN

Design/CPN [176] is a software tool for the construction, simulation and analysis of CPN models. It comprises a CPN editor (with syntax checker), simulator and state space tool, all of which include graphical user interface and use CPN ML, a variant of SML [228], for net inscriptions and declarations. More detailed descriptions can be found in [60, 128, 130, 152].

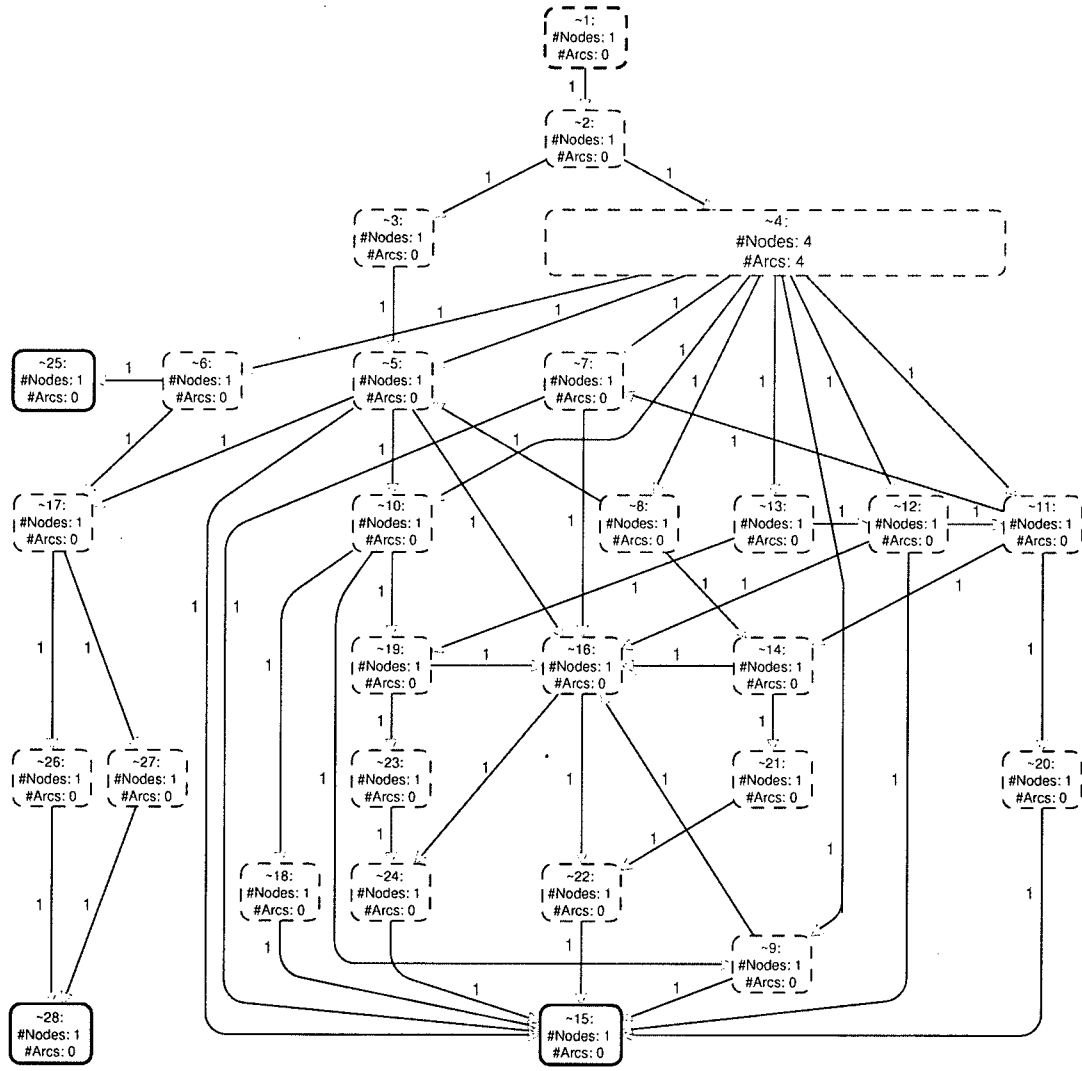


Figure 6.5: SCC graph of the state space shown in Figure 6.4.

6.3.1 CPN Editor

The CPN editor provides graphical facilities for the creation and modification of different net components such as places, transitions and arcs. It also allows the user to syntax check a CPN model. Auxiliary objects, which are non-CPN components, can be used to improve the model's readability. For each CPN model, the editor automatically generates a hierarchy page, and supports hierarchical constructs. The CPN model of IOTP Payment exchange in Section 6.1.3 was created using the CPN editor in Design/CPN.

In a hierarchical CPN, different pages (modules) can connected in two ways. Both are supported by the CPN editor. One is the use of substitution transitions. Hierarchy is introduced into the model by representing a CPN on a subpage by a single substitution transition on a higher level page. All of the input and output (socket) places associated with the substitution transition are connected to their corresponding (port) places in the subpage via a port-socket assignment. A hierarchical CPN model with substitution transitions can be constructed using facilities provided by the CPN editor, such as *creating substitution transitions* and *moving to subpages*.

The second structuring mechanism is known as *place fusion*. The basic idea behind place fusion is to create multiple copies of the same place on different pages. These copies of places

belong to the same fusion set and are called *fusion places*. Fusion places belonging to the same fusion set will have identical markings.

Further examples of substitution transitions and fusion places will be seen in Chapter 8 and 10, where detailed descriptions of the service model and the protocol model of IOTP are presented, respectively.

6.3.2 CPN Simulator

The CPN simulator supports execution and debugging of CPN models. It offers two basic simulation modes, *interactive simulation* and *automatic simulation*, which are suitable for different purposes.

The interactive simulation mode allows the user to set breakpoints, check the set of enabled transitions, select binding elements and even change markings of places. The modeller can study the dynamic behaviour of a CPN model step by step to see whether the individual net components work as expected. Interactive simulations are, in most cases, used to inspect certain parts of the CPN model in detail. Different feedback options can be set with respect to the speed of the interaction.

The automatic simulation mode focuses on the overall behaviour of a CPN model. No feedback is given to the user except when there are no enabled transitions, i.e., a dead marking is reached. The simulator makes random selections between enabled binding elements, and as a result, a random sequence of binding elements is fired. Before and after an automatic simulation, the current marking and the enabled transitions are displayed as in the interactive mode. Details about all the bindings of transitions that occur during the automatic simulation are written to a file called the *simulation report*.

Associated with the CPN simulator is the MSC library [66], which can be used to capture a single simulation run and provide a graphical view of the execution of the model. Four possible sequences of message exchanges for an IOTP Payment exchange generated using the MSC library, are shown in Figure 6.6. In Figure 6.6 (c) and (d), both Consumer and Payment Handler discard the message received after the Payment exchange has terminated at the corresponding side.

6.3.3 State Space Tool

The CPN state space tool [128] supports generation and analysis of state spaces. Full or partial state spaces can be drawn in the state space tool, displaying the markings and binding elements (see Figure 6.4). The user can define the information displayed for the markings and binding elements, which is useful when generating the language from the state space. We shall discuss this in Section 6.4 and 6.5.

When calculating a state space, it is possible for the user to control generation of the state space using *stop* and *branching options*. Stop options allow the user to specify a time or node limit under which if exceeded will stop the calculation of the state space. Branching options give the user control over the generation of successors. The generation control can be applied to identify a configuration of the system which is tractable for state space analysis, and therefore is particularly useful in early phases of state space analysis.

When a state space has been generated, a *state space report* can be produced automatically

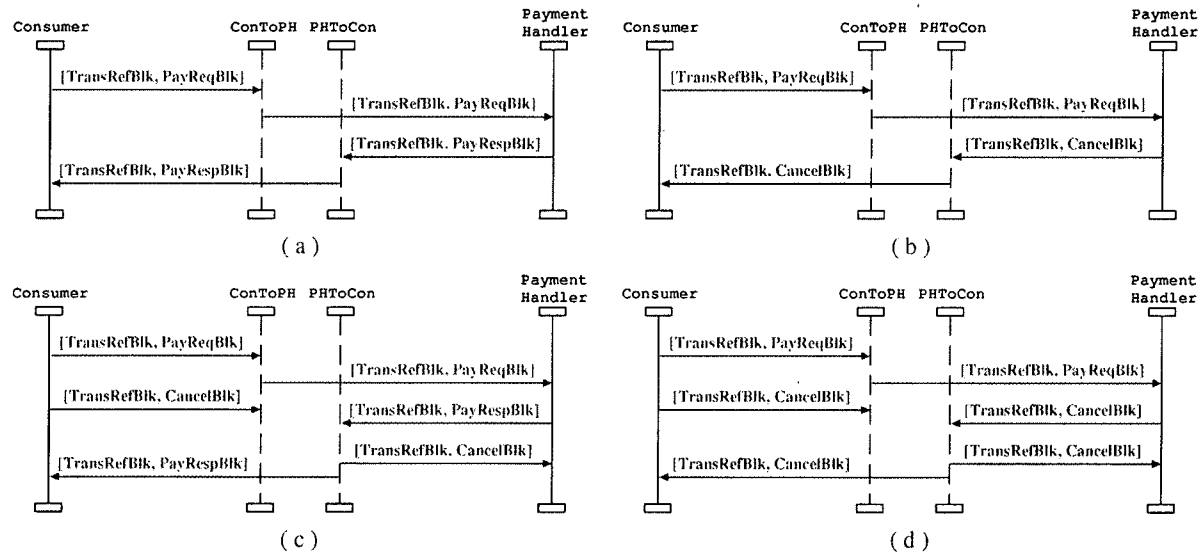


Figure 6.6: Four possible sequences of message exchanges in an IOTP Payment exchange which is: (a) complete; (b) cancelled by the Payment Handler; (c) cancelled by the Consumer but completed at the Payment Handler side; and (d) cancelled by both Consumer and Payment Handler.

to answer a set of standard questions concerning the behaviour of the CPN model. The state space report includes: *statistical information* about the size of the state space and the corresponding SCC-graph; *liveness properties* including dead markings and dead transitions; and *boundedness properties* concerning both integer and multi-set bounds of each place. The state space tool provides a set of *standard query functions* for checking standard behavioural properties of the CPN model, such as those written in the state space report. The standard query functions can be used to define advanced queries to prove more complex system-specific properties.

Apart from the above, there are also additional libraries available for analysis. These are: performance analysis [152]; state spaces with equivalences and symmetries [130]; and recently the sweep-line methods [47].

6.4 Finite State Automata

Both the service language and the protocol language of IOTP are represented by FSA. In this section we briefly introduce the definitions of automata and language, a standard automata reduction algorithm used to generate languages that only include service primitives, and automata theory for language comparison. For more details on automata definition, theory and algorithms, the reader is referred to the literature [15, 105, 138, 150, 156].

6.4.1 Automata and Language Definitions

A FSA has a finite number of *states*, including an *initial state* and, in general, a set of *final states*. Upon occurrence of an input *event*, the current state (or source state) changes to the next state (or destination state) in the FSA, according to the *next state relation* (or *transition relation*) of the FSA. Each event belongs to a finite set of pre-defined events (or *alphabet*), e.g., the set of all service primitives in IOTS (see Chapter 7). Each FSA has its minimal form which contains a minimal number of states. Based on [150], a FSA can be defined as follows.

Definition 6.1 A finite state automaton, A, can be defined as a quintuple $A=(S, E, \Delta, s_0, S_f)$, where S is a finite set of states, E is a finite set of events, $\Delta \subseteq S \times E \times S$ is the transition relation, $s_0 \in S$ is the initial state and $S_f \subseteq S$ is the set of final states. □

A FSA can be used to generate a language. Defining the language generated (or accepted) by a FSA needs some notation for sequences of events [23]:

- let E^* be the set of finite sequences of events in E, including the empty sequence, ϵ;
- let $s[e>s'$ *iff* $(s,e,s') \in A$ denote an occurrence of an event $e \in E$, in state $s \in S$, resulting in state s'; and
- a sequence of events $\sigma \in E^*$, where $\sigma = e_1 e_2 ... e_n$, starting in s_1 and finishing in state s_{n+1}, can be denoted by $s_1[\sigma > s_{n+1}$ which is defined by the state transitions:

$$s_1[e_1 > s_2[e_2 > s_3 ... [e_n > s_{n+1} \text{ where } (s_1, e_1, s_2),\ (s_2, e_2, s_3),\ ...,\ (s_n, e_n, s_{n+1}) \in A.$$

The language accepted by a FSA includes all sequences of events that start in the initial state and finish in a final state of the FSA. Two FSA are called *language equivalent* if they accept the same language. The language accepted by a FSA can be defined [23] as follows.

Definition 6.2 The language, $L(A)$, accepted by a finite state automaton, $A=(S, E, \Delta, s_0, S_f)$, can be defined as $L(A) = \{\sigma \in E^*: s_0[\sigma > s_f \text{ where } s_f \in S_f\}$. □

There are two types of FSA: a *deterministic FSA* and a *non-deterministic FSA*. A deterministic FSA is an automaton that has *at most one* output transition labelled with the same event for each state. In such a FSA, once an input event is given in a source state, the destination state is *uniquely determined.* In contrast, a non-deterministic FSA has *more than one* transition labelled with the same event leading from one state to its next state. In such a FSA, given an input event in a source state, the automaton may choose to go into any one of those valid destination states *non-deterministically*. For each non-deterministic FSA there is an equivalent (language equivalent) deterministic FSA [150].

A FSA may be represented as a directed graph. States are represented by nodes, and transitions are represented by arcs with arrows pointing from the source state to the destination state. Arcs are labelled with events. By convention, the initial state is indicated by a short arrow pointing to it. The final states are drawn as doublecircles. Figure 6.7 depicts two simple examples of FSA. FSA-(a) is deterministic, and FSA-(b) is non-deterministic. Each FSA contains five states labelled `A`, `B`, `C`, `D` and `E`. State `A` is the initial state, and states `D` and `E` are both final states. Each FSA is defined over the set of events {a,b,c,d}. The language accepted by FSA-(a) is {ab,abd,acb}, and the language accepted by FSA-(b) is {ab,ac,abd,abc}.

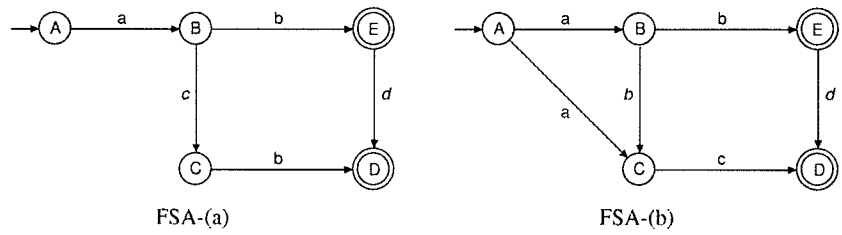


Figure 6.7: Graphical representation of two FSA.

6.4.2 Language Generation

For language analysis, the state space of a CPN model can be treated as a FSA, where the markings represent the states and the occurring binding elements correspond to the set of events in the FSA. The language accepted by the FSA defines all possible sequences of events.

The service and protocol languages may be obtained from the state spaces of the corresponding CPN models. A standard automata reduction algorithm [15] is used to generate languages that only include service primitives. The algorithm is based the following steps.

1. Removal of empty moves (ϵ-transitions) from the FSA: the empty moves are those occurring binding elements representing non-service-primitive events in the state space;
2. Removal of non-determinism (determinisation): to convert a non-deterministic FSA into its equivalent deterministic FSA; and
3. Removal of inaccessible states, and/or reduction by identifying and merging equivalent states (minimisation): to convert a deterministic FSA into its minimal form.

The minimum deterministic FSA is obtained to represent the language and also to be used for language comparison.

6.4.3 Language Comparison

The main purpose of generating service and protocol languages is to see if these two languages are the same, i.e., all the sequences of service primitives in the protocol language are those specified in the service language. Theorems and algorithms developed for the analysis of FSA can be applied to compare two FSA in terms of the languages accepted by these two FSA. The approach is based on the fact that any non-deterministic FSA with ϵ-transitions can be converted into an equivalent minimum deterministic FSA or canonical form as discussed above. If the canonical forms of two FSA are isomorphic, they are *language equivalent*.

Language equivalence can be checked using *language comparison* (or *language difference*) that is given by Definition 6.3. Using the difference between two languages, Property 6.1 defines language equivalence.

Definition 6.3 If L_A and L_B are two languages over the alphabet E. The difference between L_A and L_B is defined as [138]: L_A - $L_B = \{x\colon x \in L_A \text{ and } x \notin L_B\}$. □

Thus L_A - $L_B = \phi \Leftrightarrow L_A \subseteq L_B$.

Property 6.1 Two languages L_A and L_B over the alphabet E are identical *iff* L_A - $L_B = \phi$ and L_B - $L_A = \phi$. □

In this thesis, we use language comparison to check language equivalence: 1) between local and global service languages of IOTP (Chapter 9); and 2) between service and protocol languages of IOTP (Chapter 11).

6.5 FSM, LexTools and GraphViz

General-purpose FSA software tools are used for supporting IOTP verification based on language comparison, since Design/CPN does not provide the support for analysing FSA.

FSM. The FSM library (or FSM for short) [11] includes a set of programs for constructing, combining, minimising and searching finite state machines. We briefly describe the FSM programs that are used for automata reduction, determinisation, minimisation and comparison in verifying IOTP. Figure 6.8 shows the procedure for obtaining a minimum deterministic FSA using six FSM programs.

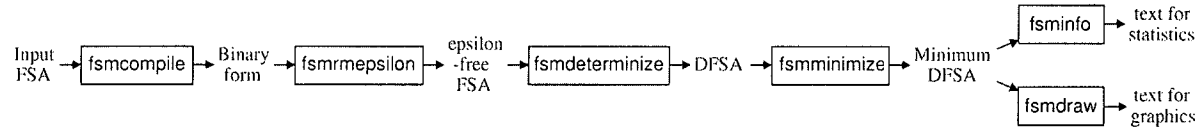


Figure 6.8: Procedure of FSA reduction, determinisation and minimisation.

Firstly, the state space output from Design/CPN is written to a text file that can be used as the input to FSM. The input FSA is usually of the form given in Figure 6.9, where S_S represents the source state, S_D the destination state, and A the transition from S_S to S_D. Information of S_S, S_D and A are all given by integers (or numbers) in the input FSA. Lines with one integer (for source state) give the final states, and transitions with number 0 correspond to ϵ-transitions.

S_S	S_D	A
1	2	1
1	3	0
2	3	1
3		

Figure 6.9: Format of a text file as the input to FSM.

Next, the text file of the input FSA is compiled into a binary form using fsmcompile. With fsmrmepsilon, all the ϵ-transitions are removed from the FSA. Then, the ϵ-free FSA (normally a non-deterministic FSA) is converted into a deterministic FSA (denoted by DFSA in Figure 6.8) using fsmdeterminize. Minimisation of the deterministic FSA is done using fsmminimize. The fsminfo takes the minimum deterministic FSA as input and returns statistical information about the FSA (e.g., the number of states, transitions and final states). The fsmdraw is used to specify the FSA in a text form that can be read by a graph drawing tool such as GraphViz [12].

In addition to those shown in Figure 6.8, FSM also provides the program fsmdifference, which takes two deterministic FSA and returns the FSA representing the language difference between the first and the second input FSA. This can be used for language comparison.

LexTools. LexTools [13] is a finite-state lexical toolkit, which provides a set of programs for converting high-level linguistic descriptions and context-free rewrite rules into FSA. It is built on FSM, and is useful for generating the language as a string representation. For example, given a FSA and a file mapping transition numbers to labels, the program *lexfsmstring* is used to write the language of an acyclic FSA to a text file.

GraphViz. GraphViz [12] is a graph drawing tool for creating and displaying graphs. Since FSM does not have an integrated visualisation facility, GraphViz is used in conjunction with FSM to generate a graphical representation of a FSA. For example, taking the output from fsmdraw (see Figure 6.8), the program *dot* is used to create a PostScript version of a FSA. GraphViz provides automatic layout for the FSA, but unfortunately does not provide facilities for graphical editing of the graphs that are automatically laid out.

Chapter 7

Internet Open Trading Service Definition

From a protocol engineer's viewpoint, obtaining a well defined service specification represents an important step in developing protocols. However, the service provided by IOTP has not yet been defined in RFC 2801 and no related work has been found. We therefore need to develop our own service specification for IOTP. This is more challenging, compared with those protocols [109, 116, 117] that already include a service definition.

In this chapter, we define the service provided by IOTP to its users, namely the *Internet Open Trading Service* (IOTS). Following the Open System Interconnection (OSI) principles [114], an abstract structure of IOTS is defined (Section 7.1). The IOTS definition comprises: 1) the definition of service primitives and their parameters (Section 7.2) and 2) the definition of all valid sequences of service primitives (Section 7.3). The outcome is a well defined IOTS, which forms a basis for developing a formal service specification for IOTP in Chapter 8.

7.1 IOTS Model

Following the OSI service conventions [114] and Transport Service definition [116], we define the abstract model of IOTS. Figure 7.1 depicts this model which comprises two conceptual components: *service users* and *service provider*.

- The service users are IOTS Users, i.e., trading role user entities. Note that the *Merchant Customer Care Provider* has been omitted since it is not involved in any trading transaction defined in IOTP.
- The IOTS Provider represents IOTP protocol entities and all the underlying communication protocols.

Also, there are two associated concepts: *service access point (SAP)* and *service primitive*.

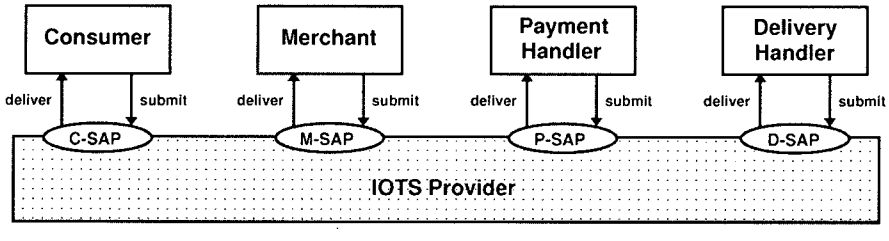


Figure 7.1: The abstract model of IOTS.

- The IOTS Provider offers service to IOTS Users via four SAPs (indicated by ellipses). Each SAP represents the boundary between the IOTS User and IOTS Provider. There is one for each and we hereafter call them *trading role SAPs*. More specifically, we use the term *Consumer-SAP* (C-SAP) to refer to the SAP that is related to the Consumer entity. Similarly, *Merchant-SAP* (M-SAP), *Payment-Handler-SAP* (P-SAP), and *Delivery-Handler-SAP* (D-SAP) are used to uniquely identify the other SAPs.
- The service primitives describe the logical interactions between the IOTS Users and IOTS Provider that can be observed at each SAP. For example, a trading role entity (e.g., a Consumer) may request a service (e.g., to inform a Merchant of a trading request for purchase) from the IOTS Provider by *submitting* a service primitive, and the IOTS Provider responds by *delivering* the corresponding service primitive to the appropriate peer trading role entity (e.g., the Merchant). In addition, there are also parameters associated with each primitive to convey the required control information and user data.

7.2 IOTS Primitives and Parameters

The Authentication and Payment-related transactions defined by IOTP can be considered as six *trading transaction services*, namely *Purchase*, *Refund*, *Deposit*, *Withdrawal*, *Value Exchange* and *Authentication*. We adopt the modular construction of transactions from document exchanges in IOTP in the service definition. To emphasize that we are at a higher level of abstraction than the protocol, and to include cancellation, we will refer to the service modules related to document exchanges as *phases*.

Hence, each trading transaction service can be considered as being provided via a combination of five phases, namely *Offer*, *Payment*, *Delivery*, *Authentication*, and *Cancel*. A Purchase transaction service consists of an Offer, a Payment, and a Delivery phase. A Deposit, Withdrawal, or Refund transaction service starts with an Offer phase followed by a Payment phase. A Value Exchange transaction service is provided via an Offer phase and two Payment phases that occur in sequence. Each of these five transaction services may also involve authentication. Finally, an Authentication transaction service comprises only an Authentication phase. In addition to the above, transactions may be aborted in a *Cancel* phase under various circumstances.

Accordingly, we define the IOTS primitives associated with each trading phase rather than each transaction. The conventions used to name these service primitives conform to [114]. A complete listing of IOTS primitives and parameters appears in Table 7.1. The primitives are named in such a way that each of them reflects the service it represents. For example, the *TradingOptions* primitives are defined for offering and selecting from a list of available trading options in the Offer phase. The name of a parameter is given as an indication of the information the parameter conveys. In the RFC [36], the IOTP message definition includes a list of trading components which carry the concrete trading information from users (see Section 3.2.3). Hence, a parameter is named using a trading component or one of its attributes defined in IOTP, if they convey the same information. For instance, the name of the *Process State* parameter associated with *PayResult*, *AuthResult* and *Cancel* primitives, is taken from the *ProcessState* attribute of the *Status Component* in IOTP, both indicating the current state of the trading phase (in IOTS) or the document exchange (in IOTP) being carried out.

Phase	Service	Primitive	Parameters
Offer	Trading offer	TradingOffer.req	(Transaction ID, Transaction Type, Trading Organisation, Order Content)
		TradingOffer.ind	(As above)
	Trading options	TradingOptions.req	(Transaction ID, Transaction Type, Trading Organisation, Selection Reply, Payment Method Options)
		TradingOptions.ind	(As above)
		TradingOptions.res	(Transaction ID, Payment Method Selection)
		TradingOptions.cnf	(As above)
	Trading result	TradingResult.req	(Transaction ID, Order Content, Payment Amount, Payment Type, Payment Instructions, Delivery Type, Delivery Instructions)
		TradingResult.ind	(As above)
Payment	Payment invoke	PayInvoke.req	(Transaction ID, Trading Organisation, Payment Amount, Payment Number, Payment Method Selection, Payment Instructions, Delivery Type, Payment Method-specific Data)
		PayInvoke.ind	(As above)
	Payment data exchange	PayData.req	(Transaction ID, Payment Method-specific Data)
		PayData.ind	(As above)
		PayData.res	(As above)
		PayData.cnf	(As above)
	Payment result	PayResult.req	(Transaction ID, Process State, Payment Receipt, Payment Note, Payment Error)
		PayResult.ind	(As above)
	Payment retry	PayRetry.req	(Transaction ID, Payment Method Selection, Payment Method-specific Data)
		PayRetry.ind	(As above)
Delivery	Delivery	Delivery.req	(Transaction ID, Trading Organisation, Order Content, Payment Receipt, Delivery Instructions)
		Delivery.ind	(As above)
		Delivery.res	(Transaction ID, Delivery Status, Delivery Note, Digital Content Data)
		Delivery.cnf	(As above)
Authentication	Authentication	Authentication.req	(Transaction ID, Trading Organisation, Transaction Type)
		Authentication.ind	(As above)
		Authentication.res	(Transaction ID, Authentication Data)
		Authentication.cnf	(As above)
	Authentication result	AuthResult.req	(Transaction ID, Process State, Authentication Error)
		AuthResult.ind	(As above)
	Authentication retry	AuthRetry.req	(Transaction ID, Authentication Data)
		AuthRetry.ind	(As above)
Cancel	Cancel	Cancel.req	(Transaction ID, Trading Phase, Process State, Cancel Reason)
		Cancel.ind	(As above)

Table 7.1: IOTS primitives and parameters.

In the following subsections, we describe in detail the IOTS primitives defined within each trading phase. For clarity, we briefly introduce the associated parameters when specifying each service primitive. A detailed description of all parameters and their usage within the associated primitives is given in Appendix B.

7.2.1 Offer Phase

All trading transactions except Authentication involve an Offer phase, where the trading options negotiation is conducted between the Consumer and Merchant role entities. Three primitives are defined to support the services of initiating a trading transaction (*TradingOffer*), offering and selecting available trading options (*TradingOptions*) and issuing the trading result (*TradingResult*). Table 7.2 lists the three primitives with associated parameters in the Offer phase. The notation for parameter usage given in Section 5.2.1 (see Table 5.2) is applied to all the primitive and parameter tables in the remainder of this chapter.

Primitive / *Parameter*	**TradingOffer**		**TradingOptions**				**TradingResult**	
	req	*ind*	*req*	*ind*	*res*	*cnf*	*req*	*ind*
Transaction ID	M	$M(=)$	M	$M(=)$	M	$M(=)$	M	$M(=)$
Transaction Type	M	$M(=)$	M	$M(=)$				
Trading Organisation	M	$M(=)$	M	$M(=)$				
Order Content	U	$C(=)$					U	$C(=)$
Selection Reply			M	$M(=)$				
Payment Method Options			M	$M(=)$				
Payment Method Selection					M	$M(=)$		
Payment Amount							M	$M(=)$
Payment Type							M	$M(=)$
Payment Instructions							M	$M(=)$
Delivery Type							M	$M(=)$
Delivery Instructions							U	$C(=)$

Table 7.2: IOTS primitives and parameters in Offer phase.

Below, we give a detailed description of the IOTS primitives appearing in Table 7.2.

- The ***TradingOffer* primitives** are defined for the Consumer to initiate a trading transaction (e.g., a Purchase) with the Merchant, after the Consumer has decided which goods/services to buy. In practice, with online shopping, this could be considered as the Consumer clicking on a "Buy Now" button (or, something similar) that is available on the Merchant's Web page, after the Consumer has filled in the order form. Initiation of a trading transaction by the Consumer represented by using the *TradingOffer* primitives, is beyond the scope of IOTP version 1.0. We have defined these primitives since IOTP version 2.0 intends to include Consumer-initiated transactions [75].

  The Transaction ID parameter uniquely identifies a trading transaction and therefore needs to be present in each primitive. Transaction Type specifies which one of the seven trading transactions is initiated. Trading Organisation provides the information of different organisations involved in the transaction. We will not describe this parameter any further when introducing each IOTS primitive. The Order Content contains the Consumer's ordering details and is only used within a Purchase or a Refund transaction.

- The ***TradingOptions* primitives**[1] are defined for the Merchant to provide the Consumer with the available trading options (e.g., a list of payment methods), and *optionally* ask for the Consumer's selection. In practice, this could be considered as the Consumer being provided with a list of credit cards (e.g., Visa, Mastercard, American Express, etc.) that can be used for the purchase. The Consumer selects one of them. The selection may or may not be sent to the Merchant (see Figure 7.2), depending on whether it will affect the price to be offered in the next stage. For example, if a 5% discount is applied to the purchase made by using a Visa card, the Merchant needs to know the Consumer's selection to determine the required payment, which will be part of its *TradingResult.req* (see the description of *TradingResult* primitives next).

  The Selection Reply parameter indicates whether or not the Merchant needs a reply from the Consumer indicating the selected payment method. Payment Method Options conveys a list of payment methods that are available. Payment Method Selection specifies which one of the payment methods is selected by the Consumer.

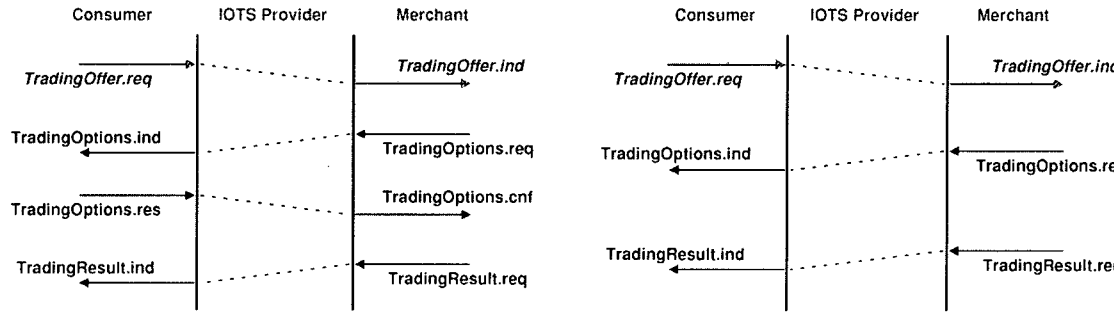
- The ***TradingResult* primitives**[2] are defined for the Merchant to notify the Consumer about the trading result at the end of the Offer phase. In practice, this could be considered as the Consumer being presented with an electronic invoice together with payment and delivery instructions.

  The Order Content parameter is copied from the *TradingOffer.ind* primitive to ensure that the Consumer's ordering details are unchanged. This parameter is optional because it only applies in a Purchase transaction. Payment Amount specifies how much needs to be paid. Payment Type specifies how many payments are required for the transaction. Note that one Payment phase involves only one payment. Payment Instructions provide information on initiating each Payment exchange. Delivery Type indicates whether a delivery is required, or whether the delivery is to be provided by the Payment Handler or Delivery Handler. Delivery Instructions set out the procedure that the Consumer needs to follow to get the goods or services delivered. This parameter is only present when there is a delivery.

Two valid sequences of IOTS primitives in the Offer phase are defined by the two time sequence diagrams (TSDs) shown in Figure 7.2. We have introduced in Chapter 5 (see Figure 5.4) the conventions of a TSD based on those defined in [114]. The TSD shown in Figure 7.2 (a) represents the scenario where the Merchant needs to know the Consumer's payment method selection. In this case, the following two correlated primitive events [*Trading-Options.res*, *TradingOptions.cnf*] MUST occur. The TSD shown in Figure 7.2 (b) represents the scenario where the Merchant does not require the Consumer's selection to be sent back. In this case, the above primitive events do NOT occur. Finally, as we mentioned before, occurrence of the first two correlated primitive events [*TradingOffer.req*, *TradingOffer.ind*] in both TSDs in Figure 7.2 supports Consumers to initiate the trading services.

[1]This is derived from Sect.9.1.2.3 on pages 199-200, Sect.9.1.2.4 on page 201, Sect.8.1 on pages 166-167 and Sect.8.2 on pages 167-168 of RFC 2801.

[2]This is derived from Sect.9.1.2.5 on page 201 and Sect.8.3 on pages 168-169 of RFC 2801.



(a) Consumer's selection response required

(b) Consumer's selection response not required

Figure 7.2: Two valid primitive sequences in the Offer phase.

7.2.2 Payment Phase

All trading transactions except Authentication involve payment(s) between the Consumer and Payment Handler. Four primitives are defined in the Payment phase to cover the services of initiating a payment (*PayInvoke*), exchanging *payment (method-specific) data* (*PayData*), issuing the payment result (*PayResult*) and retrying the payment (*PayRetry*) if it has failed. Table 7.3 lists the four primitives with associated parameters in the Payment phase.

Primitive	**PayInvoke**		**PayData**				**PayResult**		**PayRetry**	
Parameter	*req*	*ind*	*req*	*ind*	*res*	*cnf*	*req*	*ind*	*req*	*ind*
Transaction ID	*M*	*M(=)*	*M*	*M(=)*	*M*	*M(=)*	*M*	*M(=)*	*M*	*M(=)*
Trading Organisation	*M*	*M(=)*								
Payment Amount	*M*	*M(=)*								
Payment Number	*U*	*C(=)*								
Payment Method Selection	*M*	*M(=)*							*U*	*C(=)*
Payment Instructions	*M*	*M(=)*								
Delivery Type	*M*	*M(=)*								
Payment Method-specific Data	*U*	*C(=)*	*M*	*M(=)*	*M*	*M(=)*			*M*	*M(=)*
Process State							*M*	*M(=)*		
Payment Receipt							*U*	*C(=)*		
Payment Note							*U*	*C(=)*		
Payment Error							*U*	*C(=)*		

Table 7.3: IOTS primitives and parameters in the Payment phase.

Below, we give a detailed description of the IOTS primitives appearing in Table 7.3.

- The ***PayInvoke* primitives**[3] are defined for the Consumer to initiate a payment exchange with the Payment Handler. In practice, the *PayInvoke.req* could be initiated by the Consumer clicking on a "Pay Now" button (or, something similar) that is available on the shopping Web site, after the Consumer receives the invoice from the Merchant.

  The three parameters Payment Amount, Payment Instructions and Delivery Type are copied from the *TradingResult.ind* primitive of the preceding Offer phase. The Payment Method Selection parameter is copied from the *TradingOptions.res* primitive of the preceding Offer phase, in the case when the Consumer has thereby replied to the Merchant with the payment method selection. Otherwise, the Consumer should provide the information needed by this parameter before invoking the payment. Payment Number specifies whether it is the first or the second payment exchange to be initiated. This parameter is only present when there are two payment exchanges (e.g., Value Exchange).

[3]This is derived from Sect.9.1.3.2 on pages 206-207 and Sect.8.7 on pages 171-173 of RFC 2801.

Payment Method-specific Data conveys the information of the specific payment method being selected by the Consumer, e.g., the Consumer's Visa card number. The use of this parameter within the *PayInvoke* primitive is *optional*, depending on which payment system is chosen. For example, with a Secure Sockets Layer (SSL)-based system, the Consumer provides the payment card information at the time when invoking the payment exchange. While using a SET-based payment system, the Consumer will not be asked for any payment card information until the payment exchange has been initiated at the Payment Handler side. In this case, the specific payment method information will be carried by the Payment Method-specific Data parameter used within the *PayData* primitives.

- The ***PayData* primitives**[4] are defined for the Payment Handler to request the specific payment method information from the Consumer after the payment exchange has been initiated, and as a result, the Consumer replies with the required information. This constitutes a *payment (method-specific) data exchange*. For example, this could be considered as the Consumer typing in his/her Visa card number and expiration date after receiving a Visa card information inquiry from the Payment Handler.

- The ***PayResult* primitives**[5] are defined for the Payment Handler to notify the Consumer about the payment processing result. In practice, this could be considered as the Consumer being presented with, either a "transaction approved" indication accompanied by an electronic payment receipt, or a payment error indication, such as "insufficient funds", or "incorrect card number".

  The Process State parameter indicates whether the payment has completed successfully or failed. If it has completed successfully, the Payment Receipt parameter must be present to provide the Consumer with proof of payment. Also, Payment Note may be used to convey additional information such as the remaining balance on the Consumer's account after the payment has been conducted. If the payment has failed, the Payment Error parameter should be present to provide the Consumer with information about the failure, so that the Consumer can retry the payment by using the *PayRetry* primitives.

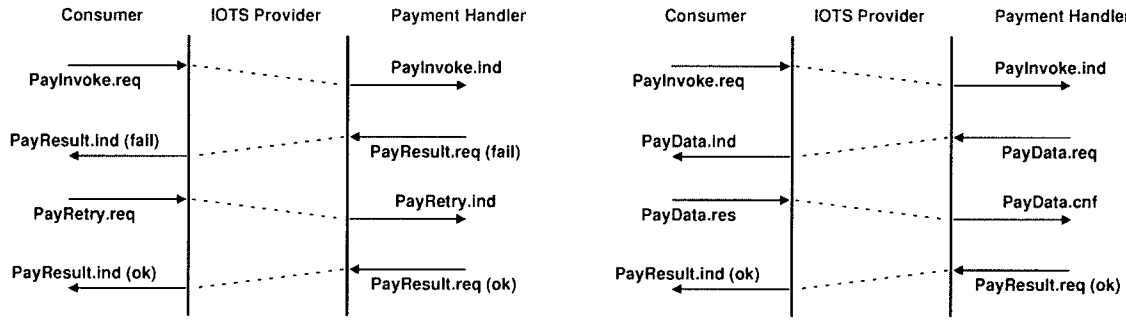
- The ***PayRetry* primitives** are defined for the Consumer to retry the payment with corrected payment method information, according to the payment error indication given by the Payment Error parameter within the preceding *PayResult* primitives. For example, the Consumer could use the American Express card to retry the payment, after receiving an indication indicated that there are insufficient funds associated with the previous Visa card that was tried in the previous attempt.

  Note that the use of the Payment Method Selection parameter is optional. It is only present when a different payment method is selected for the payment retry, e.g., an American Express card is used instead of the previous Visa card. Otherwise, this parameter is omitted if the same payment method is applied, e.g., the Consumer re-enters his/her Visa card number carefully after being notified that the card number entered before was incorrect. However, the Consumer is allowed to select a different payment

[4]This is derived from Sect.9.1.3.3 on page 208 and Sect.8.8 on page 173 of RFC 2801.
[5]This is derived from Sect.9.1.3.4 on pages 208-209 and Sect.8.9 on pages 173-174 of RFC 2801.

method only if the re-selected method does not change the trading result offered by the Merchant. For example, the Consumer cannot change to pay by American Express, after being offered a 5% discount on his/her purchase by selecting Visa card for the payment. Also, the payment method selected for the payment retry is valid only if it is one of the payment options provided by the Merchant during the preceding Offer phase.

Figure 7.3 illustrates two valid sequences of IOTS primitives in the Payment phase. The TSD shown in Figure 7.3 (a) represents the scenario where the Consumer provides specific payment method information, when invoking a payment with the Payment Handler. The payment completes successfully (via *PayResult* primitives followed by *ok*), after the Consumer has retried with the corrected information according to the payment error indication (in *PayResult* primitives followed by *fail*). The TSD shown in Figure 7.3 (b) represents the scenario where the Consumer is asked for a specific payment method (using the *PayData* primitives), after the payment request has been received by the Payment Handler. The payment completes successfully, after the Consumer replied with the required information. In Figure 7.3 (b), only one payment data exchange is considered, represented by the following four correlated primitive events [*PayData.req*, *PayData.ind*, *PayData.res*, *PayData.cnf*]. Different payment systems may require different numbers of payment data exchanges. Figure 7.3 only shows two representative examples of IOTS primitive sequences in the Payment phase.



(a) with a payment retry and no payment data exchange

(b) with a payment data exchange and no payment retry

Figure 7.3: Two valid primitive sequences in the Payment phase.

7.2.3 Delivery Phase

The Purchase transaction is the only trading transaction that involves the delivery of goods or services. Table 7.4 lists the only primitive (*Delivery*) with associated parameters defined for the Delivery phase.

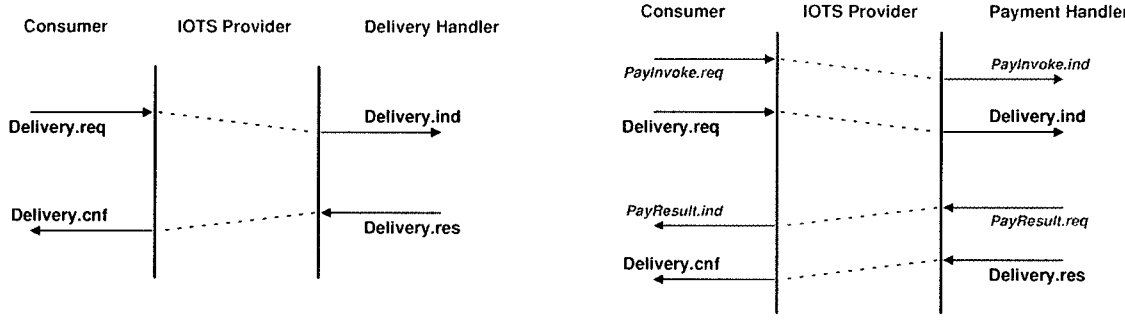
Primitive / *Parameter*	Delivery			
	req	*ind*	*res*	*cnf*
Transaction ID	*M*	*M*(=)	*M*	*M*(=)
Trading Organisation	*M*	*M*(=)		
Order Content	*M*	*M*(=)		
Payment Receipt	*U*	*C*(=)		
Delivery Instructions	*M*	*M*(=)		
Delivery Status			*U*	*C*(=)
Delivery Note			*M*	*M*(=)
Digital Content Data			*U*	*C*(=)

Table 7.4: IOTS primitives and parameters in the Delivery phase.

***Delivery* primitives**[6] are defined for the Consumer to request the delivery of goods or services from the Delivery Handler or the Payment Handler, and as a result, either the electronic goods or services, or the notification of a physical delivery is replied. For example, the Consumer may obtain an electronic journal directly via IOTP, or may receive the notification of a physical delivery (such as buying a DVD player).

The two parameters Order Content and Delivery Instructions are copied from the *TradingResult.ind* primitive of the previous Offer phase into the *Delivery.req* primitive. If the delivery is provided by the Delivery Handler, the Payment Receipt parameter must be present after the completion of the preceding Payment phase and is copied from the *PayResult.ind* primitive. Otherwise, this parameter is absent because the Consumer initiates the delivery with the Payment Handler before the payment result is issued. Delivery Status is used to provide the Consumer with information concerning progress of the delivery, e.g., the goods have been shipped, or are temporarily not available. Delivery Note contains more detailed information about the delivery. For example, it could be a delivery confirmation indicating when, where, and by which means the goods were delivered, or the notification of a deferred delivery caused by temporary unavailability of the goods. The Digital Content Data parameter conveys the content of the electronic goods, such as an electronic journal or a piece of online music.

Figure 7.4 illustrates two valid sequences of IOTS primitives in the Delivery phase. The TSD shown in Figure 7.4 (a) represents the scenario where the delivery is provided by a Delivery Handler. This always occurs after a successfully completed payment exchange between the Consumer and Payment Handler, such as either one shown in Figure 7.3. The TSD shown in Figure 7.4 (b) represents the scenario where the delivery is provided by a Payment Handler and therefore can be combined with the payment exchange. Apart from the *Delivery* primitives, the *PayInvoke* and *PayResult* primitives are also used in Figure 7.4 (b) to show how delivery occurs when combined with the payment exchange. In this case, the Consumer can ask for the delivery directly after invoking the payment exchange, but will not get the delivery response until after the payment exchange has completed.



(a) Delivery provided by the Delivery Handler.

(b) Delivery provided by the Payment Handler.

Figure 7.4: Two valid primitive sequences in the Delivery phase.

7.2.4 Authentication Phase

All trading transactions may involve an Authentication phase, where the service is provided for one trading role (the Authenticator) to verify the identity of the other trading role (the

[6]This is derived from Sect.9.1.4.2 on page 211, Sect.9.1.4.3 on pages 211-212, Sect.8.10 on pages 175-176, and Sect.8.11 on pages 176-177 of RFC 2801.

Authenticatee). The Authentication phase is the only part of an Authentication transaction, which occurs independently of any other trading transaction. It may also be carried out during any non-Authentication transaction, e.g., a Purchase, at any time when authentication service is required. Three primitives are defined to support the services of exchanging data for authentication (*Authentication*), issuing the authentication result (*AuthResult*) and retrying the authentication (*AuthRetry*) if it has failed. Table 7.5 lists the three primitives with associated parameters defined in the Authentication phase.

Primitive / *Parameter*	Authentication				AuthResult		AuthRetry	
	req	*ind*	*res*	*cnf*	*req*	*ind*	*req*	*ind*
Transaction ID	*M*	*M*(=)	*M*	*M*(=)	*M*	*M*(=)	*M*	*M*(=)
Trading Organisation	*M*	*M*(=)						
Transaction Type	*M*	*M*(=)						
Authentication Data			*M*	*M*(=)			*M*	*M*(=)
Process State					*M*	*M*(=)		
Authentication Error					*U*	*C*(=)		

Table 7.5: IOTS primitives and parameters in the Authentication phase.

Below, we give a detailed description of the IOTS primitives appearing in Table 7.5.

- The ***Authentication* primitives**[7] are defined for the Authenticator to request identity information from the Authenticatee. In practice, this may be the Consumer typing in his/her user name and password after receiving an identity inquiry from the Merchant.

  The Transaction Type parameter specifies the trading transaction within which authentication occurs. The Authentication Data parameter conveys the Authenticatee's identity for authentication.

- The ***AuthResult* primitives**[8] allows the Authenticator to notify the Authenticatee of the authentication result. This may be, for example, "authentication successful" indication information or an error indication such as "invalid password".

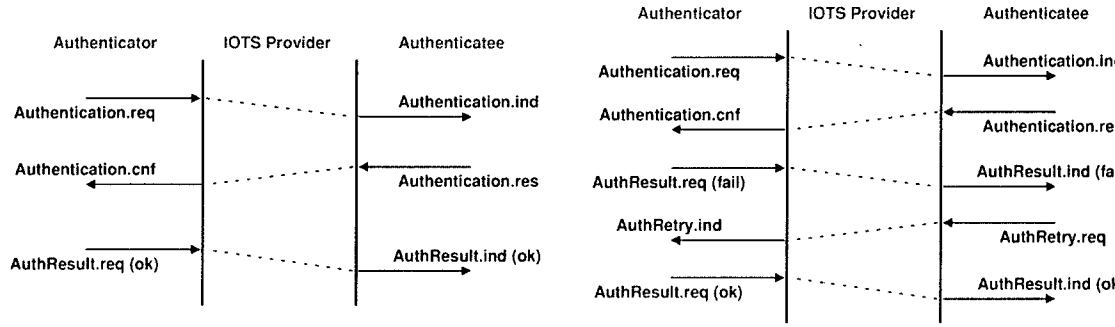
  The Process State parameter indicates whether authentication was successful or not. If successful, the Authentication phase completes at that point. Otherwise, the Authentication Error parameter must be present to inform the Authenticatee of the nature of the error detected, so that the Authenticatee can retry using the *AuthRetry* primitives.

- The ***AuthRetry* primitives** are defined for the Authenticatee to retry authentication with the corrected identity information, according to authentication error indication given in the Authentication Error parameter of the preceding *AuthResult* primitives. For example, this may correspond to the Consumer re-entering the password carefully, after being notified that the password entered before was invalid.

Figure 7.5 illustrates two valid sequences of IOTS primitives in the Authentication phase, both representing successful authentications. The distinction between these two primitive sequences concerns authentication retry, where the corresponding *AuthResult* primitive events attached by *ok* or *fail* indicate a successful or a failed authentication.

[7]This is derived from Sect.9.1.1.2 on page 191, Sect.9.1.1.3 on page 192, Sect.8.4 on pages 169-170, and Sect.8.5 on pages 170-171 of RFC 2801.

[8]This is derived from Sect.9.1.1.4 on page 193 of RFC 2801.



(a) successful authentication on first attempt (b) with authentication retry

Figure 7.5: Two valid primitive sequences in the Authentication phase.

7.2.5 Cancel Phase

A trading transaction may be cancelled at any time, by the involved trading role entities or the IOTS Provider. We consider such transaction cancellation to start the Cancel phase. Table 7.6 lists the only primitive (*Cancel*) with associated parameters defined for the Cancel phase.

Primitive / *Parameter*	Cancel	
	req	*ind*
Transaction ID	M	$M(=)$
Trading Phase	M	$M(=)$
Process State	M	$M(=)$
Cancel Reason	M	$M(=)$

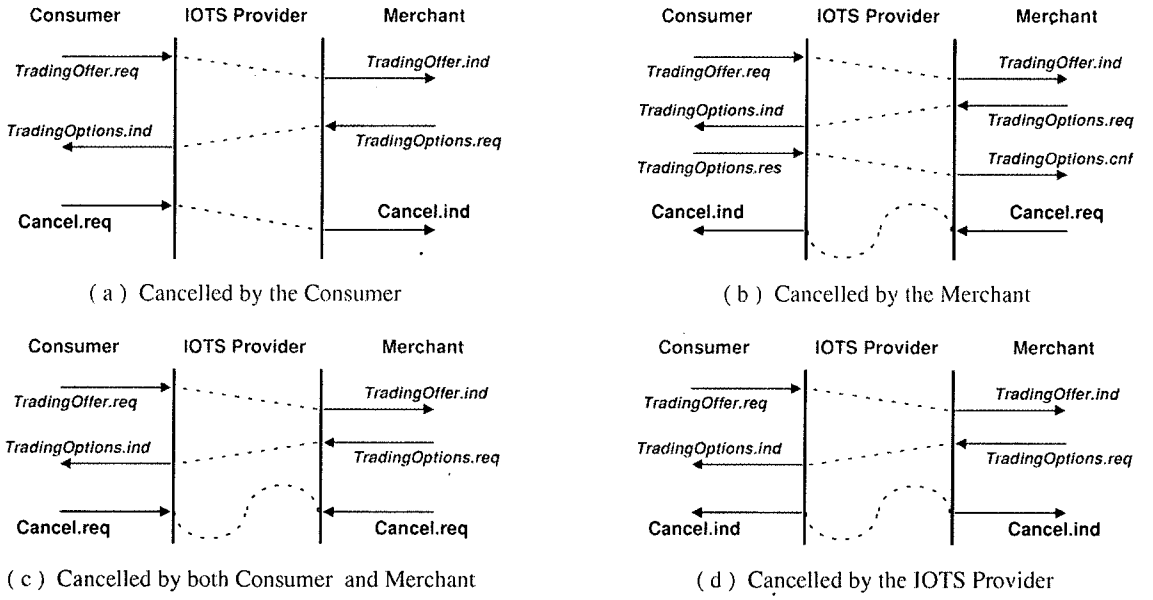
Table 7.6: IOTS primitives and parameters in the Cancel phase.

***Cancel* primitives**[9] are defined for trading roles and the IOTS Provider to cancel a transaction that is in progress. In practice, the Consumer may click on a "Cancel" button to abort an ongoing shopping procedure. For example, the "Cancel" button appearing next to a "Pay Now" button can be used by the Consumer to cancel a Purchase before invoking the payment.

The Trading Phase parameter specifies the trading phase in which cancellation occurs. Process State indicates the current state of the trading phase being carried out, i.e., whether it has completed successfully, failed or is still in progress. Cancel Reason indicates why the transaction is cancelled. This mainly includes the state or error information about that trading phase.

Figure 7.6 illustrates four valid sequences of IOTS primitives for cancelling a transaction within the Offer phase where the Consumer's trading selection is required by the Merchant. Note that these are only four representative examples of the IOTS primitive sequences for transaction cancellation. In Figure 7.6 (a), after receiving a list of trading options from the Merchant, the Consumer cancels the transaction instead of replying with his/her trading selection. In Figure 7.6 (b), on receipt of the Consumer's trading selection, the Merchant cancels the transaction instead of issuing the trading result. Also, the transaction can be cancelled by both Consumer and Merchant, or by the IOTS Provider, as shown in Figure 7.6 (c) and (d). Transaction cancellation by the IOTS Provider may happen when there is an unsolvable transport error or a nonrecoverable transaction timeout.

[9]This is derived from Sect.3.10 on pages 55-56, Sect.4.5.3 on page 70, Sect.4.6.3 on page 74, and Sect.8.18 on page 184 of RFC 2801.



(a) Cancelled by the Consumer

(b) Cancelled by the Merchant

(c) Cancelled by both Consumer and Merchant

(d) Cancelled by the IOTS Provider

Figure 7.6: Four valid primitive sequences of transaction cancellation in the Offer phase.

7.3 IOTS Local Service Languages

In this section, we specify the IOTS that can be observed at each trading role SAP, by defining all possible sequences of IOTS primitives that may occur at each trading role SAP. We refer to this complete set of primitive sequences as the *local service language* for IOTS (or, local IOTS language for short) at that trading role SAP. State transition diagrams (STDs) [150] and primitive sequence tables are used for the specification of local service languages [109, 116, 117]. For the specification of the local IOTS languages, we create a finite state automaton (FSA) for each trading role SAP using STDs. We have not used primitive sequence tables for the specification of the local IOTS languages, because their memory-less nature makes them inadequate for specifying the local IOTS languages.

To handle the complexity of IOTS, we specify the local service languages for IOTS via three steps. Firstly, we specify the local service languages for an Authentication transaction only. Secondly, we specify the local service languages for the trading transactions that do not involve authentication, e.g., a Purchase without authentication. We refer to these transactions as *non-Authentication trading transactions*. Finally, we specify the local service languages for all trading transactions that may involve an authentication at any time when it is required, e.g., a Purchase that starts with the Merchant authenticating the Consumer. We refer to these transactions as *Authentication-combined trading transactions*.

7.3.1 Authentication Transaction

The Authentication transaction service comprises an Authentication trading phase and involves only two trading role entities. Within the scope of IOTP version 1.0, the Authentication transaction can be carried out between a Consumer and any non-Consumer trading role. Either of them can act as the Authenticator to verify the identity of the other acting as the Authenticatee. It therefore suffices to define the local service language for an Authentication transaction at only two trading role SAPs: one corresponding to the Authenticator, and one corresponding to the Authenticatee.

Figure 7.7 shows the two FSA that define the local service language for an Authentication transaction at both of the two trading role SAPs. Each FSA comprises a set of states represented by rectangles with round edges, and a set of transitions represented by arcs labelled by the IOTS primitive events that may occur at that SAP. In Figure 7.7, IDLE is the initial state, and STOP is the acceptance (or final) state as indicated by a thick solid border. The meaning of all of the states is given in Table 7.7. The *Cancel* primitives are distinguished from the others by using dashed arcs, and an arc label comprising two primitives separated by a comma, such as *Cancel.req, Cancel.ind*, indicates either of the primitive events may occur. Each dashed arc labelled by *Cancel* primitives leads from any state (except IDLE and STOP) to the final state STOP. This indicates that an Authentication transaction can be cancelled at any time after it has been initiated and before it terminates at either trading role SAP. An example of an accepted sequence by the FSA in Figure 7.7 (a) is [*Authentication.req, Authentication.cnf, AuthResult.req*] (from states IDLE, AUTor_WAIT, AUTor_PROC to STOP). This sequence corresponds to the trading events occurring at the Authenticator side in the scenario shown in Figure 7.5 (a).

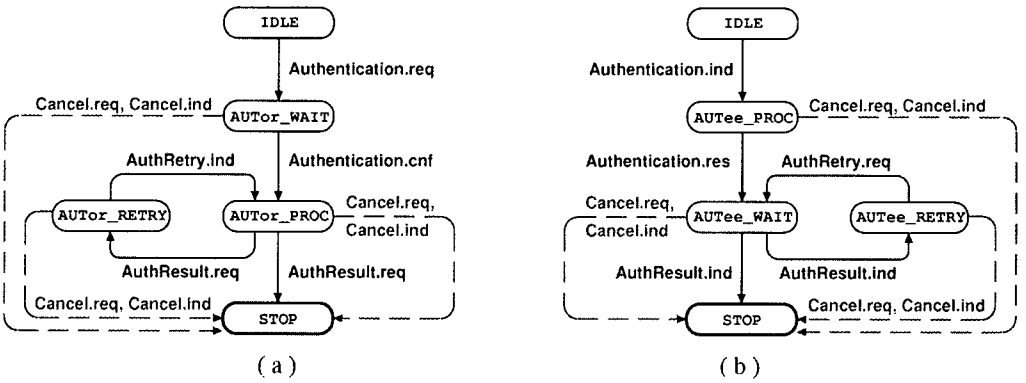


Figure 7.7: FSA defining the local service language for an Authentication transaction at (a) the Authenticator and (b) the Authenticatee trading role.

State		Meaning
prefix	generic	(in the two automata shown in Figure 7.7)
-	IDLE	The initial state reflecting there is no ongoing transaction.
AUTor/AUTee	WAIT	Waiting on the IOTS Provider to deliver an IOTS primitive.
AUTor/AUTee	PROC	Waiting on the local trading role entity to submit an IOTS primitive.
AUTor	RETRY	Waiting on the IOTS Provider to indicate that an authentication retry has been attempted by the peer trading role entity.
AUTee	RETRY	Waiting on the local trading role entity to retry the authentication by submitting an *AuthRetry.req*.
-	STOP	The final state indicating the end of a transaction.

Note: The state prefix, AUTor (or AUTee), stands for an Authentication phase where the local trading role entity acts as the Authenticator (or Authenticatee).

Table 7.7: Meaning of states defined in the FSA in Figure 7.7.

Non-determinism. Both FSA shown in Figure 7.7 are non-deterministic FSA. In Figure 7.7 (a), the *AuthResult.req* primitive occurring in state AUTor_PROC may lead to final state STOP or state AUTor_RETRY, representing a successful or a failed authentication at the Authenticator side. At the Authenticatee side, the *AuthResult.ind* primitive occurring in state AUTee_WAIT may lead to either STOP or AUTee_RETRY in Figure 7.7 (b).

Authentication retry. Both FSA shown in Figure 7.7 contain a loop of primitive events related to authentication retry. In Figure 7.7 (a), the loop of [*AuthResult.req*, *AuthRetry.ind*] (states AUTor_PROC and AUTor_RETRY) represents that the Authenticator may retry requesting the authentication from the Authenticatee a number of times, due to a failed result. In Figure 7.7 (b), the loop consisting of [*AuthResult.ind*, *AuthRetry.req*] represents that the Authenticatee may retry authentication a number times, as requested by the Authenticator. To keep the local IOTS language general, we allow *any finite number* of authentication retries to occur within one Authentication phase.

7.3.2 Non-Authentication Trading Transactions

The non-Authentication trading transaction services supported by IOTP version 1.0 consist of the following five trading transactions without authentication. These are Purchase, Refund, Deposit, Withdrawal and Value Exchange. Each of them is conducted via two or three trading phases and may involve three or four trading role entities. The three trading phases are Offer, Payment and Delivery. The four trading role entities are Consumer, Merchant, Payment Handler and Delivery Handler.

Non-Authentication Trading Transactions at Consumer-SAP

Figure 7.8 shows the FSA that defines the local service language at the Consumer-SAP (C-SAP) for non-Authentication trading transactions without cancellation. To keep the presentation simple, we initially omit all the transaction service cancellation events that may occur at the C-SAP, but we shall take them into account shortly.

The FSA shown in Figure 7.8 comprises a set of states, and a set of IOTS primitive events that may occur at the C-SAP. The meaning of each state is listed in Table 7.8. Among the set of primitive events, the *TradingOffer* primitives are not considered because they are beyond the scope of IOTP version 1.0. In Figure 7.8, all the primitive sequences occurring at the C-SAP hence start with the primitive event *TradingOptions.ind*. Each primitive sequence can be further divided into an Offer, a Payment, and optionally a Delivery phase. Furthermore, the primitive sequences in the Payment phase may represent the first payment exchange, the second payment exchange, or a payment exchange that is combined with delivery. The primitive sequences in the Delivery phase may represent the delivery that is provided by either a Payment Handler or a Delivery Handler. For example, all the primitive sequences leading from state OFR_FIN to the final state STOP in Figure 7.8, represent the payment exchange in a Payment phase and the delivery provided by the Payment Handler in a Delivery phase, where the payment exchange and delivery are combined.

Payment data exchange and payment retry. The FSA shown in Figure 7.8 contains two different loops of primitive events. One is the loop of [*PayData.ind*, *PayData.res*] between states PAY1_WAIT and PAY1_PROC or states PAY2_WAIT and PAY2_PROC. This loop represents a number of payment data exchanges that may occur at the Consumer side, within the Payment phase involving either a first or second payment exchange. The number of payment data exchanges is determined by the specific payment system used. One major design goal of IOTP is to be payment system-independent by encapsulating various payment protocols. Accordingly, we

allow *any finite number* of payment data exchanges to occur, which captures the behaviour of any specific payment system.

The other loop of primitive events in the FSA shown in Figure 7.8 is the loop of [*PayResult.ind*, *PayRetry.req*], between states PAY1_WAIT and PAY1_RETRY, PAY2_WAIT and PAY2_RETRY, DLV_PROC and PAYD_RETRY, or PAYD_WAIT and PAYD_RETRYD. This loop represents that the Consumer may retry the payment for a number of times, due to a failure in payment processing. To keep the local IOTS language general, we allow *any finite number* of payment retries to occur within one Payment phase.

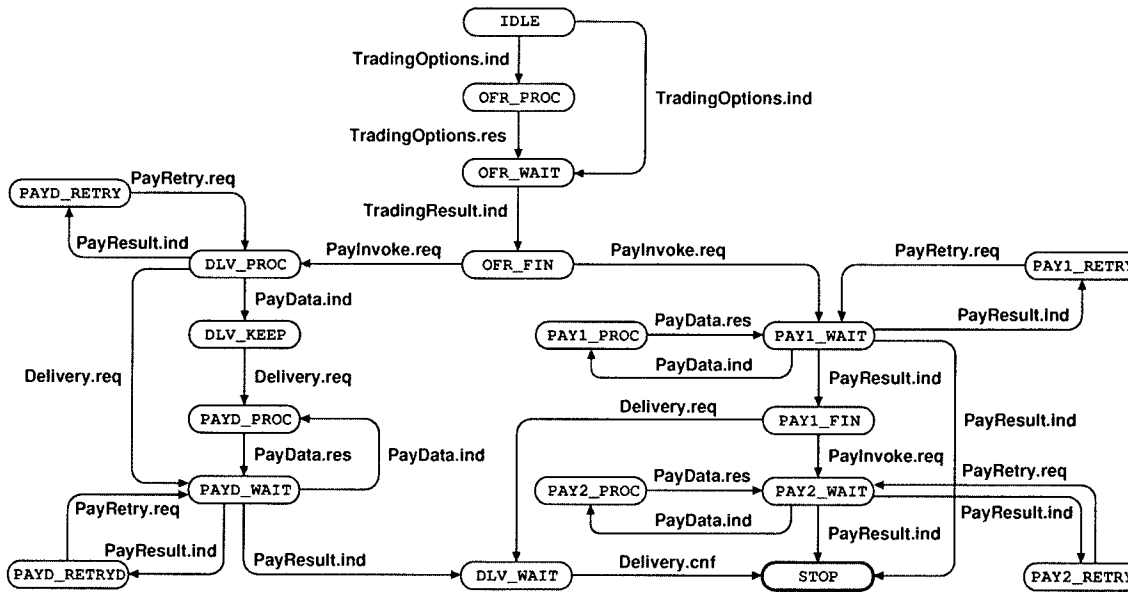


Figure 7.8: FSA for the local service language at the C-SAP for non-Authentication trading transactions without cancellation.

State		Meaning
prefix	**generic**	**(in the FSA shown in Figure 7.8)**
-	IDLE	The initial state reflecting no ongoing transaction at the C-SAP.
OFR/PAY1/PAY2 /PAYD/DLV	WAIT	Waiting on the IOTS Provider to deliver an IOTS primitive, within the trading phase indicated by the state prefix.
OFR/PAY1/PAY2 /PAYD	PROC	Waiting on the Consumer entity to submit an IOTS primitive, within the trading phase indicated by the state prefix.
OFR/PAY1	FIN	Successful completion of the Offer or the Payment phase.
DLV	PROC	Waiting on 1) the Consumer entity to request delivery; or 2) the IOTS Provider to indicate the start of payment data exchange.
DLV	KEEP	To keep waiting on the Consumer entity to request delivery.
PAY1/PAY2	RETRY	Waiting on the Consumer entity to retry the payment.
PAYD	RETRY	Waiting on the Consumer entity to retry the payment, before a *Delivery.req* is submitted.
PAYD	RETRYD	Waiting on the Consumer entity to retry the payment, after a *Delivery.req* has been submitted.
-	STOP	The final state indicating the end of a transaction.

Note: OFR stands for Offer phase, PAY1/PAY2 for a Payment phase where the first/second payment exchange is carried out, DLV for the Delivery phase where the delivery is provided by either the Payment Handler or the Delivery Handler, PAYD for a Payment phase where the payment exchange is combined with delivery.

Table 7.8: Meaning of states defined in the FSA in Figure 7.8.

Cancellation. The primitive events (*Cancel.req, Cancel.ind*), related to non-Authentication trading transaction service cancellation at the Consumer-SAP, can be represented by an arc labelled by *Cancel.req, Cancel.ind* leading from any state (except IDLE and STOP) to the final state STOP. This is similar to the definition of the primitive events related to the cancellation of an Authentication transaction service at one trading role SAP as given in Section 7.3.1. However, adding all the arcs labelled by *Cancel* primitives to the FSA shown in Figure 7.8 would result in a complex diagram. We therefore include the cancellation transitions in a formal definition of the FSA.

Using the definition of FSA given in Section 6.4.1, the FSA shown in Figure 7.8 can be represented by the quintuple FSA_c=(S_c, E_c, Δ_c, IDLE, {STOP}), where S_c is the set of states defined in the FSA, E_c is the set of IOTS primitive events (that may occur at the C-SAP) used in the FSA, IDLE is the initial state in the FSA, {STOP} is a singleton set containing the only final state in the FSA, and Δ_c is the set of labelled arcs representing the occurrences of the IOTS primitive events (at the C-SAP) in the FSA. Each element of Δ_c can be written as a triple (s, e, s')$\in S_c \times E_c \times S_c$, indicating that the local Consumer entity has either submitted or been delivered a primitive event e. For example, in Figure 7.8, the occurrence of the primitive event *TradingOptions.ind* leading from the initial state IDLE to state OFR_PROC can be written as a triple (IDLE, *TradingOptions.ind*, OFR_PROC), indicating the Consumer has been delivered a *TradingOptions.ind* primitive. With the above mathematical representation of the FSA shown in Figure 7.8, we give the following definition of the local service language for non-Authentication trading transactions with cancellation at the C-SAP.

Definition 7.1 (Local Service Language for non-Authentication Trading Transactions at the Consumer-SAP). *Let FSA_c=(S_c, E_c, Δ_c, IDLE, {STOP}) be the FSA shown in Figure 7.8 that defines the local service language at the C-SAP for non-Authentication trading transactions without cancellation. Then, the local service language for non-Authentication trading transactions with cancellation at the C-SAP is defined by the FSA:*

$$\overline{FSA_c}=(S_c,\ \overline{E}_c,\ \overline{\Delta}_c,\ IDLE,\ \{STOP\})$$

where $\overline{E}_c = E_c \cup \{Cancel.req,\ Cancel.ind\}$ *and* $\overline{\Delta}_c = \Delta_c \cup \overline{S}_c \times \{Cancel.req,\ Cancel.ind\} \times \{STOP\}$ *where* $\overline{S}_c = S_c \setminus \{IDLE,\ STOP\}$.

Non-Authentication Trading Transactions at Merchant-SAP

Figure 7.9 shows the FSA that defines the local service language for non-Authentication trading transactions with cancellation at the Merchant-SAP (M-SAP). It comprises a set of states that govern the occurrences of IOTS primitive events at the M-SAP. The meaning of each state is given in Table 7.9. Note that there are two final states: OFR_FIN and STOP, where STOP is used when the transaction service is cancelled.

All the primitive events (except *Cancel* primitives) shown in Figure 7.9 are related to the Offer phase only. Each dashed arc labelled by *both Cancel* primitives (*Cancel.req, Cancel.ind*) leads from either OFR_WAIT or OFR_PROC to the final state STOP. This indicates that either *Cancel* primitive event may occur at the M-SAP, after the transaction has been initiated (IDLE) and before the Offer phase completes (OFR_FIN). Another dashed arc, labelled by the *Cancel.ind* primitive *only*, leads from final state OFR_FIN to STOP. This indicates that, after the Offer phase has completed within an ongoing transaction, the Merchant entity cannot cancel the

transaction service by submitting a *Cancel.req*, but may be notified about the cancellation of that transaction by a *Cancel.ind* occurring later on. Of course, cancellation need not occur, in which case, the FSA remains in `OFR_FIN`. For example, a primitive sequence accepted by the FSA in Figure 7.9 is [*TradingOptions.req, TradingOptions.cnf, Cancel.req*]. This sequence corresponds to the trading events (except the first *TradingOffer.ind* primitive) occurring at the Merchant side in the scenario shown in Figure 7.6 (b), where the Merchant cancels the transaction after having received the Consumer's trading selection.

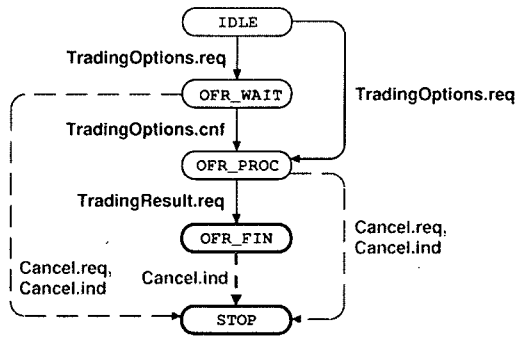


Figure 7.9: FSA for the local service language for non-Authentication trading transactions at the M-SAP.

State	Meaning
IDLE	The initial state reflecting there is no ongoing transaction at the M-SAP.
OFR_WAIT	Waiting on the IOTS Provider to deliver the *TradingOptions.ind*.
OFR_PROC	Waiting on the Merchant entity to submit the *TradingResult.req*.
OFR_FIN	The final state indicating the end of an Offer phase at the M-SAP.
STOP	The final state indicating the transaction has been cancelled.

Table 7.9: Meaning of states defined in the FSA in Figure 7.9.

Non-Authentication Trading Transactions at Payment-Handler-SAP

Figure 7.10 shows the FSA that defines the local service language at the Payment-Handler-SAP (P-SAP) for non-Authentication trading transactions without cancellation. It comprises a set of states governing the primitive events that may occur at the P-SAP. The meaning of each state is listed in Table 7.10.

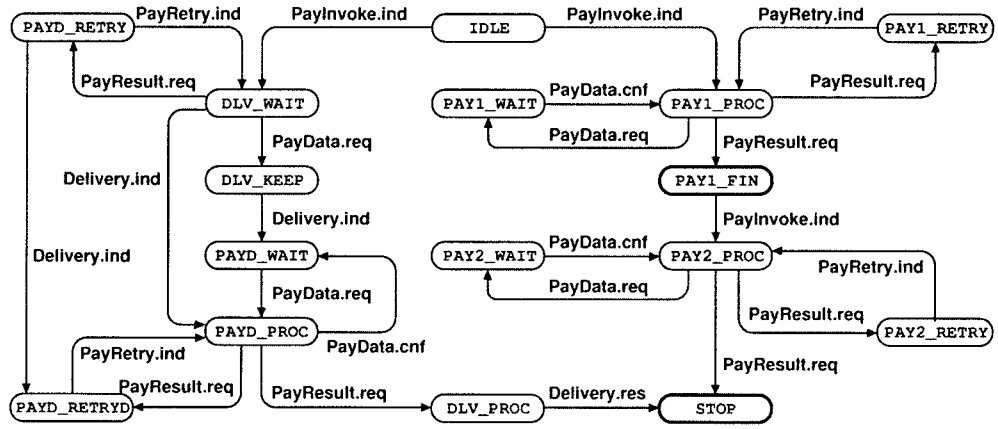


Figure 7.10: FSA for the local service language at the P-SAP for non-Authentication trading transactions without cancellation.

All the primitive events shown in Figure 7.10 are related to a Payment phase involving either a first or second payment exchange, or a Delivery phase involving a delivery provided by the Payment Handler. The primitive event loops related to payment data exchange ([*PayData.req, PayData.cnf*]) and payment retry ([*PayResult.req, PayRetry.ind*]) at the P-SAP, are similar to those that have been defined at the C-SAP. Two examples of the accepted sequences by the FSA in Figure 7.10 are [*PayInvoke.ind, PayData.req, PayData.cnf, PayResult.req*] and

[*PayInvoke.ind, Delivery.ind, PayResult.req, Delivery.res*]. The first one corresponds to the primitive events at the Payment Handler side in the scenario shown in Figure 7.3 (b) involving one payment data exchange and no payment retry. The other represents the primitive events at the Payment Handler side in the scenario shown in Figure 7.4 (b), where the delivery is provided by the Payment Handler.

State		Meaning
prefix	generic	(in the FSA shown in Figure 7.10)
-	IDLE	The initial state reflecting no ongoing transaction at the P-SAP.
PAY1/PAY2/PAYD	WAIT	Waiting on the IOTS Provider to deliver an IOTS primitive, within the trading phase indicated by the state prefix.
PAY1/PAY2/PAYD /DLV	PROC	Waiting on the Payment Handler to submit an IOTS primitive, within the trading phase indicated by the state prefix.
PAY1	FIN	Having successfully completed the first payment exchange and getting ready for the second one.
DLV	WAIT	Waiting on 1) the IOTS Provider to indicate a delivery is requested by the Consumer; or 2) the Payment Handler to request a payment data exchange.
DLV	KEEP	Keep waiting on the IOTS Provider to indicate a delivery request.
PAY1/PAY2	RETRY	Waiting on the IOTS Provider to indicate a payment retry reply.
PAYD	RETRY	Waiting on the IOTS Provider to indicate a payment retry reply, before a *Delivery.ind* is delivered.
PAYD	RETRYD	Waiting on the IOTS Provider to indicate a payment retry reply, after a *Delivery.ind* has been delivered.
-	STOP	The final state indicating the end of a transaction at the P-SAP.

Table 7.10: Meaning of states defined in the FSA in Figure 7.10.

The *Cancel* primitive events occurring at the Payment-Handler-SAP, can be represented by an arc labelled by *Cancel.req, Cancel.ind* leading from any state (except `IDLE`, `PAY1_FIN`, and `STOP`) to state `STOP`. A *Cancel.ind* can occur in `PAY1_FIN` but not a *Cancel.req*. This indicates that, after the first payment exchange has completed within a Value Exchange transaction, the Payment Handler cannot cancel the transaction but can still be notified about the cancellation of that transaction occurring later on (see Section 3.2.6).

To avoid an unreadably complex diagram, we give the following definition of the local service language for non-Authentication trading transactions with cancellation at the P-SAP.

Definition 7.2 (Local Service Language for non-Authentication Trading Transactions at the Payment-Handler-SAP). *Let* $FSA_p=(S_p, E_p, \Delta_p, IDLE, \{STOP\})$ *be the FSA shown in Figure 7.10 that defines the local service language at the P-SAP for non-Authentication trading transactions without cancellation. Then, the local service language for non-Authentication trading transactions with cancellation at the P-SAP is defined by the FSA:*

$$\overline{FSA_p}=(S_p, \overline{E}_p, \overline{\Delta}_p, IDLE, STOP\})$$

where $\overline{E}_p = E_p \cup \{Cancel.req, Cancel.ind\}$, $\overline{\Delta}_p = \Delta_p \cup \overline{S}_p \times \{Cancel.req, Cancel.ind\} \times \{STOP\}$ $\cup \{(PAY1_FIN, Cancel.ind, STOP)\}$ *where* $\overline{S}_p = S_p \backslash \{IDLE, PAY1_FIN, STOP\}$.

Non-Authentication Trading Transactions at Delivery-Handler-SAP

Figure 7.11 shows the FSA that defines the local service language for non-Authentication trading transactions (with cancellation) at the Delivery-Handler-SAP (D-SAP). It comprises a set

of states that controls the occurrences of IOTS primitives at that D-SAP. The meaning of each state is given in Table 7.11. The two primitives (except *Cancel* primitives) shown in Figure 7.11 are related to the Delivery phase only. The dashed arc labelled by both *Cancel* primitives leading from state DLV_PROC to the final state STOP, represents cancellation of delivery occurring at the D-SAP.

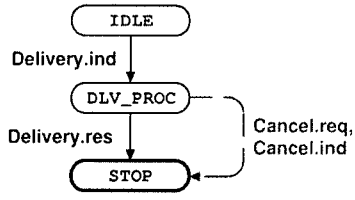


Figure 7.11: FSA for the local service language for non-Authentication trading transactions at the D-SAP.

State	Meaning
IDLE	The initial state reflecting there is no ongoing transaction at the D-SAP.
DLV_PROC	Waiting on the Delivery Handler to respond.
STOP	The final state indicating the end of a transaction at the D-SAP.

Table 7.11: Meaning of states defined in the FSA in Figure 7.11.

7.3.3 Authentication-combined Trading Transactions

Based on the non-Authentication trading transaction services defined at each trading role SAP, we now take into account the authentication(s) that may occur during any of these transactions, namely the Authentication-combined trading transaction services. Within such a transaction service, an Authentication phase is carried out as being combined with a non-Authentication phase, at the time when the authentication is required between the two trading role entities. For example, the Merchant may authenticate the Consumer before providing a list of trading options in the Offer phase, or the Consumer may authenticate the Payment Handler before initiating a payment exchange in the Payment phase.

In brief, to conduct authentication within a non-Authentication trading phase consists of three steps. Firstly, the Authenticator *suspends* the current non-Authentication phase. Secondly, the authentication between these two trading role entities is carried out in an Authentication phase. Finally, after the authentication is completed successfully, the Authenticator *resumes* the suspended original non-Authentication phase.

IOTP version 1.0 supports authentication at *any* point within a non-Authentication transaction[10] . However, the intent of the RFC [36] needs clarification, as we do not believe that authentication is meant to occur between *any* two trading roles at *any* point within a trading transaction. For example, the Payment Handler cannot authenticate the Consumer until a payment exchange has been initiated, nor will the Consumer authenticate the Payment Handler after the payment exchange has completed. We therefore explicitly make the following assumptions for conducting authentication between *certain* trading role entities at *certain* points within a trading transaction. For choosing these certain points when certain trading roles can authenticate each other, we follow a general rule as to determine whether or not such authentication has any practical meaning.

Assumption 7.1 (Conduct of Authentication between Consumer and Merchant in an Offer phase.) *Within a trading transaction, the Merchant can authenticate the Consumer before providing the available trading options at the beginning of the Offer phase, and the*

[10]See Sect 9.1.13 on page 234 of RFC 2801.

Consumer can authenticate the Merchant after receiving the trading options and before replying with the trading selection in the Offer phase.

Assumption 7.2 (Conduct of Authentication between Consumer and Payment Handler in a Payment phase or a Delivery phase.) *Within a trading transaction, the Consumer can authenticate the Payment Handler before invoking the first payment exchange at the beginning of the Payment phase, and the Payment Handler can authenticate the Consumer, either at the time when the first payment exchange is initiated (at the Payment Handler side) in the Payment phase, or at the time when it receives the Consumer's request for delivery (from the Payment Handler) in the Delivery phase.*

Assumption 7.3 (Conduct of Authentication between Consumer and Delivery Handler in a Delivery phase.) *Within a trading transaction, the Consumer can authenticate the Delivery Handler before requesting delivery (from the Delivery Handler) at the beginning of the Delivery phase, and the Delivery Handler can authenticate the Consumer at the time when it receives the Consumer's request for delivery in the Delivery phase.*

Below, we specify the local service language for Authentication-combined trading transactions at each of the four trading role SAPs. It is based on the definitions of local service languages for an Authentication transaction given in Section 7.3.1, and for non-Authentication trading transactions given in Section 7.3.2. Because the local service language for Authentication-combined trading transactions at the C-SAP is the most complicated, we define it at the end of this subsection.

Authentication-combined Trading Transactions at Merchant-SAP

Figure 7.12 shows the FSA that defines the local service language for Authentication-combined trading transactions without cancellation at the M-SAP. States drawn with shaded boxes and primitives labelling dashed arcs are related to authentication occurring at the M-SAP during the Offer phase. With Assumption 7.1, these states and primitives can be divided into two sets.

1. One set (left) containing the states with prefix `AUTor`, represents an Authentication phase in which the Merchant authenticates the Consumer before submitting a *TradingOptions.req* primitive at the beginning of an Offer phase.
2. The other set (right) containing the states with prefix `AUTee`, represents an Authentication phase in which the Merchant is authenticated by the Consumer after submitting the *TradingOptions.req* primitive.

Among the states appearing in the FSA shown in Figure 7.12, both `AUTor_FIN` and `AUTee_FIN` states (drawn with dark shaded boxes) are newly defined to specify that the authentication has completed, and as a result the original Offer phase is resumed.

Note that the FSA shown in Figure 7.12 does not contain final state `STOP`, which is related to cancellation only. To include cancellation while avoiding an unreadable diagram, we give the following definition of the local service language for Authentication-combined trading transactions with cancellation at the M-SAP.

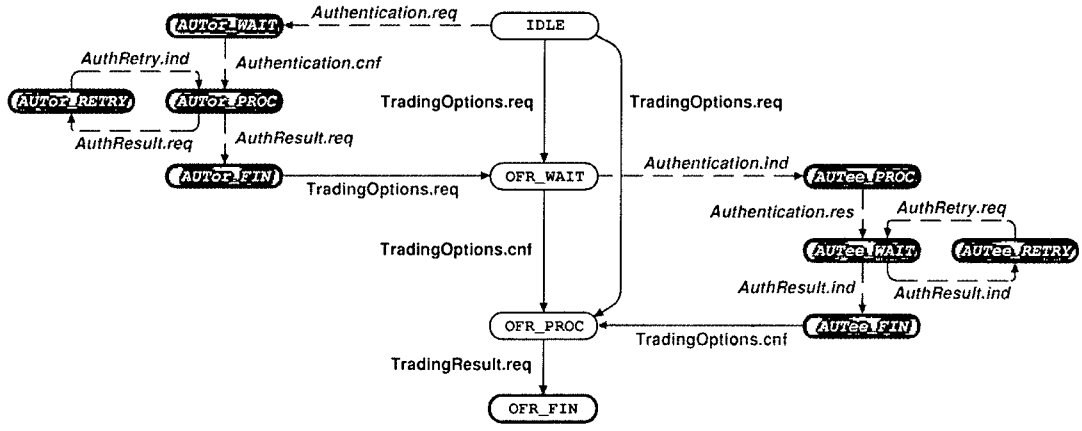


Figure 7.12: FSA for the local service language at the M-SAP for Authentication-combined trading transactions without cancellation.

Definition 7.3 (Local Service Language for Authentication-combined Trading Transactions at the Merchant-SAP). *Let* $FSA_m^A=(S_m^A, E_m^A, \Delta_m^A, IDLE, \{OFR_FIN\})$ *be the FSA shown in Figure 7.12 that defines the local service language at the M-SAP for Authentication-combined trading transactions without cancellation. Then, the local service language for Authentication-combined trading transactions with cancellation at the M-SAP is defined by the FSA:*

$$\overline{FSA_m^A}=(\overline{S_m^A}, \overline{E_m^A}, \overline{\Delta_m^A}, IDLE, \{OFR_FIN, STOP\})$$

where $\overline{S_m^A} = S_m^A \cup \{STOP\}$, $\overline{E_m^A} = E_m^A \cup \{Cancel.req, Cancel.ind\}$ *and* $\overline{\Delta_m^A} = \Delta_m^A \cup (S_m^A \backslash \{IDLE, OFR_FIN\}) \times \{Cancel.req, Cancel.ind\} \times \{STOP\} \cup \{(OFR_FIN, Cancel.ind, STOP)\}$.

Authentication-combined Trading Transactions at Payment-Handler-SAP

Figure 7.13 shows the FSA that defines the local service language for Authentication-combined trading transactions without cancellation at the P-SAP. States drawn with shaded boxes and primitives labelling dashed arcs are related to authentication occurring at the P-SAP during the Payment or the Delivery phase. According to Assumption 7.2, these states and primitives can be divided into three sets.

1. One set (right) represents an Authentication phase in which the Payment Handler is authenticated by the Consumer before a *PayInvoke.ind* primitive occurs at the beginning of the Payment phase. Once the authentication is completed, the Payment phase continues from either the `PAY1_PROC` or the `DLV_WAIT` state.

2. The other two (left) correspond to the two Authentication phases where the Payment Handler authenticates the Consumer within either the Payment or the Delivery phase.

Below, we discuss in detail the two sets of authentication-related states and primitives.

1. The set in the middle represents the Authentication phase in which the Payment Handler authenticates the Consumer after a *PayInvoke.ind* primitive occurs (from state `IDLE` to `AUTor_READY`) in the Payment phase. Once authentication is completed, the Payment phase is resumed from either `PAY1_PROC` or `DLV_WAIT` state.

2. The other set (left) containing the states whose names end with `D`, represents the Authentication phase in which the Payment Handler authenticates the Consumer after a

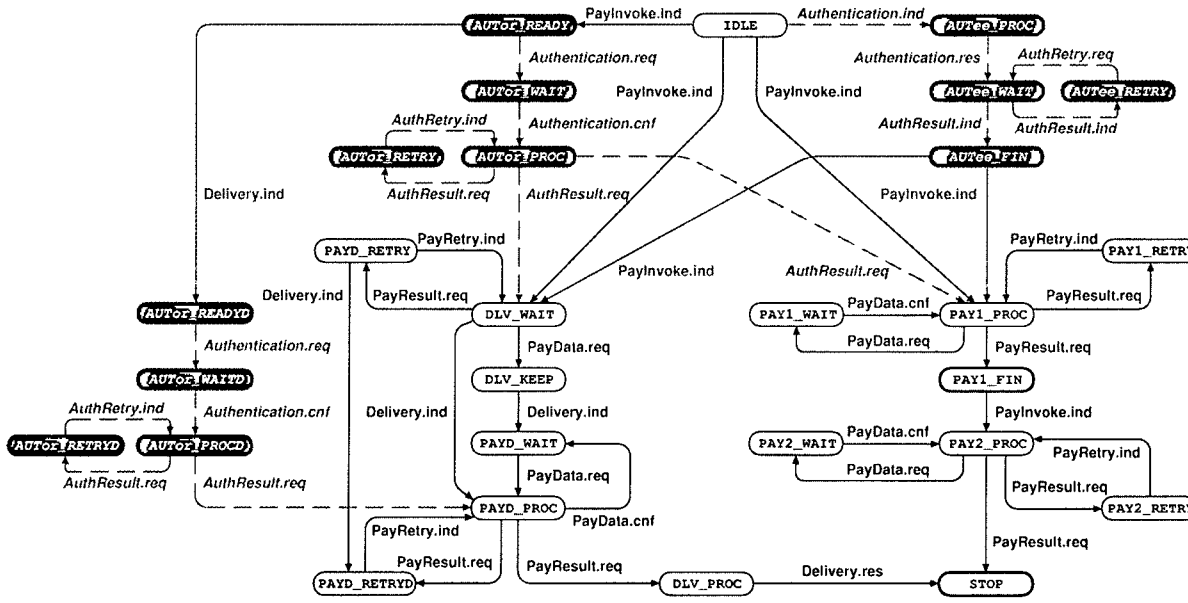


Figure 7.13: FSA for the local service language at the P-SAP for Authentication-combined trading transactions without cancellation.

Delivery.ind primitive occurs (from state AUTor_READY to AUTor_READYD) in the Delivery phase. Once authentication is completed, the Delivery phase is resumed from state PAYD_PROC.

From Figure 7.13, we can see that only one of the above two sets of primitive events can occur at the P-SAP within one trading transaction. This indicates that the Payment Handler authenticates the Consumer only once, upon being delivered either a *PayInvoke.ind* or a *Delivery.ind* primitive.

In order to include cancellation, we give the following definition of the local service language for Authentication-combined trading transactions with cancellation at the P-SAP, based on the FSA shown in Figure 7.13.

Definition 7.4 (Local Service Language for Authentication-combined Trading Transactions at the Payment-Handler-SAP). *Let* $FSA_p^A=(S_p^A, E_p^A, \Delta_p^A, IDLE, \{STOP\})$ *be the FSA shown in Figure 7.13 that defines the local service language at the P-SAP for Authentication-combined trading transactions without cancellation. Then, the local service language for Authentication-combined trading transactions with cancellation at the P-SAP is defined by the FSA:*

$$\overline{FSA_p^A}=(S_p^A, \overline{E_p^A}, \overline{\Delta_p^A}, IDLE, \{STOP\})$$

where $\overline{E_p^A} = E_p^A \cup \{Cancel.req, Cancel.ind\}$, $\overline{\Delta_p^A} = \Delta_p^A \cup \overline{S_p^A}\times\{Cancel.req, Cancel.ind\}\times\{STOP\}$ $\cup \{(PAY1_FIN, Cancel.ind, STOP)\}$ *where* $\overline{S^A}_p = S_p^A \backslash \{IDLE, PAY1_FIN, STOP\}$.

Authentication-combined Trading Transactions at Delivery-Handler-SAP

Figure 7.14 shows the FSA that defines the local service language for Authentication-combined trading transactions without cancellation at the D-SAP. States drawn with shaded boxes and primitives labelling dashed arcs are related to authentications occurring at the D-SAP during the Delivery phase. According to Assumption 7.3, these states and primitives can be divided into two sets. One set (right) represents an Authentication phase in which the Delivery Handler is authenticated by the Consumer before a *Delivery.ind* primitive occurs at the beginning

of the Delivery phase. The other (left) represents an Authentication phase in which the Delivery Handler authenticates the Consumer after a *Delivery.ind* primitive occurs and before submitting a *Delivery.res* primitive.

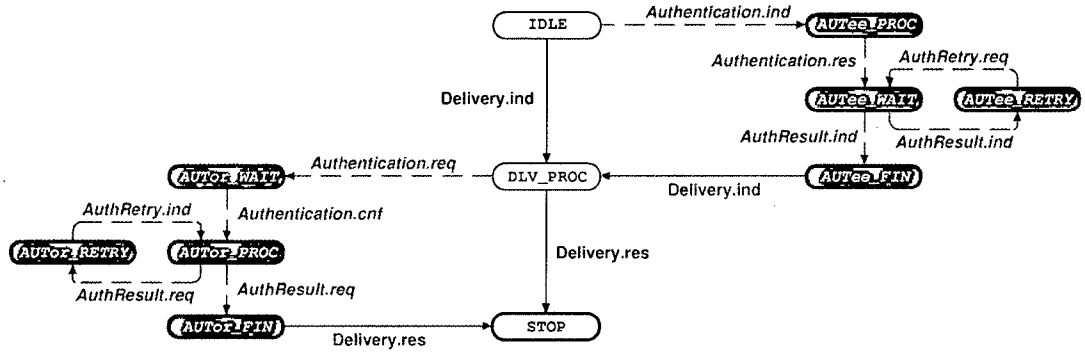


Figure 7.14: FSA for the local service language at the D-SAP for Authentication-combined trading transactions without cancellation.

Based on the FSA in Figure 7.14, we give the following definition of the local service language at the D-SAP for Authentication-combined trading transactions to include cancellation.

Definition 7.5 (Local Service Language for Authentication-combined Trading Transactions at the Delivery-Handler-SAP). *Let* $FSA_d^A=(S_d^A,\ E_d^A,\ \Delta_d^A,\ IDLE,\ \{STOP\})$ *be the FSA shown in Figure 7.14 that defines the local service language at the D-SAP for Authentication-combined trading transactions without cancellation. Then, the local service language for Authentication-combined trading transactions with cancellation at the D-SAP is defined by the FSA:*

$$\overline{FSA_d^A}=(S_d^A,\ \overline{E_d^A},\ \overline{\Delta_d^A},\ IDLE,\ \{STOP\})$$

where $\overline{E_d^A} = E_d^A \cup \{Cancel.req,\ Cancel.ind\}$, $\overline{\Delta_d^A} = \Delta_d^A \cup \overline{S_d^A}\times\{Cancel.req,\ Cancel.ind\}\times\{STOP\}$ *where* $\overline{S_d^A} = S_d^A \backslash \{IDLE,\ STOP\}$.

Authentication-combined Trading Transactions at Consumer-SAP

Figure 7.15 shows the FSA that defines the local service language for Authentication-combined trading transactions without cancellation at the C-SAP. States drawn with shaded boxes and primitives labelling dashed arcs are related to authentications occurring at the C-SAP during the Offer, Payment, or Delivery phase. According to Assumption 7.1, 7.2 and 7.3, these states and primitives can be divided into the following seven sets.

1. The set (top left) containing the states with prefix `AUTee` and suffix `1`, represents an Authentication phase in which the Consumer is authenticated by the Merchant before a *TradingOptions.ind* primitive occurs at the beginning of the Offer phase.

2. The set (top right) containing the states with prefix `AUTor` and suffix `1`, represents an Authentication phase in which the Consumer authenticates the Merchant after the occurrence of *TradingOptions.ind* but before submitting *TradingOptions.res.*

3. The set containing the states with prefix `AUTor` and suffix `2`, represents an Authentication phase in which the Consumer authenticates the Payment Handler before initiating the first payment exchange.

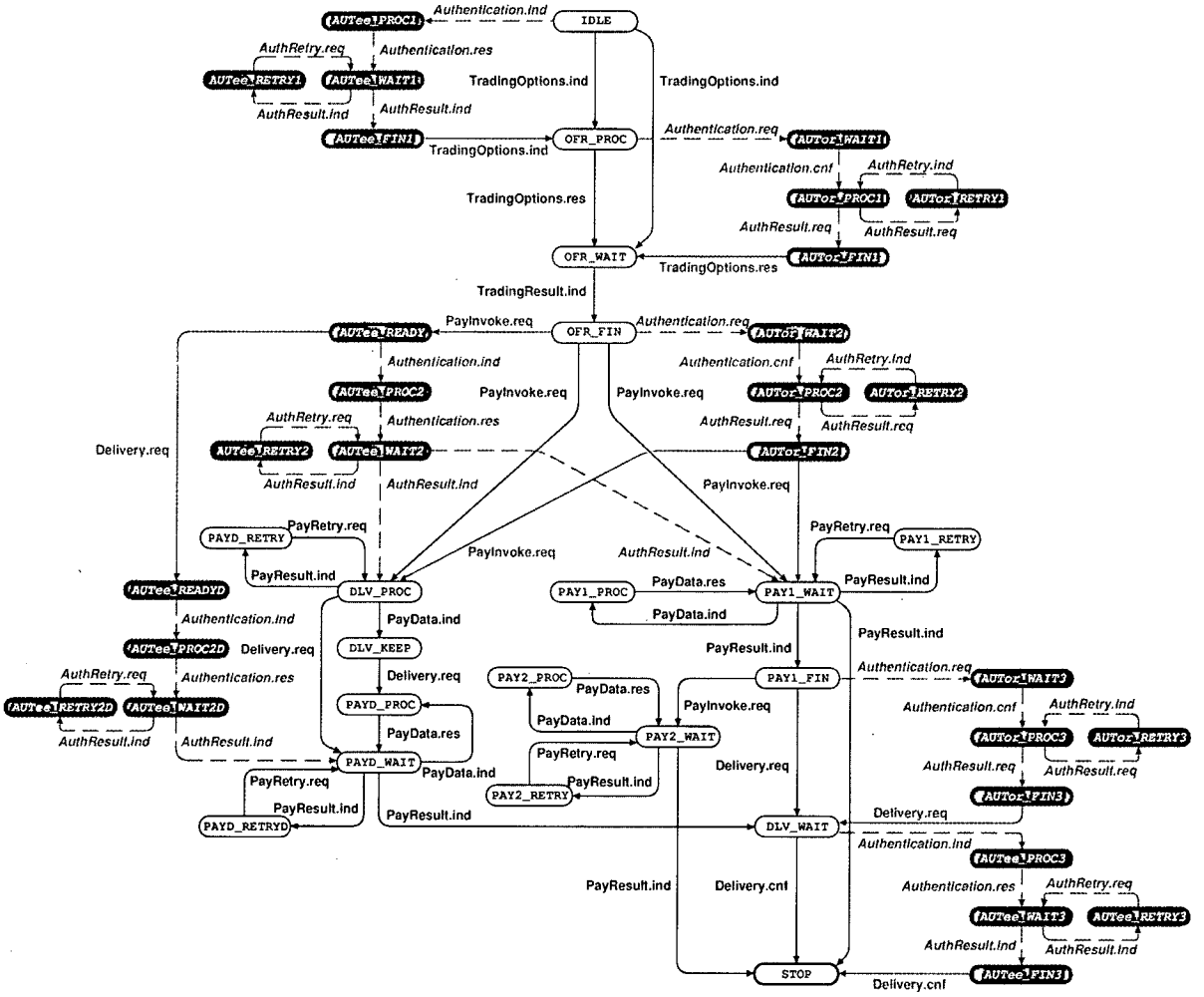


Figure 7.15: FSA for the local service language at the C-SAP for Authentication-combined trading transactions without cancellation.

4. The set containing the state `AUTee_READY` and the states with prefix `AUTee` and suffix 2, represents an Authentication phase in which the Consumer is authenticated by the Payment Handler after submitting *PayInvoke.req* but before submitting *Delivery.req*.

5. The set (bottom left) containing the state `AUTee_READYD` and the states with prefix `AUTee` and suffix `2D`, represents an Authentication phase in which the Consumer is authenticated by the Payment Handler after submitting both *PayInvoke.req* and *Delivery.req*.

6. The set containing the states with prefix `AUTor` and suffix 3, indicates an Authentication phase where the Consumer authenticates the Delivery Handler before requesting delivery.

7. The set (bottom right) containing the states with prefix `AUTee` and suffix 3, represents an Authentication phase in which the Consumer is authenticated by the Delivery Handler after submitting *Delivery.req* but before the occurrence of *Delivery.cnf*.

Compared to the FSA in Figure 7.8, the FSA in Figure 7.15 has another new state `PAY_FIN` apart from the above authentication-related states. For brevity, the two FSA are denoted by FSA 7.8 and FSA 7.15, correspondingly. In FSA 7.15, the state `PAY_FIN` represents the successful completion of the Payment phase in a Purchase transaction at the C-SAP, while `PAY1_FIN` indicates the successful completion of the first payment exchange in the Payment

phase for a Value Exchange transaction. In FSA 7.8, only PAY1_FIN is used and has the meaning of both PAY_FIN and PAY1_FIN in FSA 7.15. The reason for distinguishing between PAY_FIN and PAY1_FIN in FSA 7.15 is that an authentication can occur in PAY_FIN but not in PAY1_FIN (Assumptions 7.2 and 7.3).

Based on the FSA shown in Figure 7.15, we finally give the following definition of the local service language for Authentication-combined trading transactions to include cancellation at the C-SAP.

Definition 7.6 (Local Service Language for Authentication-combined Trading Transactions at the Consumer-SAP). *Let* $FSA_c^A=(S_c^A,\ E_c^A,\ \Delta_c^A,\ IDLE,\ \{STOP\})$ *be the FSA shown in Figure 7.15 that defines the local service language at the C-SAP for Authentication-combined trading transactions without cancellation. Then, the local service language for Authentication-combined trading transactions with cancellation at the C-SAP is defined by the FSA:*

$$\overline{FSA_c^A}=(S_c^A,\ \overline{E_c^A},\ \overline{\Delta_c^A},\ IDLE,\ \{STOP\})$$

where $\overline{E_c^A} = E_c^A \cup \{Cancel.req,\ Cancel.ind\}$, $\overline{\Delta_c^A} = \Delta_c^A \cup \overline{S_c^A}\times\{Cancel.req,\ Cancel.ind\}\times\{STOP\}$ *where* $\overline{S_c^A} = S_c^A\backslash\{IDLE,\ STOP\}$.

Chapter 8

Internet Open Trading Service CPN

In this chapter, we specify the global behaviour of IOTS based on its local behaviour at each trading role entity as defined in Chapter 7. Following the service specification procedure outlined in Chapter 5, a CPN model is developed to synthesize these local automata into a specification of the global service, which captures the correlations between primitive events at different trading role entities. Related work on formal service specification includes the OSI transport service [20], the Wireless Application Protocol (WAP) transaction service [89], and the Resource Reservation Protocol (RSVP) service [236]. However, IOTS involve *four* trading role users, which means that it is more complicated than the above services, each involving only *two* user entities.

Section 8.1 discusses three conditions under which the IOTS CPN model is developed. Section 8.2 summarizes the modelling decisions about how to synthesize the local automata into a global IOTS specification. An overview of the IOTS CPN model is given in Section 8.3, and the modelling details are described in Section 8.4. An early version of the IOTS specification, which did not involve any authentication service, can be found in [182].

8.1 General Conditions

The IOTS CPN model captures the global behaviour of all six Authentication and trading transaction services. It is developed based on the following approach.

A Single Transaction

IOTP does not define interactions between any two Payment-related transactions (i.e., trading transactions) that are in progress. For example, a Refund may be required as result of an earlier Purchase, but cannot occur until the Purchase is complete. Only an Authentication transaction may be carried out within an ongoing Payment-related transaction. However, according to the IOTS definition in Chapter 7, the authentication service within a trading transaction is conducted as an Authentication phase instead of an Authentication transaction. Hence, all transactions in IOTS are independent of each other, and it suffices to develop a CPN model of IOTS that considers each of the six transactions in isolation.

Communication between User Entities

In IOTP version 1.0, no communication takes place between non-Consumer entities. Therefore, the communication in IOTS is always between a Consumer and a non-Consumer entity.

Service Data Units (SDUs)

Communication via the service provider is by means of SDUs. Since IOTP is currently designed to operate over the HyperText Transfer Protocol (HTTP) [77], we assume within the IOTS Provider a reliable communication medium with no loss, no duplication and no re-ordering. Modelling of all the SDUs transferred via the IOTS Provider is described in Section 8.3.3.

8.2 Modelling Decisions

The purpose of creating the IOTS CPN model is to determine the global sequences of service primitives, based on the definitions of local primitive occurrences at each of the four trading role SAPs. In this section, we summarise our modelling decisions when synthesizing the local automata into a specification of global IOTS. For brevity, the FSA specifying the local service language of an Authentication-combined trading transaction at a trading role SAP in Section 7.3.3 is hereafter referred to as the *local SAP FSA*.

Synchronization of End-to-End Behaviour

Each local SAP FSA involves service primitives but not their parameters. It uses *non-determinism* to model the occurrences of certain primitives, abstracting from the parameter values conveyed by those primitives. For example, in the local M-SAP FSA (see Figure 7.12), the *TradingOptions.req* primitive labelling the arc from states `IDLE` to `OFR_WAIT` is followed by a *TradingOptions.cnf* and then a *TradingResult.req*, indicating the Selection Reply parameter conveyed by this primitive is set to Yes. Whereas the *TradingOptions.req* labelling the arc from `IDLE` to `OFR_PROC` is directly followed by a *TradingResult.req*, indicating that the Selection Reply with this primitive has a value of No. Similarly in the local C-SAP FSA (see Figure 7.15), the *TradingOptions.ind* from `IDLE` to `OFR_PROC` conveys a Selection Reply set to Yes, while the one from `IDLE` to `OFR_WAIT` conveys a Selection Reply with value No.

To synthesize the local SAP FSA at each of the four trading roles, a simple and direct way is to create CPN models of the four FSA and then connect the CPN models via a model of the IOTS Provider in which the SDUs are transferred. However, due to the non-determinism used in each local SAP FSA, the above direct synthesis of these FSA cannot guarantee the synchronized end-to-end behaviour of trading role user entities, and hence may result in some illegal primitive sequences in the global service specification. For example, we consider the valid *TradingOptions* primitive occurrences at both the Consumer and Merchant (see Figure 7.2). A *TradingOptions.req* with Selection Reply set to Yes or No at the M-SAP can only result in a *TradingOptions.ind* conveying the same parameter value at the C-SAP. Only if Selection Reply is Yes will a *TradingOptions.res* occur at the C-SAP followed by a *TradingOptions.cnf* at the M-SAP. Otherwise, with Selection Reply set to No, a *TradingResult.req* at the M-SAP followed by a *TradingResult.ind* at the C-SAP can occur directly. However, without considering the Selection Reply parameter value, one cannot guarantee the consistent occurrences of a *TradingOptions.req* at the M-SAP and the corresponding *TradingOptions.ind* at the C-SAP, let alone the valid occurrences of the following primitive events.

We use the term *significant parameters* to refer to those IOTS parameters that affect the sequencing of primitives and thereby the global service language, e.g., the parameter Selection

Reply conveyed by a *TradingOptions.req* or a *TradingOptions.ind* primitive. In the IOTS CPN model, not only the service primitives are taken into account, but also their significant parameters are modelled explicitly, so that the synchronization of end-to-end behaviour between trading role user entities can be guaranteed. Section 8.3.3 describes all significant parameters modelled in the IOTS CPN model.

Simplification of Trading Role Interface States

Each local SAP FSA defines a set of states, which provide the basis for modelling the interface states of the corresponding trading role in the IOTS CPN model. With other information in the CPN model (e.g., the significant parameter value(s) and the SDU(s) being transferred in the IOTS Provider), the trading role interface states can be simplified rather than one-to-one copied from those defined in the local SAP FSA. For example, Payment Number, as one of the significant parameters in the CPN model, indicates whether a payment is the first or the second payment in the Payment phase. Given Payment Number, the states with prefix PAY can replace the states with prefix PAY1 or PAY2 in the local C-SAP FSA (Figure 7.15) or local P-SAP FSA (Figure 7.13). This reduces the number of the payment-related states in the local SAP FSA. Section 8.3.3 defines the trading role interface states used in the IOTS CPN model.

8.3 Overview of the IOTS CPN

8.3.1 Net Structure

We model IOTS taking advantage of the hierarchical structuring mechanism of CPNs. Figure 8.1 shows the hierarchy page of the IOTS CPN model, which consists of 16 pages including the Declaration page. The top level IOTS_Global has five subpages: Offer, Payment, Delivery, Authentication and Cancel. These pages correspond to the five trading phases defined in IOTS and are therefore referred to as *trading phase pages*. The page Authentication is further decomposed into six subpages, each corresponding to an Authentication phase where a Consumer entity authenticates or is authenticated by one of the other three non-Consumer entities. Similarly, the page Cancel is further decomposed into three subpages, each corresponding to a Cancel phase where the cancellation occurs between a Consumer and a non-Consumer entity. Each subpage of page Authentication or Cancel is named in such a way that: 1) the first three or four letters (Cnl or Auth) indicates a Cancel or an Authentication phase; and 2) the last two letters are both initial letters of the two trading role entities engaged in that phase. For Auth pages, the two initial letters are given in an order such that: 1) the first one specifies the

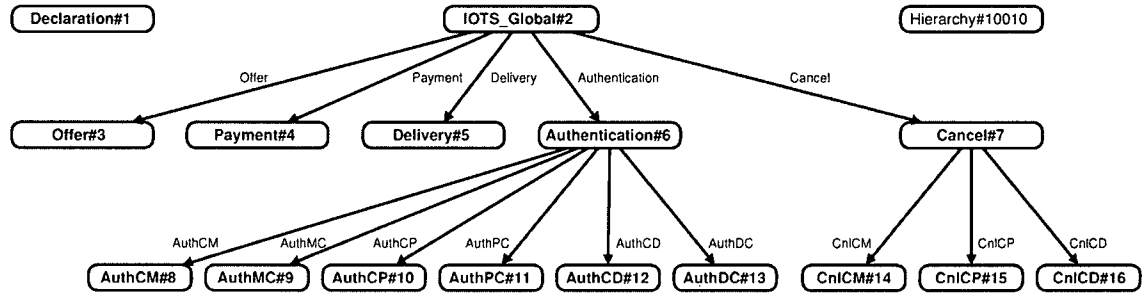


Figure 8.1: Hierarchy page for IOTS CPN.

trading role which acts as the Authenticator; and 2) the second one specifies the Authenticatee trading role. The Auth and Cnl pages are also trading phase pages.

8.3.2 The Top Level

The top level page IOTS_Global, as shown in Figure 8.2, has four places and five substitution transitions. The place C_SAP is used to model the C-SAP where the local sequences of primitives at the Consumer side are viewed. Similarly, the other three places M_SAP, P_SAP and D_SAP represent the M-SAP, P-SAP and D-SAP, respectively. We hereafter call them *SAP places*. Each SAP place has its own colour set modelling the states of the interface as observed at that trading role SAP. The definitions of these four colour sets are given in Section 8.3.3. Each substitution transition on page IOTS_Global models one of the five trading phases in IOTS, and has its associated subpage that is named accordingly. The arcs connecting the four SAP places and the five substitution transitions indicate, in an abstract way, the involvement of each trading role entity within each trading phase. For example, it can be seen that the Consumer is the only trading role that is involved in all trading phases. As another example, both the Payment Handler and Delivery Handler are modelled in the Delivery page, indicating that either of them may be involved in a Delivery phase.

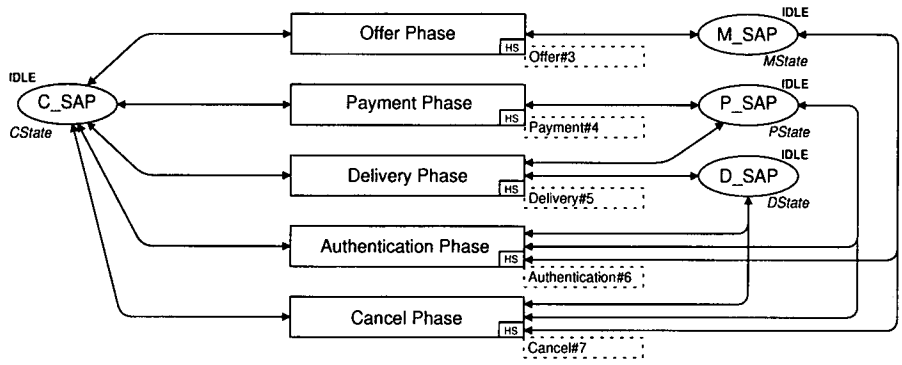


Figure 8.2: IOTS_Global page in the IOTS CPN model.

8.3.3 Global Declarations

The global declarations for the IOTS CPN model can be divided into three parts. The first part defines the colour sets and variables used to model the states of the interface at each trading role SAP (or trading role interface for short). The second part specifies the colour sets and variables related to the significant parameters and the SDUs transferred within the IOTS Provider. The third part contains a set of functions for modelling transaction service cancellation. This will be introduced when we describe the Cancel page and its three subpages in Section 8.4.6.

States of the Trading Role Interfaces

Table 8.1 lists the definition of four colour sets and variables for modelling the states of the trading role interface. Colour set State (line 1 to 12) contains the states of all trading role interfaces. It has four subsets: CState (line 13 to 23), MState (line 24 to 27), PState (line 28 to 32), and DState (line 33 to 36). The four variables sc, sm, sp, and sd (line 37) are used to select states non-deterministically.

```
1  color State = with IDLE | STOP |
2                    OFR_PROC | OFR_WAIT | OFR_FIN |
3                    PAY_PROC | PAY_WAIT | PAY_RETRY | PAY_RETRYD | PAY1_FIN | PAY_FIN |
4                    DLV_PROC | DLV_WAIT | DLV_KEEP |
5                    AUTor_READY | AUTor_READYD |
6                    AUTor_PROC | AUTor_PROC2 | AUTor_PROCD | AUTor_PROC3 |
7                    AUTor_WAIT | AUTor_WAIT2 | AUTor_WAITD | AUTor_WAIT3 |
8                    AUTor_FIN | AUTor_FIN2 | AUTor_FIN3 |
9                    AUTee_PROC | AUTee_PROC2 | AUTee_PROCD | AUTee_PROC3 |
10                   AUTee_WAIT | AUTee_WAIT2 | AUTee_WAITD | AUTee_WAIT3 |
11                   AUTee_RETRY | AUTee_RETRY2 | AUTee_RETRYD | AUTee_RETRY3 |
12                   AUTee_FIN;
13 color CState = subset State with [ IDLE, STOP,
14                   OFR_PROC, OFR_WAIT, OFR_FIN,
15                   PAY_PROC, PAY_WAIT, PAY_RETRY, PAY_RETRYD, PAY1_FIN, PAY_FIN,
16                   DLV_PROC, DLV_WAIT, DLV_KEEP,
17                   AUTor_PROC, AUTor_PROC2, AUTor_PROC3,
18                   AUTor_WAIT, AUTor_WAIT2, AUTor_WAIT3,
19                   AUTor_FIN, AUTor_FIN2, AUTor_FIN3,
20                   AUTee_PROC, AUTee_PROC2, AUTee_PROCD, AUTee_PROC3,
21                   AUTee_WAIT, AUTee_WAIT2, AUTee_WAITD, AUTee_WAIT3,
22                   AUTee_RETRY, AUTee_RETRY2, AUTee_RETRYD, AUTee_RETRY3,
23                   AUTee_FIN ];
24 color MState = subset State with [ IDLE, STOP,
25                   OFR_PROC, OFR_WAIT, OFR_FIN,
26                   AUTor_PROC, AUTor_WAIT, AUTor_FIN,
27                   AUTee_PROC, AUTee_WAIT, AUTee_RETRY ]
28 color PState = subset State with [ IDLE, STOP,
29                   PAY_PROC, PAY_WAIT, PAY_RETRY, PAY_RETRYD, PAY1_FIN,
30                   DLV_PROC, DLV_WAIT, DLV_KEEP,
31                   AUTor_READY, AUTor_READYD,
32                   AUTee_PROC, AUTee_WAIT, AUTee_RETRY, AUTee_FIN ]
33 color DState = subset State with [ IDLE, STOP,
34                   DLV_PROC, DLV_WAIT,
35                   AUTor_PROC, AUTor_WAIT, AUTor_FIN,
36                   AUTee_PROC, AUTee_WAIT, AUTee_RETRY, AUTee_FIN ]
37 var sc: CState;   var sm: MState;   var sp: PState;   var sd: DState;
```

Table 8.1: Colour sets and variables for modelling states of the interface at each trading role SAP.

Each of the four colour sets CState, MState, PState, and DState specifies the possible states of the interface at the corresponding trading role SAP. All of them are based on those defined for the local SAP FSA (Section 7.3.3), and the meaning of these interface states is given there. As mentioned in Section 8.2, some of the interface states are simplified in the IOTS CPN model. For example, the colour set CState contains *36* states, whereas the FSA shown in Figure 7.15 has *48* states. The details of how we simplify the interface states at each trading role SAP in the IOTS CPN will be given when the relevant pages in the CPN model are described.

Significant Parameters and SDUs

Table 8.2 lists the definition of the colour sets and variables related to the significant parameters and the SDUs used within IOTS. The first six colour sets define the six significant parameters that affect the sequencing of primitives. The valid parameter values for each of these six parameters are defined in Appendix B.

In Table 8.2, the colour set TransType (line 1) is used for the Transaction Type parameter, which defines the five trading transactions. The Authentication transaction service involves only two trading roles, and has an Authentication phase and/or a Cancel phase. It can be

```
    (* Significant Parameters *)
 1  color TransType = with Purchase | Refund | Deposit | Withdrawal | ValueExchange;
 2  color SelReply = with Yes | No;
 3  color PayType = with one | two;
 4  color PayNumber = with NA | Pay1 | Pay2;
 5  color DlvType = with ndlv | pdlv | ddlv;
 6  color ProState = with CompletedOk | Failed;
 7  color TTxSR = product TransType * SelReply;
 8  color PTxDT = product PayType * DlvType;
 9  color PNxDT = product PayNumber * DlvType;
10  var t: TransType;   var r: SelReply;   var p: PayType;   var n: PayNumber;
11  var d: DlvType;   var ps: ProState;

    (* Service Data Units *)
12  color SDUs = union trdoptions:TTxSR + trdselection + trdresult:PTxDT +
13                     payrequest:PNxDT + paydata + payresponse:ProState + payretry +
14                     dlvrequest + dlvresponse +
15                     authrequest + authresponse + authresult:ProState + authretry +
16                     cancel;
17  color SDUQ = list SDUs;
18  var x: SDUs;     var q: SDUQ;
```

Table 8.2: Colour sets and variables for modelling significant parameters and SDUs.

captured by considering only an Auth page and a Cnl page for the same pair of trading roles, e.g., the AuthMC and CnlCM pages, when executing the CPN model. The colour set SelReply (line 2) is used for the Selection Reply parameter, and has two values Yes and No indicating whether or not the reply of the Consumer's trading selection is required by the Merchant. The colour set PayType (line 3) is used for the Payment Type parameter, indicating whether one or two payments are involved. The colour set PayNumber (line 4) is used for the Payment Number parameter, indicating whether a payment is the first (Pay1) or the second (Pay2) in case there are two involved, otherwise this is not applicable (NA). The colour set DlvType (line 5) is used for the Delivery Type parameter, indicating whether the transaction service involves no delivery (ndlv), a delivery provided by the Payment Handler (pdlv), or a delivery provided by the Delivery Handler (ddlv). The colour set ProState (line 6) is used for the Process State parameter, indicating whether the process being carried out within a trading phase is successful (CompletedOk) or has failed (Failed). The next three colour sets TTxSR (line 7), PTxDT (line 8), and PNxDT (line 9) are all defined as a *product* type, modelling that the two corresponding parameters need to be carried in one SDU. The six variables (line 10 to 11) are used to specify the value of each significant parameter.

The last two colour sets are defined for modelling the SDUs and how they are transferred within the IOTS Provider. The colour set SDUs (line 12 to 16) is defined as a *union* type, modelling the complete set of SDUs used within IOTS. The first three SDUs (line 12) that have a name starting with trd are specific to an Offer phase. The next four (line 13) that have a name starting with pay are used for making a payment. Another two (line 14) that have a name starting with dlv are used for a delivery. The next four (line 15) that have a name starting with auth are specific to an Authentication phase. The last one cancel (line 16) is only used for transaction service cancellation. The meaning of each SDU is given in Table 8.3. Note that the significant parameters are also considered within some of the SDUs, e.g., the trdoptions SDU carries both Transaction Type and Selection Reply parameters (TTxSR). Further

details on the significant parameters carried within some of the SDUs will become apparent later. Given a reliable communication medium within the IOTS Provider (see Section 8.1), all the SDUs are delivered in a FIFO order. The colour set SDUQ (line 17) is defined as a *list* type for modelling the SDU queues. Finally (on line 18), the variable x is used to represent any SDU defined by the colour set SDUs, and q for any SDU queue within the IOTS Provider.

SDU	Meaning
trdoptions	Communicates *TradingOptions.req* parameters from Merchant to Consumer, including two significant parameters Transaction Type and Selection Reply.
trdselection	Communicates *TradingOptions.res* parameters from Consumer to Merchant.
trdresult	Communicates *TradingResult.req* parameters from Merchant to Consumer, including two significant parameters Payment Type and Delivery Type.
payrequest	Communicates *PayInvoke.req* parameters from Consumer to Payment Handler, including two significant parameters Payment Number and Delivery Type.
paydata	Communicates *PayData.req* parameters from Payment Handler to Consumer, or *PayData.res* parameters from Consumer to Payment Handler.
payresponse	Communicates *PayResult.req* parameters from Payment Handler to Consumer, including the significant parameter Process State.
payretry	Communicates *PayRetry.req* parameters from Consumer to Payment Handler.
dlvrequest	Communicates *Delivery.req* parameters from Consumer to Payment/Delivery Handler.
dlvresponse	Communicates *Delivery.res* parameters from Payment/Delivery Handler to Consumer.
authrequest	Communicates *Authentication.req* parameters from Authenticator to Authenticatee.
authresponse	Communicates *Authentication.res* parameters from Authenticatee to Authenticator.
authresult	Communicates *AuthResult.req* parameters from Authenticator to Authenticatee, including the significant parameter Process State.
authretry	Communicates *AuthRetry.req* parameters from Authenticatee to Authenticator.
cancel	Communicates *Cancel.req* parameters between a Consumer and a non-Consumer entity.

Table 8.3: Meaning of SDUs in the IOTS CPN model.

8.4 Trading Phase Pages

The rest of the pages in the IOTS CPN model the five trading phases defined in IOTS. Except for the two pages, Authentication and Cancel, all trading phase pages have a similar structure, which is described in Section 8.4.1. The modelling details of each trading phase page are given in Sections 8.4.2 to 8.4.6.

8.4.1 General Page Structure

The Offer page in Figure 8.3 is used to illustrate the general structure of a trading phase page.

Three Types of Places

There are six places, which can be divided into three categories:

SAP places. The two port places C_SAP and M_SAP are the two SAP places modelled on the Offer page. They are linked (via port-socket assignment) to the places with the same name on page IOTS_Global. C_SAP and M_SAP are typed by the colour sets CState and MState, respectively. Hence, the marking of either place can be considered as the state of the interface at the corresponding trading role SAP. For example, both C_SAP and M_SAP have an initial marking of IDLE, indicating that no primitives have occurred before the Offer phase starts.

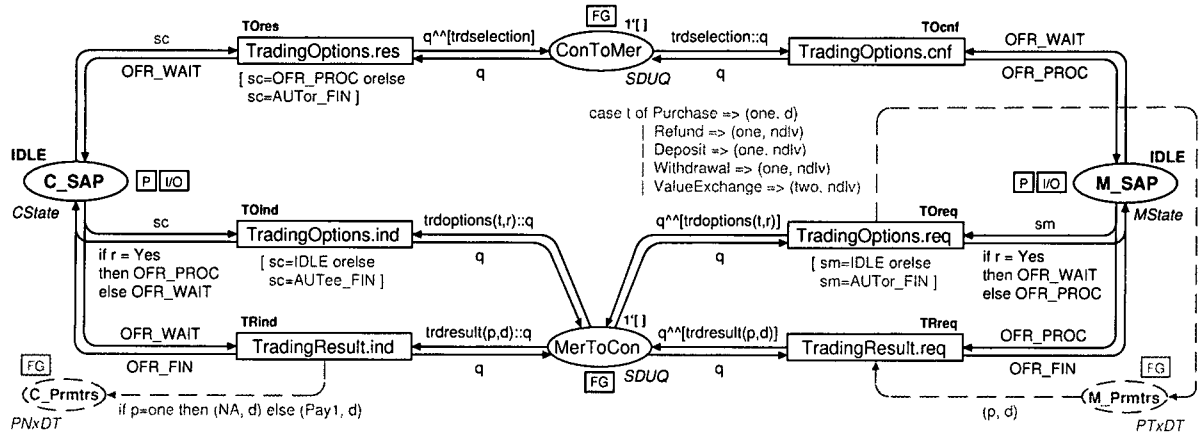


Figure 8.3: Offer page in the IOTS CPN model.

Provider places. The two fusion places ConToMer and MerToCon represent communication storage between the Consumer and Merchant via the IOTS Provider, and are called *Provider places*. All the Provider places are typed by the colour set SDUQ, indicating that each communication is modelled as delivering a SDU to and from a SDU queue within the IOTS Provider. In Figure 8.3, both ConToMer and MerToCon have an initial marking of an empty list (1'[ ]), indicating that there is initially no communication between the Consumer and Merchant.

Parameter places. The two fusion places C_Prmtrs and M_Prmtrs (shown with dashed borders) are used to store the significant parameters, and are referred to as *parameter places*. Each parameter place is typed by a colour set modelling the significant parameters. More specifically, the place C_Prmtrs typed by the colour set PNxDT, is where both Payment Number and Delivery Type parameters reside. The place M_Prmtrs typed by the colour set PTxDT, is where both Payment Type and Delivery Type parameters reside.

Primitive Transitions

There are six transitions modelling the events of submitting and delivering different IOTS primitives. These are referred to as *primitive transitions*. Each transition has an associated label (inside the transition) given by the primitive name for readability of the model, and a transition name (shown in bold and positioned above the transition). The transition name equals the abbreviated form of the corresponding primitive as it appears in the service language.

The abbreviated form of an IOTS primitive is given as a mnemonic. The first two letters in the mnemonic are the two initial letters taken from the two words constituting the primitive name. For example, TO is used for *TradingOptions*. If the two initial letters are the same for the two different primitives, a third letter is then added to distinguish between these two primitives. For example, PRS is used for *PayResult*, and PRT for *PayRetry* in the Payment phase. The next three letters indicate the primitive type, i.e., *req, ind, res* or *cnf*. Some primitives may also have one more letter in their abbreviated form for naming the primitive transitions. The meaning of primitive transitions and their transition names used in the IOTS CPN model will be introduced when they appear in the corresponding CPN pages.

Modelling local interface state changes. Each arc from a SAP place to a primitive transition has an inscription specifying the required interface state at that SAP for the primitive event to occur. The arc in the opposite direction has an inscription specifying the state of the

SAP interface after the primitive event occurs. For example, the occurrence of transition TOcnf (top right of Figure. 8.3) changes the M-SAP interface state from OFR_WAIT to OFR_PROC.

Queuing Operations. The two arcs connecting a Provider place and a primitive transition, specify how the SDUs are submitted to or delivered by the IOTS Provider. Each arc inscription has a variable q, which is bound to the token in the Provider place that represents a queue of SDUs. The submission of a SDU to the SDU queue from a trading role entity is modelled as the concatenation of that SDU to the end of the list of SDUs. The delivery of a SDU to a peer trading role entity is modelled as removing the first element from a non-empty list of SDUs in the queue. For example, in Figure 8.3, the occurrence of transition TOres (top left) at the C-SAP adds a trdselection to the end of the queue (q) in place ConToMer. When the trdselection becomes the first element in the queue, transition TOcnf (top right) at the M-SAP is enabled, and its occurrence removes the trdselection from the queue. This modelling approach is applied to ensure that all SDUs within the IOTS Provider are delivered in FIFO order.

Modelling the effects of parameters. These are shown as dashed arcs. Each arc from a parameter place to a primitive transition has an inscription specifying the required value of the significant parameter(s) for the primitive event to occur. The arc in the opposite direction has an inscription specifying the value of the significant parameter(s) to be stored after the primitive event occurs. For example, when a Purchase transaction is initiated, transition TOreq (middle right of Figure 8.3) will occur in a binding with t=Purchase. As a result, a token with value (one,d) will be put into place M_Prmtrs, where d is a variable that may be bound to one of the three values of DlvType. In this case, there will be only one payment (p=one). In another example, if transition TRind (bottom left of Figure. 8.3) occurs, both payment number and delivery type information will be stored in C_Prmtrs. The arc inscription indicates how payment type determines payment number: if there is only one payment (p=one), the payment number will not be applicable (NA). Otherwise, the following Payment phase will only involve the first payment (Pay1) within the current transaction service.

8.4.2 Offer Phase Page

The Offer page (see Figure 8.3) models the global sequences of IOTS primitives during an Offer phase. The *TradingOptions* primitives are modelled by the four upper transitions, and the *TradingResult* primitives are modelled by the lowest two transitions. The meaning of each of these transitions is given in Table 8.4.

Transition	Primitive	Meaning
TOreq	TradingOptions.req	Trading Options request submitted by the Merchant
TOind	TradingOptions.ind	Trading Options indication delivered to the Consumer
TOres	TradingOptions.res	Trading Options response submitted by the Consumer
TOcnf	TradingOptions.cnf	Trading Options confirm delivered to the Merchant
TRreq	TradingResult.req	Trading Result request submitted by the Merchant
TRind	TradingResult.ind	Trading Result indication delivered to the Consumer

Table 8.4: Meaning of primitive transitions on the Offer page.

Initially, both C-SAP and M_SAP places are marked with IDLE and both Provider places are marked with an empty list (1'[ ]). As mentioned before, IOTP version 1.0 specifies that a

trading transaction service is always initiated by the Merchant. Accordingly on the Offer page, TOreq is the only primitive transition enabled in the beginning, with the state variable sm bound to IDLE and the list variable q bound to the empty list. As defined in the local M-SAP FSA (see Figure 7.12), the Merchant may also authenticate the Consumer at the beginning of an Offer phase (when the M-SAP interface is IDLE), in which case, the Merchant will continue within the Offer phase only if the authentication is complete successfully. According to this, the guard on TOreq is used to specify that TOreq can be enabled in either of the following two cases. One is that the Merchant does not want to authenticate the Consumer, in which case, the M-SAP interface is IDLE (sm=IDLE). The other is that the Merchant authenticates the Consumer and the authentication is completed successfully, in which case, the M-SAP interface enters AUTor_FIN (sm=AUTor_FIN) before TOreq can occur.

On occurrence of TOreq, the M-SAP interface enters either OFR_WAIT or OFR_PROC, a trdoptions SDU conveying both the transaction type and selection reply information is added to the SDU queue (in place MerToCon), and both payment type and delivery type information is stored in the parameter place M_Prmtrs at the Merchant side.

In the arc expression trdoptions(t,r), the variable t or r is used for non-deterministically selecting one of the values of TransType or SelReply. The transaction type (t) determines both the payment type and the delivery type. The case of a Purchase transaction (t=Purchase) has been described at the end of Section 8.4.1. A Refund transaction (t=Refund) involves one payment (p=one) and no delivery (d=ndlv), and so does a Deposit (t=Deposit) or Withdrawal transaction (t=Withdrawal). A Value Exchange transaction (t=ValueExchange) involves two payments (p=two) and no delivery (d=ndlv). The selection reply (r) determines the interface state at the M-SAP upon occurrence of TOreq and thereby the transition(s) to be enabled next. If Yes, the M-SAP interface enters OFR_WAIT, in which case, only TOind is enabled. Otherwise, if No, the M-SAP interface enters OFR_PROC, in which case, both TOind and TRreq are enabled. The enabling of TRreq indicates that the Merchant can submit a *TradingResult.req* directly following a *TradingOptions.req* primitive when the Selection Type parameter is set to No.

At the Consumer side, the guard on TOind specifies that the Consumer may receive a *TradingOptions.ind* either without being authenticated (sc=IDLE), or after being successfully authenticated by the Merchant (sc=AUTee_FIN). The occurrence of TOind removes trdoptions, and changes the interface state at the C-SAP to OFR_PROC if r is bound to Yes, or to OFR_WAIT if r is No. When C-SAP interface is in OFR_PROC, TOres is the only transition enabled (on page Offer). Similarly to TOreq, the guard on TOres specifies it may also be enabled in a binding with sc=AUTor_FIN. If the C-SAP interface is in AUTor_FIN before TOres occurs, it indicates that the Consumer has successfully authenticated the Merchant after receiving a *TradingOptions.ind* and before submitting a *TradingOptions.res*. On occurrence of TOres, the C-SAP interface enters OFR_WAIT waiting for the delivery of a *TradingResult.ind* primitive, and a trdselection SDU is added to the SDU queue from the Consumer to Merchant (in place ConToMer). Given that the M-SAP interface is in OFR_WAIT, transition TOcnf is then enabled when trdselection is at the head of the queue. The occurrence of TOcnf removes trdselection from the queue and changes the interface state at the M-SAP to OFR_PROC.

No matter whether selection reply is Yes or No, the M-SAP interface must be in OFR_PROC and the C-SAP interface in OFR_WAIT, before the two transitions TRreq and TRind can occur. After TRreq has occurred, the M-SAP interface enters OFR_FIN, and a trdresult is added to the

SDU queue, carrying both payment type and delivery type parameters (p,d) obtained from the Merchant (place M_Prmtrs). At the Consumer side, TRind is enabled once trdselection is at the head of the SDU queue. On its occurrence, the trdresult is removed from the queue, both payment number and delivery type information is stored in C_Prmtrs, and the C-SAP interface enters OFR_FIN. The dashed arc from TRind to C_Prmtrs has an inscription specifying how the payment type determines the payment number information. A detailed description for this has been given at the end of Section 8.4.1. When the C-SAP and the M-SAP interfaces are both in OFR_FIN, the Offer phase completes and the IOTS can enter the Payment phase.

From the above, it can be seen that the Selection Reply parameter value controls the occurrences of both TOres and TOcnf transitions. When selection reply is No, the C-SAP interface cannot enter OFR_PROC nor can the M-SAP interface enter OFR_WAIT. According to the local C-SAP FSA defined in Figure 7.15, the Consumer is not allowed to authenticate the Merchant except when the C-SAP interface is in OFR_PROC. This indicates that the C-SAP interface will never be in AUTor_FIN without entering OFR_PROC first, which disables transitions TOres and then TOcnf. As mentioned before, TOind and TRreq can be enabled *concurrently* when selection reply is No. The occurrences of these two transitions are *independent* of each other. This indicates that the two primitive events: the Merchant submitting a *TradingResult.req* and the Consumer being delivered a *TradingOptions.ind*, can occur *simultaneously*.

8.4.3 Payment Phase Page

The Payment page shown in Figure 8.4 models the global sequences of IOTS primitives during the Payment phase. The two SAP places are C_SAP and P_SAP. P_SAP has an initial marking of IDLE, indicating the P-SAP interface is IDLE before the Payment phase starts. The two Provider places, ConToPH and PHToCon, model the communication storage between the Consumer and Payment Handler. The two parameter places, C_Prmtrs and P_Prmtrs, are used to store the payment number (n) and the delivery type (d) for the Consumer and Payment Handler, respectively. C_Prmtrs belongs to a fusion set which also contains the place with the same name on the Offer page, representing the only parameter storage for the Consumer. Table 8.5 lists the meaning of each of the twelve primitive transitions on page Payment. The name of a *PayResult* primitive transition ends with the letter s or f, indicating a successful or a failed payment result.

Simplification of Payment-related States

The local interface state changes at the C-SAP or P-SAP during the Payment phase are quite complicated. Without the well-defined local SAP FSA in Section 7.3.3, the development of a CPN such as the Payment page shown in Figure 8.4, would be a very time-consuming and error-prone procedure. As mentioned in Section 8.2, the interface states used in the IOTS CPN model are however simplified compared to those defined in the corresponding local SAP FSA. A major simplification is made to the states related to the Payment phase. Table 8.6 shows the mapping between the payment-related states in the CPN model and those in the local C-SAP or P-SAP FSA (see Figure 7.15 and 7.13). With parameter values of payment number and delivery type (n, d), the state prefix PAY is used in the CPN model instead of PAY1, PAY2, and PAYD defined in the local FSA, for all payment-related states except PAY1_FIN.

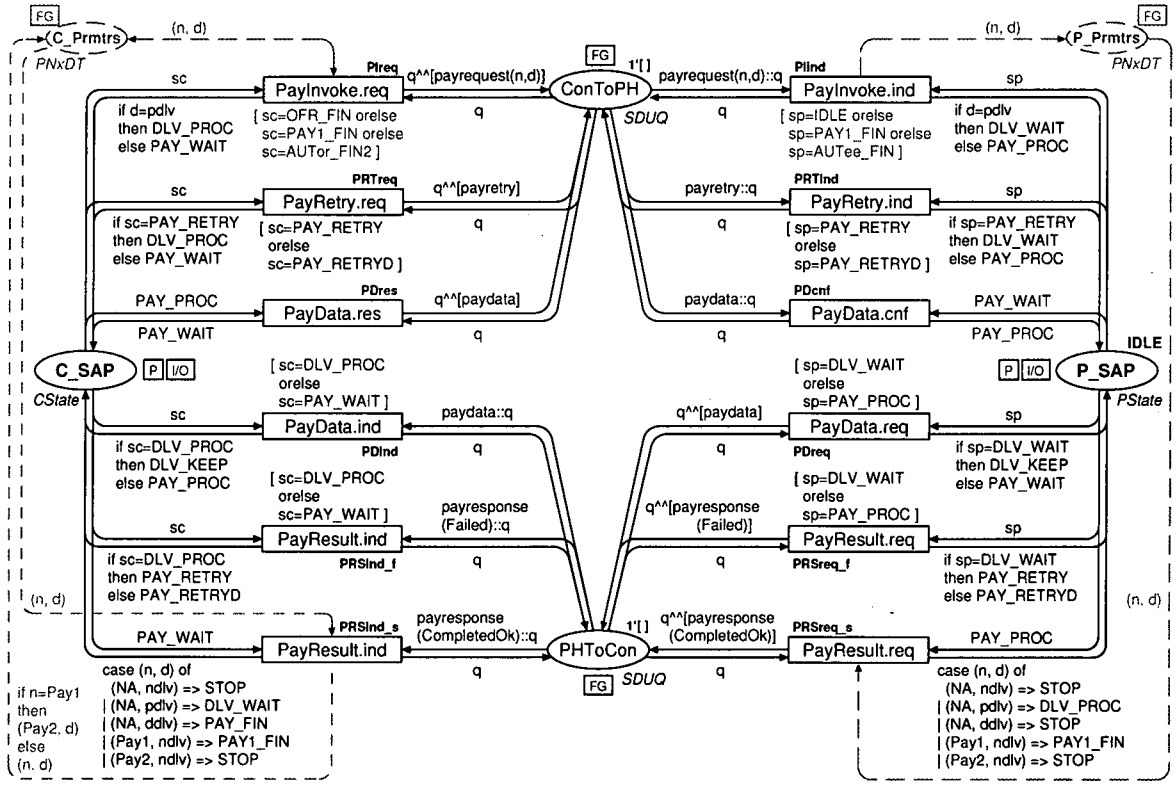


Figure 8.4: Payment page in the IOTS CPN model.

Transition	Primitive	Meaning
PIreq	PayInvoke.req	Payment Invoke request submitted by the Consumer
PIind	PayInvoke.ind	Payment Invoke indication delivered to the Payment Handler
PDreq	PayData.req	Payment Data request submitted by the Payment Handler
PDind	PayData.ind	Payment Data indication delivered to the Consumer
PDres	PayData.res	Payment Data response submitted by the Consumer
PDcnf	PayData.cnf	Payment Data confirm delivered to the Payment Handler
PRSreq_f	PayResult.req	Payment Result request submitted by the Payment Handler, conveying a failed payment result
PRSreq_s	PayResult.req	Payment Result request submitted by the Payment Handler, conveying a successful payment result
PRSind_f	PayResult.ind	Payment Result indication delivered to the Consumer, conveying a failed payment result
PRSind_s	PayResult.ind	Payment Result indication delivered to the Consumer, conveying a successful payment result
PRTreq	PayRetry.req	Payment Retry request submitted by the Consumer
PRTind	PayRetry.ind	Payment Retry indication delivered to the Payment Handler

Table 8.5: Meaning of primitive transitions on the Payment page.

IOTS CPN Model		Local C-SAP or P-SAP FSA States
Interface States	Parameter Values (n, d)	
PAY_PROC/WAIT/RETRY	(NA, ndlv) or (NA, ddlv) or (Pay1, ndlv)	PAY1_PROC/WAIT/RETRY
	(Pay2, ndlv)	PAY2_PROC/WAIT/RETRY
	(NA, pdlv)	PAYD_PROC/WAIT/RETRY
PAY_RETRYD	(NA, pdlv)	PAYD_RETRYD
PAY1_FIN	(Pay1, ndlv)	PAY1_FIN
PAY_FIN	(NA, ddlv)	PAY_FIN

Table 8.6: Mapping of payment-related states between the IOTS CPN model and the local C-SAP or P-SAP FSA.

General Description of Modelling a Payment Phase

At the beginning of the Payment phase, the C-SAP interface is in one of the following three states: OFR_FIN indicating the end of an Offer phase, PAY1_FIN indicating the completion of the first payment in a Value Exchange transaction, and AUTor_FIN2 indicating the Consumer has successfully authenticated the Payment Handler before initiating the first payment. Transition PIreq is enabled. Upon its occurrence, the C-SAP interface enters DLV_PROC if there is a delivery provided by the Payment Handler (d=pdlv), or PAY_WAIT otherwise. A payrequest conveying both the payment number and delivery type is added to the SDU queue from the Consumer to Payment Handler (in place ConToPH).

At the Payment Handler side, the P-SAP interface is in state IDLE, PAY1_FIN, or AUTee_FIN before a *PayInvoke.ind* can occur. Transition PIind is enabled when the first element of the SDU queue in ConToPH is a payrequest with the parameter pair (n,d) bound to any valid value of PNxDT. After PIind has occurred, the payrequest is removed from the queue, the payment number and delivery type carried by (n,d) is stored in P_Prmtrs, and the P-SAP interface enters DLV_WAIT if the Payment Handler needs to provide delivery (d=pdlv), or PAY_PROC otherwise.

Next, when P-SAP interface is in DLV_WAIT or PAY_PROC, both PDreq and PRSreq_f are enabled, and PRSreq_s is also enabled if the P-SAP interface is in PAY_PROC *only*. Further, when one of the three transitions PDreq, PRSreq_f, and PRSreq_s occurs the other two are disabled, and the occurrence of each of them results in a different procedure. The occurrence of PDreq indicates the beginning of a payment (system-specific) data exchange, by depositing paydata into the SDU queue in ConToPH. The occurrence of PRSreq_f adds a payresponse(Failed) to the SDU queue, indicating that the payment has failed and a payment retry may take place, in which case, transition PRTind will be enabled if a payretry appears at the head of the queue in ConToPH. Finally, the occurrence of PRSreq_s which adds a payresponse(CompletedOk) to the queue, indicates the successful completion of the payment phase at the P-SAP.

At the Consumer side, the enabling of transition PDind, PRSind_f or PRSind_s is determined by the SDU placed at the head of the queue in PHToCon, given that the C-SAP interface is in DLV_PROC or PAY_WAIT. If paydata is at the head of the queue, PDind can occur. If a payresponse(Failed) is at the head of the queue, PRSind_f can occur which enables PRTreq. When a payresponse(CompletedOk) is received, on occurrence of PRSind_s, indicating the Payment phase has been successfully completed at the C-SAP.

Modelling of Payment Data Exchange and Payment Retry

On the Payment page, a payment data exchange is modelled by the four primitive transitions that occur in the order of PDreq, PDind, PDres and PDcnf. It is always initiated by the Payment Handler when P-SAP interface is in DLV_WAIT or PAY_PROC thereby enabling PDreq, and finishes on occurrence of PDcnf which changes the state of the P-SAP interface back to PAY_PROC enabling PDreq again. This models that the above sequence of four *PayData* primitive transitions can be repeated indefinitely. As defined in the local C-SAP and P-SAP FSA in Section 7.3.3, we allow an unbounded number of payment data exchanges to take place at the C-SAP or P-SAP *locally*, which results in an unbounded number of payment data exchanges that may occur *globally*. Note that, the occurrence of PDreq changes the state of the P-SAP interface from DLV_WAIT to DLV_KEEP. When the P-SAP interface is in DLV_KEEP,

the Payment phase cannot continue until a *Delivery.ind* primitive is delivered to the Payment Handler, which is modelled on page Delivery in Section 8.4.4. In a similar way at the Consumer side, once the C-SAP interface enters DLV_KEEP (on occurrence of PDind), the Payment phase cannot continue until the Consumer submits a *Delivery.req* primitive.

Similarly, the payment retry is modelled by the four primitive transitions PRSreq_f, PRSind_f, PRTreq, and PRTind. The occurrences of these four transitions can be repeated indefinitely, modelling an unbounded number of payment retries.

Modelling of Payment Phase Completion

The inscription on the arc from transition PRSreq_s to place P_SAP and the inscription on the arc from transition PRSind_s to place C_SAP, determine the interface state at the P-SAP and the C-SAP upon successful completion of the Payment phase. The state is determined by the payment number and the delivery type. The parameter pair (n,d) may have five different values corresponding to the five different scenarios of a trading transaction.

Firstly, if the transaction involves only one payment with no delivery (NA,ndlv), both the P-SAP and the C-SAP interfaces will enter STOP, indicating the transaction service completes at the end of the Payment phase. Secondly, if there is a payment with delivery provided by the Payment Handler (NA,pdlv), the P-SAP interface will enter DLV_PROC and the C-SAP interface will enter DLV_WAIT. In this case, the IOTS moves into the Delivery phase (which is combined with the Payment phase). Next, if there is a payment followed by delivery provided by the Delivery Handler (NA,ddlv), the P-SAP interface will enter STOP and the C-SAP interface will enter PAY_FIN. In this case, the IOTS will move into a standalone Delivery phase. The last two scenarios are concerned with a Value Exchange transaction which involves two payments but no delivery. The first (Pay1,ndlv) models the situation where the payment completed in the current Payment phase is the first payment in the transaction and is to be followed by a second one. In this case, both the P-SAP and the C-SAP interfaces will enter PAY1_FIN, and the IOTS will then carry out the second payment. In the second case (Pay2,ndlv), when the second payment is completed successfully, both the P-SAP and the C-SAP interfaces will enter STOP, indicating the completion of the transaction service. In addition, the inscription of the dashed arc from transition PRSind_s to place C_Prmtrs is used for updating the payment number from the first (n=Pay1) to the second (n=Pay2) upon completion of the first payment.

Discussion: Modelling of *PayResult* Primitives

With respect to a successful or a failed payment result, a *PayResult.req* or a *PayResult.ind* is modelled by a pair of primitive transitions PRSreq_s and PRSreq_f or PRSind_s and PRSind_f. Alternatively, it is also possible to fold each pair of transitions into one, modelling the *PayResult* primitive with either a successful or a failed result. This will however require more complex arc inscriptions, reducing the clarity of the specification. We discuss three aspects:

1. Different enabling conditions would need to be combined. For example, the transition PRSreq_f can occur when the P-SAP interface is in either DLV_WAIT or PAY_PROC, whereas PRSreq_s can only occur when the P-SAP interface is in PAY_PROC. Folding the two transitions into one, i.e, a *PayResult.req* primitive transition, requires the combination of their enabling conditions. However, the two different payment results should still

be distinguished from each other. Given that the P-SAP interface is in DLV_WAIT, the *PayResult.req* primitive for a successful result cannot occur. This restriction needs to be specified by the inscription on the arc from the new *PayResult.req* primitive transition to place PHToCon, which would result in a very complex arc inscription.

2. The SAP interface state changes, which are determined in different ways, would need to be combined. For example, if PRSreq_s occurs, the P-SAP interface state is determined by the value of the parameter pair (n,d), and such state change is only valid when the P-SAP interface was in PAY_PROC before PRSreq_s occurs. Whereas, if PRSreq_f occurs, the P-SAP interface entering either PAY_RETRY or PAY_RETRYD is independent of any parameter value, but determined by whether it was in DLV_WAIT or PAY_PROC before PRSreq_f occurs. Combining these two sets of interface state changes into one arc inscription would result in a much more complicated arc inscription with less clarity.

3. Conditions for updating the payment number (n) on completion of a payment would need to be combined. At the C-SAP, the occurrence of transition PRSind_s with n bound to Pay1 changes the payment number to Pay2. Folding two transitions PRSind_s and PRSind_f into one, i.e., a *PayResult.ind* primitive transition, must ensure that the update of payment number from Pay1 to Pay2 cannot occur if the payment fails. This would complicate the inscription on the arc leading from the combined *PayResult.ind* primitive transition to place C_Prmtrs.

We thus use two transitions to model each *PayResult* primitive, one for successful processing of the payment and the other when the payment fails. This provides a simpler and more readily understood specification of the Payment phase.

8.4.4 Delivery Phase Page

The Delivery page shown in Figure 8.5 models the global sequences of IOTS primitives during the Delivery phase. Delivery may be provided by the Payment Handler or the Delivery Handler, resulting in two possible ways (modes) in which it can occur. On page Delivery, there are three SAP places: C_SAP, P_SAP and D_SAP. D_SAP has an initial marking of IDLE, indicating the D-SAP interface is IDLE before the Delivery phase starts. The two Provider places, ConToPH

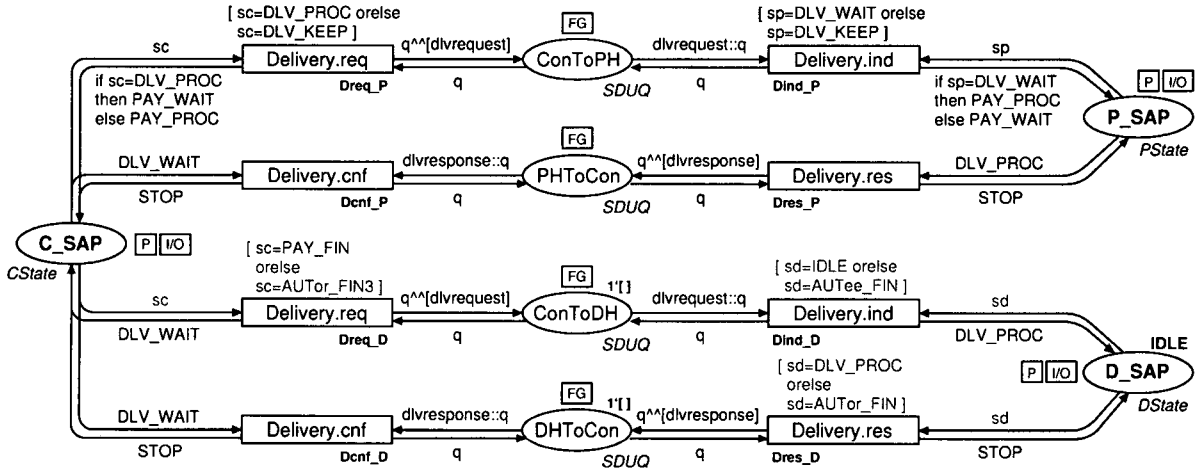


Figure 8.5: Delivery page in the IOTS CPN model.

and PHToCon, are the same as those used in the Payment page (Figure 8.4). The other two, ConToDH and DHToCon, model the FIFO queues between the Consumer and Delivery Handler. The parameter place C_Prmtrs is used for providing the payment number and the delivery type information during the Delivery phase. Table 8.7 lists the meaning of each of the eight primitive transitions on page Delivery. The name of each transition ends with the letter P or D to indicate which trading role (Payment Handler or Delivery Handler) provides delivery.

Transition	Primitive	Meaning
Dreq_P	Delivery.req	Delivery request submitted by the Consumer, where the delivery needs to be provided by the Payment Handler
Dind_P	Delivery.ind	Delivery indication delivered to the Payment Handler
Dres_P	Delivery.res	Delivery response submitted by the Payment Handler
Dcnf_P	Delivery.cnf	Delivery confirm delivered to the Consumer, where the delivery is provided by the Payment Handler
Dreq_D	Delivery.req	Delivery request submitted by the Consumer, where the delivery is provided by the Delivery Handler
Dind_D	Delivery.ind	Delivery indication delivered to the Delivery Handler
Dres_D	Delivery.res	Delivery response submitted by the Delivery Handler
Dcnf_D	Delivery.cnf	Delivery confirm delivered to the Consumer, in which case, the delivery is provided by the Delivery Handler

Table 8.7: Meaning of primitive transitions on the Delivery page.

Only one of the two Delivery modes specified in the Delivery page can occur during a trading transaction service. The choice of mode depends on the delivery type information stored at the Consumer side. When delivery is provided by the Payment Handler, the Delivery phase is always combined with the Payment phase. Otherwise, the Delivery phase is separate and follows a successfully completed Payment phase.

Delivery between Consumer and Payment Handler

This is given by the first half of the Delivery page (upper part of Figure 8.5), involving two SAP places: C_SAP and P_SAP, and two Provider places: ConToPH and PHToCon. Since it is strongly related to the modelling of the Payment phase, we shall refer to the Payment page (see Figure 8.4) in the following discussion.

The enabling of transition Dreq_P on page Delivery requires the C-SAP interface in DLV_PROC or DLV_KEEP. Revisiting the Payment page shows that the C-SAP interface enters DLV_PROC on occurrence of PIreq in a binding with d=pdlv. This enables Dreq_P on page Delivery, which indicates that the Consumer can submit a *Delivery.req* directly following a *PayInvoke.req*.

Considering further when C-SAP interface is in DLV_PROC, the transition PDind on page Payment will be enabled if paydata is at the head of the SDU queue in PHToCon. In this case, both PDind on page Payment and Dreq_P on page Delivery will be enabled, and either of them can occur. If PDind occurs first, the C-SAP interface will enter DLV_KEEP which still enables Dreq_P, and the occurrence of Dreq_P will change the state of the C-SAP interface from DLV_KEEP to PAY_PROC. Otherwise, if Dreq_P and then PDind occur, the C-SAP interface will also enter PAY_PROC.

Considering again when C-SAP interface is in DLV_PROC, the transition PRSind_f on page Payment will be enabled if payresponse(Failed) is at the head of the SDU queue in PHToCon. In this case, both PRSind_f on page Payment and Dreq_P on page Delivery will be enabled.

If PRSind_f occurs first, the C-SAP interface will enter PAY_RETRY, which enables transition PRTreq on page Payment but disables Dreq_P on page Delivery. The occurrence of PRTreq will then change the state of the C-SAP interface back to DLV_PROC, which again enables Dreq_P. This indicates that the Consumer will firstly retry the payment upon the indication of a payment failure, and then continue to request the delivery. Otherwise, if Dreq_P and then PRSind_f occur, the C-SAP interface will enter PAY_RETRYD.

Once the C-SAP interface enters PAY_PROC or PAY_RETRYD, transition Dreq_P will not be enabled any more, because a *Delivery.req* primitive is not allowed to be submitted twice. In this case, the Consumer will continue with the payment until the end of the Payment phase. Upon successful completion of the payment, the C-SAP interface will then enter DLV_WAIT, in which case, transition Dcnf_P on page Delivery will be enabled if a dlvresponse is at the head of the SDU queue in PHToCon. After Dcnf_P occurs, the dlvresponse will be removed and the C-SAP interface will enter STOP, indicating the completion of the transaction service.

At the Payment Handler side, when P-SAP interface is in DLV_WAIT and a dlvrequest is at the head of the SDU queue in ConToPH, the three enabled transitions are: Dind_P on page Delivery, and both PDreq and PRSreq_f on page Payment. Among these, Dind_P may occur first, changing the P-SAP interface state from DLV_WAIT to PAY_PROC; or PDreq may occur first, changing the P-SAP interface state from DLV_WAIT to DLV_KEEP and again enabling Dind_P. Once Dind_P occurs, the Payment Handler will then continue with the payment until the end of the Payment phase. Upon successful completion of the payment, the P-SAP interface will enter DLV_PROC, enabling transition Dres_P on page Delivery. After Dres_P occurs, the P-SAP interface will enter STOP and a dlvresponse will be added to the end of the SDU queue in PHToCon, indicating the transaction service is complete at the Payment Handler side.

Delivery between Consumer and Delivery Handler

This is given by the other half of the Delivery page (lower part of Figure 8.5), involving two SAP places: C_SAP and D_SAP, and two Provider places: ConToDH and DHToCon.

Transition Dreq_D can be enabled when the C-SAP interface is in PAY_FIN or AUTor_FIN3. If in AUTor_FIN3, the Consumer has successfully authenticated the Delivery Handler before requesting a delivery. Otherwise, if in PAY_FIN, the Consumer can request a delivery from the Delivery Handler. After Dreq_D occurs, the C-SAP interface enters DLV_WAIT and a dlvrequest is added to the end of the SDU queue (in place ConToDH).

At the Delivery Handler side, the D-SAP interface is either IDLE or enters AUTee_FIN after the Delivery Handler has been successfully authenticated by the Consumer. If a dlvrequest is at the head of the SDU queue in ConToDH, transition Dind_D is enabled. Its occurrence removes the dlvrequest from the queue and changes the D-SAP interface state to DLV_PROC. When the D-SAP interface is in DLV_PROC, the Delivery Handler can authenticate the Consumer. As a result, the D-SAP interface may either stay in DLV_PROC (without such authentication) or enter AUTor_FIN after the Delivery Handler has successfully authenticated the Consumer. This enables Dres_D and when it occurs the D-SAP interface enters STOP, indicating the completion of the transaction service at the Delivery Handler side, and a dlvresponse is placed at the end of the SDU queue (in place DHToCon). Transition Dcnf_D is then enabled and when it occurs, the dlvresponse is removed from the queue, and the C-SAP interface enters STOP, indicating that the whole transaction service has completed successfully.

8.4.5 Authentication Phase Pages

Figure 8.6 depicts the Authentication page, which provides an overview of all possible Authentication phases that may be conducted between a Consumer and any of the other three non-Consumer entities. There are four SAP places and six substitution transitions. The four SAP places are linked to the places with the same names on page IOTS_Global (Figure 8.2). Each of the six substitution transitions models an Authentication phase where either a Consumer authenticates a non-Consumer entity, or vice versa. Each transition has its name (shown in bold) positioned (left) above it, and has an associated subpage that is named accordingly.

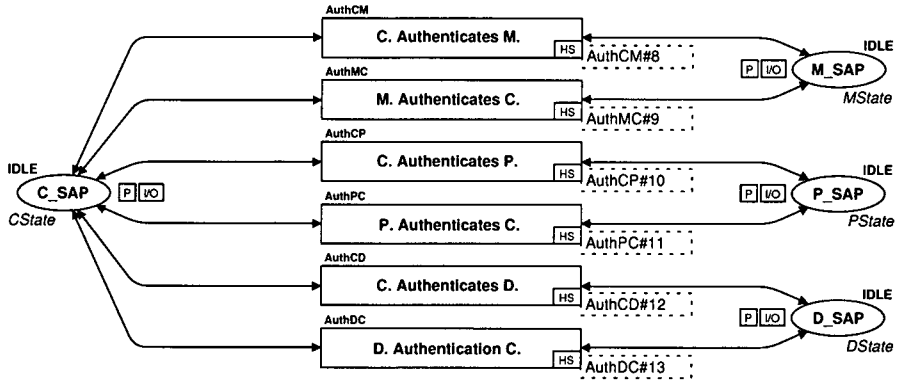


Figure 8.6: Authentication page in the IOTS CPN model.

Each of the six subpages models the global sequences of IOTS primitives during an Authentication phase between a Consumer and a non-Consumer entity. Except page AuthPC, all the other five pages model the eight *Authentication-related* primitive events in a similar way. The AuthPC page is much more complicated in that it models the two possible scenarios of an Authentication phase where the Consumer is authenticated by the Payment Handler during a Payment phase or a Delivery phase. In the following, we take the AuthCP page as a representative example to illustrate our basic modelling approach. Based on this, we then discuss the modelling details on the AuthPC page. Each of the other four pages will be briefly mentioned.

Consumer Authenticates Payment Handler

Figure 8.7 depicts the AuthCP page modelling the global sequences of primitives for an Authentication phase, where the Consumer authenticates the Payment Handler before initiating

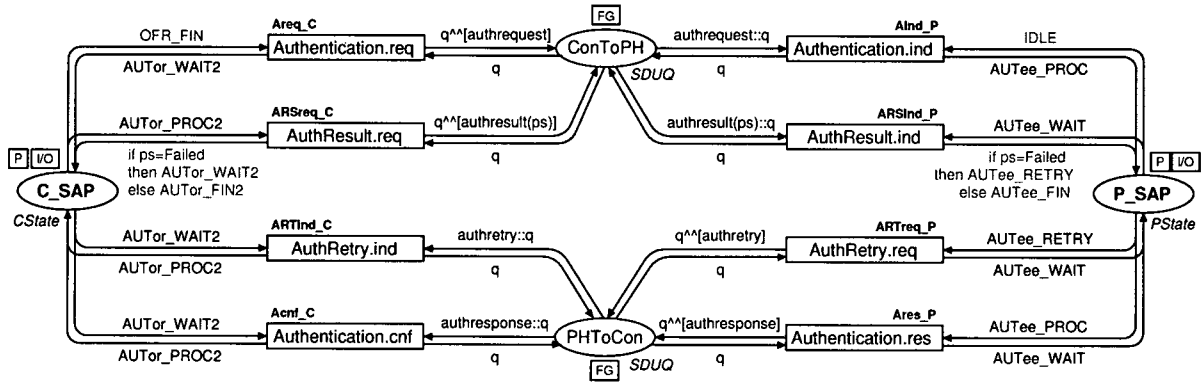


Figure 8.7: AuthCP page in the IOTS CPN model.

a payment at the beginning of the Payment phase. Table 8.8 lists the meaning of each primitive transition on page AuthCP. Note that each transition name ends with the letter C or P, indicating the primitive occurs at the Consumer or Payment Handler.

Transition	Primitive	Meaning
Areq_C	Authentication.req	Authentication request submitted by the Consumer
Aind_P	Authentication.ind	Authentication indication delivered to the Payment Handler
Ares_P	Authentication.res	Authentication response submitted by the Payment Handler
Acnf_C	Authentication.cnf	Authentication confirm delivered to the Consumer
ARSreq_C	AuthResult.req	Authentication Result request submitted by the Consumer
ARSind_P	AuthResult.ind	Authentication Result indication delivered to the Payment Handler
ARTreq_P	AuthRetry.req	Authentication Retry request submitted by the Payment Handler
ARTind_C	AuthRetry.ind	Authentication Retry indication delivered to the Consumer

Table 8.8: Meaning of primitive transitions on the AuthCP page.

The C-SAP interface is in OFR_FIN and the P-SAP interface is initially IDLE. Transition Areq_C is enabled. Its occurrence changes the state of the C-SAP interface to AUTor_WAIT2 and adds an authrequest to the end of the SDU queue in ConToPH, indicating that the Consumer starts to authenticate the Payment Handler. When authrequest is at the head of the queue, transition Aind_P is enabled. Its occurrence removes the authrequest and changes the state of the P-SAP interface to AUTee_PROC, indicating that the Payment Handler starts to be authenticated by the Consumer.

Next, the two transitions Ares_P and then Acnf_C occur, modelling the submission of an *Authentication.res* primitive by the Payment Handler which results in the delivery of an *Authentication.cnf* primitive to the Consumer. The authresponse SDU is used to carry *Authentication.res* parameters from the Consumer to Payment Handler.

When the C-SAP interface is in AUTor_PROC2, transition ARSreq_C occurs, with the Process State parameter variable ps bound to either CompletedOk or Failed. If Failed, the C-SAP interface enters AUTor_WAIT2 waiting for the reply of an authentication retry from the Payment Handler. Otherwise, if CompletedOk, the C-SAP interface enters AUTor_FIN2 indicating that the Consumer has successfully authenticated the Payment Handler and thereby can resume the payment within the (original) Payment phase specified on page Payment (see Figure 8.4). At the Payment Handler side, the occurrence of transition ARSind_P changes the state of the P-SAP interface to AUTee_RETRY if an authresult(Failed) is received. Otherwise, upon receiving an authresult(CompletedOk) the P-SAP enters the state AUTee_FIN representing the successful completion of the Authentication phase.

Now, we consider the modelling of sequences of primitives related to authentication retry. If the P-SAP interface is in AUTee_RETRY and the C-SAP interface is in AUTor_WAIT2, transition ARTreq_P and then ARTind_C can occur (once authretry is at the head of the queue), modelling the Payment Handler submitting an *AuthRetry.req* primitive and the Consumer receiving an *AuthRetry.ind* primitive. Again, the C-SAP interface enters AUTor_PROC2, enabling ARSreq_C. If the process state is failed, the occurrences of ARSreq, ARSind, ARTreq, and ARTind will repeat, modelling an unbounded number of authentication retries.

Simplification of authentication-related states. There is a one-to-one mapping of the authentication-related states between the AuthCP page and the local C-SAP or P-SAP FSA

(see Figure 7.15 and 7.13). The only exception is the state AUTor_RETRY2 appearing in the local C-SAP FSA but not in the AuthCP page. In Figure 8.7, if the C-SAP interface is in AUTor_WAIT2, transition ARTind_C can occur upon arrival of an authretry SDU from place PHToCon. Therefore, the state AUTor_WAIT2 with the arrival of the authretry SDU on page AuthCP provides the same information for controlling the occurrence of an *AuthRetry.ind*, as the state AUTor_RETRY2 defined in the local C-SAP FSA. As mentioned in Section 8.2, the interface states used in the CPN model can be simplified with information such as the SDU(s) transferred in the IOTS Provider. Accordingly, the following simplification rule is applied to all six Authentication phase pages. Given an authretry SDU at the head of the queue in the Provider place connected with an AuthRetry.ind primitive transition, the IOTS CPN model uses an AUTor_WAIT (with any suffix) state to replace each AUTor_RETRY (with the same suffix) state in the local SAP FSA.

Payment Handler Authenticates Consumer

Figure 8.8 depicts the AuthPC page modelling the global sequences of primitives for two possible scenarios of an Authentication phase, where the Payment Handler authenticates the Consumer after being delivered a *PayInvoke.ind* or a *Delivery.ind* primitive. The meaning of each Authentication-related primitive transition on page AuthPC is given in Table 8.9.

The AuthPC page has a structure different from the previous AuthCP page (see Figure 8.7) in the following three ways.

1. Two parameter places C_Prmtrs and P_Prmtrs are needed to provide the parameter information at the beginning or the end of the Authentication phase;
2. The *AuthResult* primitives conveying either a successful or a failed authentication result are modelled separately; and
3. Apart from the set of Authentication-related primitive transitions, the *PayInvoke.ind* primitive transition PIind and a *Delivery.ind* primitive transition DInd_P are also required.

Below, we illustrate the modelling of the primitive events and the interface state changes at the P-SAP and then the C-SAP, upon occurrences of primitive events at each side.

Transition	Primitive	Meaning
Areq_P	Authentication.req	Authentication request submitted by the Payment Handler
Aind_C	Authentication.ind	Authentication indication delivered to the Consumer
Ares_C	Authentication.res	Authentication response submitted by the Consumer
Acnf_P	Authentication.cnf	Authentication confirm delivered to the Payment Handler
ARSreq_f_P	AuthResult.req	Authentication Result request submitted by the Payment Handler, conveying a failed authentication result
ARSreq_s_P	AuthResult.req	Authentication Result request submitted by the Payment Handler, conveying a successful authentication result
ARSind_f_C	AuthResult.ind	Authentication Result indication delivered to the Consumer, conveying a failed authentication result
ARSind_s_C	AuthResult.ind	Authentication Result indication delivered to the Consumer, conveying a successful authentication result
ARTreq_C	AuthRetry.req	Authentication Retry request submitted by the Consumer
ARTind_P	AuthRetry.ind	Authentication Retry indication delivered to the Payment Handler

Table 8.9: Meaning of Authentication-related primitive transitions on page AuthPC.

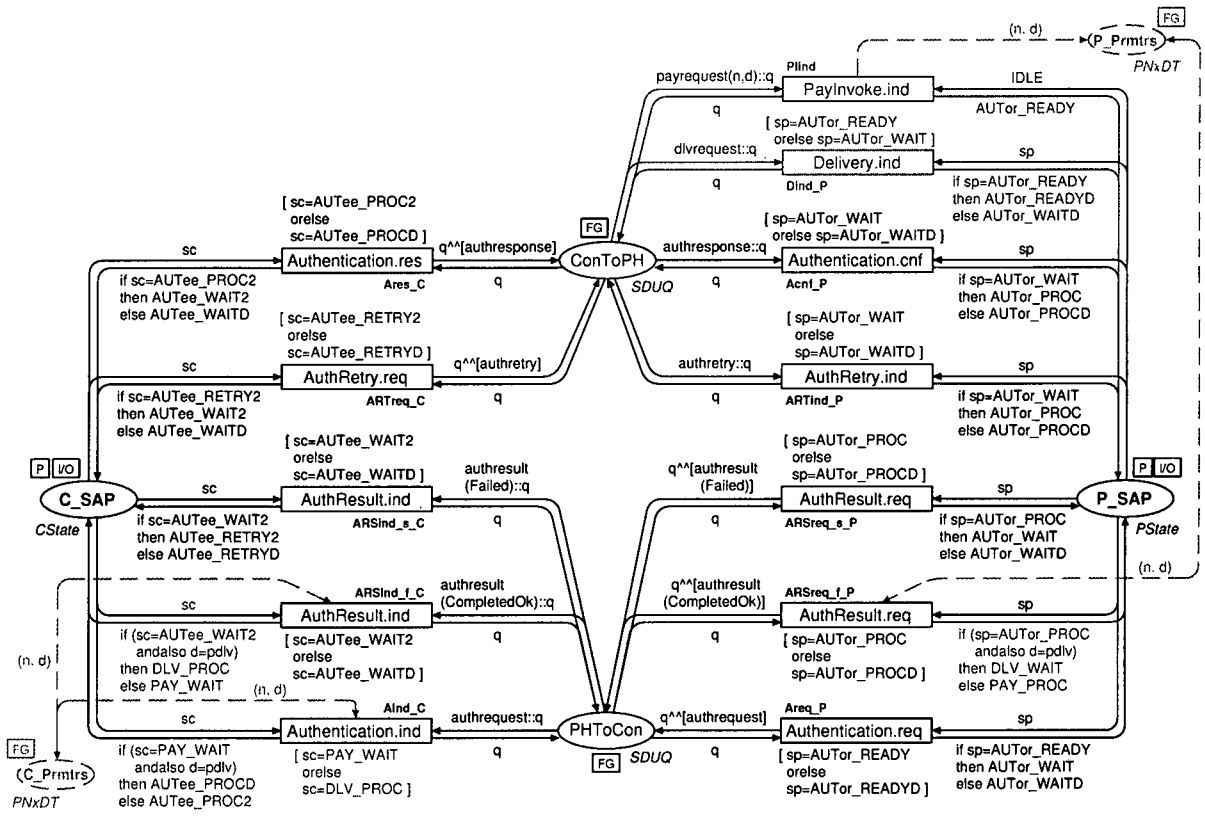


Figure 8.8: AuthPC page in the IOTS CPN model.

Modelling at the Payment Handler side. When the P-SAP interface is IDLE, the transitions Plind on both AuthPC and Payment pages are enabled, but they are in conflict with each other. If Plind on page AuthPC occurs, the P-SAP interface enters AUTor_READY indicating the Payment Handler is about to authenticate the Consumer, after being delivered a *PayInvoke.ind* but before continuing with the Payment phase. Otherwise, the occurrence of Plind on page Payment models the Payment Handler continuing without authenticating the Consumer.

Next, which one of the two Dind_P transitions on both AuthPC and Delivery pages will be enabled depends on which one of the above two Plind transitions has occurred. If Plind occurs on page AuthPC, only Dind_P on the same page can be enabled, due to the state of the P-SAP interface. If Dind_P on page AuthPC occurs with the state variable sp bound to AUTor_READY or AUTor_WAIT, the P-SAP interface will enter AUTor_READYD or AUTor_WAITD, correspondingly. It can be seen from the local P-SAP FSA that, once the P-SAP is in state AUTor_READYD or state AUTor_WAITD, it will never enter the state DLV_WAIT nor DLV_KEEP because the *Delivery.ind* primitive has already occurred. Hence, the transition Dind_P on page Delivery will not occur.

When the P-SAP interface is in AUTor_READY or AUTor_READYD, the only transition enabled on page AuthPC is Areq_P (bottom right of Figure 8.8) and its occurrence models the Payment Handler starting to authenticate the Consumer. This results in two different scenarios. One occurs when the P-SAP interface is AUTor_READY (indicating that a PayInvoke.ind has occurred) and the other when the P-SAP interface is in AUTor_READYD, in which case, both the PayInvoke.ind and the Delivery.ind primitives have occurred.

The two Authentication scenarios are modelled by the same set of *Authentication* primitive transitions with two different sets of P-SAP interface states that are defined in the local P-SAP FSA. Each set of states is specific to one Authentication scenario and is distinguished

from the other by the different ending (with or without the letter D) of their state names, representing that the authentication occurs before (without D) or after (with D) a *Delivery.ind* occurs at the Payment Handler. Both sets of P-SAP interface states are needed to model the two Authentication scenarios separately, because the Payment Handler cannot change from one scenario to the other when authenticating the Consumer unless a *Delivery.ind* primitive occurs. Accordingly, the guard on each *Authentication* primitive transition at the P-SAP specifies two states of the P-SAP interface when the transition can be enabled, and each of these two states corresponds to one of the two Authentication scenarios.

The *AuthResult* and *AuthRetry* primitives on page AuthPC are modelled in a similar way to the *PayResult* and *PayRetry* primitives on page Payment. At the P-SAP, an *AuthResult.req* primitive is modelled by two transitions ARSreq_s_P and ARSreq_f_P, where the second last letter of each transition name is used to indicate a successful (s) or a failed (f) authentication result. Both transitions can be enabled when the P-SAP interface is in AUTor_PROC or AUTor_PROCD. However, their occurrences result in different state assignments to the P-SAP interface. More specifically, if ARSreq_f_P occurs, the P-SAP interface will enter AUTor_WAIT or AUTor_WAITD, indicating that the Payment Handler continues with the Authentication phase. Otherwise, the Payment Handler completes the authentication of the Consumer upon the occurrence of ARSreq_s_P. If the Authentication phase completes before the *Delivery.ind* is delivered to the Payment Handler (i.e. ARSreq_s_P occurs with sp bound to AUTor_PROC and d bound to pdlv), the P-SAP interface will enter DLV_WAIT, waiting for the *Delivery.ind*. Otherwise, the P-SAP interface will enter PAY_PROC, continuing with the payment until the end of the Payment phase.

Modelling at the Consumer side. The modelling of the above two Authentication scenarios at the Consumer side concerns Authentication-related primitive events only. It is similar but simpler than the modelling at the Payment Handler side. The Consumer can be authenticated when the C-SAP interface is in PAY_WAIT or DLV_PROC. On occurrence of transition Aind_C (bottom left of Figure 8.8), the Consumer will enter one of the two Authentication scenarios, depending on both the C-SAP interface state and the delivery type. A binding with sc=PAY_WAIT and d=pdlv indicates that a *PayInvoke.req* primitive and a *Delivery.req* primitive have been submitted by the Consumer. If Aind_C occurs in this binding, the C-SAP interface will enter AUTee_PROCD, representing that the Consumer is being authenticated after the *Delivery.req* has been submitted. Otherwise, the occurrence of AIND in any other binding will change the state of the C-SAP interface to AUTee_PROC2. In this case, the Consumer is being authenticated after the *PayInvoke.req*, and will not continue with the Payment or the Delivery phase until authentication completes.

When the C-SAP interface is in AUTee_PROC2 or AUTee_PROCD, transition Ares_C is enabled. After it occurs, the Consumer starts waiting for the authentication result from the Payment Handler. If an authresult(Failed) SDU appears at the head of the queue in PHToCon, transitions ARSind_f_C and then ARTreq_C will occur, modelling authentication retry by the Consumer. Otherwise, if ARSind_s_C occurs, the Consumer will complete the authentication and resume the process within the (original) Payment or Delivery phase. As specified by the inscription on the arc from ARSind_s_C to C_SAP, on occurrence of ARSind_s_C, the C-SAP interface will return to the same state (DLV_PROC or PAY_WAIT) as when the authentication started.

Simplification of authentication-related states. Mapping of authentication-related states between the AuthPC page and the local C-SAP and P-SAP FSA shows that the local C-SAP FSA has two more states. One is `AUTee_READY` used after a *PayInvoke.req* but before a *Delivery.req* primitive is submitted. The other is `AUTee_READYD` used after both primitives have been submitted. Whereas on page AuthPC, with the information of an authrequest SDU in place PHToCon and the delivery type (d) in place C_Prmtrs, it is not necessary to introduce both `AUTee_READY` and `AUTee_READYD` states. More specifically, the CPN model uses the following information to replace `AUTee_READY`, which comprises the state `PAY_WAIT`, the delivery type pdlv and the arrival of the authrequest SDU from PHToCon. Instead of `AUTee_READYD`, the CPN model uses the information of the state `DLV_PROC` upon arrival of the authrequest.

Other Authentication Phase Pages

The other four subpages of the Authentication page, as shown in Figures 8.9 to 8.12, correspond to the four Authentication phases where the Consumer authenticates or is authenticated by the Merchant or the Delivery Handler. Each of these four pages is modelled in a similar way to the AuthCP page (Figure 8.7) and is not described in detail except the modelling of the initiation and completion of each Authentication phase. Also, the meaning of the primitive transitions on these pages are similar to those on page AuthCP, except that the transitions have their names ending with the letter M or D represents the primitive occurrences at the Merchant or the Delivery Handler.

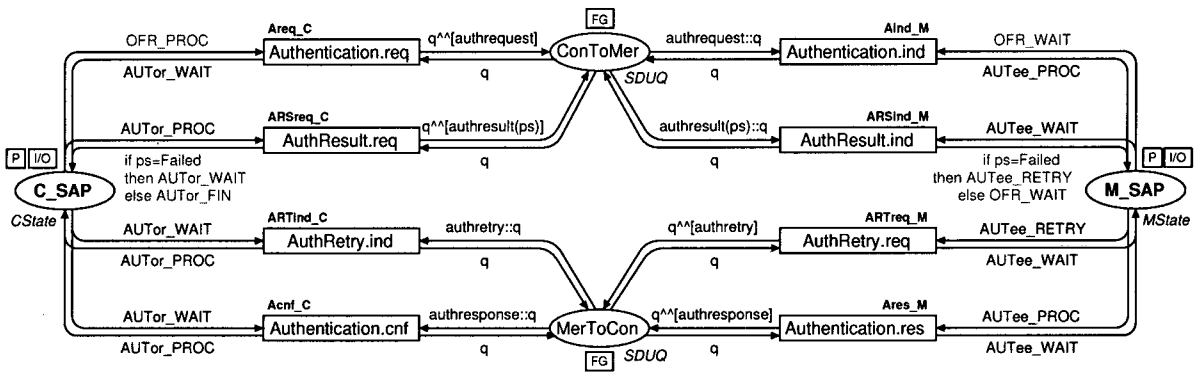


Figure 8.9: AuthCM page in the IOTS CPN model.

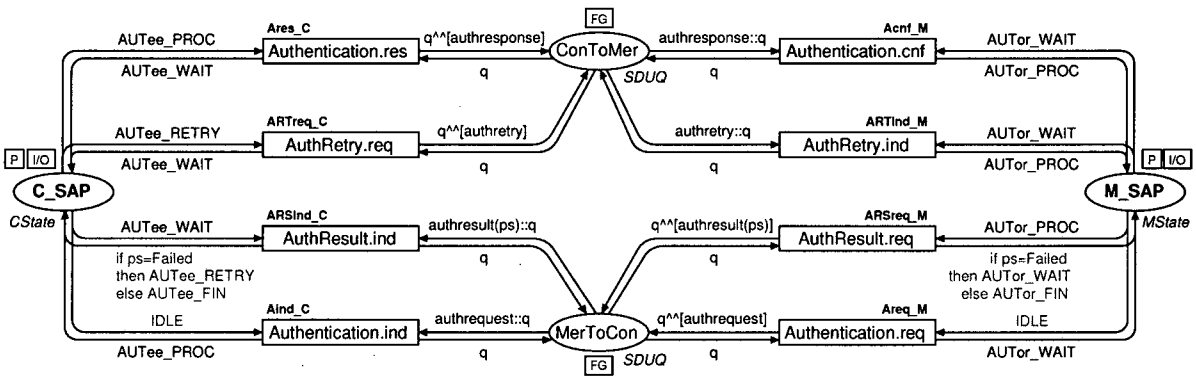


Figure 8.10: AuthMC page in the IOTS CPN model.

The Consumer can authenticate the Merchant during an Offer phase (see page Offer in Figure 8.3). This is modelled by the AuthCM page in Figure 8.9. The Consumer initiates the authentication when the C-SAP interface is in `OFR_PROC` (on occurrence of a *TradingOptions.ind*), and completes the authentication when it enters `AUTor_FIN` (where a *TradingOptions.res* can be submitted). At the Merchant side, the authentication starts when the M-SAP interface is in `OFR_WAIT` and completes when it returns to this state (waiting for a *TradingOptions.cnf*).

The Merchant may also authenticate the Consumer at the beginning of the Offer phase, as specified by the AuthMC page in Figure 8.10. The Merchant initiates authentication when the M-SAP interface is `IDLE`, and completes the authentication when it enters `AUTor_FIN` (where a *TradingOptions.req* can be submitted). Similarly, at the Consumer side, authentication starts when the C-SAP interface is `IDLE` and completes when it enters `AUTee_FIN` (waiting for a *TradingOptions.ind*).

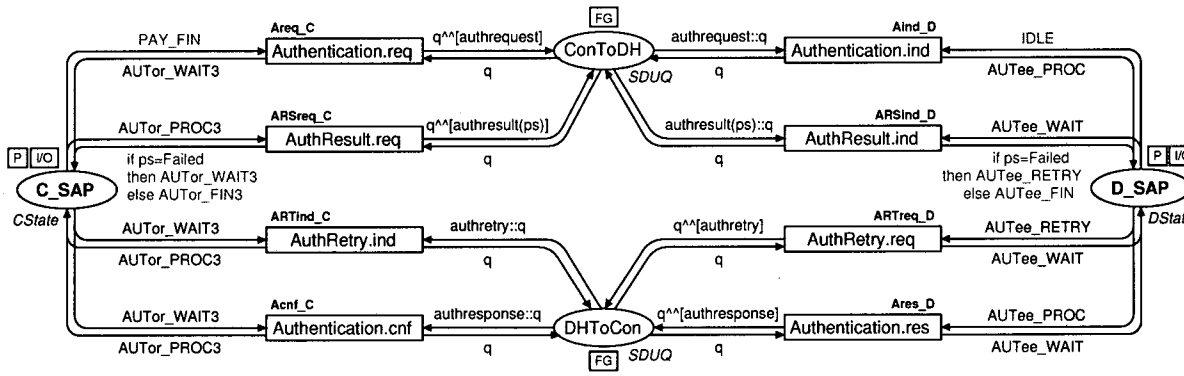


Figure 8.11: AuthCD page in the IOTS CPN model.

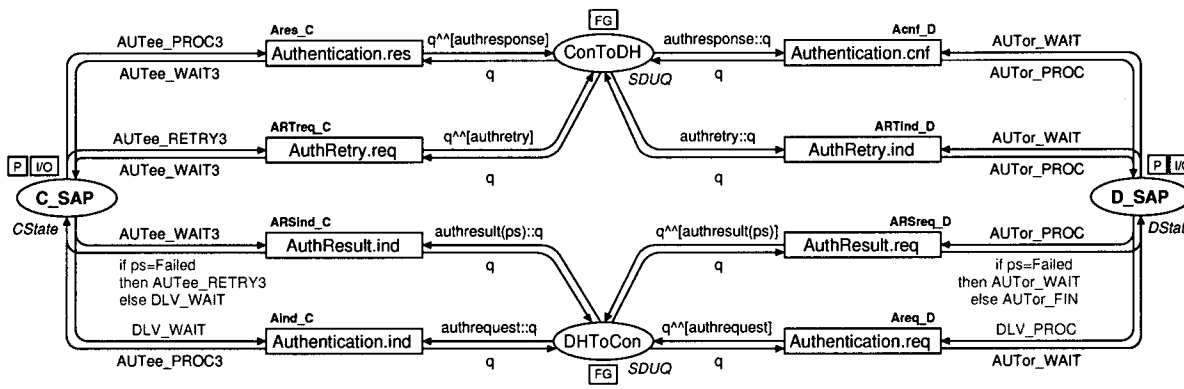


Figure 8.12: AuthDC page in the IOTS CPN model.

The Consumer may authenticate the Delivery Handler before the Delivery phase (see the Delivery page in Figure 8.5) starts. This is modelled by the AuthCD page in Figure 8.11. The Consumer initiates authentication when the C-SAP interface is in `PAY_FIN`, and completes the authentication when it enters `AUTor_FIN3` (where a *Delivery.req* can be submitted). At the Delivery Handler side, authentication starts when the D-SAP interface is `IDLE` and completes when it enters `AUTee_FIN` (waiting for a *Delivery.ind*).

The Delivery Handler can also authenticate the Consumer during the Delivery phase, as specified by the AuthDC page in Figure 8.12. The Delivery Handler initiates authentication when the D-SAP interface is in `DLV_PROC` (after a *Delivery.ind* occurs), and completes the authentication when it enters `AUTor_FIN` (where a *Delivery.res* can be submitted). At the

Consumer side, authentication starts when the C-SAP interface is in DLV_WAIT and completes when it returns to this state (waiting for a *Delivery.cnf*). The state AUTee_FIN3 defined in the local C-SAP FSA is not used, as it is replaced by the DLV_WAIT state and the arrival of an authresult(CompletedOk) SDU from place DHToCon on page AuthDC.

8.4.6 Cancel Phase Pages

The Cancel page shown in Figure 8.13 provides an overview of the three Cancel phases, each involving two different trading role entities. The page consists of four SAP places and three substitution transitions. The four SAP places are linked to the places with the same name on IOTS_Global (Figure 8.2), and the three substitution transitions correspond to the three Cancel phases. Each transition has its name (shown in bold) positioned (left) above it, and has an associated subpage that is named accordingly.

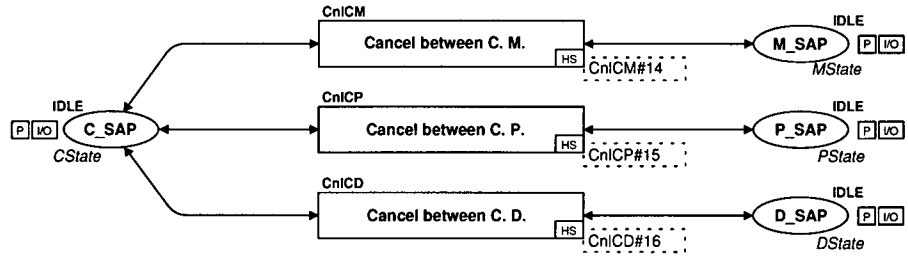


Figure 8.13: Cancel page in the IOTS CPN model.

Each of the three subpages models the global behaviour of cancelling an ongoing transaction service between a Consumer and one of the three non-Consumer entities. However, to capture all the possible cancellation scenarios in the IOTS CPN, we first need to obtain an overall understanding that how cancellations occur *globally* given their *local* behaviour that has been defined in Section 7.3.3. Based on this, we then discuss how the transaction service cancellations are modelled in each of the three subpages of the Cancel page.

Discussion of Transaction Service Cancellation

Cancellation can be initiated by a trading role entity submitting a *Cancel.req* or by the IOTS Provider delivering a *Cancel.ind*. Definition 7.4 to 7.6 in Section 7.3.3 specify all valid *Cancel* primitive events at each of the four trading role SAPs. Below, we review the local behaviour of the transaction service cancellations given by these four definitions.

- At each local trading role SAP, neither a *Cancel.req* nor a *Cancel.ind* primitive can occur if the SAP interface is in the initial state IDLE or the final state STOP.
- According to Definition 7.3, when the M-SAP interface is in OFR_FIN (upon submitting a *TradingResult.req*), a *Cancel.req* primitive cannot occur but a *Cancel.ind* can occur.
- According to Definition 7.4, when the P-SAP is in PAY1_FIN (upon submitting a successful *PayResult.req*), a *Cancel.req* primitive cannot occur but a *Cancel.ind* can occur.

Based on the above, we discuss the impact that the *local Cancel* primitive events have on the *global* trading transaction services. First of all, no cancellation can occur when the M-SAP interface is IDLE because no transaction service has yet started, nor can any cancellation

occur once the C-SAP interface enters STOP because the overall transaction service has already terminated. In terms of primitive events, a *Cancel.req* or *Cancel.ind* primitive may occur after a *TradingOptions.req* or *Authentication.req* primitive has been submitted by the Merchant but before a *PayResult.ind* or a *Delivery.cnf* is delivered to the Consumer. In the following, we consider the different cases when cancellations occur.

Cancellation *during* a trading phase. This refers to a cancellation that occurs after a trading phase has been initiated but before it completes at *both* a Consumer entity and a non-Consumer entity. In general, there are four cases when *both* trading roles involved in the trading phase are aware that a cancellation has occurred *during* that trading phase. The four representative examples can be seen in Figure 7.6, which illustrates the four cancellation scenarios between a Consumer and a Merchant during the Offer phase.

1. One trading role submits a *Cancel.req*, and as a result, the peer trading role is delivered a *Cancel.ind* (Figure 7.6 (a)).
2. One trading role submits a *Cancel.req*, whereas the peer trading role is delivered a *Cancel.ind* as a result of a cancellation that is initiated by the IOTS Provider and is independent of that *Cancel.req* (Figure 7.6 (b)).
3. Both trading roles submit *Cancel.req* primitives (Figure 7.6 (c)).
4. Both trading roles are delivered *Cancel.ind* primitives, due to a cancellation initiated by the IOTS Provider (Figure 7.6 (d)).

Cancellation *at the beginning of* a trading phase. This refers to a cancellation that occurs when the transaction service has been initiated at *only one* of the two trading role entities within a certain trading phase. In this case, the IOTS provider may receive a *Cancel.req* from or deliver a *Cancel.ind* to one entity before initiating the transaction service at the peer entity. Therefore, it is legal for the *Cancel* primitives to occur only at the entity which has started the transaction service, but not the peer entity which is still *idle*.

Figure 8.14 illustrates two valid primitive sequences of cancellations at the beginning of an Offer phase, where a *Cancel.req* issued by the Merchant (in (a)) or a *Cancel.ind* by the IOTS Provider (in (b)) after a *TradingOptions.req*, is not delivered to the Consumer if no *TradingOptions.ind* has yet occurred.

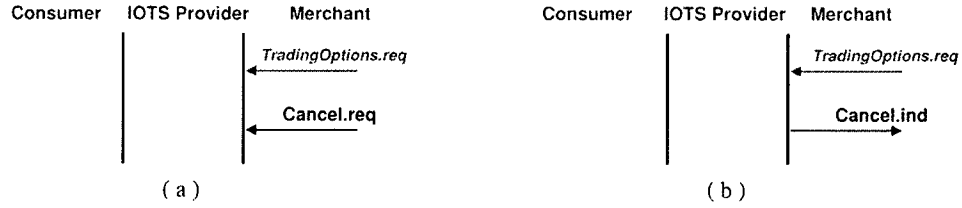


Figure 8.14: Two valid primitive sequences of cancellations at the beginning of an Offer phase with *TradingOptions.req* followed by (a) *Cancel.req* or (b) *Cancel.ind* at the Merchant entity only.

Cancellation *between* two trading phases. This refers to a cancellation that occurs after a trading phase completes but before the next one starts, i.e., between Offer and Payment, or between Payment and Delivery (where the Delivery Handler provides delivery). In this case, cancellations occur to the Consumer and Merchant but not the Payment Handler and Delivery

Handler which are either in state *idle* or *stop*. Since the M-SAP interface is in OFR_FIN, the Merchant cannot submit a *Cancel.req* but may be delivered a *Cancel.ind*, and this *Cancel.ind* only occurs as a result of the cancellation occurring to the Consumer.

Figure 8.15 shows three typical scenarios where cancellations occur between two trading phases. In (a), the Consumer initiates cancellation between Offer and Payment, and in (b), the Consumer initiates cancellation between Payment and Delivery. Each scenario consists of the events of Consumer submitting a *Cancel.req* followed by the corresponding *Cancel.ind* being delivered to the Merchant. In (c), the IOTS Provider initiates cancellation by delivering a *Cancel.ind* to both the Consumer and the Merchant.

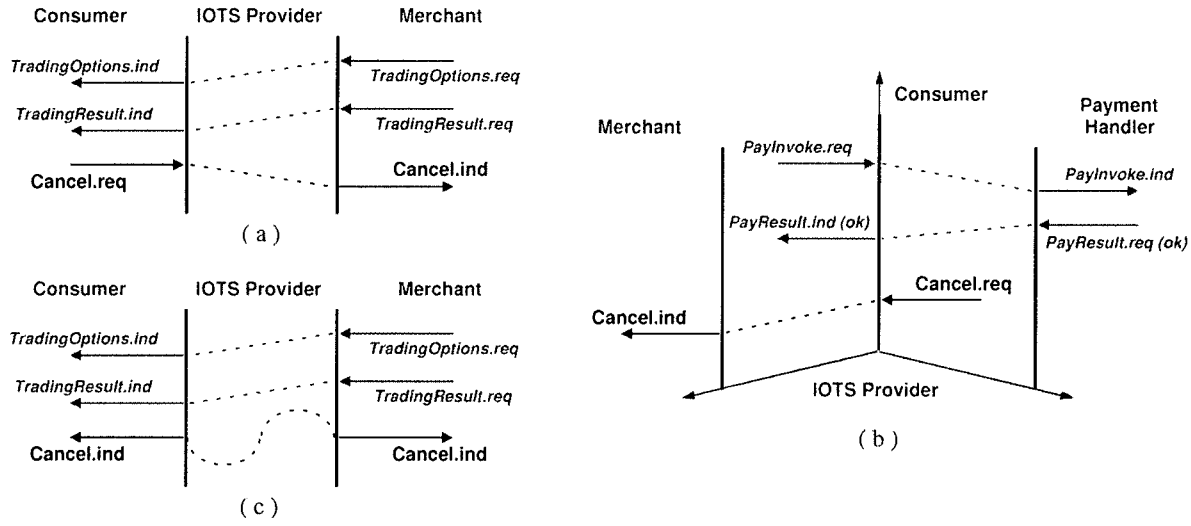


Figure 8.15: Three scenarios of cancellations between two trading phases, where (a) the Consumer initiates cancellation between Offer and Payment, (b) the Consumer initiates cancellation between Payment and Delivery, and (c) the IOTS Provider initiates cancellation between Offer and Payment.

Cancellation *after* the Offer phase. This includes all the above cases of cancellations after the Offer phase. The Merchant, which is the only trading role that indeed provides the goods or services as well as authorises a refund, should be informed of cancellation at any time when the transaction service is in progress. This is also the key reason why a *Cancel.ind* can still occur at the Merchant when the M-SAP interface is in one of its *final* states OFR_FIN.

Figure 8.16 depicts four possible scenarios where cancellations occur at the beginning of or during the Payment phase. In (a), the Consumer issues cancellation before payment is initiated at the Payment Handler, and therefore only the Merchant is notified about the cancellation. In (b), the Consumer cancels the transaction service after payment has started at the Payment Handler. Both the Merchant and the Payment Handler are notified about the cancellation. In (c), the Payment Handler initiates cancellation, and as a result, both the Consumer and the Merchant are informed of the cancellation. Finally, in (d), the IOTS Provider initiates cancellation which occurs at all three trading roles: the Consumer, the Merchant and the Payment Handler.

Modelling of Transaction Service Cancellation

The three pages, CnlCM, CnlCP and CnlCD, as shown in Figure 8.17 to 8.19, model transaction service cancellations. Each page has six primitive transitions modelling the *Cancel* primitive events and one non-primitive transition capture used to capture the cancellation scenarios

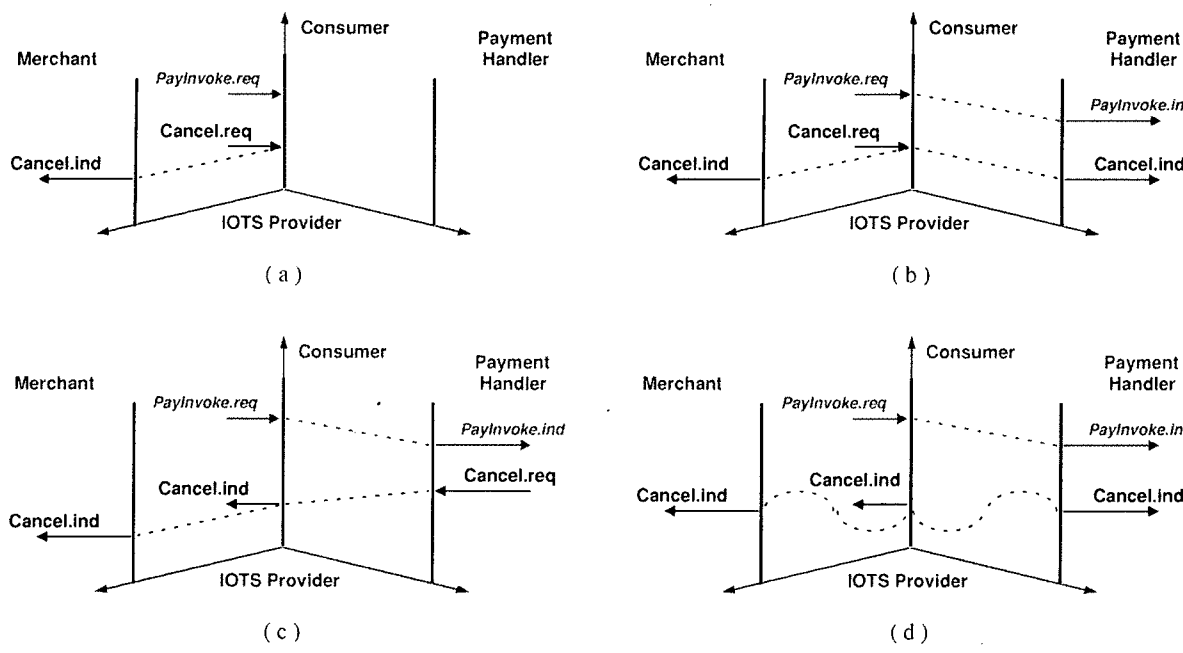


Figure 8.16: Four scenarios of cancellations where (a) the Consumer submits a *Cancel.req* before a *PayInvoke.ind* and the corresponding *Cancel.ind* occurs at the Merchant only, (b) the Consumer submits a *Cancel.req* after a *PayInvoke.ind* and the *Cancel.ind* occurs at both the Merchant and the Payment Handler, (c) the Payment Handler submits a *Cancel.req* and the *Cancel.ind* occurs at the Consumer and then the Payment Handler, and (d) the IOTS Provider issues *Cancel.ind* to the Consumer, the Merchant and the Payment Handler, respectively.

occurring at the beginning of a trading phase. The name of each *Cancel* primitive transition (e.g., Creq_C) comprises an abbreviated primitive name (e.g., Creq for *Cancel.req*) and the initial letter of the trading role at which the primitive occurs (e.g., C for Consumer). The meaning of these transitions is given in Table 8.10.

Transition	Primitive	Meaning
Creq_C	Cancel.req	Cancel request submitted by the Consumer
Cind_C	Cancel.ind	Cancel indication delivered to the Consumer, resulting from the cancellation initiated by the peer non-Consumer entity
Cindp_C	Cancel.ind	Cancel indication delivered to the Consumer, resulting from the cancellation initiated by the IOTS Provider
Creq_M	Cancel.req	Cancel request submitted by the Merchant
Cind_M	Cancel.ind	Cancel indication delivered to the Merchant, resulting from the cancellation occurring at the Consumer
Cindp_M	Cancel.ind	Cancel indication delivered to the Merchant, resulting from the cancellation initiated by the IOTS Provider
Creq_P	Cancel.req	Cancel request submitted by the Payment Handler
Cind_P	Cancel.ind	Cancel indication delivered to the Payment Handler, resulting from the cancellation initiated by the Consumer
Cindp_P	Cancel.ind	Cancel indication delivered to the Payment Handler, resulting from the cancellation initiated by the IOTS Provider
Creq_D	Cancel.req	Cancel request submitted by the Delivery Handler
Cind_D	Cancel.ind	Cancel indication delivered to the Delivery Handler, resulting from the cancellation initiated by the Consumer
Cindp_D	Cancel.ind	Cancel indication delivered to the Delivery Handler, resulting from the cancellation initiated by the IOTS Provider

Table 8.10: Meaning of *Cancel* primitive transitions on CnlCM, CnlCP and CnlCD pages.

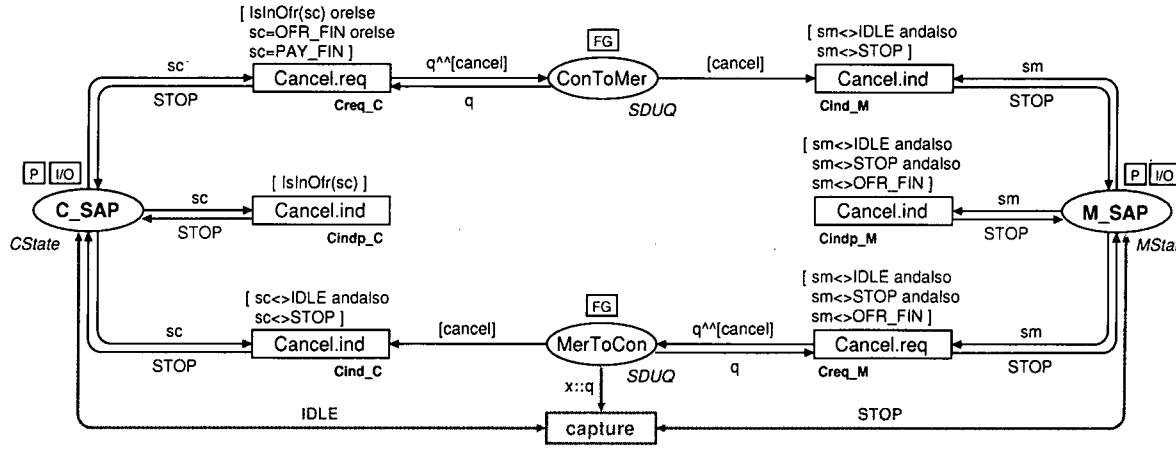


Figure 8.17: CnICM page in the IOTS CPN model.

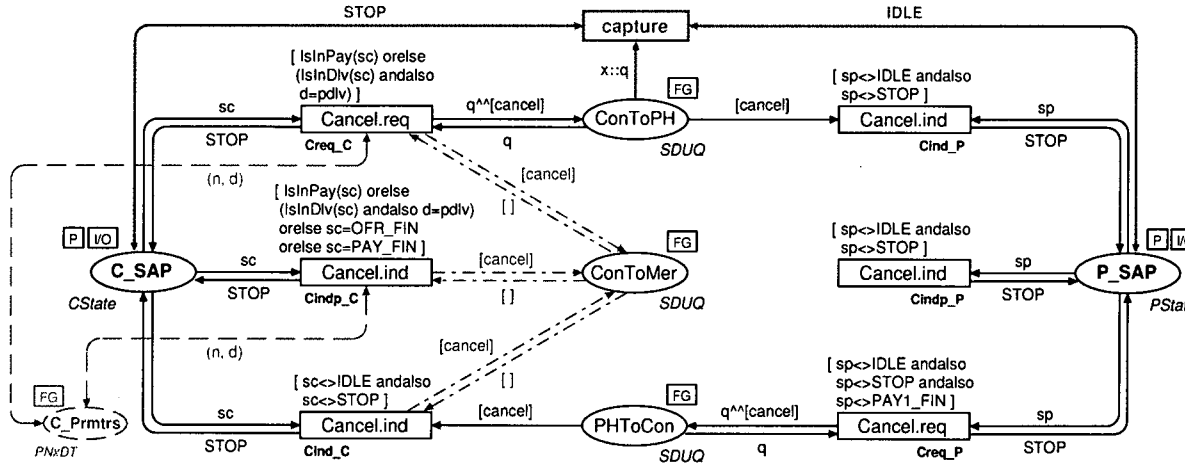


Figure 8.18: CnICP page in the IOTS CPN model.

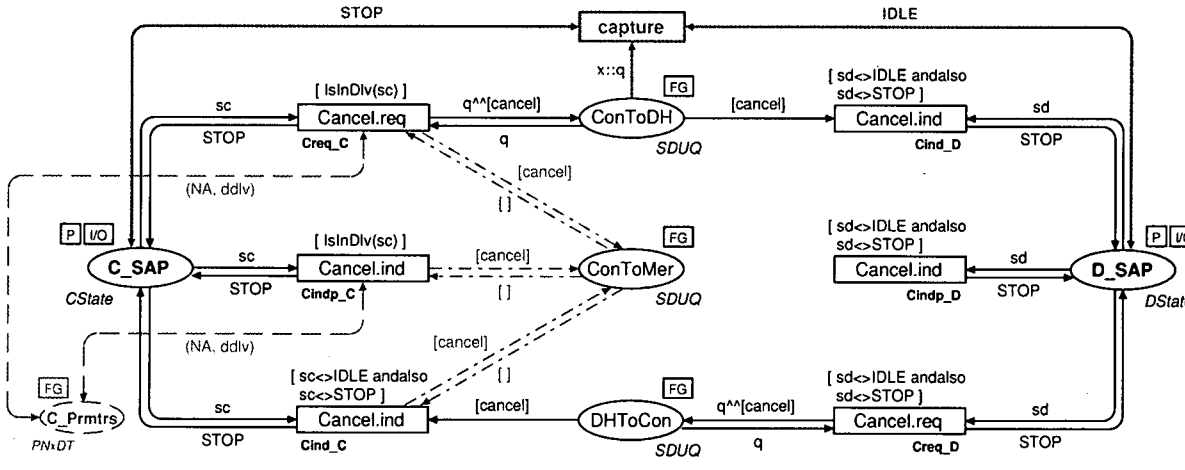


Figure 8.19: CnICD page in the IOTS CPN model.

The CnlCM page. The Consumer side primitive transitions each has an input arc from and an output arc to place C_SAP. The input arc has an inscription of sc, which can be bound to any value in the colour set CState. The output arc has an inscription of STOP, representing the C_SAP interface enters the final state STOP upon occurrence of the corresponding *Cancel* primitive event. The arcs between each primitive transition and place M_SAP at the Merchant side are modelled in a similar way.

The arcs between transition Creq_C and place ConToMer or between transition Creq_M and place MerToCon, model adding a cancel SDU to the end of the SDU queue within the IOTS Provider when the Consumer or the Merchant initiates cancellation. Transitions Cind_C and Cind_M each has an input arc from place MerToCon or ConToMer, modelling the receipt of the cancel SDU at the local trading role SAP as a result of the cancellation occurring at the peer trading role SAP. Each arc is inscribed with a value of a single element list [cancel], because no more SDUs will be added to the queue after a cancellation has occurred.

At the Consumer side, the guard on Creq_C or Cindp_C specifies the set of C-SAP interface states that enable the transition, and applies a boolean function IsInOfr defined in Table 8.11. IsInOfr identifies whether a C-SAP interface state (sc) is one of those occurring within an Offer phase (involving the combined Authentication phase). Using such a function as opposed to showing the complete list of valid C-SAP interface states in the guard on Creq_C or Cindp_C, increases the clarity of the CnlCM page. Apart from the states specified in function IsInOfr, Creq_C can also be enabled in OFR_FIN or PAY_FIN, modelling the *Cancel.req* submitted by the Consumer between Offer and Payment or between Payment and Delivery. The guard on Cind_C indicates that the transition is not enabled if the C-SAP interface is IDLE or in STOP.

```
1  fun IsInOfr ( sc:CState ) : bool =
2      sc=OFR_PROC orelse sc=OFR_WAIT orelse
3      sc=AUTor_PROC orelse sc=AUTor_WAIT orelse sc=AUTor_FIN orelse
4      sc=AUTee_PROC orelse sc=AUTee_WAIT orelse sc=AUTee_RETRY orelse
5      sc=AUTee_FIN;
```

Table 8.11: Definition of the function IsInOfr.

At the Merchant side, neither Creq_M nor Cindp_M can be enabled if the M-SAP interface is IDLE or in any of the two final states OFR_FIN and STOP. Transition Cind_M has a different guard specifying that the transition is not enabled in the IDLE and STOP states (but not OFR_FIN). The occurrence of Cind_M with sm bound to OFR_FIN models the *Cancel.ind* at the Merchant after the Offer phase is complete. This *Cancel.ind* only occurs as a result of the cancellation (either *Cancel.req* or *Cancel.ind*) occurring to the Consumer, and is therefore different from the one modelled by transition Cindp_M.

The occurrences of transitions Creq_C, Cindp_C, Creq_M and Cindp_M are *non-deterministic*, indicating the corresponding *Cancel* primitive events that occur at the Consumer and the Merchant entities are independent of each other. For example, it could be the occurrences of Creq_C (a *Cancel.req* issued by the Consumer) followed by Cindp_M (a *Cancel.ind* initiated by the IOTS Provider), or vice versa, given that both the C-SAP and M-SAP interface states satisfy the boolean guard expressions on the corresponding transitions.

Finally, the only non-primitive transition capture is used to capture all possible scenarios of cancellations that only occur to the Merchant at the beginning of an Offer phase. In such a scenario, the M-SAP enters STOP and the C-SAP remains IDLE when the transaction service

is cancelled. This is modelled by the directional arc between capture and place M_SAP and the one between capture and place C_SAP. The SDU queue from the Merchant to Consumer contains at least one SDU resulting from the first primitive submitted by the Merchant to initiate a transaction service, and all the SDUs in this SDU queue will be discarded after the transaction service has been cancelled. This is modelled by the arc from place MerToCon to capture, with an inscription of x::q representing a non-empty SDU queue.

Since the Merchant can only submit a non-*Cancel* primitive (e.g., a *TradingOptions.req*) when the M-SAP interface is IDLE, the first element (e.g., a trdoptions) in the above SDU queue is always a non-*Cancel* SDU. After that, a *Cancel.req* (Creq_M) or a *Cancel.ind* (Cindp_M) may occur at the Merchant, upon which the M-SAP interface enters STOP. However, given that the C-SAP interface is IDLE, the corresponding primitive (e.g., a *TradingOptions.ind*) can always occur on arrival of the first non-*Cancel* SDU in the queue, initiating the transaction service at the C-SAP. With transition capture, the above SDU queue may be removed from MerToCon once the M-SAP interface enters STOP. This prevents the occurrence of the first non-*Cancel* primitive at the C-SAP, and thus makes it possible on page CnlCM to capture the transaction service termination where the C-SAP interface is IDLE and the M-SAP interface in STOP.

The CnlCP page. This page is similar to page CnlCM but has two more places: the parameter place C_Prmtrs and the Provider place ConToMer. Below, we do not describe in detail the CnlCP page, except the guards on primitive transitions, and the arcs between the Consumer side primitive transitions and the above two places.

At the Consumer side, the two functions IsInPay and IsInDlv, as defined in Table 8.12, are used in the guards on Creq_C and Cindp_C. These two functions identify whether a C-SAP interface state (sc) is one of those *during* a Payment phase (IsInPay) or a Delivery phase (IsInDlv). Together with IsInDlv, the delivery type information is necessary in determining whether a Payment Handler or Delivery Handler provides delivery in the Delivery phase (i.e., d=pdlv or d=ddlv). Creq_C and Cindp_C can be enabled on page CnlCP if d=pdlv (see page CnlCD for d=ddlv). In addition, Cindp_C can also be enabled in OFR_FIN or PAY_FIN, modelling the *Cancel.ind* issued by the IOTS Provider to the Consumer between Offer and Payment or between Payment and Delivery. The guard on Cind_C is as same as that in the CnlCM page.

```
1  fun IsInPay ( sc:CState ) : bool =
2      sc=PAY_PROC orelse sc=PAY_WAIT orelse sc=PAY_RETRY orelse
3      sc=PAY_RETRYD orelse sc=PAY1_FIN orelse
4      sc=AUTor_PROC2 orelse sc=AUTor_WAIT2 orelse sc=AUTor_FIN2 orelse
5      sc=AUTee_PROC2 orelse sc=AUTee_WAIT2 orelse sc=AUTee_RETRY2 orelse
6      sc=AUTee_PROCD orelse sc=AUTee_WAITD orelse sc=AUTee_RETRYD;
7  fun IsInDlv ( sc:CState ) : bool =
8      sc=DLV_PROC orelse sc=DLV_WAIT orelse sc=DLV_KEEP orelse
9      sc=AUTor_PROC3 orelse sc=AUTor_WAIT3 orelse sc=AUTor_FIN3 orelse
10     sc=AUTee_PROC3 orelse sc=AUTee_WAIT3 orelse sc=AUTee_RETRY3;
```

Table 8.12: Definitions of two functions IsInPay and IsInDlv.

At the Payment Handler side, a *Cancel.ind* primitive modelled by either Cind_P or Cindp_P cannot occur if the P-SAP interface is IDLE or in the final state STOP. The Payment Handler is not allowed to submit a *Cancel.req* when the P-SAP interface is in PAY1_FIN. Thus, transition Creq_P has a different guard specifying it is not enabled in IDLE, STOP and also PAY1_FIN.

Each bidirectional arc between place C_Prmtrs and transition Creq_C or Cindp_C is used to provide the delivery type required by the transition guard. The arcs between each Consumer side primitive transition and place ConToMer model adding a cancel SDU to the empty queue ([ ]) between the Consumer and Merchant, on occurrence of a *Cancel* primitive at the C-SAP. As a result, transition Cind_M on page CnlCM will occur on arrival of such a cancel SDU, indicating the Merchant is informed of each cancellation between the Consumer and Payment Handler after the Offer phase.

The CnlCD page. This page applies the similar modelling mechanism as that on page CnlCP. Transitions Creq_C and Cindp_C are only enabled if the C-SAP interface state is one of those occurring within a Delivery phase (IsInDlv(sc)) and also the delivery is provided by the Delivery Handler (d=ddlv). At the Delivery Handler side, each *Cancel* primitive transition cannot occur if the D-SAP interface is IDLE or in STOP. The arcs between each Consumer side primitive transition and place ConToMer ensure that the Merchant is informed of each cancellation between the Consumer and Delivery Handler after the Offer phase.

Chapter 9

Validation of the Internet Open Trading Service

In Chapter 7 we used FSA to define the local IOTS language for each of the four trading roles. In Chapter 8 we developed a CPN model as a specification of the global service, in order to determine the global sequences of service primitives for IOTS.

In this chapter, we validate the IOTS by checking the consistency between the global service language obtained from the CPN model and the local primitive sequences defined by each local FSA. Due to the inconsistencies detected between the global and the local behaviour of IOTS in Section 9.1, we revise the IOTS CPN to remove the errors and analyse the global service language obtained from the revised CPN model in Section 9.2. An early version of the IOTS validation was published in [182].

9.1 Validation of the IOTS

The IOTS is validated using the methodology outlined in Section 5.4.3. In this section, we present the results of the IOTS state space analysis, language generation and comparison.

9.1.1 State Space Analysis

The full state space of the IOTS CPN model is calculated on a computer with the following configuration: Intel Pentium III (700 MHz), 512 Mb RAM, running Design/CPN 4.0.4 on a Linux Red Hat 7.0 Operating System. The state space report produced by Design/CPN is included in Appendix C.

Table 9.1 gives the statistics for the IOTS state space, showing that the SCC graph generated from the IOTS CPN model has a smaller size than the corresponding occurrence graph (or state space). This means that the IOTS state space contains cycles. We recall the modelling details on the Payment page (in Section 8.4.3) and each of the Authentication phase pages (in Section 8.4.5) in the IOTS CPN model. The global behaviour of IOTS regarding payment data exchange, payment retry and authentication retry results in cycles in the IOTS state space.

	Occurrence Graph	SCC Graph
Nodes	3056	2975
Arcs	6717	5009
Calculation Time (secs)	16	1

Table 9.1: State space statistics of the IOTS CPN model.

The IOTS state space has 799 dead markings (see Appendix C). These mainly comprise different combinations of the markings of the six Provider places (ConToMer, MerToCon, ConToPH, PHToCon, ConToDH and DHToCon) in the IOTS CPN model, and are a result of different sequences of service primitives leading to the termination (particularly the cancellation) of a transaction service. For example, if both the Consumer and Merchant submit *Cancel.req* primitives, then the transaction is terminated with cancel SDUs remaining in the places ConToMer and MerToCon. However, if the Consumer submits a *Cancel.req* adding a cancel SDU to the SDU queue in ConToMer and a *Cancel.ind* is delivered to the Merchant on receipt of that cancel SDU from ConToMer, the places ConToMer and MerToCon would then be empty.

A ML function named InValidTerminal (see the function code in Appendix C) is written to check that each dead marking in the IOTS state space is desired, and returns the dead markings that are undesired. A dead marking is desired if it has the form of one of the markings given in Tables 9.2 and 9.3.

Place	**Marking (M)**			
	$M1$	$M2$	$M3$	$M4$
C_SAP	1'STOP	1'STOP	1'STOP	1'STOP
M_SAP	1'OFR_FIN	1'OFR_FIN	1'OFR_FIN	1'OFR_FIN
P_SAP	1'STOP	1'STOP	1'STOP	1'STOP
D_SAP	1'IDLE	1'IDLE	1'IDLE	1'STOP
C_Prmtrs	1'(NA,ndlv)	1'(NA,pdlv)	1'(Pay2,ndlv)	1'(NA,ddlv)
M_Prmtrs	empty	empty	empty	empty
P_Prmtrs	empty	empty	empty	empty
ConToMer	1'[ ]	1'[ ]	1'[ ]	1'[ ]
MerToCon	1'[ ]	1'[ ]	1'[ ]	1'[ ]
ConToPH	1'[ ]	1'[ ]	1'[ ]	1'[ ]
PHToCon	1'[ ]	1'[ ]	1'[ ]	1'[ ]
ConToDH	1'[ ]	1'[ ]	1'[ ]	1'[ ]
DHToCon	1'[ ]	1'[ ]	1'[ ]	1'[ ]

Table 9.2: Forms of the desired dead markings for successful transactions in the IOTS state space.

Place	**Marking Set (MS)**			
	$MS1$	$MS2$	$MS3$	$MS4$
C_SAP	1'IDLE	1'STOP	1'STOP	1'STOP
M_SAP	1'STOP	1'STOP	1'STOP	1'STOP
P_SAP	1'IDLE	1'IDLE	1'STOP	1'STOP
D_SAP	1'IDLE	1'IDLE	1'IDLE	1'STOP
C_Prmtrs	empty	<don't care>	non-empty	1'(NA,ddlv)
M_Prmtrs	<don't care>	<don't care>	empty	empty
P_Prmtrs	empty	empty	<don't care>	empty
ConToMer	1'[ ]	<don't care>	empty	empty
MerToCon	empty	<don't care>	1'[ ]	1'[ ]
ConToPH	1'[ ]	1'[ ] or empty	<don't care>	1'[ ]
PHToCon	1'[ ]	1'[ ]	<don't care>	1'[ ]
ConToDH	1'[ ]	1'[ ]	1'[ ] or empty	<don't care>
DHToCon	1'[ ]	1'[ ]	1'[ ]	<don't care>

Table 9.3: Forms of the desired dead markings for cancelled transactions in the IOTS state space.

In Table 9.2, $M1$ to $M4$ represent successful completions of a transaction service that involves a Delivery Handler ($M4$) or not ($M1$, $M2$, $M3$) (see the two places D_SAP and C_Prmtrs). The desired dead markings for a cancelled transaction service are divided into four sets, based on the states of the interfaces at the four trading role SAPs. Table 9.3 gives the forms of these four sets of markings. For each marking in the marking set $MS1$, a transaction service is cancelled at the Merchant only, before it can be initiated at the Consumer. The place M_Prmtrs may have different markings, which results in the set of markings comprising $MS1$. For example, if cancellation occurs before the transition TOreq on the Offer page (Figure 8.3), the place M_Prmtrs is empty, otherwise it may have a marking 1'(one, ndlv), 1'(one, pdlv), 1'(one, ddlv) or 1'(two, ndlv). $MS2$ is a set of markings representing cancelled transaction services between the Consumer and Merchant before payment starts at the Payment Handler. In $MS3$, the transaction services are cancelled before delivery starts at the Delivery Handler, and in $MS4$, the transaction services are cancelled during the Delivery phase that involves the Delivery Handler. Applying the function InValidTerminal to the IOTS state space indicates that all dead markings are desired (see the results in Appendix C).

9.1.2 Language Generation

The IOTS state space can be considered as a FSA to obtain the global service language, a necessary prerequisite for language analysis. Since Design/CPN does not provide the facilities required for doing this, the FSM tool (or, FSM) [11] is used, following the procedure of automata reduction, determinisation and minimisation illustrated in Figure 6.8 (Section 6.5). However, before applying that procedure, three steps are required to write the Design/CPN state space output into a text file that can be used as the input to FSM.

1. Mapping binding element occurrences in the IOTS state space to the alphabet of the language. The primitive transitions in the IOTS CPN model are assigned different non-*zero* integers according to Table D.1 (see Appendix D.1). The others, i.e., the non-primitive transition capture on pages CnlCM, CnlCP and CnlCD (Figures 8.17 to 8.19), are marked as *zero* (i.e., ϵ-transitions).

2. Specifying the halt (or, final) states of the FSA by the dead markings in the IOTS state space. The analysis results in Section 9.1.1 show that all dead markings in the IOTS state space represent the desired terminations of transaction services, and hence the possible completions of sequences of service primitives.

3. Writing the IOTS state space to a text file with the format given in Figure 6.9 that can be read by FSM. The source and destination state numbers correspond to the numbers of the markings in the state space. The arc numbers are those assigned to the binding element occurrences in the state space, determined in Step 1 above. The halt state numbers are given by the numbers of the dead markings in the IOTS state space, according to Step 2.

The ML code for implementing the above three steps is given in Appendix D.2.

9.1.3 Validating the Global IOTS Language

We wish to ensure that the global IOTS language preserves the local service language as specified for each trading role SAP in Section 7.3. This can be done by comparing the global

service language and the corresponding local service language at each of the four trading role SAPs. The procedure illustrated in Figure 5.8 (Section 5.4.3) is applied.

A local SAP FSA specifies the defined-local service language at the corresponding trading role SAP. A *SAP projected-FSA* represents the projected-local service language at that SAP and is obtained from the global service language, by projecting the global FSA onto the set of service primitives that can be viewed at that SAP only. All transition occurrences that are not labelled by primitives occurring at the SAP of interest are marked as ϵ-transitions in the IOTS state space, and hence they are removed in the automata reduction.

Both the local SAP FSA and the SAP projected-FSA need to be converted into minimal deterministic FSA, which are then compared to check for language equivalence (i.e., whether or not the languages represented by the two FSA are identical). Any difference indicates there are either missing sequences or invalid sequences in the global service language obtained from the IOTS CPN model. Language comparison (see Section 6.4.3) is conducted for each of the four trading role SAPs, using the fsmdifference program provided by FSM (see Section 6.5). It reveals inconsistencies between the defined- and the projected-local service languages at only the Payment Handler SAP.

Missing Sequences in the Projected-Local Service Language at the P-SAP

Comparison between the two FSA at the P-SAP reveals that there are primitive sequences defined in the local P-SAP FSA but missing in the P-SAP projected-FSA. GraphViz (see Section 6.5) is used for visualisation of FSA. Figure 9.1 shows the FSA for the missing sequences in the P-SAP projected-FSA. Each primitive sequence starts at the initial state (node 0) and finishes at a final state indicated by a double circle (node 7). The names of service primitives in the FSA are given in their abbreviated form as listed in Table D.1. Note that an arc labelled by two *Cancel* primitives which are separated by a comma, represents the occurrence of either of these two primitives [26]. For example, the arc from node 4 to node 7 in the FSA is labelled by *Creq_P, Cind_P*, indicating that either *Creq_P* or *Cind_P* occurs.

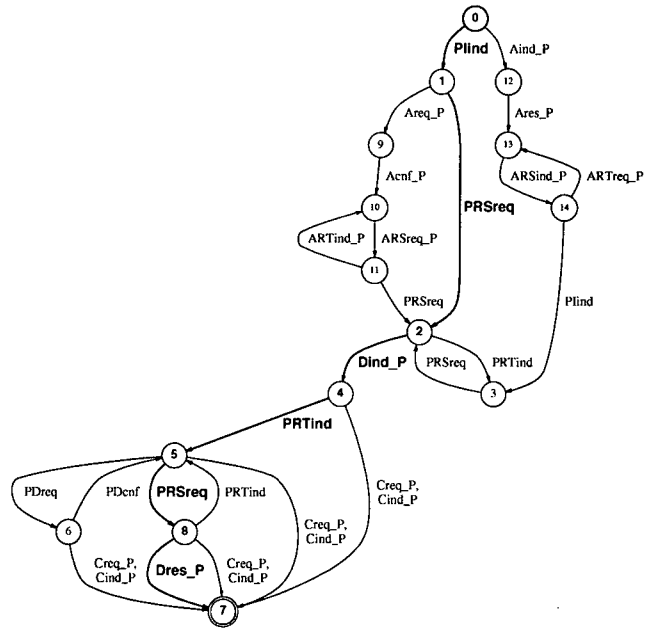


Figure 9.1: The FSA specifying primitive sequences that are defined in the local P-SAP FSA but missing in the P-SAP projected-FSA.

Figure 9.2 depicts the scenario of one of the missing sequences in the P-SAP projected-FSA, which is defined by the node sequence (0,1,2,4,5,8,7) in the FSA in Figure 9.1. It is a valid primitive sequence occurring at the P-SAP during a Purchase transaction with delivery provided by the Payment Handler. The sequence starts with a *PayInvoke.ind* (PIind from node 0 to 1), followed by a *PayResult.req* conveying a failed payment result (PRSreq from node 1 to 2), a *Delivery.ind* (Dind_P from node 2 to 4), a *PayRetry.ind* (PRTind from node 4 to 5), a *PayResult.req* conveying a successful payment result and finally a *Delivery.res* (PRSreq from node 5 to 8 followed by Dres_P from node 8 to 7). In Figure 9.2, the corresponding primitive sequence occurring at the C-SAP is also given.

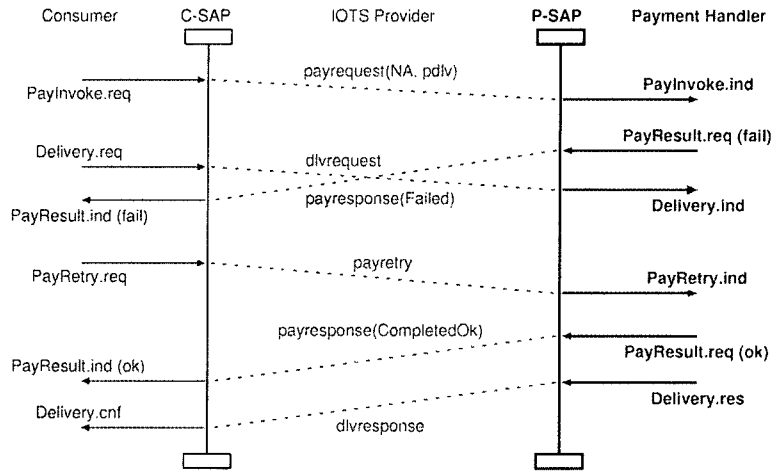


Figure 9.2: A valid primitive sequence at the P-SAP not preserved in the P-SAP projected-FSA.

The missing sequences in the P-SAP projected-FSA reveals that there are missing sequences in the global FSA (from which the P-SAP projected-FSA is obtained). Next, we discuss how to remove the above inconsistencies between the defined- and the projected-local service languages at the P-SAP.

9.2 Revised IOTS CPN model and its Analysis

The missing sequences in the projected-local service language at the P-SAP help us to correct the IOTS CPN model.

9.2.1 Revised IOTS CPN Model

From Figure 9.1 it can be observed that all the missing sequences in the P-SAP projected-FSA involve a *Delivery.ind*, as represented by Dind_P labelling the arc from node 2 to 4. This *Delivery.ind* only occurs after a failed *PayResult.req* but before the corresponding *PayRetry.ind*. This is evidenced by PRSreq labelling the arc from node 1, 3 or 11 to node 2 and PRTind labelling the arc from node 4 to 5 in the FSA in Figure 9.1. Thus, we examine the modelling of the Delivery phase between the Consumer and Payment Handler on page Delivery (Figure 8.5) in the IOTS CPN to discover the reason for the missing sequences in the P-SAP projected-FSA.

On the Delivery page shown in Figure 8.5, a *Delivery.ind* will never occur when the P-SAP interface is in PAY_RETRY, according to the guard on transition Dind_P. PAY_RETRY is defined as an interface state after a failed *PayResult.req* but before a *PayRetry.ind* occurs at the P-SAP.

The corrected Delivery page, as shown in Figure 9.3, therefore involves modifications made to transition Dind_P. The modified Dind_P allows a *Delivery.ind* to occur at the Payment Handler, when the P-SAP interface is in PAY_RETRY (see the modified transition guard) and a dlvrequest is at the head of the SDU queue in ConToPH. The occurrence of Dind_P changes the P-SAP interface state from PAY_RETRY to PAY_RETRYD (see the modified inscription on the arc from Dind_P to place P_SAP). When the P-SAP interface enters PAY_RETRYD, the Payment Handler is notified of the delivery request from the Consumer, and does not expect to receive it again.

Due to the complexity of IOTS, it is expected that the sort of modelling errors above will occur. Comparison between the defined- and the projected-local service languages at each entity is a way of detecting and removing these errors.

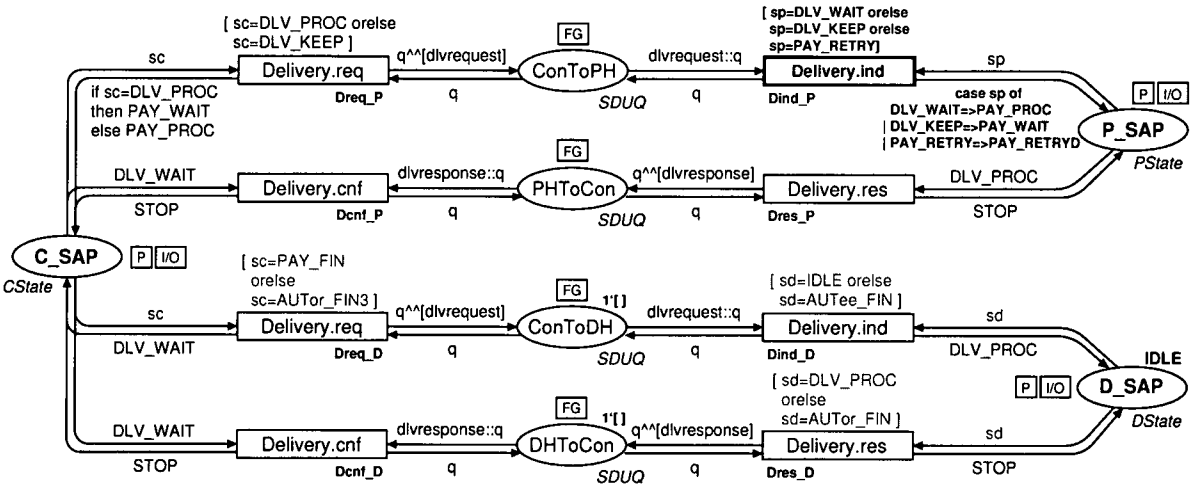


Figure 9.3: Corrected Delivery page in the IOTS CPN model.

9.2.2 Validation of the Revised IOTS CPN Model

This involves state space analysis and language validation as conducted on the previous IOTS CPN model in Section 9.1. Table 9.4 gives the statistics of the state space generated from the revised IOTS CPN model, and the state space report is included in Appendix C. It contains the same number of states and SCC nodes as the state space of the previous IOTS CPN model (see Table 9.1). The differences are in the number of arcs and the corresponding SCC arcs, where the state space of the revised CPN model has 15 more arcs and SCC arcs than the state space of the previous model.

	Occurrence Graph	SCC Graph
Nodes	3056	2975
Arcs	6732	6624
Calculation Time (secs)	16	1

Table 9.4: State space statistics of the revised IOTS CPN model.

With the revised IOTS CPN model, we conduct language comparison between the SAP projected-FSA and the local SAP FSA, which shows that the projected-local service language matches the defined-local service language at each trading role SAP. We are therefore highly confident that the global IOTS language obtained from the revised CPN model is correct. As an example, Figure 9.4 depicts the P-SAP projected-FSA for the revised IOTS CPN model.

Its language is identical to that of the local P-SAP FSA in Definition 7.4. The set of sequences defined by nodes connected via thick arcs in Figure 9.4 are those missing sequences in the P-SAP projected-FSA for the previous CPN model.

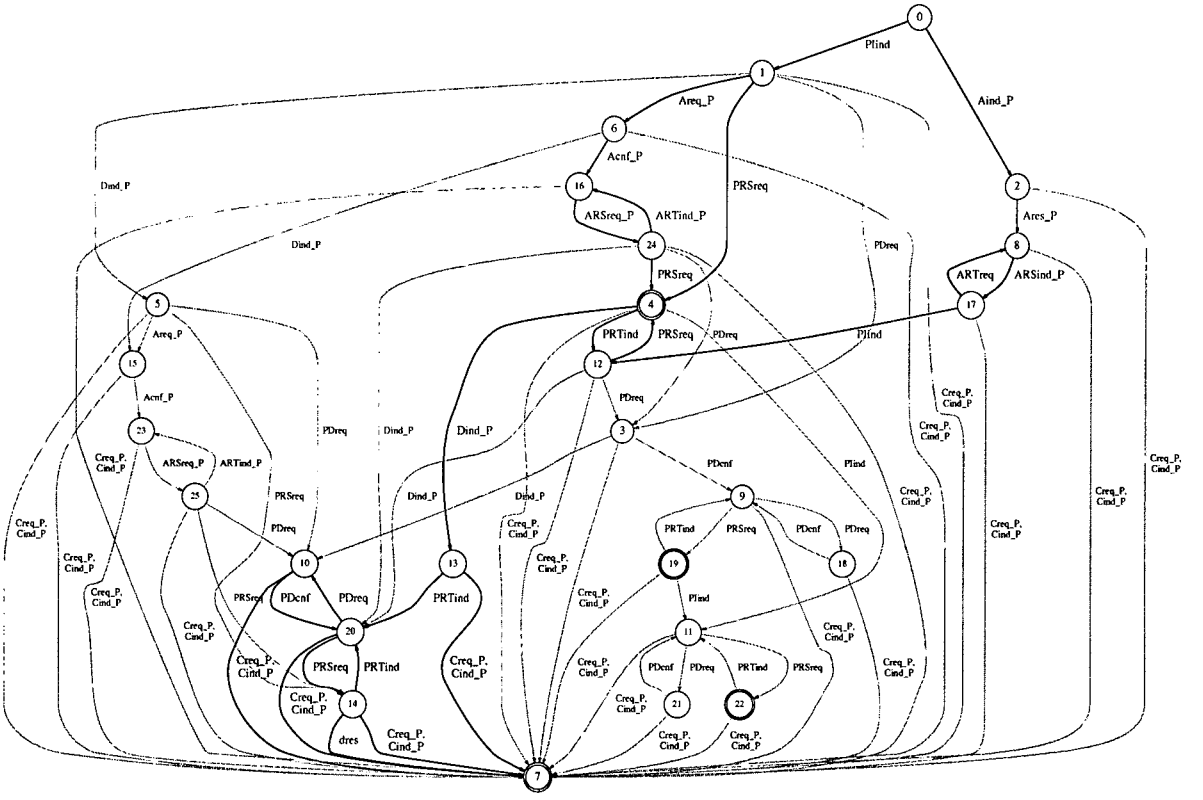


Figure 9.4: P-SAP projected-FSA obtained from the revised IOTS CPN model.

9.2.3 Language Analysis

The minimal (reduced) FSA of the revised CPN model is generated, using FSM version 3.6. It represents the global IOTS language. The FSA contains 296 states, 1206 arcs and 12 final states. It is thus much smaller than the corresponding state space in Table 9.4.

Visualisation is possible for FSA of small size, with support of GraphViz [12]. For illustration purpose, we consider an example of a Purchase transaction service with no delivery and no authentication. Figure 9.5 shows the minimized FSA for the global service language of this example using GraphViz. It is obtained from the state space of the revised CPN model with p=one and d=ndlv but without executing the Authentication page (Figure 8.6).

In Figure 9.5, the primitive sequences accepted by node 26 (i.e., a final state) represent successfully completed transaction services, indicated by thick arcs. The subsequence defined by nodes (16,18,23,30,16) represents a payment data exchange, and the subsequence defined by nodes (16,19,26,33,16) corresponds to a payment retry. The two subsequences start and end at the same node (i.e., node 16), indicating that payment data exchange may occur before or after payment retry and both can occur any number of times. Hence, the language represented by the FSA comprises an infinite number of service primitive sequences.

The rest of the primitive sequences in Figure 9.5 are accepted by one of the other four final states given by nodes 4, 14, 22 and 35, and represent all cancelled transaction services. Using dashed arcs, we mark out several of them to illustrate how they capture those cancellation scenarios shown in Figures 8.14 to 8.16 (Section 8.4.6).

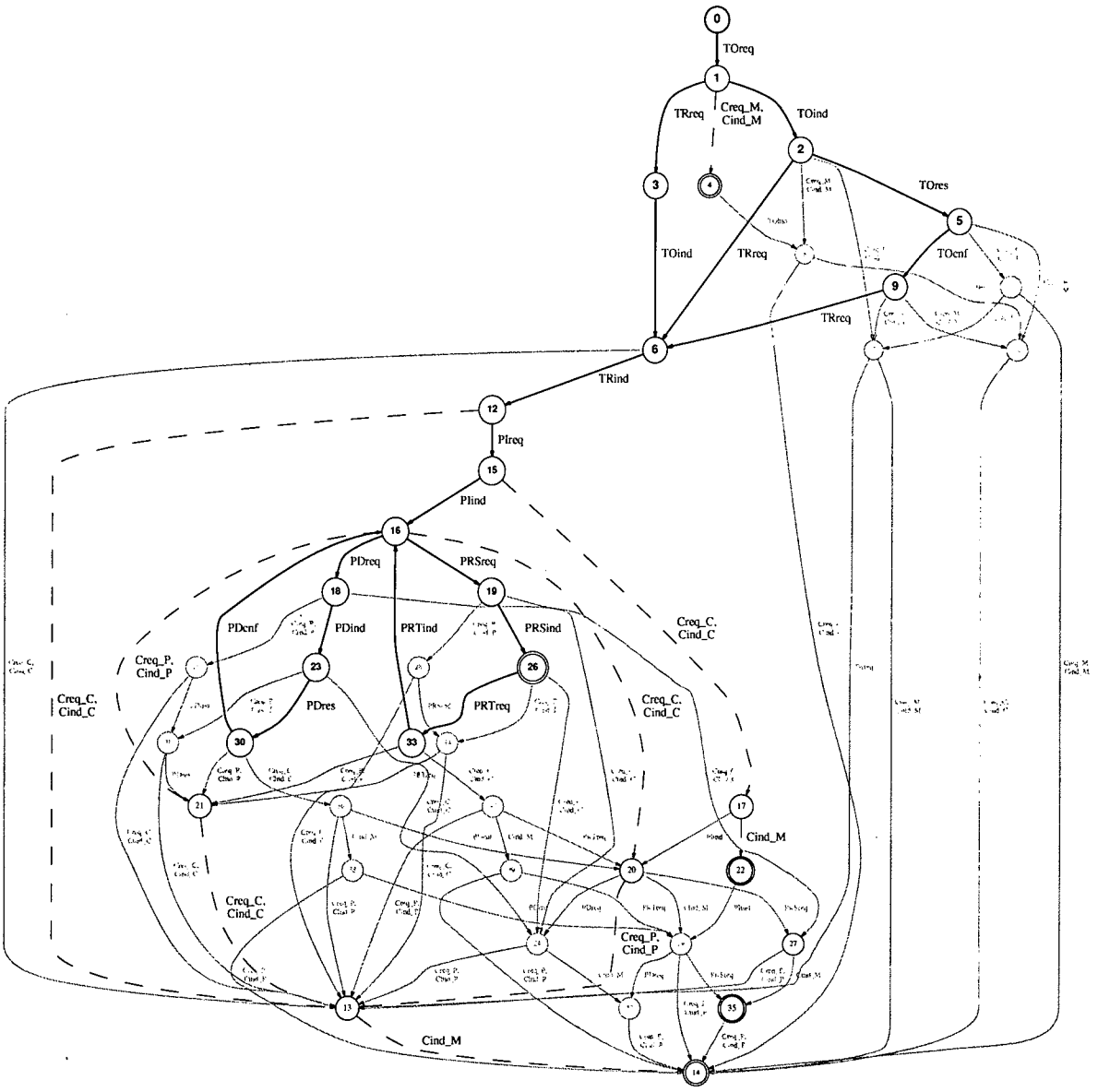


Figure 9.5: The FSA for the global service language of Purchase without delivery and authentication.

1. The two sequences (TOreq, Creq_M) and (TOreq, Cind_M) defined by nodes (0,1,4) correspond to the two cancellation scenarios in Figure 8.14 (a) and (b);
2. The sequences ending with Creq_C or Cind_C from nodes 12 to 13 followed by Cind_M from node 13 to 14, represent cancellations initiated by the Consumer or the IOTS Provider after the Offer phase has completed but before the Payment phase starts. The corresponding scenarios are shown in Figure 8.15 (a) and (c);
3. The sequences ending with Creq_C or Cind_C from nodes 15 to 17 followed by Cind_M from node 17 to 22, capture the cancellations occurring after the Payment phase has been initiated at the Consumer only, such as the scenario shown in Figure 8.16 (a);
4. Finally, the sequences ending with Creq_C or Cind_C from nodes 16 to 20 followed by Creq_P or Cind_P from nodes 20 to 13 and then Cind_M from nodes 13 to 14, or the sequences ending with Creq_P or Cind_P from nodes 16 to 21 followed by Creq_C or Cind_C from nodes 21 to 13 and then Cind_M from nodes 13 to 14, are examples of cancellations occurring after the payment has been initiated at both the Consumer and the Payment Handler. Three of these examples are shown in Figure 8.16 (b)-(d).

Chapter 10

Internet Open Trading Protocol CPN

RFC 2801 [36] contains an informal narrative description of IOTP and suffers from ambiguities and incompleteness. We create a CPN model of IOTP to obtain a formal specification of IOTP that truly reflect the intent of its RFC specification. When ambiguities and gaps are detected, assumptions are made to clarify the ambiguities and fill in the gaps. The model is validated against the IOTP description by checking that the model does correspond to the RFC. This chapter presents the CPN model of IOTP. In Chapter 11 the model is used for the analysis of a set of desired properties of IOTP, and in Chapter 12 it is verified against IOTS in terms of the global sequences of service primitives.

The CPN modelling and analysis of IOTP followed an incremental approach and went through several refinements. The work started with the modelling and analysis of the Deposit transaction [180]. This initial analysis revealed some deficiencies in the Deposit transaction procedure known as unspecified receptions. We constructed a partial hierarchical CPN model for the complete set of IOTP Authentication and Payment-related transactions [179]. This led to a proposal for the simplified conceptual framework for IOTP given in Section 4.1. To obtain a simpler and clearer model that better captured IOTP's transactions, the IOTP CPN model was then improved by modifying the modules for basic transaction procedures [181]. The analysis of this improved IOTP CPN model has revealed a lack of synchronization between the participating entities upon transaction cancellation (see Chapter 11). However, all the above work did not take into account all the possible circumstances for transaction cancellations (called arbitrary transaction cancellation) and error handling, which are essential for a complete specification of IOTP. The CPN model presented in this chapter includes these two procedures, and as a result, the complexity of the model (and hence its analysis) has increased significantly. The refined specification further exploits the modularity present in IOTP and enhances its readability by introducing a new hierarchy level associated with transactions, which re-uses modular exchange mechanisms in IOTP. A brief description of this IOTP CPN model and its initial analysis results can also be found in [178].

This chapter begins by defining an abstract model of IOTP in Section 10.1. Section 10.2 lists the scope of the work and the modelling assumptions. Section 10.3 describes the hierarchical structure of the model. The four layers of the IOTP CPN model are presented top-down in Sections 10.4 to 10.8.

10.1 IOTP Structure

The IOTS in Chapters 7 and 9 is defined by the service primitives and their parameters, and the complete set of valid primitive sequences between the IOTS users and the IOTS Provider. Figure 7.1 shows an abstract structure of the IOTS, where IOTP and its underlying protocol layers form the IOTS Provider. In this section, we decompose the IOTS Provider into IOTP and its underlying service provider, and thus obtain the structure of IOTP.

IOTP is a refinement of IOTS. It defines the procedures of the IOTP entities for providing IOTS to its users. The IOTP entities are trading role protocol entities and IOTS users are trading role users. For brevity, we hereafter refer to a trading role protocol entity as TRole_PE and a trading role user as TRole_User. There are four TRole_PEs: the Consumer protocol entity (Consumer_PE), the Merchant protocol entity (Merchant_PE), the Payment Handler protocol entity (PHandler_PE) and the Delivery Handler protocol entity (DHandler_PE). The corresponding four TRole_Users are: the Consumer user (Consumer_User), the Merchant user (Merchant_User), the Payment Handler user (PHandler_User) and the Delivery Handler user (DHandler_User). Figure 10.1 depicts the structure of IOTP and its relationship with the TRole_Users and the IOTP Transport Service Provider. For illustration purposes, we divide IOTP structure into four parts, each concerning a Consumer_PE and a non-Consumer_PE during certain document exchanges. For clarity, all the IOTS primitives are given while the primitive parameters are omitted from the structure. Below, we take the part in Figure 10.1 (a) as a representative example to describe how the structure of IOTP is defined.

In Figure 10.1 (a), the Consumer_PE and Merchant_PE exchange *protocol data units* (PDUs) to realise the required trading services to the corresponding Consumer_User and Merchant_User. These PDUs are the *IOTP messages* (see Chapter 3): *TPO*, *TPO Selection*, *Offer Response*, *Error* and *Cancel*. TPO, TPO Selection and Offer Response are only transferred in an Offer document exchange to provide the services of trading options and trading result during the Offer phase. Both Error and Cancel messages are common to all document exchanges. An Error message is used by a TRole_PE to signal to the peer TRole_PE that an error has been detected when processing an IOTP message received from the peer TRole_PE. The Cancel message is used to abort the transaction at both TRole_PEs, e.g., as a result of a *Cancel.req* submitted by a TRole_User.

We now define the IOTP Transport Service primitives used by TRole_PEs, to be *IOTP-Message* (with request and indication type) and *TransportError* (with only indication type). The *IOTPMessage* primitive is used by a TRole_PE to send (*IOTPMessage.req*) and receive (*IOTPMessage.ind*) IOTP messages. The *TransportError* primitive is only used when the IOTP Transport Service Provider fails to transfer IOTP messages. According to RFC 2935 [77], IOTP will operate over HTTP [79], in which case, the IOTP Transport Service is provided by HTTP (and its underlying protocol layers). In the future, IOTP is also expected to run over other transport mechanisms [36] such as SMTP [192] and TCP [193].

10.2 Scope and Assumptions of the IOTP CPN

This section discusses the scope of IOTP and its components shown in Figure 10.1 to be modelled and lists the assumptions made in the modelling process. The scope of the IOTP

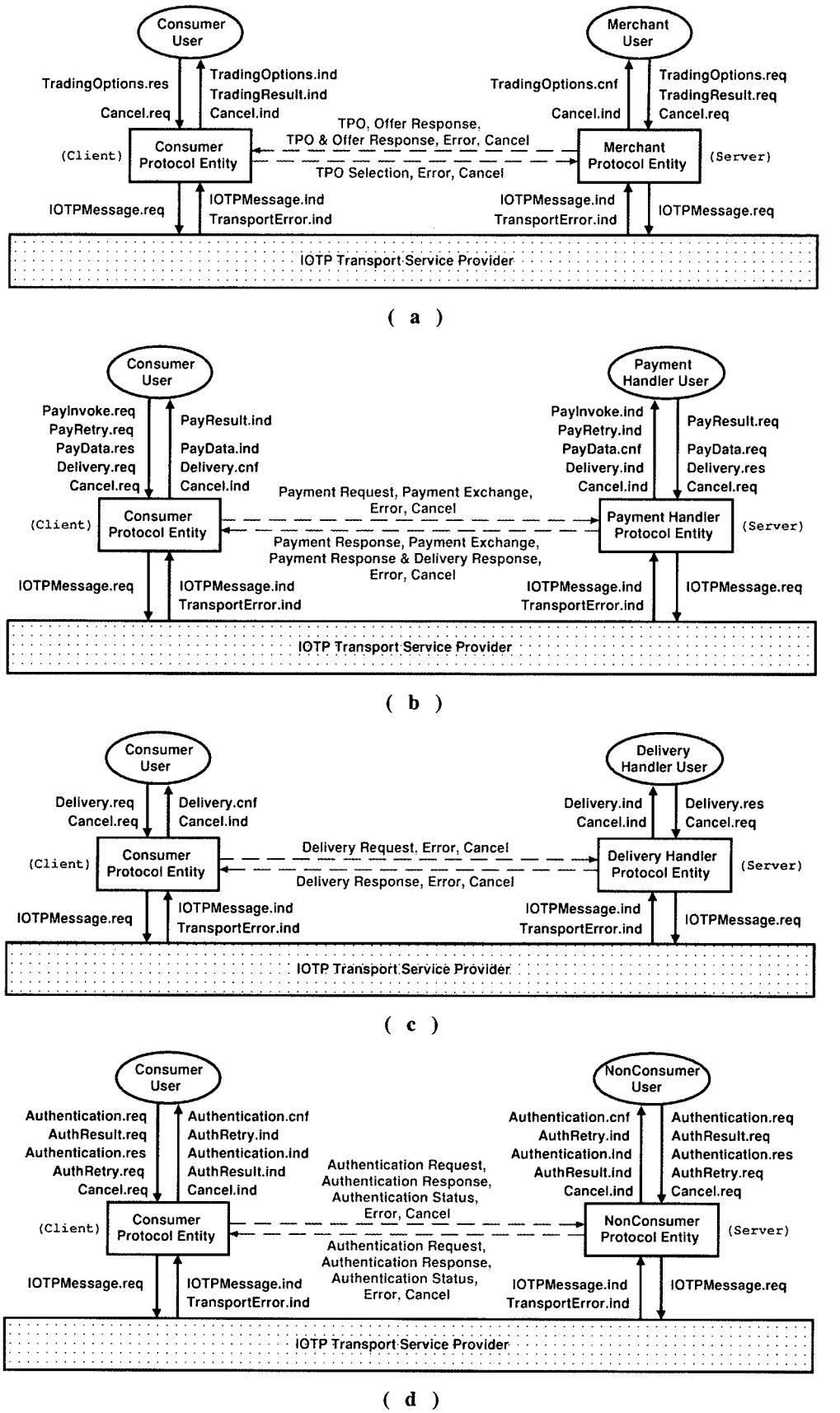


Figure 10.1: Structure of IOTP constituted by four parts: (a) the part between Consumer_PE and Merchant_PE in Offer document exchanges; (b) the part between Consumer_PE and PHandler_PE in Payment and Delivery document exchanges; (c) the part between Consumer and DHandler_PE in Delivery document exchanges; and (d) the part between a Consumer_PE and a non-Consumer_PE in Authentication document exchanges.

CPN comprises a set of limitations that we impose to meet our objectives in the verification of IOTP. These limitations are given in terms of *restrictions* (that may have an impact on the analysis result) and *simplifications* (that would not affect the analysis result) [88]. Also, *assumptions* are necessary when the description of IOTP in RFC 2801 [36] is ambiguous or incomplete.

10.2.1 Transactions and Exchanges

RFC 2801 defines in total eight Baseline IOTP transactions (see Section 3.2.2). The first restriction we introduce limits our investigation to the six Authentication and Payment-related transactions. These are: *Authentication*, *Purchase*, *Deposit*, *Withdrawal*, *Refund* and *Value Exchange*. The two Infrastructure transactions, *Transaction Status Inquiry* and *Ping*, are designed for non-trading purposes and therefore not considered in the IOTP CPN model.

Restriction 10.1 (Baseline IOTP Authentication and Payment-related Transactions). *Only Authentication and Payment-related transactions of IOTP version 1.0 (as described in RFC 2801 [36]) are modelled.* □

The interactions between transactions can be divided into two categories. One category concerns the interrelationship between transactions. For example, one transaction occurs as a result of another transaction. IOTP defines a *Related To Component* (in a *Transaction Reference Block*) to link the interrelated transactions (see Section 3.2.3). Two instantiations of this are specified in the RFC [36]. A Refund transaction may occur as a result of an earlier Purchase transaction. In this case, the Refund occurs *after* the Purchase transaction is complete, and therefore they do not affect each other when either of them is *in progress*. Next, an Authentication transaction may occur at any time when authentication is required within a Payment-related transaction. As discussed in Section 3.2.2, it is simpler to replace the interrelated Authentication transaction by an Authentication document exchange. Using this approach, an Authentication transaction can only occur as a standalone transaction, and therefore no interactions will occur between an Authentication transaction and a Payment-related transaction. These considerations allow us to investigate each of the six Authentication and Payment-related transactions in isolation. Other scenarios of interrelated transactions that are not specified in the RFC are also possible. For example, a Deposit transaction may be needed when insufficient funds are available to make the payment during a Purchase transaction.

The other category is related to interference between transactions. For example, a common problem is the confusion arising between two transactions that have the same transaction identifier due to the wrapping of the transaction identifiers. The definition of IOTP transaction identifiers conforms to the rules for the format of message identifiers of Internet text messages [63]. "The uniqueness of the message identifier" and thereby the IOTP transaction identifier "is guaranteed by the host which generates it"[1] , e.g., an IOTP Merchant server. This further indicates that no interference occurs between two transactions, allowing us to develop the IOTP CPN model for one transaction at a time.

Simplification 10.1 (Transaction Independence). *All six Authentication and Payment-related transactions are independent of each other.* □

[1] See Section 4.6.1 (Message-Id/Resent-Message-Id) on page 22 in [63] for the quoted statement.

Simplification 10.1 means no interactions are considered between transactions. As a consequence, the Related To Component and the IotpTransId attribute of the Transaction Id Component in a Transaction Reference Block can be ignored in the IOTP CPN model.

Given no interactions between transactions, we can simplify our investigation of IOTP by considering only one instantiation of each of the four trading role entities in each transaction, without impacting the analysis results. This allows us to model four TRole_PEs in the IOTP CPN model, each corresponding to one of the four TRole_Users in IOTS.

Simplification 10.2 (Configuration of the TRole_Users). *Transactions occur between one Consumer_User, one Merchant_User, one PHandler_User and one DHandler_User.* □

As we discussed in Section 4.1.1, the concept of document exchanges is sufficient in the construction of IOTP trading transactions and the concept of trading exchanges can be deleted from the RFC [36]. Accordingly, we use the set of document exchanges as the only conceptual building blocks for the modelling of transactions in the IOTP CPN model. More specifically, an Authentication transaction is modelled by a single Authentication document exchange, and each Payment-related transaction is modelled as a combination of at least two document exchanges one of which must be a Payment document exchange.

Simplification 10.3 (Document Exchanges and Trading Exchanges). *All document exchanges but no trading exchanges are considered in the modelling of Authentication and Payment-related transactions.* □

RFC 2801 specifies that an Authentication document exchange can only occur at the beginning of a Payment-related transaction for the Merchant to authenticate the Consumer. The authentication required at any other time during the transaction is implemented by an Authentication transaction, which however can be replaced by an Authentication document exchange according to our discussion in Section 4.1.3. Using an Authentication document exchange instead of an Authentication transaction in a Payment-related transaction allows the Authentication document exchange to be carried out at any time during the Payment-related transaction. To include this in the IOTP CPN model would introduce significant complexity into the model. As previously mentioned, using our incremental approach we limit the scope of investigation to modelling the core behaviour of IOTP. The following restriction is made because each (standalone) Payment-related transaction specified in IOTP represents the core behaviour of the trading transactions in that category.

Restriction 10.2 (Occurrence of Authentication Document Exchange in Payment related Transactions). *The only Authentication document exchange modelled in a Payment-related transaction is the one that occurs between the Consumer_PE and Merchant_PE at the beginning of the transaction.* □

An Authentication transaction may occur between any two trading roles: one acting as Authenticator, the other as Authenticatee. In IOTP version 1.0, communication between non-Consumer_PEs is however not considered. This limits the Baseline Authentication transaction to take place between a Consumer and a non-Consumer trading role. Further, each Authentication transaction follows the same transaction procedure, whichever two trading roles are

involved or whichever of them acts as Authenticator or Authenticatee. It is therefore sufficient to model one of the Authentication transactions, e.g., the one that is carried out by a Merchant to authenticate a Consumer.

Simplification 10.4 (Representative (Standalone) Authentication Transaction). *Modelling (standalone) Authentication transactions can be instantiated as modelling the one where the Merchant acts as the Authenticator and the Consumer acts as the Authenticatee.* □

The definition of a Payment-and-Delivery document exchange in RFC 2801 involves three IOTP messages exchanged between the Consumer_PE and PHandler_PE. These are: a Payment Request Message, a Payment Exchange Message, and a combined Payment Response & Delivery Response Message. Such a Payment-and-Delivery document exchange is however incomplete, due to the lack of a Delivery Request Message which is necessary in generating the Delivery Response. Assumption 10.1 applies a combined Payment Request & Delivery Request Message to replace the Payment(*-only*) Request Message used in the RFC for the Consumer_PE initiating a Payment-and-Delivery document exchange with the PHandler_PE.

Assumption 10.1 (Using the combined Payment Request & Delivery Request Message in Payment-and-Delivery document exchanges). *To start a Payment-and-Delivery document exchange, a combined Payment Request & Delivery Request Message, instead of only the Payment Request Message, needs to be sent from the Consumer_PE to PHandler_PE.* □

With the combined Payment Request & Delivery Request Message, the number of IOTP messages used in the revised Payment-and-Delivery document exchange under Assumption 10.1 remains the same as the one defined in RFC 2801.

10.2.2 Service Primitives

All IOTS primitives, except *PayRetry* and *AuthRetry*, submitted by and delivered to the TRole_Users in Figure 10.1 are modelled. The *PayRetry* and *AuthRetry* primitive events are not modelled, because RFC 2801 does not specify how the protocol deals with a retried authentication or payment (see Section 3.2.7). Since RFC 2801 lacks a service specification for IOTP, it is necessary to relate the occurrence of each primitive event with the protocol operation defined in the RFC.

Assumption 10.2 (Atomic Events). *A primitive submission from a TRole_User results in an IOTP message being sent by the (local) TRole_PE, and a primitive delivery to a TRole_User occurs as a consequence of a message received at the TRole_PE. Both are considered as atomic events at that TRole_PE.* □

Assumption 10.2 allows each IOTS primitive and the corresponding operation of the IOTP to be modelled as one transition in the CPN model.

10.2.3 Security Considerations

The security issues addressed in IOTP include the use of digital signatures and certificates, symmetric and asymmetric cryptography, data privacy and payment protocol security (see Sections 5 and 6 of [36]). These are however not considered in the IOTP CPN model, because we have restricted our scope to the core functional behaviour of IOTP (see Section 3.3).

Simplification 10.5 (Omission of Digital Signatures and Certificates). *Digital signatures and certificates in Authentication and Payment-related transactions are not modelled.*

A consequence of Simplification 10.5 is that the Signature Block in an IOTP message (described in Section 3.2.3) is not modelled.

10.2.4 Transaction Cancellation

All possible transaction cancellation scenarios (discussed in Section 3.2.6) are modelled in the IOTP CPN model. The *Cancel Message* used to cancel an ongoing transaction has a *Status Block*. The Status Block uses a *Status Component* to report the reason for cancelling the transaction. Information about the cancel reason however does not affect the behaviour of transaction cancellation, and therefore can be ignored in the IOTP CPN model.

Simplification 10.6 (Omission of Cancel Reasons). *The reason for cancelling an ongoing transaction is not modelled.* □

As a consequence of Simplification 10.6, the Status Component is omitted from the Status Block that is used in a Cancel Message when modelling transaction cancellation.

10.2.5 Error Handling

As discussed in Section 3.2.7, the IOTP error handling specified in RFC 2801 is mainly concerned with technical errors. Although business errors are considered, definitions of how to handle and recover from business errors are not given in the RFC. Therefore, only technical errors and their error handling procedures are specified in the IOTP CPN model. As a consequence, the following message information or primitive events are not modelled: 1) the *ProcessState* and *CompletionCode* attributes of the *Status Component* used to indicate and encode business errors in a response block or a *Cancel Block*; and 2) the *PayRetry* and *AuthRetry* primitives defined for business error recovery in IOTS.

Information about a technical error is reported in the *Severity* and the *ErrorCode* attributes of an *Error Component* that is used to form an *Error Block*. The *Severity* attribute indicates a *Warning Error*, a *Transient Error* or a *Hard Error*. An Error Block reporting a Warning Error is always contained within a non-Error Message. Any trading role that receives a non-Error Message reporting a Warning Error, can ignore the error information and process the message as usual. An Error Message reporting a Transient Error results in resetting the timer for message retransmission. An Error Message reporting a Hard Error results in the failure of the transaction, which must be reported to the user (via a *Cancel.ind* primitive).

From the above, the occurrence of a Warning Error does not affect the operation of the protocol, and therefore can be ignored in modelling the error handling procedures.

Simplification 10.7 (Omission of Technical Errors with Severity of Warning). *All possible technical errors with their Severity attribute value set to Warning are not modelled.* □

The *ErrorCode* attribute of the *Error Component* has 22 error code values, representing 18 technical errors at the three levels of error depth (one at the transport level, the others at the message level as well as the block/component level) and 4 other errors. Modelling the transport level errors is discussed in Section 10.2.7.

Each error at the message or the block/component level has a Severity of either Warning Error or Hard Error. The IOTP CPN model is currently developed at such an abstraction level that the operation of the protocol may only be affected by the severity of the error. This means that the details of an error at the message or the block/component level has no impact on the core behaviour of transactions. The following simplification is made so that the 17 error code values representing the message level and the block/component level errors can be replaced by one abstract value *MsgErr* in the IOTP CPN model.

Simplification 10.8 (Abstraction of Message and Block/Component Level Technical Errors). *All possible technical errors occurring at the message level and the block/component level are modelled as one category of errors. The error code for any error within this category is assigned with the same abstract value named MsgErr.* □

Another two error code values, *MsgBeingProc* and *SystemBusy*, are used with a Severity of Transient Error. SystemBusy indicates an IOTP server (operating a non-Consumer_PE) is currently not available and therefore cannot handle the message received. However, this is not modelled in the IOTP CPN model where only a single transaction is considered at a time (Simplification 10.1) and all TRole_PEs are available for that transaction. MsgBeingProc indicates the previous message received by the peer TRole_PE is still being processed. The TRole_PE that receives an Error Message reporting MsgBeingProc, should resent the previous message if no response is received by the time indicated by the *MinRetrySecs* attribute of the Error Component[2] . The last two error code values, *Reserved* (for specific use by software vendor/developer) and *UnknownError* (for any unspecified problem), are unnecessary in verifying the core behaviour of IOTP and therefore are not modelled.

A TRole_PE re-sends the previous message upon a time-out due to the retransmission timer expiring. The timer is reset either upon message retransmission or upon receipt of an Error Message that reports a Transient Error. RFC 2801 specifies that re-sending a message can only occur up to a maximum number of times and the TRole_PE should terminate the transaction upon another time-out. The maximum number allowed for message retransmission is however not given in the RFC. The following assumption is made, because a message retransmission counter is needed by the TRole_PE to record the number of times a message has been retransmitted, and a limit of retransmissions needs to be included.

Assumption 10.3 (Message retransmission Counter). *Each TRole_PE uses a message retransmission counter to record the number of times a message has been retransmitted. The counter is reset to zero upon sending a new message, and is incremented by one upon retransmitting a message. It has a maximum value indicated by RCmax.* □

10.2.6 IOTP Messages

According to Restriction 10.1, the IOTP messages that are used in the six Authentication and Payment related transactions must be modelled. This complete set of messages can be seen in Figure 10.1. The other IOTP messages related to the two Infrastructure transactions are not modelled. These are: *Inquiry Request*, *Inquiry Response*, *Ping Request* and *Ping Response*.

[2]See Section 7.21.2 on pages 160 to 161 in RFC 2801 [36].

Trading Blocks

Each IOTP message comprises a number of trading blocks. Except for the *Inquiry Request*, *Inquiry Response*, *Ping Request*, *Ping Response*, and *Signature* blocks (Simplification 10.5), all other trading blocks that are listed in the XML definition of an IOTP message (see Figure 3.3) are needed for constructing the IOTP messages in the CPN model.

Trading Components

Each trading block consists of several trading components. Most of the trading components are not modelled, due to one of the above restrictions or simplifications, or because the information conveyed by those trading components have no impact on the core behaviour of IOTP. For example, the *Signature* and the *Certificate* components used to construct a *Signature* block are not modelled according to Simplification 10.5. In another example, the *Organisation* component is not modelled because the protocol operation is independent of the trading role organisation information conveyed by this component. Only three trading components, Transaction Id, Message Id and Error, are used in the IOTP CPN model.

Attributes

Each trading component has a set of attributes used to convey concrete information. However, we are only interested in those that may have impact on the core behaviour of IOTP. The following simplification lists the six attributes of the above five trading components that may affect the protocol operation and are therefore modelled in the IOTP CPN.

Simplification 10.9 (Selected Attributes of Trading Components). *For all trading components, only three of them with four attributes are modelled. These are: 1) the IotpTransType attribute of the Transaction Id component, 2) the ID attribute of the Message Id component, and 3) the Severity and ErrorCode attributes of the Error component.* □

Message Identifier

Each IOTP message has a message identifier which uniquely identifies that message. Any duplicate messages must have the same message identifier. A message identifier is conveyed by the ID attribute of the Message Id Component in the Transaction Reference Block in an IOTP message (see Section 3.2.3). It is necessary when retransmissions or duplicates occur within the error handling procedure and therefore needs to be taken into account when each IOTP message is specified in the CPN model.

10.2.7 Transport Service Provider

The TRole_PEs communicate with each other using the IOTP Transport Service. The characteristics of the Transport Service Provider must be known so that an abstract model of it can be created. IOTP version 1.0 currently uses HTTP as its transport mechanism [77]. With future extension, IOTP is also expected to operate over SMTP, TCP, and others. HTTP, SMTP and TCP all provide reliable delivery of IOTP messages with no losses, duplicates or re-ordering. However, there is delay in message delivery due to transmission over the Internet.

Assumption 10.4 (IOTP Transport Service Provider Characteristics). *The IOTP Transport Service Provider provides reliable delivery of IOTP messages with no loss, duplication or re-ordering, but message delivery delay is possible. The capacity of the IOTP Transport Service Provider is unbounded.* □

RFC 2801 defines an error code value *TransportError* to indicate a transport level error "preventing the (IOTP) message from being received" (page 161, [36]). It also specifies that "the only reasonable automatic action when faced with transport level errors is to retry" (page 58, [36]). Thus, the only effect of transport level errors on the protocol operation is the retransmission of IOTP messages. This allows us to ignore any details of the transport level errors but consider it as a trigger event for message retransmission in the IOTP CPN model.

10.3 Hierarchy Page

Figure 10.2 shows the hierarchy page for the IOTP CPN model. It consists of 32 pages which are divided into four hierarchical levels. This four-level hierarchy of the IOTP CPN model provides a logical structure that can be validated against RFC 2801 [36]. It [178] is developed by refining the structure of our previous specifications [179–181] to better reflect IOTP's definition and to improve modularity and re-use. The specification is also completed by including IOTP's procedures for error handing and arbitrary transaction cancellation.

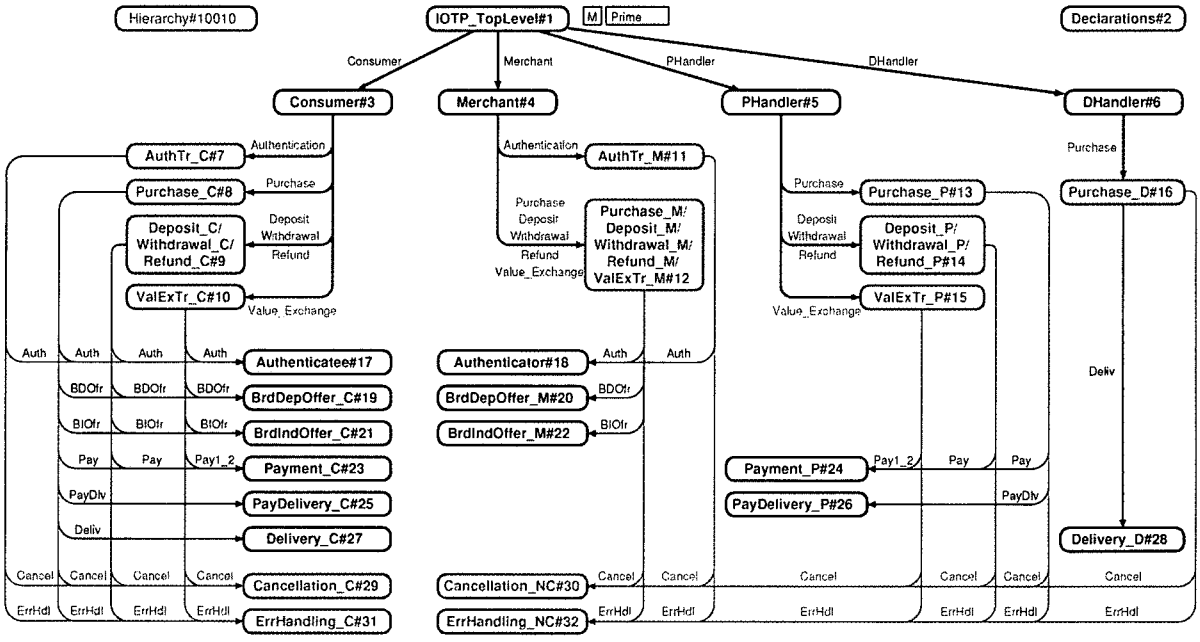


Figure 10.2: Hierarchy page for IOTP CPN.

The first (top) level presents an abstract view of IOTP on one page, namely IOTP_TopLevel. The second level comprises four pages: Consumer, Merchant, PHandler and DHandler, corresponding to four TRole_PEs. We refer to these pages as *trading role pages*.

Each trading role page has a set of subpages specifying the possible Authentication and Payment-related transactions for that TRole_PE. All these subpages, which we call *transaction pages*, constitute the third level of the IOTP CPN model. For example, the page Consumer has four subpages modelling six transactions at the Consumer_PE. The three transactions

Deposit, Withdrawal and Refund use the same procedure, therefore they are modelled on one page named Deposit_C/Withdrawal_C/Refund_C. The initial letter of each trading role name is used as a suffix of the name of transaction pages modelled for that trading role. For example, the subpages AuthTr_C and AuthTr_M represent an Authentication transaction between the Consumer and Merchant. It can be seen that both Consumer and Merchant are involved in all six transactions, the Payment Handler is engaged in all transactions except Authentication and the Delivery Handler is only involved in Purchase transactions.

Each transaction page is further decomposed into a set of subpages modelling the document exchanges that are used to construct that transaction as well as the error handling and cancellation procedures. All these subpages, which we call *exchange level pages*, constitute the fourth level of the IOTP CPN model. For example, the page Purchase_C has six subpages modelling the six document exchanges used to implement a Purchase transaction at the Consumer (e.g., the Authenticatee page for an Authentication document exchange where the Consumer acts as the Authenticatee) and two others modelling how the Consumer_PE handles technical errors and cancels an ongoing transaction (i.e., the Cancellation_C and ErrHandling_C pages). Since a document exchange always involves two trading roles, we have modelled each exchange as a pair of pages, which are respectively suffixed by the initial letter of the corresponding trading role name. For example, the pages Delivery_C and Delivery_D represent a Delivery document exchange between the Consumer and Delivery Handler. In addition, the procedures of error handling or transaction cancellation are different between Consumer and non-Consumer entities. We have also modelled each of these two procedures using only a pair of pages - one (whose name ends with the letter C) for the Consumer entity (realised by IOTP client), the other (whose name ends with the two letters NC) for any non-Consumer entity (realised by IOTP server). Each pair of ErrHandling or Cancellation pages are used by all transaction pages.

In the following sections, we describe the IOTP CPN pages on each level of the hierarchy. The Declarations page defines all types, constants, variables and functions used in the CPN model, and is introduced incrementally as we progress through the descriptions of the CPN pages. The complete set of declarations are given in Appendix E.

10.4 Overview Page

Figure 10.3 depicts the IOTP_TopLevel page that provides the most abstract view of IOTP. The four TRole_PEs are modelled by four substitution transitions Consumer, Merchant, PHandler and DHandler. They communicate with each other by sending and receiving IOTP messages via the IOTP Transport Service Provider, which is modelled by the place Transport typed by the colour set TRxTRxMQ. The details of the TRxTRxMQ colour set will be introduced in Section 10.5. For now it is sufficient to know that it defines the different IOTP messages transferred between two TRole_PEs.

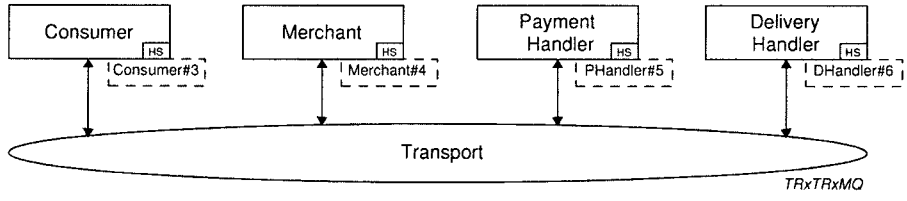


Figure 10.3: IOTP_TopLevel page in the IOTP CPN model.

The IOTP_TopLevel page does not show the TRole_Users, as compared to the structure of IOTP in Section 10.1. The reason is that we choose to model the TRole_Users implicitly via the occurrence of transitions representing the submission and delivery of IOTS primitives. These transitions are in the exchange level pages to be described in Section 10.8. On the IOTP_TopLevel page, each substitution transition is decomposed into a subpage that provides further details on the protocol operation at the corresponding TRole_PE. Section 10.6 describes the subpages.

10.5 IOTP Messages and Communication Queue

Table 10.1 defines the colour sets and variables used to model IOTP messages and the communication queue between TRole_PEs. For clarity, we divide them into four groups.

```
    (* Selected Attributes of Trading Components *)
1   color IotpTransType = with Authentication | Purchase | Deposit | Withdrawal | Refund |
2                         ValueExchange;
3   color Severity = with TransientError | HardError;
4   color ErrorCode = with MsgErr | MsgBeingProc;
5   color TwoAttrOfErrComp = product Severity * ErrorCode;
6   var trtype: IotpTransType;

    (* Trading Blocks and IOTP Messages *)
7   color TradingBlk = union TransRefBlk:IotpTransType +
8                            AuthReqBlk + AuthRespBlk + AuthStatusBlk +
9                            TpoBlk + TpoSelectionBlk + OfferRespBlk +
10                           PayReqBlk + PayExchBlk + PayRespBlk +
11                           DeliveryReqBlk + DeliveryRespBlk +
12                           ErrorBlk:TwoAttrOfErrComp +
13                           CancelBlk;
14  color IotpMsg = list TradingBlk;
15  var m, rm, sm: IotpMsg;

    (* IOTP Message with Message Identifier *)
16  color MsgId = int;
17  color Message = product IotpMsg * MsgId;
18  var id, rid, sid: MsgId;

    (* Trading Role Communication Queue *)
19  color MsgQueue = list Message;
20  color TradingRole = with Consumer | Merchant | PHandler | DHandler;
21  color TRxTRxMQ = product TradingRole * TradingRole * MsgQueue;
22  var role: TradingRole;
23  var q, rq, sq: MsgQueue;
```

Table 10.1: Colour sets for modelling IOTP messages and trading role communication queue.

The first group (lines 1 to 5) concerns three selected attributes of the corresponding trading components that are given in Simplification 10.9 (except the ID attribute of a Message Id Component to be specified in the third group). Each attribute is modelled by a colour set with the same name. The colour set IotpTransType has six values corresponding to the six Authentication and Payment-related transactions (Restriction 10.1). The colour set Severity has two values: TransientError and HardError (Simplification 10.7). The colour set ErrorCode contains two values: MsgErr used to abstractly represent a technical error at the message level

or the block/component level (Simplification 10.8), and MsgBeingProc used to indicate the previous message received is still being processed. Finally, the colour set TwoAttrOfErrComp is defined as a *product* type, modelling that the two attributes *Severity* and *ErrorCode* need to be carried in one Error Component.

The second group (lines 7 to 14) models trading blocks and IOTP messages, based on their XML definitions given in RFC 2801. The colour set TradingBlk is defined as the *union* of *14* different types of trading blocks, where the name of each trading block is given in its abbreviated form as specified in the XML definition. For example, TransRefBlk stands for a Transaction Reference Block. Among these *14* trading blocks, two of them contain the selected attributes (that are modelled in the first group) of the corresponding trading components. These are: TransRefBlk with IotpTransType attribute of the Transaction Id Component, and ErrorBlk with both Severity and ErrorCode attributes of the Error Component. It can be observed that all trading components are omitted and only their selected attributes are used in modelling trading blocks. Hence, the representation of trading blocks in the IOTP CPN model is a simplified and abstract representation of those specified in RFC 2801. Next, the colour set IotpMsg is defined as a *list* of TradingBlks denoting each message consists of one or more trading blocks. This conforms to the XML definition of an IOTP message (see Figure 3.3) given in the RFC.

The third group (line 16 to 17) specifies the IOTP message with its message identifier. As described in Section 3.2.3, a message identifier consists of a letter and one or more digits. The colour set MsgId is defined as *integers* and models the digits in the message identifier. The letter representing the trading role that sends the message, is modelled by the TradingRole colour set in the fourth group (see below). Next, the colour set Message is defined as a *product* type, modelling that each IotpMsg must have a message identifier represented by MsgId. Also in Section 3.2.3, a message identifier is conveyed by the ID attribute of the Message Id Component in a Transaction Reference Block. However, we model the message identifier as being separate from the IOTP message structure, so that we can omit the Transaction Reference Block from an IOTP message conveying only the message identifier needed. With such a modelling mechanism, we obtain a simplified and abstract representation of IOTP messages.

The last group (line 19 to 21) defines the communication queue between TRole_PEs. With a reliable transport service considered in Assumption 10.4, the colour set MsgQueue is defined as a *list* of Messages, modelling that the IOTP messages are transferred (via transport media) in FIFO order. Also, no bound is imposed on the length of a queue of IOTP messages. The colour set TradingRole models the TRole_PEs and contains four values corresponding to each of the four different TRole_PEs (Simplification 10.2). The final colour set, TRxTRxMQ, models the MsgQueue between two (distributed) TradingRoles. The order of the two TradingRoles in the TRxTRxMQ colour set specifies the direction in which messages are transferred. The first TradingRole represents the TRole_PE sending the message and the second one represents the peer TRole_PE receiving the message. With this approach, we can omit from the MsgId colour set the representative letter used to indicate the sender trading role.

Finally, a number of variables (line 6, 15, 18, 22 and 23) are declared for use in arc inscriptions and guards in the IOTP CPN model.

10.6 Trading Role Pages

The second level of the IOTP CPN model consists of four trading role pages, modelling the set of Authentication and Payment-related transactions that can be carried out at each of the four TRole_PEs. The Consumer page is shown in Figure 10.4, and the Merchant, PHandler and DHandler pages are shown in Figures 10.5 to 10.7. Since the four trading role pages have a similar structure, we illustrate our approach with the Consumer page. The three non-Consumer trading role pages are briefly described afterwards.

10.6.1 Consumer Trading Role Page

The IOTP transactions are consumer-oriented, and hence the Consumer is the most important trading role in IOTP. The Consumer page in Figure 10.4 provides an overview of the six Authentication and Payment-related transactions that can occur at the Consumer_PE. Each transaction is abstractly represented by a substitution transition, which has the same name as that transaction and is decomposed into a subpage that models the set of document exchanges used to implement the transaction at the Consumer_PE. Section 10.7 describes these subpages.

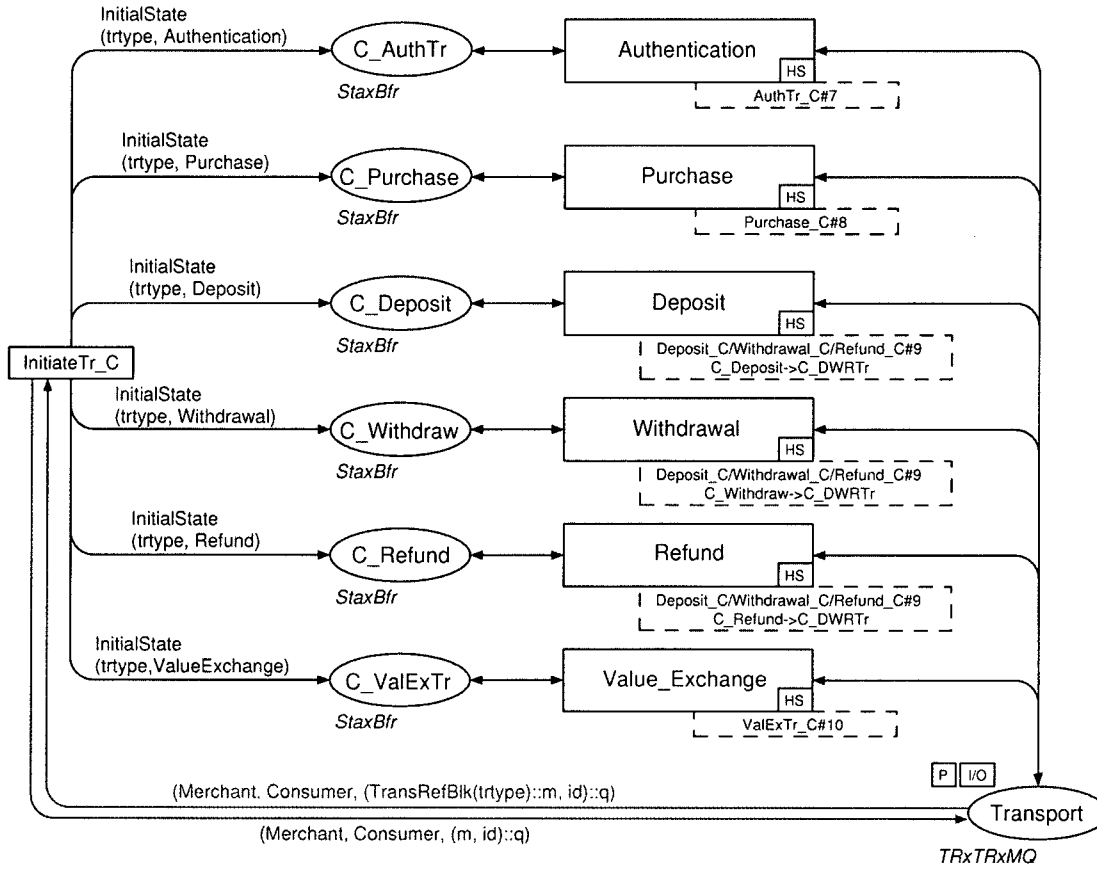


Figure 10.4: Consumer page in the IOTP CPN model.

States of Trading Role Protocol Entities

The Consumer page introduces six new places. Each of them has a name starting with the initial letter of Consumer and is hereafter referred to as a *C_place*. All the six C_places are typed by the colour set StaxBfr, modelling the state of the Consumer_PE with its message

retransmission/combining buffer in each of the six different transactions. Table 10.2 defines the colour sets and variables used to model the (compound) state of the TRole_PEs.

```
1  color GeneralState = with READY | COMPLETED | CANCELLED |
2                               LISTEN | WAIT | HOLD | HOLD_WAIT | FINISHED;
3  color Exchange = with Auth | BDOfr | BIOfr | Pay1 | Pay2 | PayDlv | Deliv | NoExch;
4  color R_Counter = int;      (* Message retransmission Counter *)
5  color MIdxRC = product MsgId * R_Counter;
6  color State = product GeneralState * IotpTransType * Exchange * MIdxRC * MsgId;
7  color StaxBfr = product State * IotpMsg;
8  var s: GeneralState;   var exch: Exchange;   var rc: R_Counter;
```

Table 10.2: Colour sets for modelling states of TRole_PEs.

The colour set GeneralState (line 1) specifies the following 8 general states of a TRole_PE.

- READY, COMPLETED and CANCELLED represent the initial state, the successful terminal state and the aborted terminal state, respectively, of a TRole_PE in a transaction.
- LISTEN is used when a TRole_PE is waiting for an IOTP message from the peer TRole_PE.
- WAIT is used when a TRole_PE is expecting any required transaction information from the (local) TRole_User (via the submission of the corresponding IOTS primitive).
- HOLD is used to indicate a TRole_PE is currently processing a *combined* IOTP message (e.g., a combined *Payment Response & Delivery Response Message*), in which case, any subsequent message received by the TRole_PE will not be processed until processing of the previous combined message has completed.
- HOLD_WAIT is a special WAIT state that must be used when a TRole_PE needs to send a combined IOTP message to the peer TRole_PE. It indicates the TRole_PE has already received part of the transaction information from the local TRole_User, and keeps waiting for the rest before the combined IOTP message can be sent.
- FINISHED represents a TRole_PE has completed a document exchange and is ready to move on to the next document exchange in order to continue with the transaction.

The colour set Exchange (line 3) defines six document exchanges that may be carried out at a TRole_PE during an Authentication or Payment-related transaction. All these document exchanges are given by their short names: Auth for Authentication exchange, BDOfr and BIOfr for Brand Dependent Offer or Brand Independent Offer exchange, Pay1 and Pay2 for 1st or 2nd Payment exchange, PayDeliv for Payment-and-Delivery exchange, and Deliv for Delivery exchange. Also, another value called NoExch indicates no document exchange is conducted at the TRole_PE before a transaction is initiated.

The colour set R_Counter (line 4) models the message retransmission counter (Assumption 10.3) and is defined as the set of *integers*. The colour set MIdxR (line 5) represents the MsgId of an IOTP message with its R_Counter. The value of the R_Counter is incremented by one upon each message retransmission.

Next, the colour set State (line 6) models the (compound) state of the TRole_PEs. Each state of a TRole_PE consists of a general state of the TRole_PE (GeneralState), the transaction (IotpTransType) and the document exchange (Exchange) being conducted at the TRole_PE, the

message identifier of the last IOTP message sent to the peer TRole_PE with its retransmission counter (MIdxRC), and the message identifier of the last IOTP message received from the peer TRole_PE (MsgId). Both IotpTransType and Exchange are necessary in specifying the state of a TRole_PE, because the set of GeneralStates are common to all types of transactions and document exchanges. Recall from Section 3.2.7 that IOTP error handling involves message retransmission as well as handling duplicates. Hence, by taking into account the error handling procedure, the MIdxR of the previously sent message and the MsgId of the previously received message also need to be considered as part of the state information of the TRole_PE.

Finally, the colour set StaxBfr (line 7) represents the State of a TRole_PE with the IotpMsg residing in the message retransmission/combining buffer at the TRole_PE.

Transaction Initiation

The page Consumer has only one ordinary transition named InitiateTr_C. It is used to initiate an appropriate transaction at the Consumer_PE, and has an input arc and an output arc to the Transport place. The input arc has an inscription specifying the first IOTP message (represented by (TransRefBlk(trtype)::m,id)) from the Merchant_PE. The output arc indicates processing the Transaction Reference Block (TransRefBlk(trtype)) is complete, and the block is removed from the input message buffer once the transaction is initiated at the Consumer_PE. Also, there are six output arcs of transition InitiateTr_C to each of the six trading role places. Each arc is inscribed by a function called InitialState, modelling how the initial state of the Consumer_PE is determined upon a certain transaction type. Table 10.3 defines this function.

```
1  fun InitialState(trtype, trtypevalue:IotpTransType): StaxBfr
2    = if trtype=trtypevalue
3      then 1'((READY, trtype, NoExch, (0, 0), 0), [ ])
4      else empty
```

Table 10.3: Function used to determine the initial state of a TRole_PE.

Function InitialState has two parameters: trtype and trtypevalue, both typed by the colour set IotpTransType. If the two parameters have the same value, the function returns ((READY, trtype, NoExch,(0,0),0),[ ]) (of StaxBfr), which represents the initial state of a TRole_PE. Otherwise, the function does not return any token value, defined as empty. The initial state of a TRole_PE specifies that the TRole_PE is READY to carry out a transaction of trtype but does not know which document exchange to start before processing the first IOTP message from the Merchant_PE.

When using InitialState in the Consumer page, the transaction type values are given, in different ways, to the two parameters in the function. The trtype has the transaction type value provided in the first IOTP message from the Merchant, whereas the trtypevalue is set to a certain transaction type in each function application. Upon the occurrence of transition InitiateTr_C, only one of the six function applications returns an initial state value, which is then put into the corresponding C_place. As a result, only the substitution transition representing the transaction with the type indicated by trtype can be executed. With this approach, the six Authentication and Payment-related transactions are modelled independently of each other.

For modelling transaction initiation on page Consumer, it can be observed that the receipt of the first IOTP message from the Merchant_PE is decomposed into two events in the CPN

model. The first event is the Consumer processing the TransRefBlk(trtype) in the message to initiate the transaction of trtype. We consider this a transaction level operation and therefore model it on trading role pages. The second event concerns processing the rest of the information in the message (e.g., an Authentication Request Block) which are operations for document exchanges, and is therefore specified in the corresponding exchange level page (e.g, the Authenticatee page).

10.6.2 Merchant Trading Role Page

Figure 10.5 depicts the page Merchant which is structured in a similar way to the Consumer page with six M_places and six substitution transitions abstractly representing the six independent transactions at the Merchant_PE. A major difference on page Merchant is the modelling of how transactions are initiated by the Merchant_PE. Since the Merchant_PE is the one that generates and sends the first IOTP message in a transaction, the place Initialize is used to provide the transaction type information for initiating the corresponding transaction at the Merchant_PE. It is typed by the colour set IotpTransType, and has an initial marking of a constant Transaction (see Appendix E) which conveys one of the six transaction type values. With place Initialize, the CPN model can be executed for different transactions by changing the value of the constant Transaction in the global declaration node only.

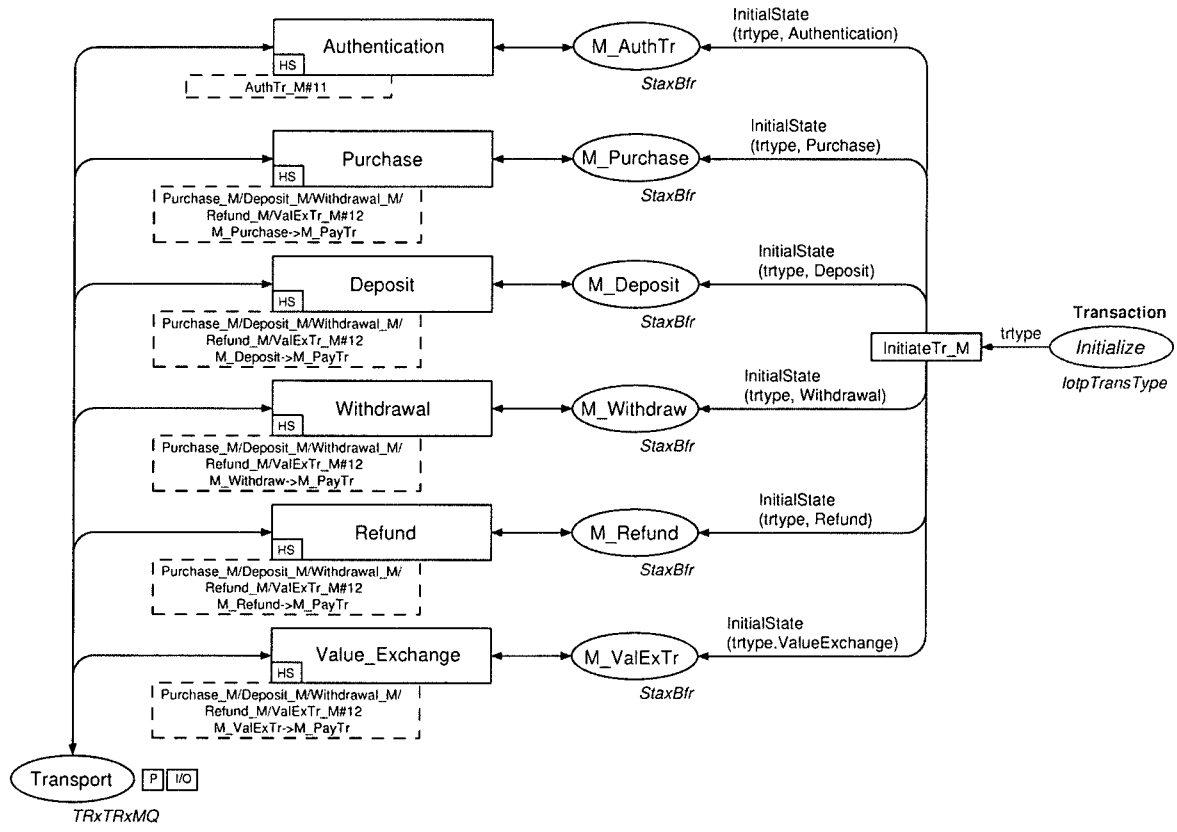


Figure 10.5: Merchant page in the IOTP CPN model.

10.6.3 Payment Handler Trading Role Page

Figure 10.6 depicts the page PHandler which contains five P_places and five substitution transitions for each of the five Payment-related transactions at the PHandler_PE. With Simplification 10.4, modelling Authentication transactions in the IOTP CPN model is instantiated as modelling the one between the Merchant and Consumer, and therefore is not considered on the other trading role pages. Similarly to the Consumer page, the transition InitiateTr_P is used to initiate the appropriate transaction at the PHandler_PE, on receipt of the first IOTP message from the Consumer_PE at the beginning of a Payment or a Payment-and-Delivery document exchange.

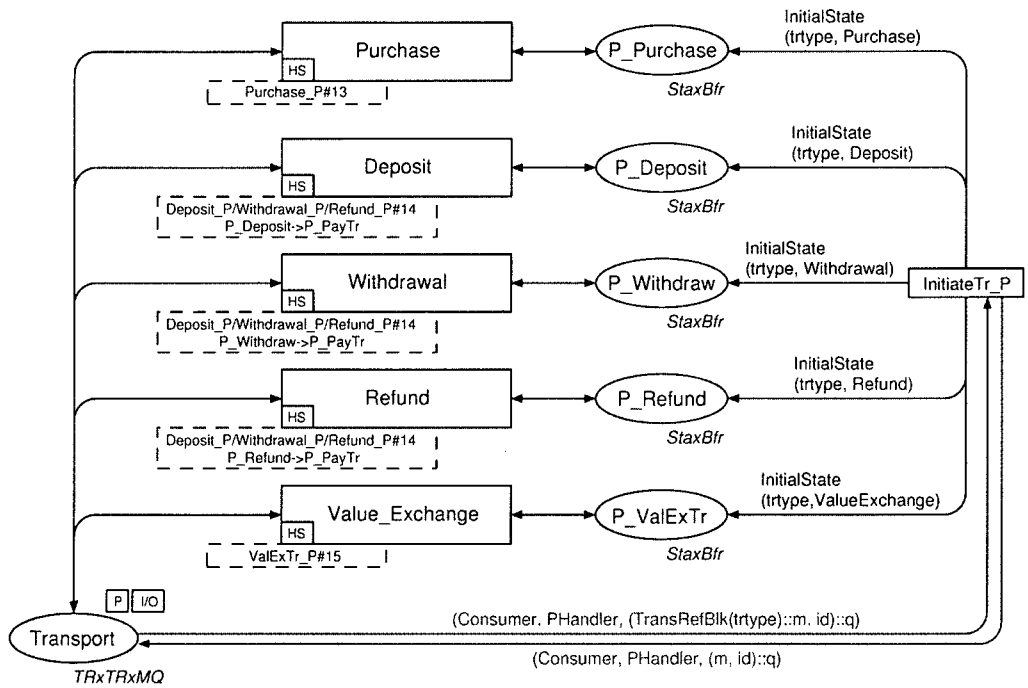


Figure 10.6: PHandler page in the IOTP CPN model.

10.6.4 Delivery Handler Trading Role Page

The Delivery Handler is only involved in Purchase transactions. Accordingly, the DHandler page shown in Figure 10.7 has just one D_place and one substitution transition modelling a Purchase transaction at the DHandler_PE. Similarly, occurrence of the transition InitiateTr_DE indicates that the DHandler_PE initiates a Purchase transaction upon receiving the first IOTP message from the Consumer_PE at the beginning of a Delivery document exchange.

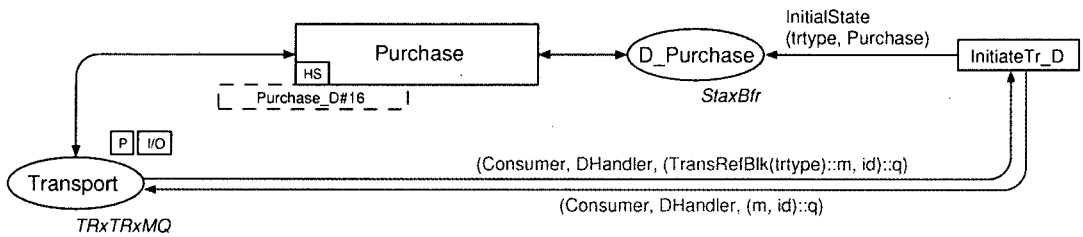


Figure 10.7: DHandler page in the IOTP CPN model.

10.7 Transaction Pages

The Authentication and Payment-related transactions are implemented via combinations of document exchanges at each TRole_PE. The transaction pages, which constitute the third level of the IOTP CPN model, specify valid combination of the set of document exchanges to construct a transaction at each TRole_PE. All these pages have a similar structure. Hence, we illustrate the modelling of transaction level pages using Authentication and Purchase transactions as two representative examples. The differences to the other transaction pages will be briefly mentioned. Note that both error handling and transaction cancellation procedures are common to all six transactions and therefore appear as substitution transitions on each transaction page.

10.7.1 Authentication Transaction

The Authentication is the simplest transaction involving just two TRole_PEs and using only the Authentication document exchange. Figures 10.8 and 10.9 depict the AuthTr_C and AuthTr_M pages, modelling an Authentication transaction at the Consumer_PE and Merchant_PE, respectively. Each page consists of a C_AuthTr or M_AuthTr, a Transport place and three substitution transitions. The C_AuthTr (or M_AuthTr) place is linked to the place that has the same name on page Consumer (or Merchant). The three substitution transitions abstractly represent the Authentication document exchange (Auth), the error handling procedure (ErrHdl) and the transaction cancellation procedure (Cancel) at the Consumer_PE or Merchant_PE.

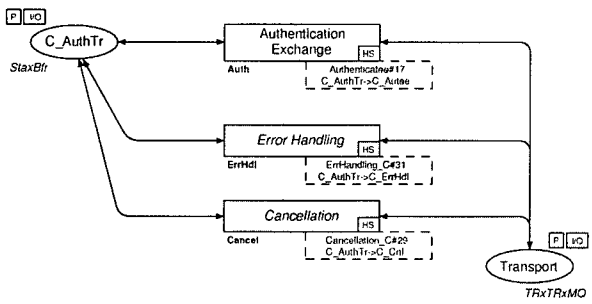


Figure 10.8: AuthTr_C page.

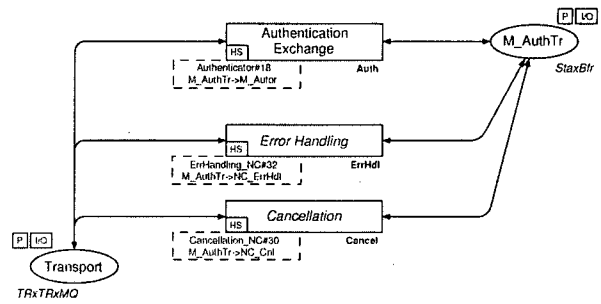


Figure 10.9: AuthTr_M page.

10.7.2 Purchase Transaction

The Purchase is the most complex transaction involving four TRole_PEs and using six document exchanges. It consists of an optional Authentication document exchange (Auth), followed by an Offer document exchange (BDOfr or BIOfr), and then a Payment document exchange (Pay) only, or a Payment followed by a Delivery document exchange (Deliv), or a Payment-and-Delivery document exchange (PayDlv). Figures 10.10 to 10.13 depict the four transaction pages used to model a Purchase transaction at each of the four TRole_PEs.

The Consumer is the only trading role involved in the whole transaction procedure, and thus all the six document exchanges are specified by the six substitution transitions on page Purchase_C shown in Figure 10.10. The Payment Handler is involved in a Payment or a Payment-and-Delivery document exchange, as represented by the Pay and PayDlv transitions on page Purchase_P in Figure 10.11. The Delivery Handler is involved in a Delivery document exchange only, as represented by the Deliv transition on page Purchase_D in Figure 10.12.

Finally, the Merchant may be involved in an Authentication document exchange and either of the two Offer document exchanges in a Purchase transaction as well as the other four Payment-related transactions. These are modelled by the Auth, BDOfr and BIOfr transitions on page Purchase_M/Deposit_M/Withdrawal_M/Refund_M/ValExTr_M in Figure 10.13, which is used for modelling all the five Payment-related transactions at the Merchant_PE.

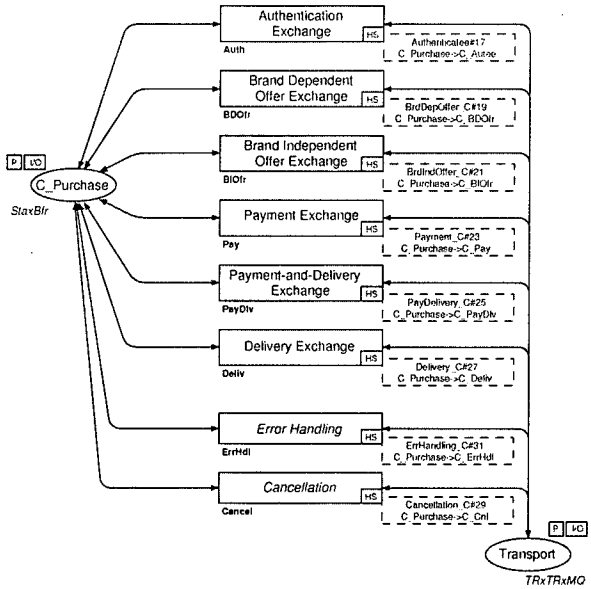


Figure 10.10: Purchase_C page.

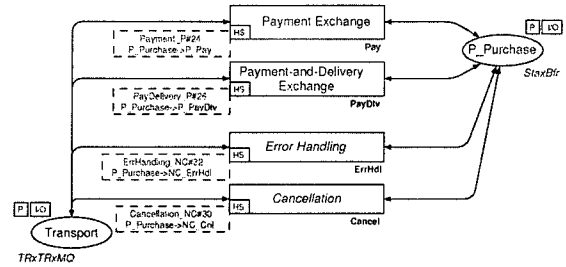


Figure 10.11: Purchase_P page.

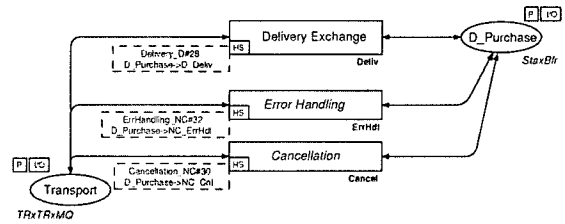


Figure 10.12: Purchase_D page.

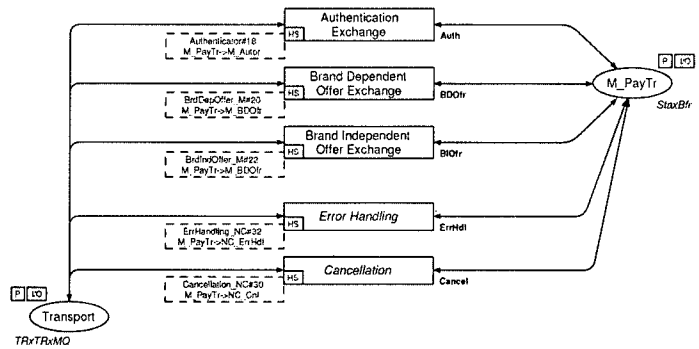


Figure 10.13: Purchase_M/Deposit_M/Withdrawal_M/Refund_M/ValExTr_M page.

10.7.3 Other Payment-related Transactions

The other transaction pages are modelled in a similar way. A Deposit, Withdrawal or Refund transaction involves three TRole_PEs (Consumer_PE, Merchant_PE and PHandler_PE) and uses the same four document exchanges (Auth, BDOfr, BIOfr and Pay). Hence, the three transactions are modelled by the same transaction page at each TRole_PE. The two transaction level pages modelling a Deposit, Withdrawal or Refund transaction at the Consumer_PE and PHandler_PE are shown in Figures 10.14 and 10.15, respectively. The other transaction page used at the Merchant_PE is shown in Figure 10.13.

A Value Exchange transaction involves three TRole_PEs (as above) and uses five document exchanges (as above but with two Payment document exchanges (Pay1 and Pay2)). The transaction is modelled by the three transaction pages, and two of them used for the Consumer_PE and PHandler_PE are shown in Figures 10.16 and 10.17, respectively.

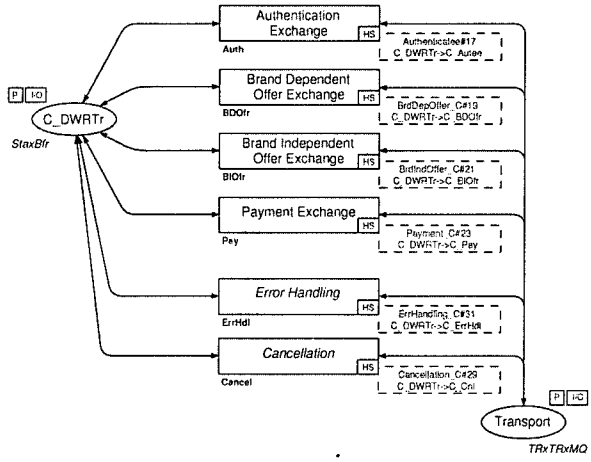


Figure 10.14: Deposit_C/Withdrawal_C/Refund_C page.

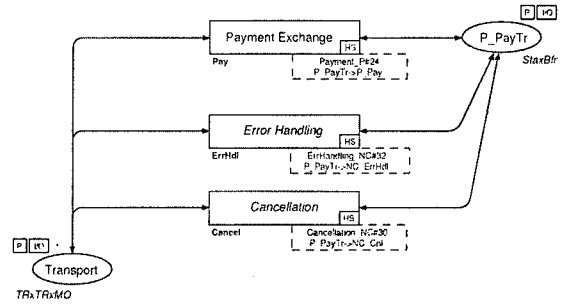


Figure 10.15: Deposit_P/Withdrawal_P/Refund_P page.

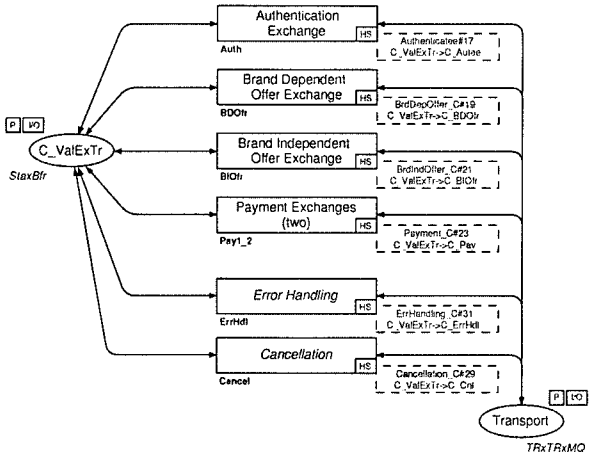


Figure 10.16: ValExTr_C page.

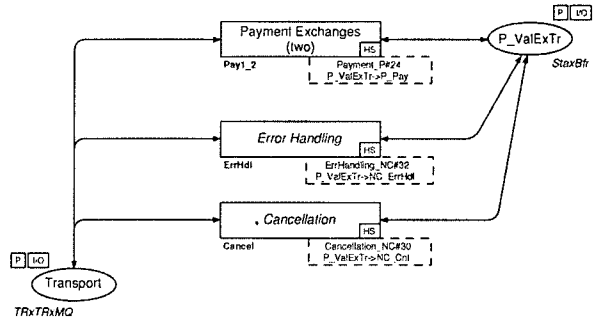


Figure 10.17: ValExTr_P page.

10.8 Exchange Level Pages

All the subpages of the transaction pages are exchange level pages, modelling the six document exchanges, the transaction cancellation and the error handling procedures. These constitute the fourth level of the IOTP CPN model. Each document exchange, error handling or transaction cancellation occurs between a Consumer_PE and a non-Consumer_PE, and is therefore specified on a pair of exchange level pages. All these pages have a similar structure, which we outline when illustrating our approach with the Authentication document exchange in Section 10.8.1.

RFC 2801 [36] defines a set of message processing guidelines for each document exchange without considering error handling and cancellation, as briefly described in Section 3.2.4. Based on this, Section 10.8.2 to 10.8.6 present the modelling of six document exchanges. Both transaction cancellation and error handling procedures, as described in Section 3.2.6 and 3.2.7, are modelled separately from document exchanges and presented in Section 10.8.7 and 10.8.8.

10.8.1 General Page Structure

The Authentication document exchange modelled in the IOTP CPN model is used by the Merchant to authenticate the Consumer (Simplification 10.4). In this case, the Merchant acts

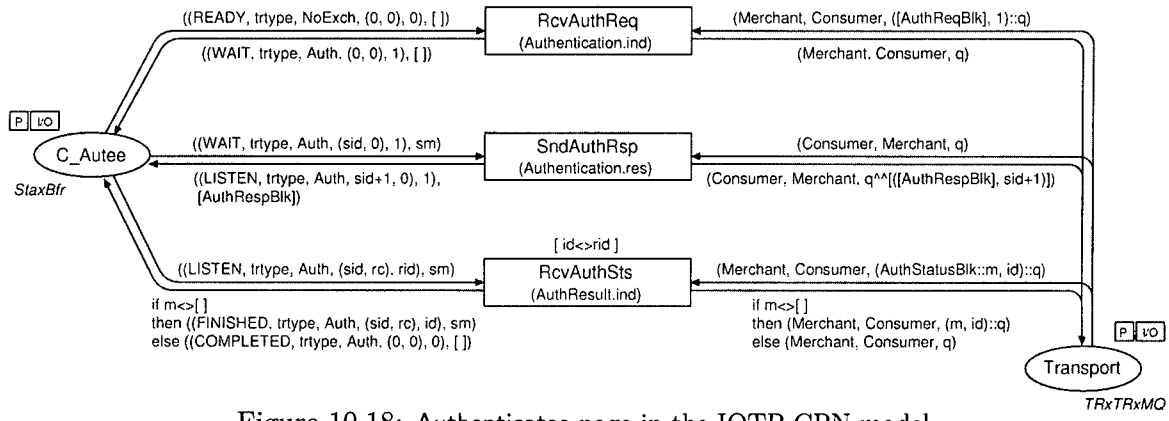


Figure 10.18: Authenticatee page in the IOTP CPN model.

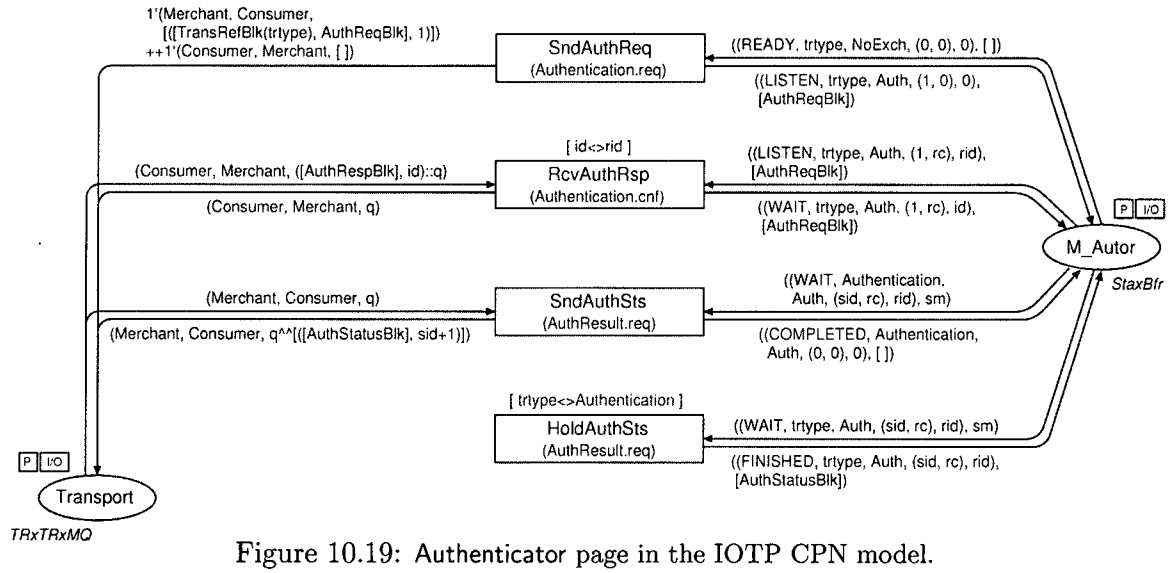


Figure 10.19: Authenticator page in the IOTP CPN model.

as the Authenticator and the Consumer as the Authenticatee. Figures 10.18 and 10.19 depict the Authenticatee and Authenticator pages that model the Authentication document exchange at the Consumer_PE and Merchant_PE, respectively. In the following, we outline the general page structure of exchange level pages using these two pages as representative examples. Each page consists of a C_place or M_place, a Transport place and a set of (ordinary) transitions.

C_place or M_place. The place C_Autee or M_Autor is linked to the C_place or M_place in each transaction page modelled at the Consumer_PE or Merchant_PE, e.g., the place C_AuthTr on page AuthTr_C, or the place M_AuthTr on page AuthTr_M.

Transitions. Most of the transitions are used to model the events of a TRole_PE sending or receiving IOTP messages. Each transition name comprises two parts. The first part indicates whether to send (Snd) or to receive (Rcv) a message, and the second part represents the message name in brief. For example, the transition RcvAuthReq (on page Authenticatee) models that the Consumer_PE receives the Authentication Request Message from the Merchant_PE.

Apart from Snd and Rcv transitions, there are other transitions whose name starts with Hold and hence are referred to as Hold transitions. A Hold transition is used to model the event of a TRole_PE holding a message which needs to be combined with the next message. For example, the transition HoldAuthOk (on page Authenticator) models the Merchant_PE holding a successful Authentication Status Message which is to be combined with a TPO Message in

the following Offer document exchange. Note that only one Transaction Reference Block and thereby only one message identifier occurs in a combined IOTP message.

Also, attached to each transition is a label given under the transition name. The label indicates a *primitive submission* resulting in sending a message, or a *primitive delivery* caused by receiving a message (Assumption 10.2). The primitive name is enclosed in parentheses. For example, the transition RcvAuthReq has a label (Authentication.ind), which represents the Consumer_PE delivering the *Authentication.ind* primitive to the Consumer_User upon the occurrence of RcvAuthReq.

Guards on the Rcv transitions restrict their enabling based on the message identifier of the IOTP message being received. Each guard comprises one boolean expression (id<>rid) to check whether the message is a duplicate. This is done by comparing the currently received message identifier id against the previously received message identifier rid. Procedure of handling a duplicate (id<>rid) is specified on two ErrHandling pages in Section 10.8.8.

Trading role state changes and message buffer operations. Each transition has input and output arcs associated with the trading role place. These arcs are used to model the state change of a TRole_PE and the operation on the message retransmission/combining buffer at that TRole_PE upon the occurrence of a transition. The colour set, StaxBfr, used to type each trading role place, comprises two parts of information: the state information of a TRole_PE and the message information in the message retransmission/combining buffer at the TRole_PE.

The state information of a TRole_PE is specified by the State colour set. It consists of six state variables (see Table 10.2). Below, we illustrate how each of the six state variables change their values when the TRole_PE sends, receives or holds an IOTP message.

- The first state variable (e.g., READY) represents the general state of the TRole_PE, and changes its value upon transition occurrence. In general, except at the beginning and the end of a document exchange, a TRole_PE is in state WAIT for a Snd transition to occur, and enters the LISTEN state after the occurrence of the transition. Similarly, for a Rcv transition, the state of the TRole_PE changes from LISTEN to WAIT. The TRole_PE is READY at the beginning of a transaction. When a document exchange is completed successfully, the TRole_PE enters the state COMPLETED or FINISHED depending on whether or not it is the last document exchange in the transaction.

- The second state variable (e.g., trtype) indicates the type of the transaction being conducted at the TRole_PE. It has the same value throughout one transaction procedure.

- The third state variable (e.g., Auth) specifies the document exchange being carried out at the TRole_PE. It remains in the same value after the current document exchange has been initiated and before the next one starts. The value NoExch is only used in the initial state of a TRole_PE that cannot tell which document exchange is to take place.

- The next is a pair of state variables (e.g., (sid, rc)) records the message identifier of the last IOTP message sent to the peer TRole_PE with its retransmission counter. The recorded message identifier (sid) has an initial value 0. It is overwritten by the message identifier (sid+1) of each *new* message sent (e.g., on occurrence of transition SndAuthRsp on page Authenticatee), and reset to 0 if the TRole_PE enters COMPLETED. The message retransmission counter (rc) also starts with an initial value 0 but increments by one upon

each message retransmission (specified on ErrHandling pages in Section 10.8.8). It is reset to 0 when the TRole_PE sends each *new* message or enters COMPLETED.

- The last state variable (e.g., rid) records the message identifier of the last IOTP message received from the peer TRole_PE. Similarly, it has an initial value 0 and is reset to 0 once the TRole_PE has COMPLETED the transaction. Upon receiving any non-duplicate message, its message identifier (id) overwrites the currently recorded message identifier for message received (rid) (e.g., on occurrence of RcvAuthRsp on page Authenticator).

The message information in the message retransmission/combining buffer at the TRole_PE is specified by the colour set IotpMsg. The buffer is initially empty ([ ]), and is reset to empty once the TRole_PE has COMPLETED the transaction. During the transaction, the message buffer (sm) is updated upon sending/combining each new message. For example, if transition SndAuthRsp on page Authenticatee occurs, the information stored in sm is replaced by the Authentication Response Message ([AuthRespBlk]). How this message buffer is used for combining messages will be discussed when we describe the exchange level pages in detail.

Queuing Operations. The input and output arcs of each Snd or Rcv transition associated with the Transport place specify the IOTP message sent to or received from the peer TRole_PE.

Except for the output arc of transition SndAuthReq (on page Authenticator) to place Transport, each arc inscription has a variable q that represents the message queue from the local TRole_PE to the peer TRole_PE, or in the opposite direction. According to Assumption 10.4, reliable message delivery is assumed within the IOTP Transport Service Provider. Therefore, any message being sent to the peer TRole_PE can be modelled by concatenating the corresponding message to the end of the queue, and any message being received from the TRole_PE is represented by the first element of the queue.

Transition SndAuthReq (on page Authenticator) models the event of sending the first IOTP message from the Merchant_PE to Consumer_PE. It has only an output arc to place Transport with an arc inscription initialising the two communication queues between the two TRole_PEs.

10.8.2 Authentication Document Exchange

Modelling an Authentication document exchange is based on its message processing guidelines in Section 9.1.1 of RFC 2801 with neither error-handling nor cancellation. Figure 10.20 shows a possible scenario of an Authentication exchange within a transaction. This is used to illustrate how the two pages Authenticatee (Figure 10.18) and Authenticator (Figure 10.19) model the Authentication exchange. In the IOTP CPN model, the Authentication exchange

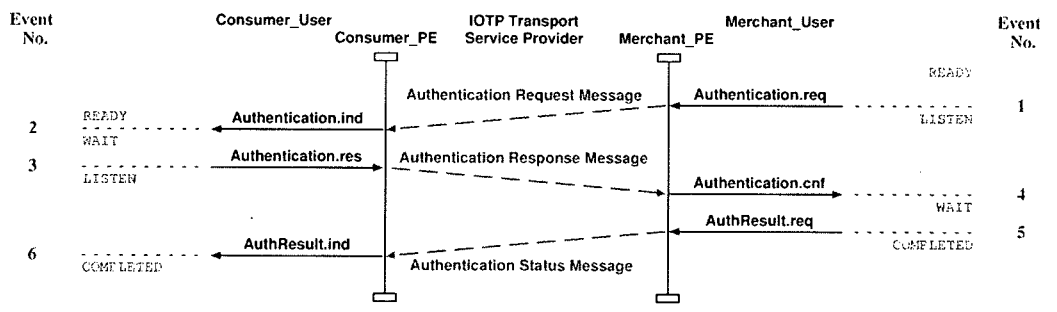


Figure 10.20: A completed Authentication document exchange.

occurs between the Consumer_PE and Merchant_PE at the beginning of a transaction (Simplification 10.4). Therefore, details of the two trading role pages Consumer (Figure 10.4) and Merchant (Figure 10.5) are also involved in the following illustration of modelling the Authentication document exchange.

Initially, the place Initialize on page Merchant has an initial marking of a transaction type value given by the constant Transaction. The occurrence of transition InitiateTr_M initiates the appropriate transaction at the Merchant_PE, by depositing into the corresponding M_place a token with the value returned from executing the function InitialState(). For example, if Transaction has a value of Authentication, then only the M_AuthTr place will have a marking of 1'((READY,Authentication,NoExch,(0,0),0),[ ]) on occurrence of InitiateTr_M (with trtype bound to Authentication).

As a result, the place M_Autor on page Authenticator, which is linked to place M_AuthTr on page AuthTr_M (and then place M_AuthTr on page Merchant via port-socket assignment), will have the same marking as the place M_AuthTr on page Merchant. Transition SndAuthReq can occur, modelling event 1 shown in Figure 10.20. Upon occurrence of an *Authentication.req* primitive, the Merchant_PE initiates an Authentication document exchange, by sending an *Authentication Request Message* to request authentication information (e.g., user account and password) from the Consumer_PE. As the first IOTP message in the transaction, the Authentication Request Message is represented by [TransRefBlk(trtype), AuthReqBlk] with a message identifier 1. The variable trtype can take any value of transaction type, modelling that the Authentication document exchange can occur in any transaction. After sending the message, the Merchant_PE enters the `LISTEN` state in Auth document exchange, updates the sent message identifier (from 0) to 1, initialises the message retransmission counter to 0, and stores the message information [AuthReqBlk] into the empty message retransmission buffer.

When the above Authentication Request Message arrives at the Consumer_PE, the transition InitiateTr_C on page Consumer is enabled. When it occurs, the Consumer_PE initiates the transaction of the type indicated by trtype from the TransRefBlk in the message. Also, the corresponding C_place will have a marking determined by the function InitialState. For example, if trtype has a value Authentication, the place C_AuthTr will have an initial marking of 1'((READY,Authentication,NoExch,(0,0),0),[ ]).

Similarly, the place C_Autee on page Authenticatee, linked to place C_AuthTr on page AuthTr_C, will have the same marking as the place C_AuthTr on page Consumer. Also, after the TransRefBlk(trtype) has been removed from the first IOTP message (on occurrence of transition InitiateTr_C on page Consumer), the rest of the message remains in the place Transport. Transition RcvAuthReq can occur, modelling event 2 in Figure 10.20. On receipt of the first (error-free) Authentication Request Message, the Consumer_PE changes from state `READY` to `WAIT`, records the received message identifier 1, and starts an Authentication document exchange by delivering an *Authentication.ind* primitive to the Consumer_User.

Event 3 in Figure 10.20 is modelled by transition SndAuthRsp on page Authenticatee. On occurrence of an *Authentication.res* primitive, the Consumer_PE replies with an *Authentication Response Message* (e.g., containing the requested user account and password) to the Merchant_PE. The message is represented by [AuthRespBlk] and has a message identifier sid+1 which is obtained by incrementing the previously sent message identifier by one. Note that,

before the *Authentication.res* occurs, the Consumer_PE may send message(s) reporting transient errors to the Merchant_PE (see page ErrHandling_C in Section 10.8.8). This increments the sent message identifier and overwrites the message retransmission buffer. Hence we use the variable sid rather than the constant 0 for the sent message identifier, and sm rather than [ ] for the message retransmission buffer. After sending the message, the Consumer_PE enters the LISTEN state, increments the sent message identifier (from sid) to sid+1, retains the value 0 for the message retransmission counter, and updates the message information (from sm) to [AuthReqBlk] in the message retransmission buffer.

Event 4 in Figure 10.20 is modelled by transition RcvAuthRsp on page Authenticator. On receipt of a non-duplicate (id<>rid) Authentication Response Message, the Merchant_PE enters the state WAIT, updates the received message identifier (from rid) to id, and delivers an *Authentication.cnf* primitive to the Merchant_User. Note that, before receiving the Authentication Response Message, the Merchant_PE may resend the previous (Authentication Request) message upon timeout, and then may receive message reporting transient error from the Consumer_PE (see page ErrHandling_NC in Section 10.8.8). Therefore, we use two variables rc and rid for the message retransmission counter and received message identifier, respectively.

Next, the two transitions SndAuthSts and HoldAuthStsOk on page Authenticator are both enabled, but only one of them can occur. The two transitions model the same primitive *AuthResult.req* but the different operations of the Merchant_PE upon primitive occurrence. On receipt of an *AuthResult.req* from the Merchant_User, the Merchant_PE generates an *Authentication Status Message* to inform the Consumer_PE of the authentication result. This message will be sent on its own in an Authentication transaction, while it needs to be combined with the first message in the following Offer document exchange[3] in any Payment-related transaction starting with an Authentication document exchange. Accordingly,

- Transition SndAuthSts models the Merchant_PE sending a standalone Authentication Status Message to the Consumer_PE on occurrence of an *AuthResult.req* in an Authentication transaction. After sending the message, the Merchant_PE enters state COMPLETED, the sent message identifier (sid), message retransmission counter (rc) and received message identifier (rid) are all reset to their initial value (0), and the message retransmission/combining buffer (sm) is cleared. This corresponds to event 5 in Figure 10.20.

- Transition HoldAuthSts (with guard [trtype<>Authentication]) models the Merchant_PE holding the Authentication Status Message, which needs to be combined with the first message in the following Offer document exchange in any Payment-related transaction. Upon occurrence of HoldAuthStsOk, the Merchant_PE enters state FINISHED and writes the message [AuthStatusBlk] to the message combining buffer. The absence of arcs between HoldAuthSts and place Transport indicates no message is sent when HoldAuthSts occurs.

Finally, the transition RcvAuthSts on page Authenticatee can occur with variable m bound to an empty list ([ ]). It models the receipt of a standalone Authentication Status Message, resulting in the occurrence of an *AuthResult.ind* at the Consumer_PE. This corresponds to event 6 in Figure 10.20. Otherwise, if m is not empty, the transition RcvAuthSts specifies the

[3]This is defined in Section 9.1.12 (Valid Combinations of Document Exchanges) of RFC 2801 [36].

receipt of an Authentication Status Message that is combined with the first message in the following Offer document exchange. In this case, RcvAuthSts can only be enabled after the Merchant_PE initiates an Offer document exchange and sends the combined message to the Consumer_PE (see Section 10.8.3).

10.8.3 Offer Document Exchanges

IOTP defines two different Offer document exchanges, namely Brand Dependent Offer and Brand Independent Offer. In Section 3.2.4 we have discussed the distinction between these two exchanges. Based on the message processing guidelines in Section 9.1.2 of RFC 2801, a Brand Dependent Offer uses three IOTP messages, *TPO*, *TPO Selection* and *Offer Response*, whereas a Brand Independent Offer only requires one message, i.e., the combined *TPO & Offer Response* message. Figure 10.21 shows the message exchanges in each of the two Offer exchanges at the beginning of a Payment-related transaction. As mentioned in Section 10.8.2, if the Offer follows an Authentication exchange, the first message, either TPO or TPO & Offer Response, needs to be combined with the previous Authentication Status Message.

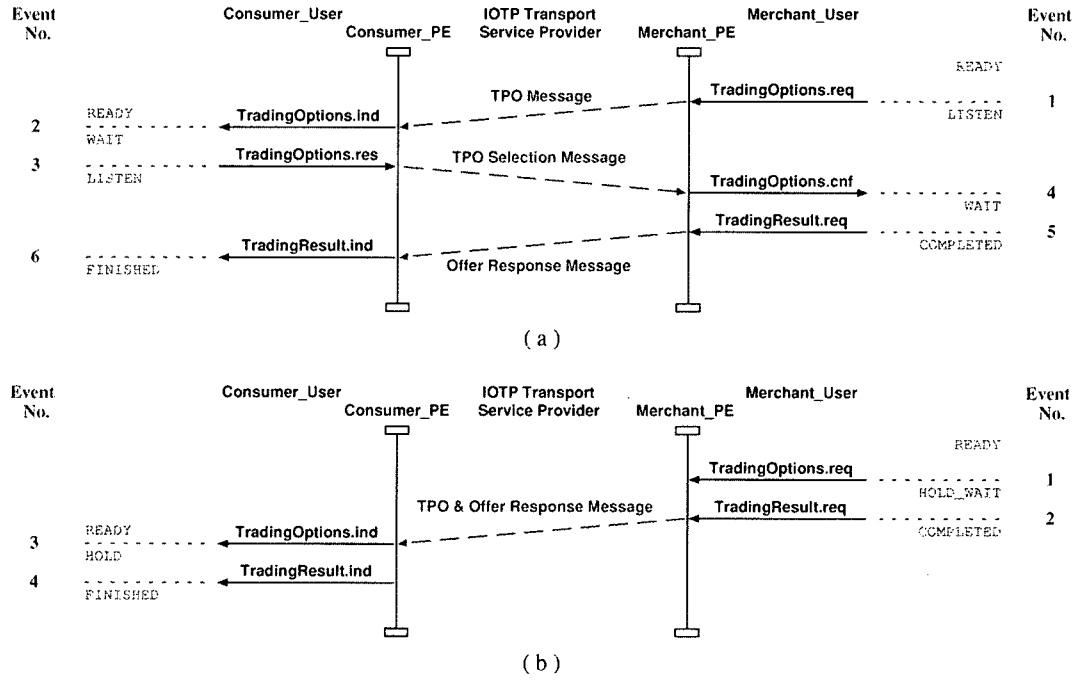


Figure 10.21: Two sequences of message exchanges in (a) a Brand Dependent Offer and (b) a Brand Independent Offer document exchange.

Figure 10.22 (a) depicts page BrdDepOffer_C modelling a Brand Dependent Offer exchange at the Consumer_PE, and (b) page BrdDepOffer_M at the Merchant_PE. Similarly, Figure 10.23 (a) and (b) depict two pages, BrdIndOffer_C and BrdIndOffer_M, modelling a Brand Independent Offer exchange at the Consumer_PE and Merchant_PE, respectively.

The place M_BDOfr on page BrdDepOffer_M, place M_BIOfr on page BrdIndOffer_M, and place M_Autor on page Authenticator (Figure 10.19) are conceptually one place as linked to place M_PayTr on page Purchase_M/Deposit_M/Withdrawal_M/Refund_M/ValExTr_M (Figure 10.13), and therefore have the same marking. Initially, the three places have the marking, e.g., 1'((READY, Purchase,NoExch,(0,0),0),[ ]) in a Purchase transaction, which is the marking of place M_Purchase on page Merchant (Figure 10.5). As a result, for each Payment-related

transaction (i.e., trtype<> Authentication), the three transitions, SndAuthReq on page Authenticator, SndTPO on page BrdDepOffer_M and HoldTPO on page BrdIndOffer_M are all enabled, but only one of them can occur. This indicates that which one of the three document exchanges (Authentication, Brand Dependent Offer or Brand Independent Offer) to be carried out at the beginning of the transaction is modelled *non-deterministically*. If an Authentication exchange occurs and then completes, the two places M_BDOfr and M_BIOfr will have the same marking as M_Autor on occurrence of transition HoldAuthSts on page Authenticator (Figure 10.19). In this case, the two transitions, SndAuthSts_TPO on page BrdDepOffer_M and HoldAuthSts_TPO on page BrdIndOffer_M, are both enabled but their occurrences are in conflict. This means that whether a Brand Dependent Offer or a Brand Independent Offer occurs after the Authentication exchange is modelled *non-deterministically*.

Similarly, three places modelling the Consumer_PE, which are C_BDOfr on page BrdDepOffer_C, C_BIOfr on page BrdIndOffer_C, and C_Autee on page Authenticatee (Figure 10.18) are conceptually one place and will have the same marking. Initially, they are marked by the token residing in one of the six C_places (e.g., C_Purchase) after transition InitiateTr_C occurs (e.g., with trtype bound to Purchase) on page Consumer (Figure 10.4).

Below, we illustrate the modelling details for each of the two Offer document exchanges.

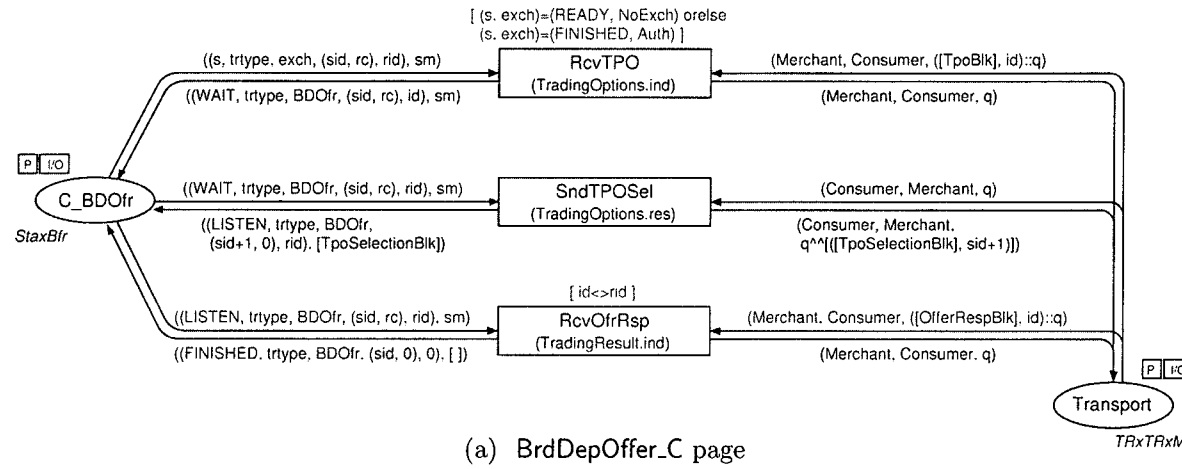
Brand Dependent Offer Document Exchange

Initially, the Merchant_PE is `READY` with NoExch carried out. Transition SndTPO on page BrdDepOffer_M is enabled, modelling the occurrence of a Brand Dependent Offer document exchange at the beginning of a Payment-related transaction. This corresponds to event 1 in Figure 10.21 (a). As the first message in the transaction, the TPO message have a message identifier 1. The TransRefBlk(trtype) is also modelled as the part of the message information, where the variable trtype can be bound to any transaction type value except Authentication.

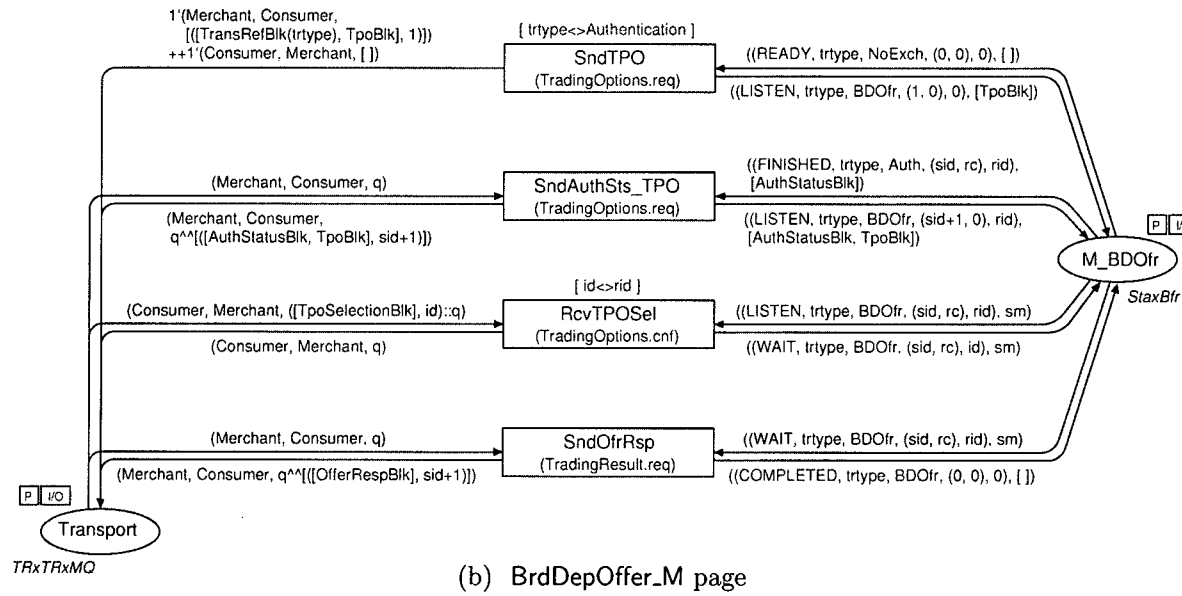
Otherwise, following an Authentication exchange, the Merchant_PE should have `FINISHED` that Auth exchange and hold [AuthStatusBlk] in the message combining buffer. Transition SndAuthSts_TPO can occur, modelling the Merchant_PE sending the combined Authentication Status & TPO Message ([AuthStatusBlk, TpoBlk]) with a new message identifier sid+1 to the Consumer_PE. No matter whether SndTPO or SndAuthSts_TPO occurs, the Merchant_PE initiates a BDOfr exchange and enters the `LISTEN` state.

When the Consumer_PE receives a TPO Message as the first message in the transaction, transition InitiateTr_C on page Consumer (Figure 10.4) occurs first, which removes [TransRefBlk(trtype)] from the message and deposits a token 1'((READY,trtype,NoExch,(0,0),0),[ ]) in the corresponding C_place. Place C_BDOfr on page BrdDepOffer_C then has an initial marking as that C_place, and transition RcvTPO can occur (with s bound to `READY` and exch to NoExch). This corresponds to event 2 in Figure 10.21 (a). Otherwise, if the Consumer_PE receives an Authentication Status & TPO Message, transitions HoldAuthSts on page Authenticatee (Figure 10.18) and then RcvTPO (with s bound to `FINISHED` and exch to Auth) occur, specifying the receipt of [AuthStatusBlk] and then [TpoBlk] from the combined message. Here we model the receipt of one (combined) Authentication Status & TPO Message using two transitions to separate the occurrences of *AuthResult.ind* and *TradingOptions.ind* primitives.

Next, the Consumer_PE replies with a TPO Selection Message ([TpoSelectionBlk]) on occurrence of a *Trading-Options.res* primitive (event 3 in Figure 10.21 (a)). This is modelled by



(a) BrdDepOffer_C page



(b) BrdDepOffer_M page

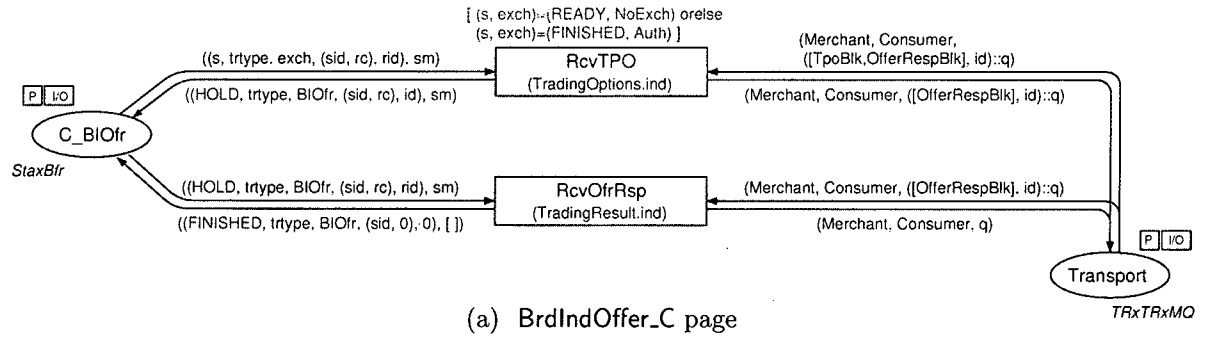
Figure 10.22: Two pages modelling a Brand Dependent Offer document exchange.

transition SndTPOSel on page BrdDepOffer_C. The receipt of the TPO Selection Message at the Merchant_PE and thereby the occurrence of *TradingOptions.ind* (event 4 in Figure 10.21 (a)) is specified by transition RcvTPOSel on page BrdDepOffer_M.

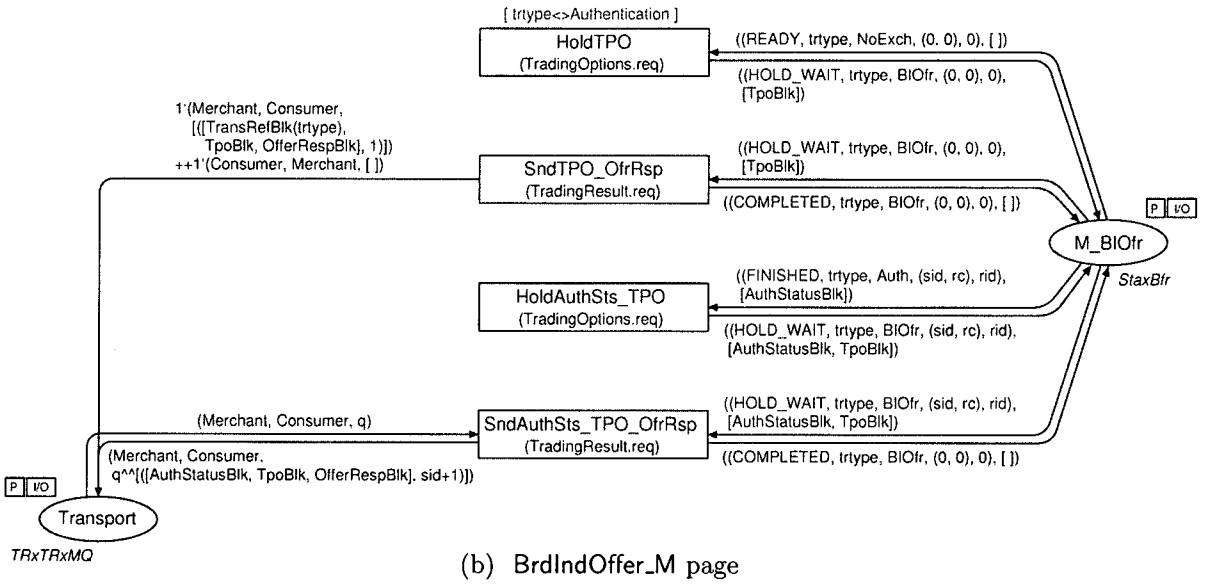
The last two events 5 and 6 in Figure 10.21 (a) occur, in which case, an Offer Response Message ([OfferRespBlk]) is sent from the Merchant_PE to Consumer_PE, and as a result the Brand Dependent Offer exchange is complete. These are modelled by transitions SndOfrRsp on page BrdDepOffer_M and RcvOfrRsp on page BrdDepOffer_M, respectively. Upon occurrence of SndOfrRsp, the Merchant_PE enters the successful terminal state COMPLETED, resets sid, rc and rid to their initial value (0), and clears buffer sm. Upon occurrence of RcvOfrRsp, the Consumer_PE enters the FINISHED state and moves onto the following document exchange, i.e., Payment, in the transaction. Both rc and rid are reset to 0 and sm is cleared, whereas sid retains its current value because it will be used by the Consumer_PE to generate the message identifier of the next IOTP message in the transaction.

Brand Independent Offer Document Exchange

There is only one message sent from the Merchant_PE to Consumer_PE during a Brand Independent Offer document exchange. The message is either a TPO & Offer Response Message ([TpoBlk, OfferRespBlk]) or an Authentication Status Message & TPO & Offer Response Message ([AuthStatusBlk, TpoBlk, OfferRespBlk]). Which of the two messages to be sent depends on whether the Brand Independent Offer occurs at the beginning of a transaction or follows an Authentication exchange in the transaction.



(a) BrdIndOffer_C page



(b) BrdIndOffer_M page

Figure 10.23: Two pages modelling a Brand Independent Offer document exchange.

On page BrdIndOffer_M, HoldTPO and SndTPO_OfrRsp transitions, when taken together, model events 1 and 2 in Figure 10.21 (b) where the Brand Independent Offer occurs as the first document exchange in the transaction. The other two transitions, HoldAuthSts_TPO and SndAuthSts_TPO_OfrRsp, are used for generating and sending [AuthStatusBlk, TpoBlk, OfferRespBlk] when the Brand Independent Offer follows an Authentication exchange in the transaction. After a Hold transition (HoldTPO or HoldAuthSts_TPO) but before a Snd transition (SndTPO_OfrRsp or SndAuthSts_TPO_OfrRsp) occurs, the Merchant_PE is in the HOLD_WAIT state, expecting the trading result information from the Merchant_User. Such information is necessary in generating [OfferRespBlk] in each of the above two combined messages.

Before any transition is enabled on page BrdIndOffer_C, either the transition InitiateTr_C on page Consumer (Figure 10.4) or the transition RcvAuthOk on page Authenticatee (Figure 10.18)

occurs first, which removes [TransRefBlk(TrType)] or [AuthStatusBlk] and leaves [TpoBlk, OfferRespBlk] in the message. On page BrdIndOffer_C, the receipt of [TpoBlk, OfferRespBlk] is modelled by two transitions RcvTPO and then RcvOfrRsp, which results in the occurrences of two primitives *TradingOptions.ind* followed by *TradingResult.ind* at the Consumer_PE (events 3 and 4 in Figure 10.21 (b)). Between the occurrences of these two transitions, the Consumer_PE is in the HOLD state, waiting for the rest of the message to be processed. Again, we use two transitions for specifying one combined message, in order to separate the two corresponding primitives occurrences and therefore match the service.

10.8.4 Payment Document Exchanges

The Payment document exchange allows a payment to be made by the Consumer to the Payment Handler, and is always initiated by the Consumer in IOTP. Each Payment-related transaction, except Value Exchange, involves one Payment document exchange. A Value Exchange transaction includes two Payment document exchanges which occur in sequence. Each Payment document exchange uses the same set of IOTP messages: *Payment Request*; *Payment Exchange*; and *Payment Response*. Figure 10.24 shows a possible sequence of message exchanges in a Payment document exchange.

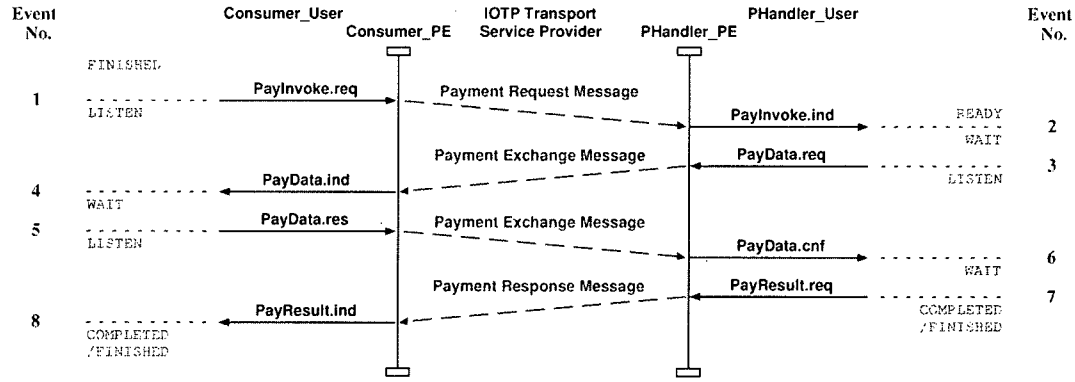


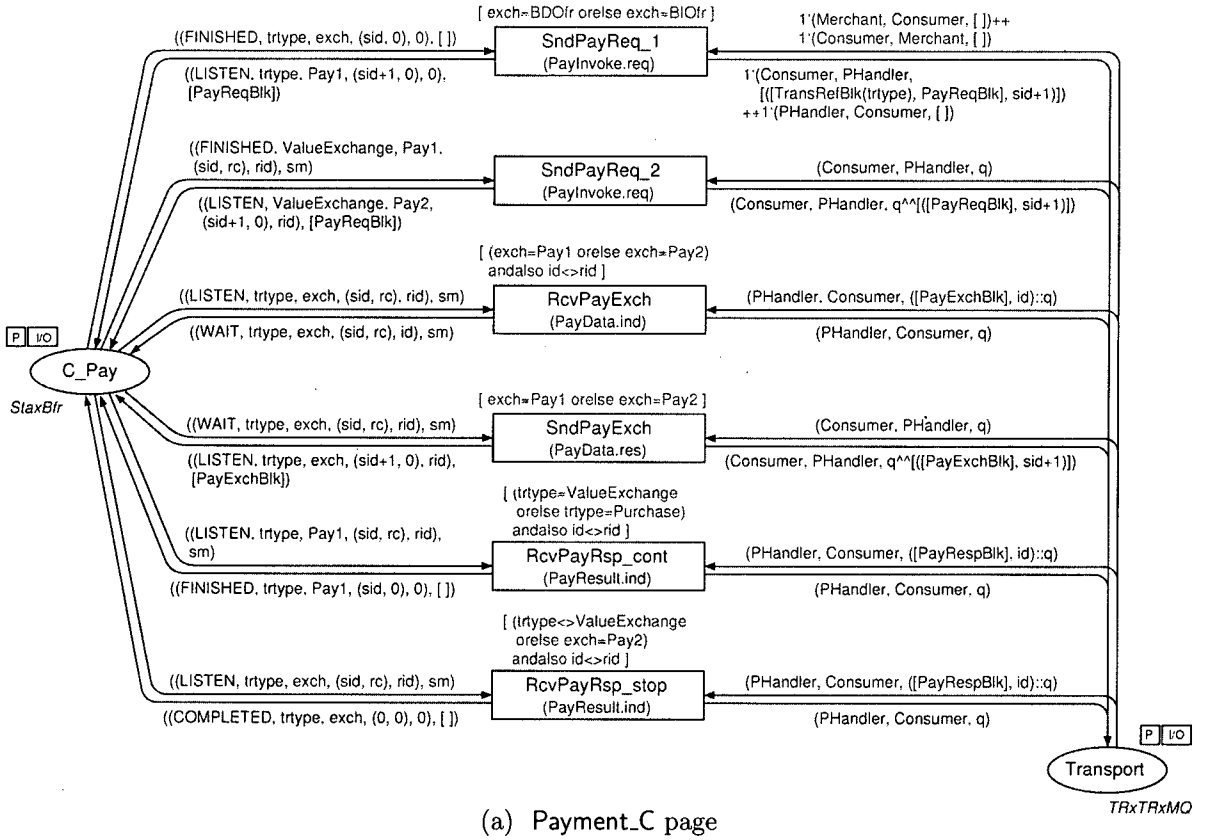
Figure 10.24: A possible sequence of message exchanges in a Payment document exchange.

Two exchange level pages, Payment_C and Payment_P, as illustrated in Figure 10.25, capture one or two Payment document exchanges based on the message processing guidelines in Section 9.1.3 of RFC 2801. By means of port-socket assignment, the place C_Pay on page Payment_C has the same marking as place C_BDOfr on page BrdDepOffer_C (Figure 10.22 (a)) or place C_BIOfr on page BrdIndOffer_C (Figure 10.23 (a)), and the place P_Pay on page Payment_P is linked to one of the P_places on page PHandler (Figure 10.6).

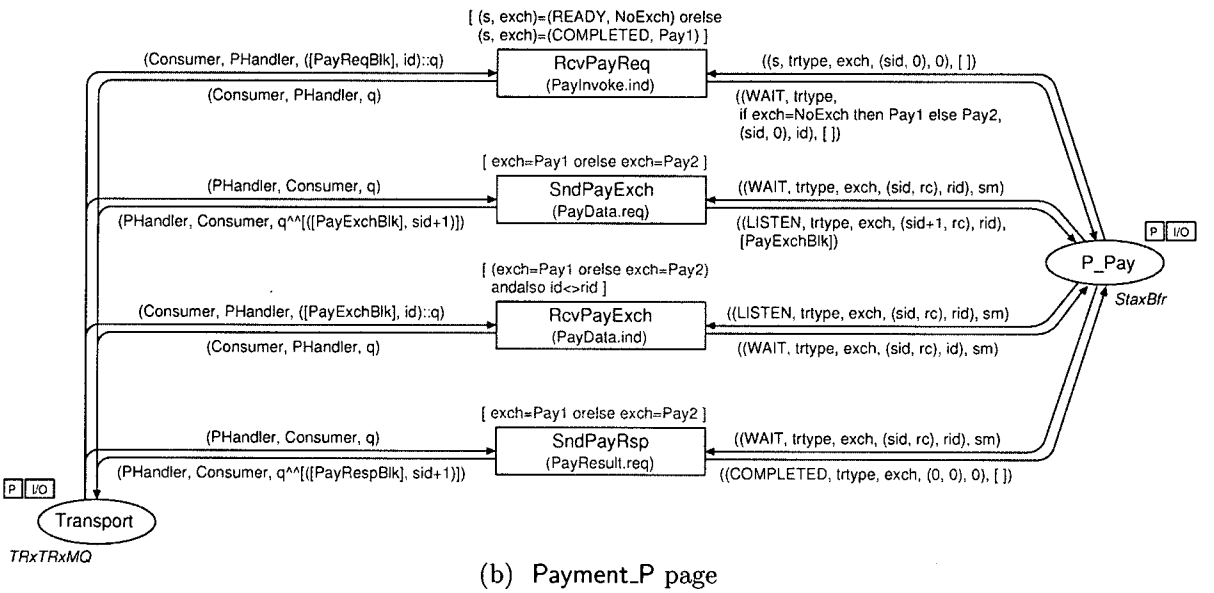
On page Payment_C, transition SndPayReq_1 models the Consumer_PE initiating the first (or, the only) Payment document exchange (Pay1) after having FINISHED the preceding Offer exchange (either BDOfr or BIOfr). Transition SndPayReq_2 specifies the start of the second Payment document exchange (Pay2) after the first one is FINISHED, and is only used for a ValueExchange transaction. When either of these two transitions occurs, a Payment Request Message is sent from the Consumer_PE to the PHandler_PE (event 1 in Figure 10.24).

The arc from place Transport to SndPayReq_1 is used to model that: 1) the message queues between the Consumer_PE and Merchant_PE should be empty before a Pay1 exchange starts; and 2) they are removed once the Pay1 exchange has started. Conceptually, the message queues will be available between the two TRole_PEs until the communication between them

is disconnected. However, in the IOTP CPN model, we consider to delete the empty queues between the Consumer_PE and Merchant_PE when the Consumer_PE switches from communicating with the Merchant_PE to communicating with the PHandler_PE. The reason for doing so is to simplify the modelling of transaction cancellation and error handling that occur after a Payment exchange is initiated. This is further explained in Section 10.8.7 and 10.8.8.



(a) Payment_C page



(b) Payment_P page

Figure 10.25: Two pages modelling the Payment document exchange(s).

The Payment Request Message for Pay1 is the first IOTP message for initiating the transaction at the PHandler_PE. Hence, the receipt of this Message by the PHandler_PE (event 2

in Figure 10.24) is modelled by the occurrences of transitions InitiateTr_P on page PHandler (Figure 10.6) and then RcvPayReq (with s bound to READY and exch to NoExch) on page Payment_P. Otherwise, receiving the Payment Request Message for Pay2 is captured by the occurrence of only RcvPayReq transition (with s bound to FINISHED and exch to Pay1).

No matter whether Pay1 or Pay2 is initiated, both SndPayExch and SndPayRsp transitions on page Payment_P are enabled, but their occurrences are in conflict. Conceptually, which transition will occur depends on whether a particular payment system is required for processing of payment information. In the IOTP CPN model, this is specified *non-deterministically* as we do not model the detailed payment information in the Payment document exchange.

If a payment system (e.g., SET) is used, the PHandler_PE and Consumer_PE will then start to exchange Payment Exchange Messages encapsulating payment system-specific data (e.g., SET messages), which are called *payment data exchanges* between these two TRole_PEs (Section 3.2.4). *Event 3* to *6* in Figure 10.24 represent one payment data exchange, and are captured by the successive occurrences of the following four transitions in the CPN model: SndPayExch on page Payment_P, RcvPayExch and then SndPayExch on page Payment_C, and finally RcvPayExch on Payment_P. After RcvPayExch occurs on page Payment_P, place P_Pay will have a marking 1'((Wait,trtype,exch,(sid,rc),rid),sm), which again enables transition SndPayExch and thereby triggers another payment data exchange. As a result, this can occur for any number of times, catering for various payment systems to be encapsulated by IOTP.

On page Payment_P, whenever the SndPayExch transition is enabled, the SndPayRsp transition can also occur. Transition SndPayRsp models the PHandler_PE sending a Payment Response Message to complete the Payment document exchange (event 7 in Figure 10.24), and therefore its occurrence disables transition SndPayExch. The arc from SndPayRsp to place P_Pay indicates that, upon sending the Payment Response Message, the PHandler_PE enters the FINISHED state and continues if the current document exchange is Pay1 in a ValueExchange transaction, or otherwise enters the COMPLETED state and stops any further processing.

Finally, receiving the Payment Response Message at the Consumer_PE (event 8 in Figure 10.24) is captured by the occurrence of either RcvPayRsp_cont or RcvPayRsp_stop transition on page Payment_C. RcvPayRsp_cont models the Consumer_PE moves onto the second Payment document exchange or Delivery document exchange after having FINISHED the current Pay1, and therefore can only occur in a ValueExchange or Purchase transaction, as specified by its guard expression. RcvPayRsp_stop models the Consumer_PE terminating the transaction once the current Payment document exchange is COMPLETED, and therefore its enabling conditions are given as whether this document exchange is the last one in the transaction. More specifically, a ValueExchange transaction stops on completion of Pay2, while the other Payment-related transactions, including a Purchase without delivery, stop at the end of Pay1. The guard associated with RcvPayRsp_stop contains a simplified representation of the above conditions, where the boolean expression exch=Pay2 is equivalent to (trtype=ValueExchange andalso exch=Pay2) and trtype<>ValueExchange to trtype<>ValueExchange andalso exch=Pay1, because Pay2 is only possible in ValueExchange transactions. Furthermore, in the case of a Purchase transaction, both RcvPayRsp_cont and RcvPayRsp_stop transitions are enabled on completion of Pay1, indicating the transaction may continue to the next document exchange (Delivery, Section 10.8.6) or stop (if there is no delivery). Which transition to occur is modelled in a *non-deterministic* way.

10.8.5 Payment-and-Delivery Document Exchange

The Payment Handler can make the delivery as well as process the payment for purchase. The Payment-and-Delivery document exchange, which is *only* possible in Purchase transactions, should be used in this case. A sequence of message exchanges is shown in Figure 10.26 based on the message processing guidelines in Section 9.1.5 of RFC 2801. Compared to the Payment document exchange in Figure 10.24, a Payment-and-Delivery uses two combined messages: *Payment Request & Delivery Request* to initiate the exchange (Assumption 10.1), and *Payment Response & Delivery Response* to complete the exchange (page 213, [36]).

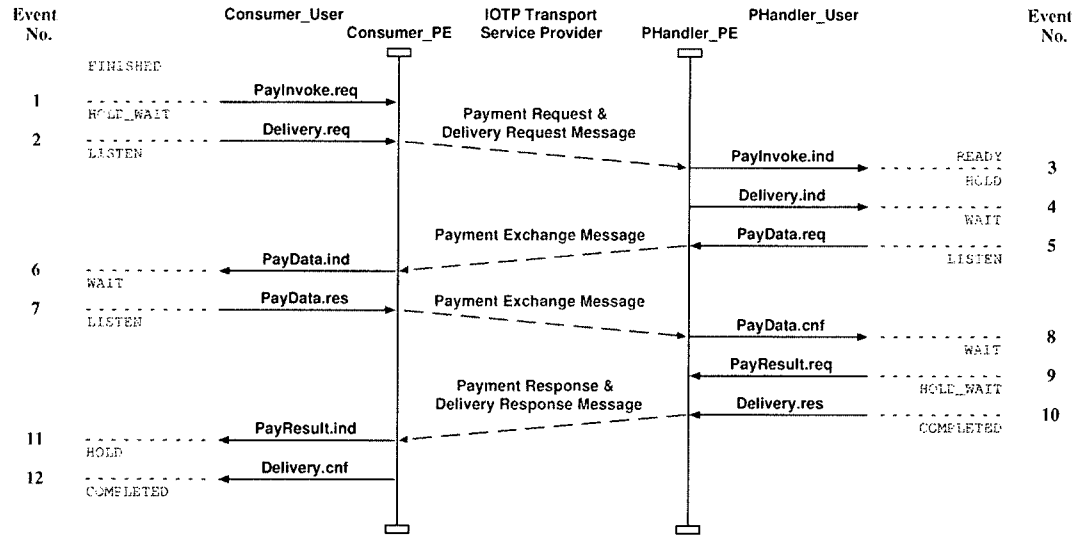


Figure 10.26: A sequence of message exchanges in a Payment-and-Delivery document exchange.

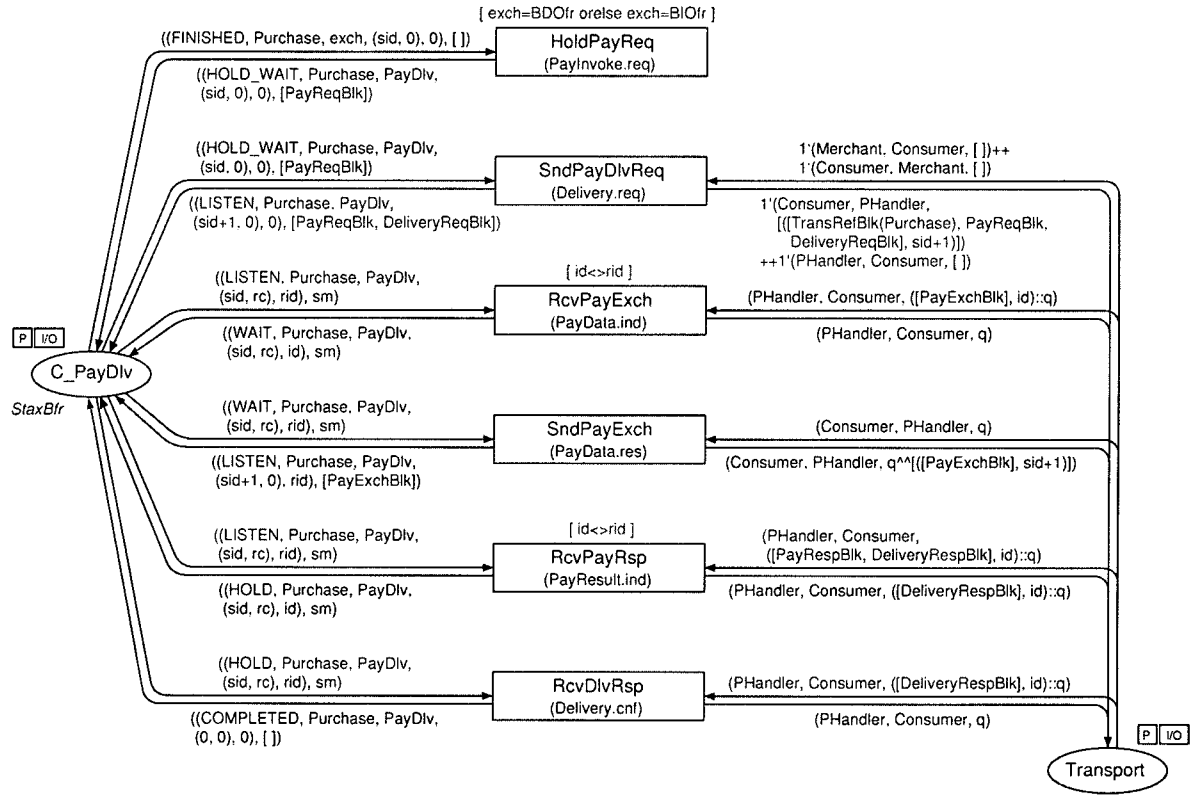
Figure 10.27 depicts the two pages, PayDelivery_C and PayDelivery_P, for the Payment-and-Delivery document exchange at the Consumer_PE and PHandler_PE. It is very similar to the modelling of Payment document exchange except when initiating and completing the exchange.

Initially, two transitions HoldPayReq and then SndPayDlvReq on page PayDelivery_C occur. This represents the Consumer_PE sending a Payment Request & Delivery Request Message (*events 1* and *2* in Figure 10.26). Receiving PayRespBlk and then DeliveryRespBlk from this message at the PHandler_PE (*events 3* and *4* in Figure 10.26) is modelled by the successive occurrences of the RcvPayReq and RcvDlvReq transitions on page PayDelivery_P.

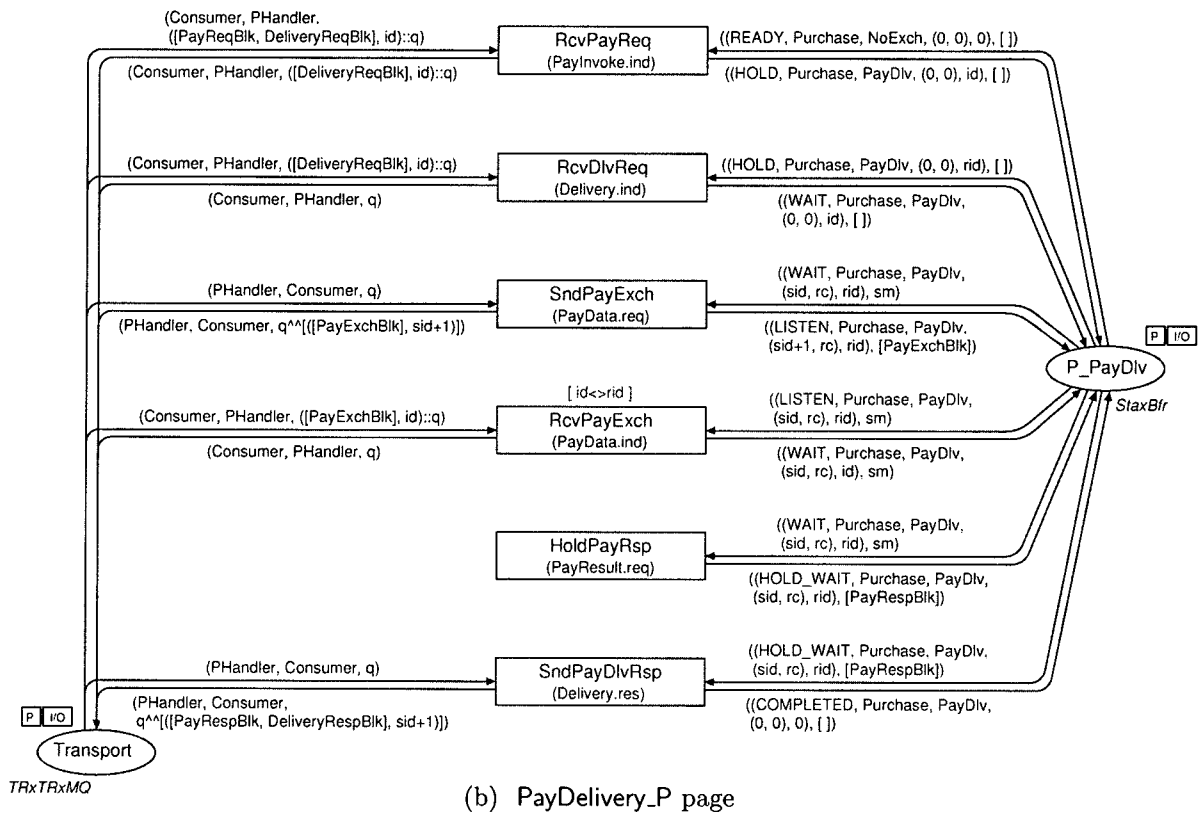
At the end of the exchange, two transitions HoldPayRsp and then SndPayDlvRsp on page PayDelivery_P occur. This results in sending a Payment Response & Delivery Response Message by the PHandler_PE (*events 9* and *10* in Figure 10.26). The receipt of this message at the Consumer_PE (*events 11* and *12* in Figure 10.26) is modelled by the successive occurrences of the RcvPayRsp and RcvDlvRsp transitions on page PayDelivery_C. On completion of the Payment-and-Delivery document exchange, both TRole_PEs enter the COMPLETED state and stop any further processing in the Purchase transaction.

10.8.6 Delivery Document Exchange

The only communication between the Consumer_PE and DHandler_PE occurs during a Delivery document exchange. Like the Payment-and-Delivery (Section 10.8.5), the Delivery document exchange is only carried out in a Purchase transaction. It uses two messages: *Delivery*



(a) PayDelivery_C page



(b) PayDelivery_P page

Figure 10.27: Two pages modelling the Payment-and-Delivery document exchange.

Request and *Delivery Response*. Figure 10.28 shows the message exchanges in a Delivery document exchange, according to the message processing guidelines in Section 9.1.4 of [36].

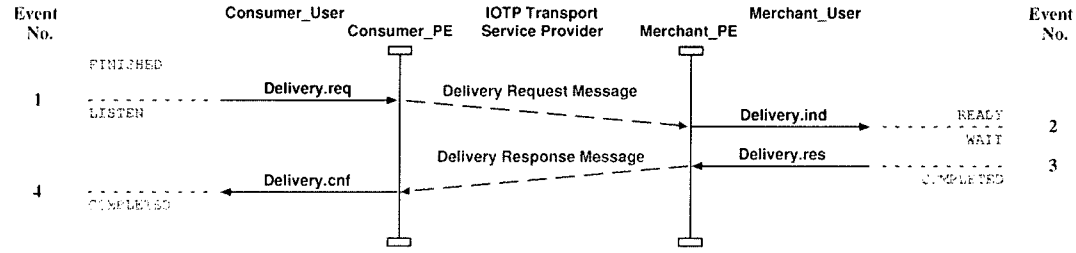
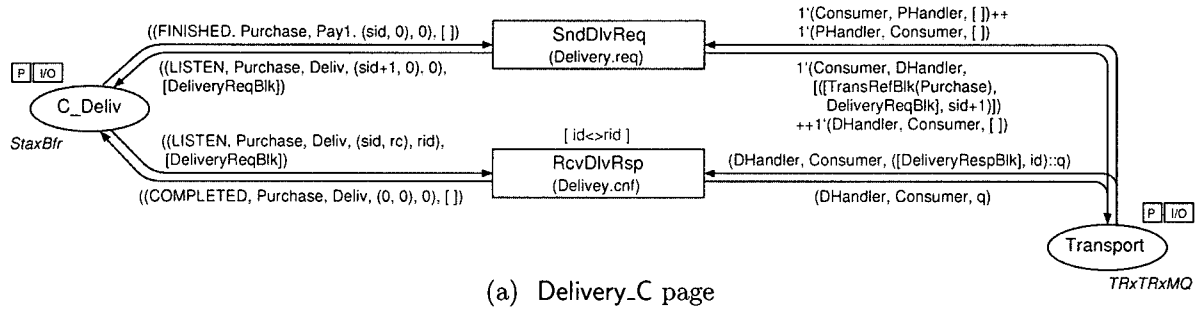
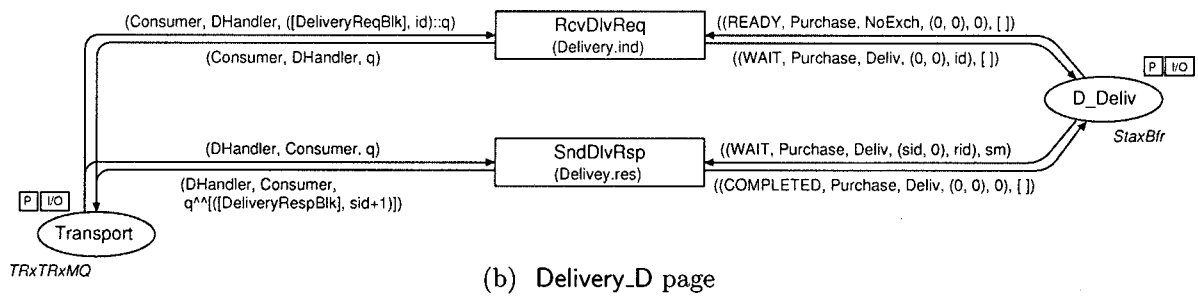


Figure 10.28: The message exchanges in a Delivery document exchange.

The Delivery_C and Delivery_D pages, used to model a Delivery document exchange at the Consumer_PE and DHandler_PE, are illustrated in Figure 10.29.



(a) Delivery_C page



(b) Delivery_D page

Figure 10.29: Two pages modelling the Delivery document exchange.

The Delivery document exchange starts at the Consumer_PE. Transition SndDlvReq on page Delivery_C corresponds to *event 1* in Figure 10.28. On its occurrence, the Consumer_PE switches communicating with the PHandler_PE to communicating with the DHandler_PE, where the empty message queues between the Consumer_PE and PHandler_PE are removed, and a Delivery Request Message is sent to the DHandler_PE.

Since the Delivery Request Message is the first message received by the DHandler_PE in the transaction, *event 2* in Figure 10.28 is captured by the successive occurrences of transitions InitiateTr_D on page DHandler (Figure 10.7) and RcvDlvReq on page Delivery_D.

Next, transition SndDlvRsp on page Delivery_D occurs first, followed by transition RcvDlvRsp on page Delivery_C, corresponding to *event 3* and 4 in Figure 10.28. On occurrences of these two transitions, both Consumer_PE and DHandler_PE enter the COMPLETED state, signaling the completion of the Delivery document exchange and thereby the Purchase transaction.

10.8.7 Transaction Cancellation

RFC 2801 [36] defines a transaction cancellation procedure separately from the specification of each document exchange. In Section 3.2.6 we have introduced how each trading role initiates or is notified about the cancellation of an ongoing transaction.

Since transaction cancellation is defined as a general procedure, we have modelled it on two pages: Cancellation_C for the Consumer_PE, Cancellation_NC shared by all the three non-Consumer_PEs. The two pages are modelled in a similar way, as shown in Figures 10.30 and 10.31, respectively. We discuss the modelling approach in detail for page Cancellation_C, and then describe page Cancellation_NC focusing on the differences with respect to page Cancellation_C.

Transaction Cancellation at the Consumer_PE

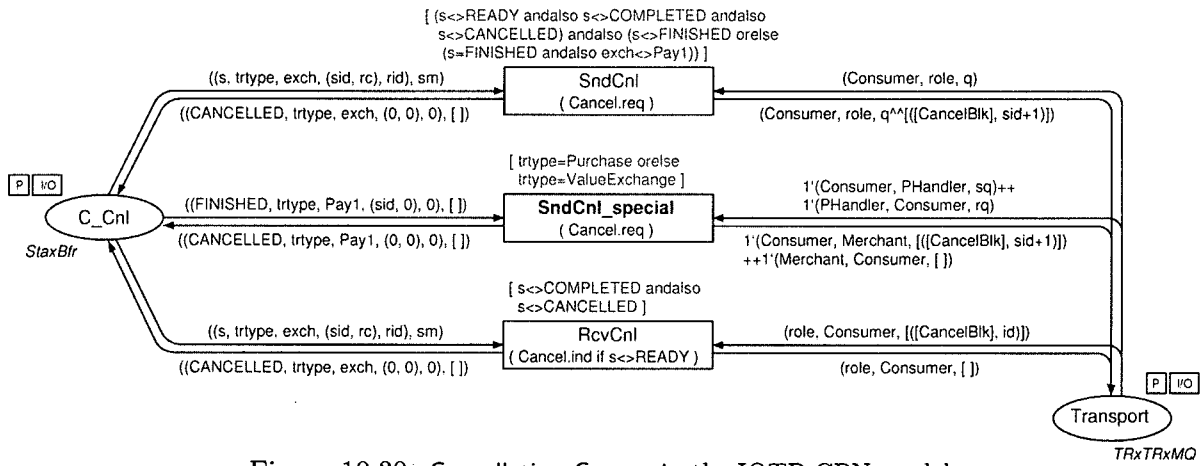


Figure 10.30: Cancellation_C page in the IOTP CPN model.

A Consumer_PE may cancel a transaction at any time when the transaction is in progress: 1) as a result of a *Cancel.req* primitive submitted by the (local) Consumer_User, or 2) on receipt of a *Cancel Message* (represented by [CancelBlk]) from the peer non-Consumer_PE. On occurrence of *Cancel.req*, the Consumer_PE sends a *Cancel Message* to the peer non-Consumer_PE. On receipt of the Cancel Message, the Consumer_PE delivers the corresponding *Cancel.ind* to the user. On page Cancellation_C, sending the Cancel Message is modelled by the SndCnl and the SndCnl_special transition, and receiving the Cancel Message by the RcvCnl transition.

As described in Section 3.2.6, when the Consumer_PE cancels a transaction, the Cancel Message is sent, in most cases, to the peer non-Consumer_PE. Transition SndCnl models all the corresponding events. The inscription on each arc between SndCnl and place Transport has a variable role representing the above non-Consumer_PE. However, there are two *special* cases, where the Cancel Message should be sent to the Merchant rather than the Payment Handler, after the Consumer_PE has finished the (first) Payment document exchange but 1) before the next Delivery document exchange occurs in a Purchase transaction or 2) before the next (second) Payment document exchange occurs in a Value Exchange transaction. Transition SndCnl_special models these special cases. It has input and output arcs associated with place Transport, indicating the Consumer_PE switches communicating with PHandler_PE to communicating with Merchant_PE to which the Cancel Message needs to be sent.

The guard on transition SndCnl comprises two parts of boolean expressions specifying different conditions on the state of the Consumer_PE for the transition to occur.

- The first part, (s<>READY andalso s<>COMPLETED andalso s<> CANCELLED), indicates when it is possible for the Consumer_user to submit a *Cancel.req*. The initial state READY is used when the Consumer_PE has just started a transaction but not yet notified the user about the transaction. The two terminal states, COMPLETED and CANCELLED, are used when the Consumer_PE has already terminated the transaction. Hence, it is not possible for the Consumer_User to submit a *Cancel.req* in any of the above three states.

- The second part, (s<>FINISHED orelse (s=FINISHED andalso exch<>Pay1)), separates the special case modelled by SndCnl_special from those common cancellations modelled by SndCnl. It is therefore complementary to the enabling conditions of SndCnl_special when the Consumer_PE is in FINISHED.

Similarly, the guard on RcvCnl specifies that the Consumer_PE will not accept a Cancel Message, after having COMPLETED or CANCELLED the transaction. Otherwise, the Consumer_PE, which can be in any state except READY, will deliver a *Cancel.ind* to the user on receipt of a Cancel Message. When the Consumer_PE is READY, no primitive has yet occurred to the user. Hence, it is not sensible to deliver a *Cancel.ind* to the user that is not aware of the transaction.

On occurrence of each of the above three transitions, the Consumer_PE enters the aborted terminal state CANCELLED, resets the sent message identifier sid, message retransmission counter rc and received message identifier rid to their initial value (0) and purges the message retransmission/combining buffer sm (to [ ]).

Transaction Cancellation at the non-Consumer_PE

On page Cancellation_NC, both SndCnl and LocalCnl transitions model the non-Consumer_PE submitting a *Cancel.req* for cancelling the transaction. As specified by the guard on SndCnl, it is not possible for a non-Consumer_User to submit a *Cancel.req* when the (local) non-Consumer_PE is in one of the three states: READY, COMPLETED and CANCELLED.

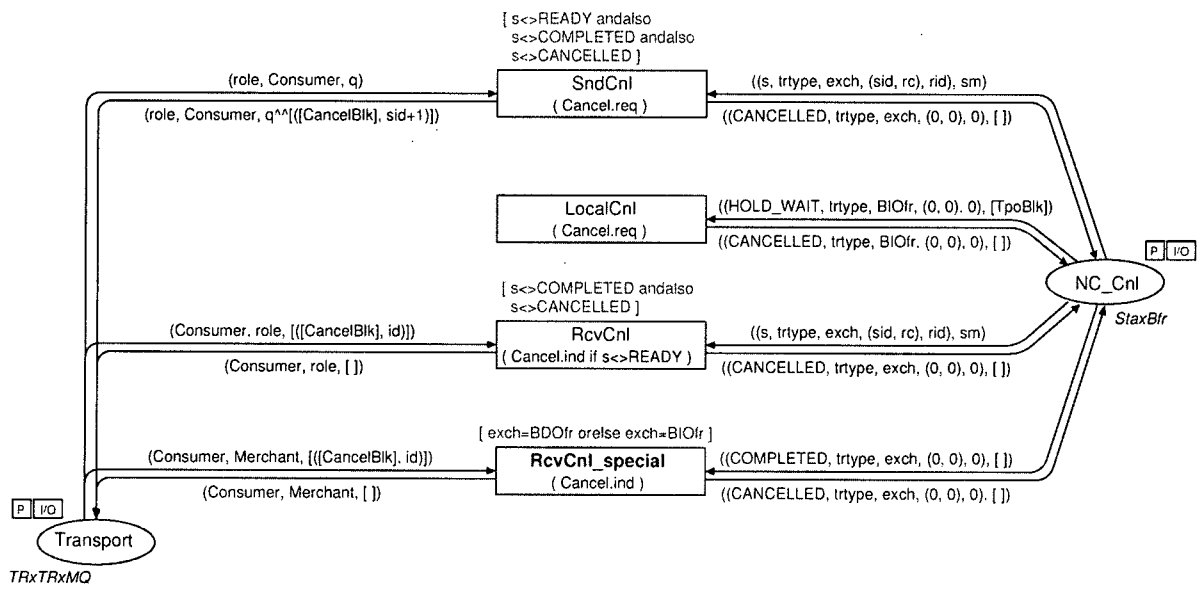


Figure 10.31: Cancellation_NC page in the IOTP CPN model.

Transition LocalCnl is used to capture a special event of cancellation at the Merchant_PE, which occurs as a result of the Merchant_User submitting a *Cancel.req* but before the Merchant_PE starts to communicate with the Consumer_PE. It has only input and output arcs associated with place NC_Cnl. The input arc inscription indicates the transition is only used when the Merchant_PE receives a *Cancel.req* during a Brand Independent Offer at the beginning of a transaction (see page BrdIndOffer_M in Figure 10.23 (b)). Both sent and received message identifiers have an initial value 0, indicating no message has yet been sent or received by the Merchant_PE. There are no arcs between LocalCnl and place Transport, because the Merchant_PE has not yet started to communicate with the Consumer_PE.

Both RcvCnl and RcvCnl_special transitions specify the receipt of a Cancel Message which terminates the transaction. RcvCnl models the behaviour of a non-Consumer_PE upon receiving a Cancel Message in a similar way to the transition RcvCnl on page Cancellation_C. The guard associated with RcvCnl_special specifies the Merchant_PE can still accept the Cancel Message and thereby deliver a *Cancel.ind* to the Merchant_User after having COMPLETED either a BDOfr or BIOfr exchange. Furthermore, the above two Rcv transitions on page Cancellation_NC are modelled in a consistent manner with the two Snd transitions on page Cancellation_C. This means that RcvCnl always occurs as a result of the occurrence of SndCnl, and RcvCnl_special only occurs as a result of the occurrence of SndCnl_special.

10.8.8 Error Handling

As mentioned in Section 3.2.7, IOTP error handling concerns how a TRole_PE processes received messages containing technical errors, or those that are duplicates. A Consumer_PE is an IOTP client, while non-Consumer_PEs are IOTP servers. The server and client role processing sequences, as defined in Sections 4.5 and 4.6 of RFC 2801, outline the IOTP error handling procedure for both the Consumer_PE (client) and non-Consumer_PEs (servers) in six major steps. We re-organise these six steps in a logical order and present them using a flow chart shown in Figure 10.32. Like transaction cancellation, error handling is defined as a general procedure. The two pages, ErrHandling_C and ErrHandling_NC, as shown in Figures 10.33 and 10.34, model the error handling procedure at the Consumer_PE and each non-Consumer_PE. Page ErrHandling_NC is a subpage shared by all the transaction pages specified for all the three non-Consumer_PEs (see Hierarchy page in Figure 10.2). How the pages in Figures 10.33 and 10.34 reflect the error handling procedure is described in each of the steps shown in Figure 10.32. The number written at the lower right hand corner of each transition on these two pages indicates which step is modelled by the transition.

For brevity, we refer to an IOTP message received by a TRole_PE as an *input message*, and a message sent by a TRole_PE as an *output message*. Also, we use the two terms, a *message in error* and a *message reporting an error*, which have been introduced in Section 3.2.7.

1. *The first step is to check whether there is any input message. If there is, the TRole_PE continues to Step 2, but if not and the TRole_PE is expecting an input message, it carries out two sub-steps concerning message retransmission upon time-out:*

   1.1 *Check whether the retransmission timer expires or a transport level error occurs. If it does, the TRole_PE continues to Step 1.2, otherwise, it keeps waiting for an input message;*

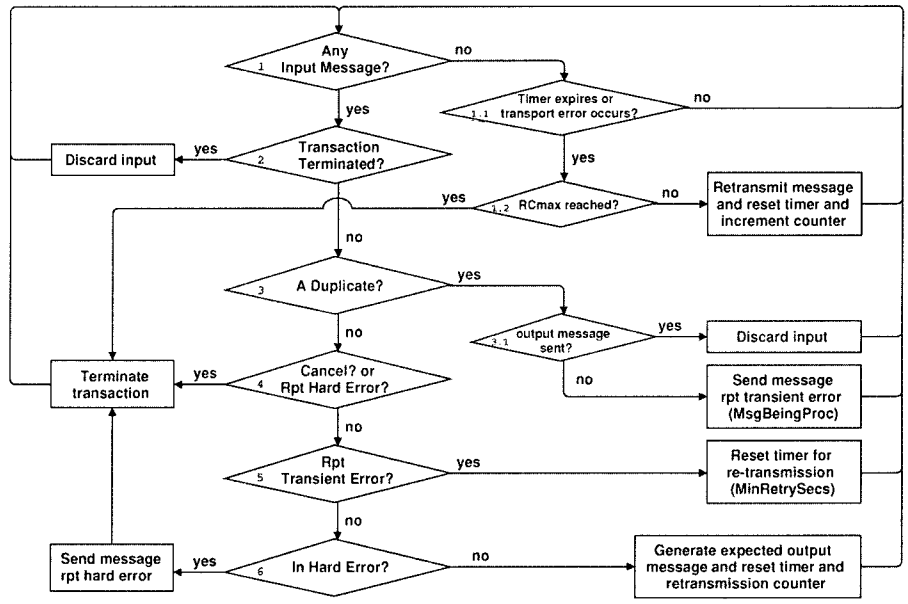


Figure 10.32: Error Handling Procedure at a TRole_PE.

1.2 *Check whether the retransmission counter reaches its maximum value (RCmax). If it has, the TRole_PE stops any further processing in the transaction, otherwise, it re-sends the previous output message, resets the timer and increments the counter by one.*

On each ErrHandling page, the two transitions Retransmit_RstTimer and Stop_1 are used when the retransmission timer expires or a transport level error occurs. Retransmit_RstTimer models the TRole_PE re-sending the previously sent message, sm, if the timer expires or a transport level error occurs, and also the message retransmission counter, rc, is less than *RCmax* (*RCmax_c* for the Consumer_PE, *RCmax_m* for the Merchant_PE, *RCmax_p* for the PHandler_PE and *RCmax_d* for the DHandler_PE, see Appendix E). Upon message retransmission, the timer is reset, and the retransmission counter is incremented (rc+1). The transition Stop_1 models the TRole_PE entering the CANCELLED state and delivering a *Cancel.ind* to the TRole_User, due to a time-out or a transport level error occurring after rc reaches *RCmax*. In this case, no message (e.g., a Cancel Message) is sent to the peer TRole_PE upon transaction termination.

In the above modelling, we ignore any details of the timer or the transport level errors and only consider their effects on the protocol operation, i.e., triggering the message retransmission. Also, both the timer expiring and the transport level error occurrence are modelled non-deterministically. This means that the transition Retransmit_RstTimer can occur at any time when a TRole_PE is in LISTEN state except if *RCmax* is reached.

2. *The second step is to check whether or not the transaction has terminated. If this is true, the TRole_PE discards any input message, but if not, it continues to Step 3.*

A TRole_PE (except the Merchant_PE) in either COMPLETED or CANCELLED discard any messages that arrive. The Merchant_PE, which has COMPLETED an Offer exchange, will still be able to enter CANCELLED on receiving a Cancel Message from the Consumer_PE (see the RcvCnl_special transition on page Cancellation_NC in Figure 10.31).

On page ErrHandling_C, transition Discard_Clear models discarding any message that arrives at the Consumer_PE after transaction completion or cancellation. On page ErrHandling_NC, two transitions Discard_Clear_P and Discard_Clear_D model similar behaviour at the PHandler_PE and DHandler_PE, respectively. Transition Discard_Clear_M is used for the Merchant_PE. It has a guard indicating that a Cancel Message from the Consumer_PE cannot be discarded after the Merchant_PE has COMPLETED the Offer document exchange (either a BDOfr or a BIOfr). These four transitions guarantee that the communication channel between a Consumer_PE and a non-Consumer_PE is empty after the transaction has terminated.

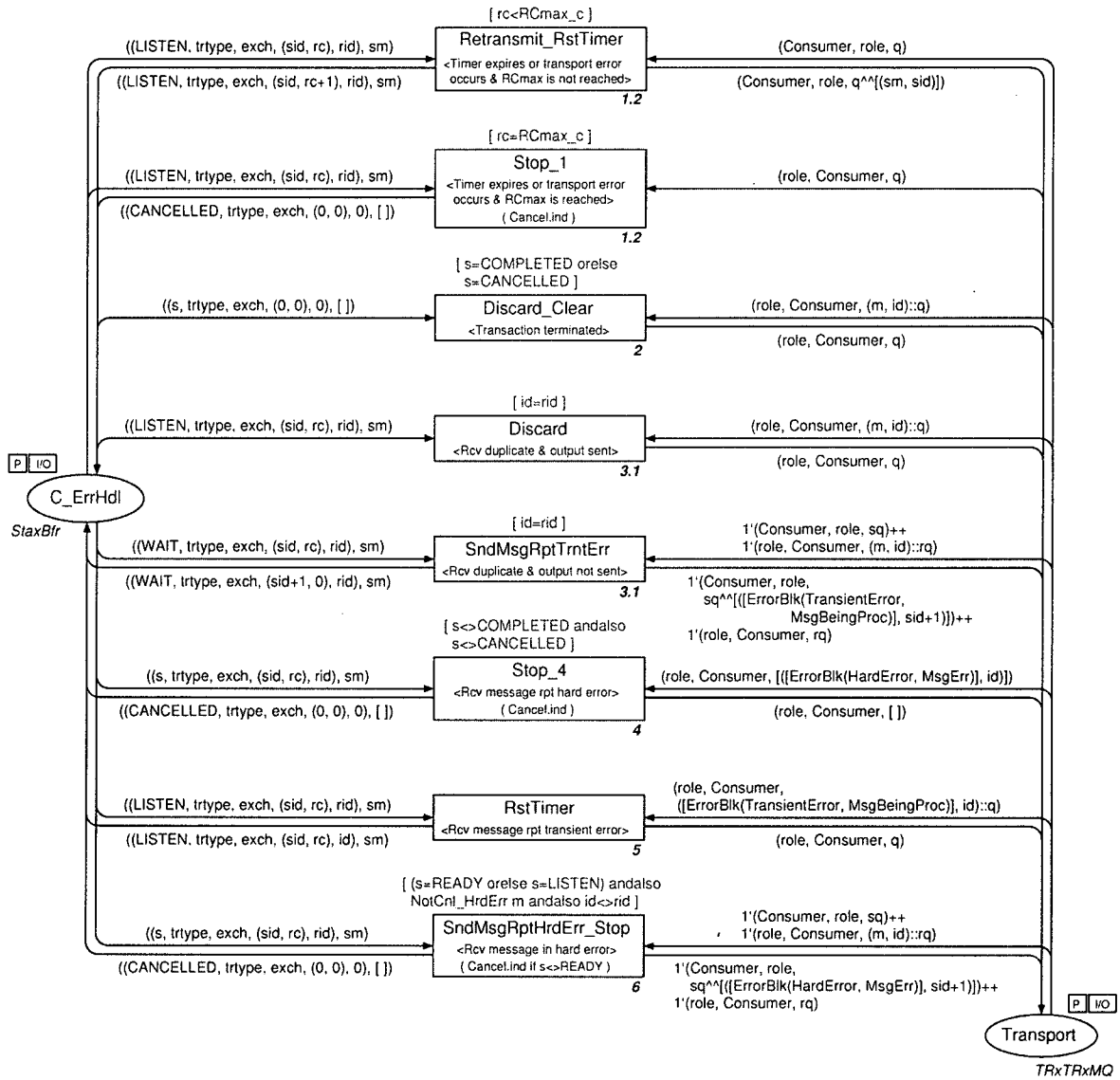


Figure 10.33: ErrHandling_C page in the IOTP CPN model.

3. *The third step is to check whether or not the input message has been received before (i.e., a duplicate). If it has, the TRole_PE carries out the following sub-step, but if not, it continues to Step 4 to handle the non-duplicate message.*

   3.1 *Check whether an output message has been sent. If it has, the TRole_PE discards the duplicate and expects the next input message. Otherwise, if the previous input message is still being processed, the TRole_PE sends a message reporting a transient error of MsgBeingProc to the peer TRole_PE.*

Processing each non-duplicate input message results in an output message at the TRole_PE, except when the input message is the last message in a document exchange. Accordingly, if the output message has already been sent by the TRole_PE, processing of the previous input message must have been complete. In this case, the TRole_PE enters the state `LISTEN`, waiting for the next message from the peer TRole_PE. Otherwise, if the input message is still being processed, the TRole_PE remains in `WAIT` or `HOLD_WAIT` (for a non-Consumer_PE only), waiting for the information from the user required for generating the output message.

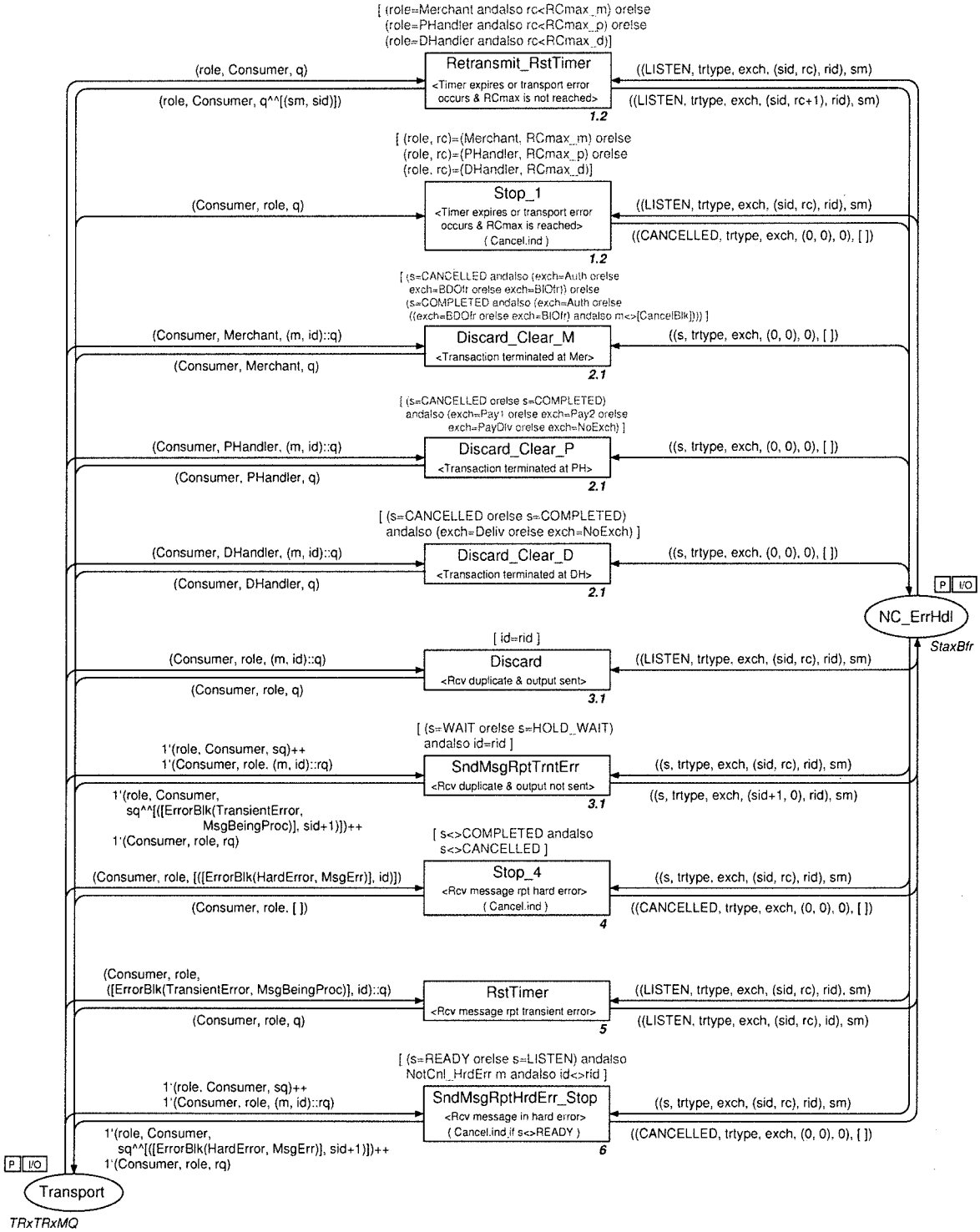


Figure 10.34: ErrHandling_NC page in the IOTP CPN model.

On both ErrHandling pages, the transitions named Discard model the TRole_PE discarding a duplicate message if processing the previous message has completed. This is indicated by the bi-directional arc between Discard and place C_ErrHdl or NC_ErrHdl on each page. A duplicate message is detected when the message identifier of the input message is the same as that of the previously received message (id=rid).

Another two transitions named SndMsgRptTrntErr model the TRole_PE sending to the peer TRole_PE a message reporting a transient error ([ErrorBlk(TransientError, MsgBeingProc)]) if the previous message is still being processed. RFC 2801 states that any output message reporting a transient error should not be saved because "it will be processed afresh when the (new) input message is received" (page 69, [36]). Accordingly, on sending a message reporting a transient error, only the sent message identifier (sid) is updated (to sid+1), which is necessary for calculating the message identifier of the next output message.

4. *The fourth step is to check whether or not the input message is a Cancel Message or a message reporting a hard error. If it is, the TRole_PE cancels any further processing in the transaction, but if not, it continues to Step 5.*

The receipt of a Cancel Message is modelled on the Cancel pages described in Section 10.8.7. On both ErrHandling pages, the two transitions named Stop_4 model that a TRole_PE receives a message reporting a hard error ([ErrorBlk(HardError, MsgErr)]) from the peer TRole_PE. As with the receipt of a Cancel Message, the TRole_PE enters the CANCELLED state and delivers a *Cancel.ind* to the user upon receiving a message reporting a hard error.

5. *The fifth step is to check whether the input message reports a transient error. If it does, the TRole_PE resets the message retransmission timer, but if not, it continues to Step 6.*

The transitions named RstTimer on both ErrHandling pages model the receipt of a message reporting a transient error of MsgBeingProc at a TRole_PE. In this case, the TRole_PE should reset the timer for message retransmission and only re-send the message (represented by sm) once a pre-defined amount of time (by the *MinRetrySecs* attribute of the Error Component) has elapsed. Retransmitting the message sm is modelled by the Retransmit_RstTimer transition on the same page (in Step 1.2).

6. *The last step is to check whether or not the input message contains a hard error. If it does, the TRole_PE sends a message reporting a hard error to the peer TRole_PE and stops any further processing, otherwise, it continues to process the input message.*

When a TRole_PE detects a hard error in the input message, it must send a message reporting a hard error to the peer TRole_PE. The TRole_PE stops any further processing on sending or receiving (in Step 4) a message reporting a hard error. Also, a TRole_PE (except the Merchant_PE) in the initial state READY may detect the message is in hard error after receiving the first IOTP message from the peer TRole_PE. This is not the case for the Merchant_PE, because, when it is READY it initiates a transaction by sending the first IOTP message.

Transitions called SndMsgRptHrdErr_Stop on both ErrHandling pages model the above step. When the Merchant_PE is READY, transition RcvMsgInHrdErr on page ErrHandling_NC cannot be enabled because no IOTP message will arrive at the Merchant_PE. Also, the guard on each transition indicates that the input message does not report a hard error and is not a Cancel Message, as specified by the function NotCnl_HrdErr in Table 10.4.

```
1  fun NotCnl_HrdErr ( [ ErrorBlk ( HardError, MsgErr ) ] ) = false
2    | NotCnl_HrdErr ( [ CancelBlk ] ) = false
3    | NotCnl_HrdErr ( _ ) = true
```

Table 10.4: Function used to check a message reporting a hard error or a Cancel Message.

The function NotCnl_HrdErr is used to check if an input message reports a hard error or is a Cancel Message. It is defined using *pattern matching* in Standard ML, where the argument provided to the function is matched in turn against the patterns and evaluated accordingly. In NotCnl_HrdErr, the only argument is an IOTP message represented as a list of trading blocks. If this message matches the pattern given by a message reporting a hard error ([ErrorBlk(HardError,MsgErr)]) or a Cancel Message ([CancelBlk]), the function returns false. Otherwise it evaluates to true indicating the input message does not report a hard error and is not a Cancel Message.

Finally, upon sending a message reporting a hard error, the TRole_PE, which is in the `LISTEN` state, delivers a *Cancel.ind* to the (local) TRole_User and enters the `CANCELLED` state. Whereas, a TRole_PE in `READY` enters `CANCELLED` without delivering a *Cancel.ind* to the TRole_User which is not aware of the transaction.

Chapter 11

Analysis of Internet Open Trading Protocol Properties

The Coloured Petri Net in Chapter 10 models IOTP as described in RFC 2801 [36] with a certain set of assumptions and simplifications. The state space of the IOTP CPN can be calculated, allowing several properties of IOTP (e.g., transaction termination) to be verified. In this chapter, we analyse the properties of IOTP, following the approach presented in Section 11.1. Section 11.2 defines a set of desired properties of IOTP. In Sections 11.3 to 11.5, we investigate and formally reason about these properties. The deficiencies revealed in the protocol design will be dealt with in Chapter 12.

11.1 Analysis Approach

11.1.1 Transaction Analysis Simplification

Our investigation of IOTP focuses on the six Authentication and Payment-related transactions. The Deposit, Withdrawal and Refund transactions consist of the same set of document exchanges, i.e., a procedure starting with an optional Authentication document exchange, followed by an Offer document exchange (either Brand Dependent Offer or Brand Independent Offer) and then a Payment document exchange. When examining the functional behaviour of these three transactions, we only need to consider one of them (such as a Deposit) as a representative example. The other three transactions, Authentication, Purchase and Value Exchange, are examined separately. Thus we only need to analyse four transactions in total.

IOTP is designed to be payment system-independent. A Payment document exchange may involve payment data exchanges that can occur any finite number of times, catering for various payment systems to be encapsulated by IOTP. In the IOTP CPN model, the message identifier is used for checking duplicates, and is incremented with each payment data exchange. As a result, we obtain infinite state space if the number of payment data exchanges is unbounded. However, in the current design of IOTP [36], a payment data exchange is only used to encapsulate payment system-specific messages (e.g., a SET message) and its occurrence in a (Payment-related) transaction is optional. This allows us to hide any payment data exchange from the analysis of the core behaviour of each Payment-related transaction. This abstraction of hiding payment data exchanges in the analysis of IOTP arose in discussions with Professor Jonathan Billington.

11.1.2 Parameters and Configurations of IOTP

The analysis of each transaction starts with selecting appropriate values for a set of parameters, and thereby defining the configurations of IOTP.

As described in Chapter 10, the IOTP CPN models six Authentication and Payment-related transactions in a modular fashion. According to Simplification 10.1, all six transactions are independent of each other and it is therefore valid to analyse one transaction at a time. For the analysis of a given transaction, the CPN model needs to be initialized for the execution of that transaction *only*. The maximum number of message retransmissions (*RCmax*) is also a parameter in the CPN model. The configuration is specified in the global declarations of the IOTP CPN model (see Appendix E) using the following three parameters:

1. Transaction: The initial marking of place Initialize on page Merchant (Figure 10.5). This parameter indicates the type of the transaction to occur and has one of the six values from the colour set IotpTransType;
2. RCmax_c: The maximum value of the message retransmission counter at an IOTP client (used in Consumer_PE, see page ErrHandling_C in Figure 10.33);
3. RCmax_nc: The maximum value of the retransmission counter at an IOTP server (used in non-Consumer_PE, i.e., RCmax_m for the Merchant_PE, RCmax_p for the PHandler_PE or RCmax_d for the DHandler_PE, see page ErrHandling_NC in Figure 10.34). Note that IOTP defines error handling procedures in terms of clients and servers, and does not specify whether the *RCmax* can be different at different non-Consumer_PE. To simplify the configurations of IOTP, we consider that all IOTP servers have the same maximum value for *RCmax* (i.e., given by RCmax_nc).

For brevity, we may refer to a configuration by the values assigned to the parameters ordered as in the above list [88]. When referring to the value of the Transaction parameter, we only use its first letter. For example, *Config P-1-2* represents a Purchase transaction in which the Consumer_PE cannot resend a message more than once and any non-Consumer_PE cannot resend a message more than twice.

RFC 2801 [36] does not define the maximum number of message retransmissions allowed at a client (operating Consumer_PE) or a server (operating non-Consumer_PE) side. This means that the value of each RCmax parameter (i.e., RCmax_c or RCmax_nc) is unbounded, resulting in an infinite number of possible configurations. In IOTP, the retransmission mechanism is not designed for message loss (because IOTP is designed to operate over a reliable medium), but for recovery from transient errors, transport level errors, or the delay in message delivery causing time-out at the TRole_PE that is waiting for the message. Therefore, we start the analysis with the assumption of no more than one retransmission at each TRole_PE for testing the basic features of IOTP. Accordingly, the RCmax parameters will have a value of either *0* or *1* in an examined configuration.

11.2 Desired Properties of Transactions

We define three properties that IOTP transactions should satisfy: proper termination in terms of absence of undesired terminal states (deadlocks); absence of undesired cycles (livelocks); and absence of procedures that are specified but never executed (dead code).

Property 11.1 (Valid Transaction Termination). *The dead markings of the state space of each transaction represent all valid terminal states of the corresponding transaction.*

The dead markings of the state space of each transaction define the state of the transaction upon termination. If a transaction terminates properly, it must satisfy two conditions:

1. Each of the TRole_PEs that have started the transaction must enter a valid terminal state. In the IOTP CPN model, there are two valid terminal states for a TRole_PE: `COMPLETED` and `CANCELLED`. `COMPLETED` represents that a transaction terminates successfully, while `CANCELLED` is used when the transaction is cancelled; and
2. Each communication queue between two peer TRole_PEs should be empty after the transaction has terminated.

Furthermore, it follows from Property 11.1 that any dead marking that does not represent the valid transaction termination is undesirable, i.e., a deadlock. So, if Property 11.1 is true, we can prove that the state space of the corresponding transaction is deadlock free.

Property 11.2 (Absence of Livelocks). *There must be no livelocks in the state space of each transaction.*

A transaction, which provides meaningful operations and terminates properly, should be free from livelocks. As discussed in Chapter 6, if the state space is isomorphic to the SCC graph, and contains no self-loops, then the state space has no livelocks. If Property 11.2 is true, we can prove that it is always possible for a transaction to terminate in one of the terminal states represented by dead markings.

Property 11.3 (Absence of Unexpected Dead Transitions). *There must be no unexpected dead transitions in the state space of each transaction, unless those transitions model events that are not activated because of the selected initial parameter values.*

A dead transition in the state space of a transaction indicates the protocol operation modelled by that transition never occurs. We expect transitions that model protocol operations that are not activated for a particular set of initial parameter values to be dead. An unexpected dead transition means a protocol operation is redundant or that unexpected behaviour has occurred, resulting in that operation not being utilized.

In the IOTP CPN model, each transaction page is specific to the transaction indicated by its page name (see the Hierarchy page of the model in Figure 10.2). Suppose that the model is executed for an Authentication transaction (by selecting Authentication for the Transaction parameter). Only two transaction pages, AuthTr_C and AuthTr_M, and their subpages, the six exchange level pages, Authenticatee, Authenticator, Cancellation_C, Cancellation_NC, ErrHandling_C and ErrHandling_NC will be executed. The remaining transaction pages and exchange level pages will not be activated, and therefore all the transitions on these pages are expected to be dead. In a similar way, we can see which transaction pages and exchange level pages are activated for each of the other five transactions.

Apart from the above, Table 11.1 lists a number of transitions that are expected to be dead on the activated exchange level pages for each of the six transactions. The parameter value for an expected dead transition is also given. Each transition in Table 11.1 is assigned a unique

number so that it can be referred to using this number. For example, transition HoldAuthSts on page Authenticator is hereafter referred to as transition 1.

No.	*Parameter Value*	*Dead Transition*	*Page*
1	Authentication transaction	HoldAuthSts	Authenticator
2	Payment-related transaction	SndAuthSts	
3	Purchase/Deposit/Withdrawal/Refund transaction	SndPayReq_2	Payment_C
4	Deposit/Withdrawal/Refund transaction	RcvPayRsp_cont	
5	Authentication transaction	LocalCnl	Cancellation_NC
6	Authentication/Deposit/Withdrawal/Refund	SndCnl_special	Cancellation_C
7	transaction	RcvCnl_special	Cancellation_NC
8	Authentication transaction	Discard_Clear_P	
9	Authentication/Deposit/Withdrawal/Refund/ Value Exchange transaction	Discard_Clear_D	ErrHandling_NC
10	RCmax_c=0	Retransmit_RstTimer	ErrHandling_C
11		RstTimer	
12		Discard	ErrHandling_NC
13		SndMsgRptTrntErr	
14	RCmax_nc=0	Retransmit_RstTimer	ErrHandling_NC
15		RstTimer	
16		Discard	ErrHandling_C
17		SndMsgRptTrntErr	

Table 11.1: Expected dead transitions under certain parameter values in the IOTP CPN.

The first two transitions (see Figure 10.19) listed in Table 11.1 model the last event in an Authentication document exchange at the Merchant_PE. An Authentication Status Message must be sent on its own to complete a standalone Authentication transaction, or needs to be held until it is combined with the first message of the following Offer document exchange within any Payment-related transaction. Therefore, transitions 1 and 2 will be dead for an Authentication or a Payment-related transaction, respectively.

Transition 3 (Figure 10.25 (a)) models the Consumer_PE sending a Payment Request Message to start the second Payment document exchange. Hence, it should not occur in any transaction except Value Exchange. On the same page, transitions 4 models the receipt of a Payment Response Message at the Consumer_PE in the first Payment document exchange before the second one starts. A Deposit, Withdrawal or Refund transaction terminates at the end of the first Payment document exchange, in which case, transition 4 cannot occur.

Transitions 5 to 7 (Figures 10.30 and 10.31) model special events of transaction cancellation. Transition 5 will be dead in an Authentication transaction because it can only occur in a Brand Independent Offer document exchange. Transitions 6 and 7 are used between a Payment and a Delivery document exchange in a Purchase transaction or between two Payment document exchanges in a Value Exchange transaction. Hence, they cannot occur in any other transaction.

Transitions 8 to 9 (Figures 10.34) model the clearing of message buffers at a PHandler_PE and at a DHandler_PE, respectively. Both transitions will be dead in an Authentication transaction. Since the DHandler_PE is only involved in a Purchase transaction, transition 9 is not needed in any other transactions.

The rest of the transitions (Figure 10.33 and 10.34) listed in Table 11.1 concern message retransmissions, receipt of duplicate messages and messages reporting transient errors at a

TRole_PE during error handling procedures. When the maximum value RCmax_c is *0*, transition 10 allowing the Consumer_PE to resend the previous message can never occur. Without retransmission, it is not possible for the peer non-Consumer_PE to receive any duplicate messages and thereby to send messages reporting transient error. This results in transitions 12, 13 and 11 being dead. Likewise, transitions 14 to 17 will be dead when RCmax_nc is set to *0*.

11.3 Transaction Termination

One important issue in validating the functional correctness of IOTP is to make sure that a transaction always terminates properly. For example, it should not be possible for a transaction to be cancelled by any TRole_PE after it has completed successfully. RFC 2801 [36] however lacks a complete definition of transaction termination. With the IOTP CPN model, the dead markings of the state space of each transaction represent all possible terminal states of that transaction based on the current design of IOTP in the RFC. In this section, we investigate and formally reason about the termination properties of all six transactions by analysing the set of dead markings calculated from the IOTP CPN model. Earlier work on the analysis of IOTP transaction termination properties was published in [181].

Six configurations of IOTP are used for our analysis. They represent each of the six Authentication and Payment-related transactions and have both RCmax parameters set to a value of *1*, allowing a single retransmission of an IOTP message at each TRole_PE. These are: *Config A-1-1* (for Authentication), *Config P-1-1* (for Purchase), *Config D-1-1* (for Deposit), *Config W-1-1* (for Withdrawal), *Config R-1-1* (for Refund) and *Config V-1-1* (for Value Exchange). Hereafter, we refer to them as Configuration Group 1-1.

The occurrence graphs (OG) are generated for the models with configurations in Group 1-1. The corresponding state space reports are presented in Appendix F. Table 11.2 summarises the statistics of the state space size, the number of dead markings and the generation time, for each configuration. Details of each dead marking will be examined in Section 11.3.1.

Configuration	A-1-1	P-1-1	D-1-1/W-1-1/R-1-1	V-1-1
Nodes	276	3695	2465	2919
Arcs	759	10021	7021	8148
Dead markings	4	37	19	25
Time (sec)	1	28	16	20

Table 11.2: State space size and dead markings for Configuration Group 1-1.

11.3.1 Inspection of Terminal States

By examining each dead marking in the state space of a transaction, we can determine whether the transaction terminates properly. Below, we illustrate our approach to the inspection of all possible terminal states, using the Authentication transaction as a representative example. Details of dead markings for Payment-related transactions and the inspection of the corresponding terminal states are also given.

Authentication Transaction

Figure 11.1 shows the four dead markings in the state space of an Authentication transaction in *Config A-1-1*. Places that have an empty marking are not included. The four dead markings in Figure 11.1 represent all possible terminations of an Authentication transaction. Each TRole_PE enters the COMPLETED or CANCELLED state in the Auth document exchange, except M_{20} where the Consumer_PE is in NoExch. Each communication queue between two TRole_PEs is empty of messages, represented by the empty list [ ].

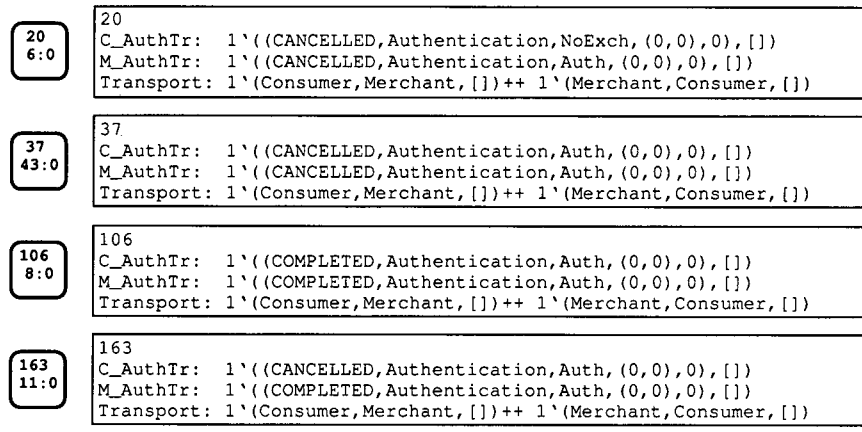


Figure 11.1: The four dead markings of an Authentication transaction in *Config A-1-1*.

The Authentication transaction is completed in M_{106}, whereas it is cancelled in M_{20} and M_{37}. In M_{20}, the transaction is cancelled before an Auth document exchange can be initiated at the Consumer_PE, and therefore the Consumer_PE remains in NoExch. In M_{163}, the transaction is completed at the Merchant_PE but cancelled at the Consumer_PE. Figure 11.2 shows a path leading to M_{163} in the state space of the transaction with *Config A-1-1*. A time sequence diagram of the corresponding occurrence sequence is given in Figure 11.3, where

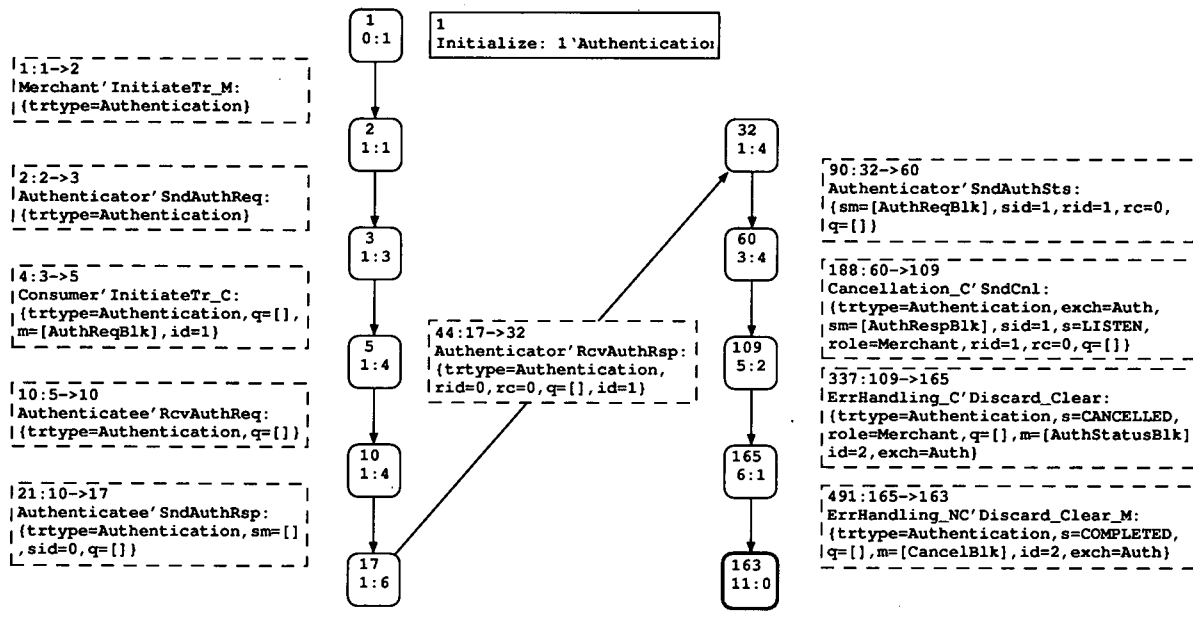


Figure 11.2: Path in the Authentication transaction (*Config A-1-1*) state space leading to M_{163}.

- Upon submission of the *AuthResult.req* primitive from the Merchant_User, the Merchant_PE sends the Authentication Status Message to the Consumer_PE and then enters

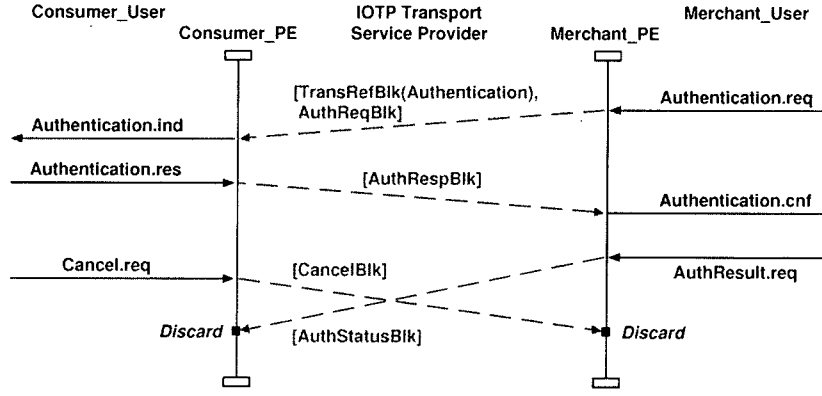


Figure 11.3: Time sequence diagram of the path shown in Figure 11.2.

the COMPLETED state. This indicates the completion of the Authentication transaction at the Merchant_PE, and any message in the transaction (e.g., the Cancel Message) received afterwards will be *discarded*; and

- At the Consumer_PE, the Consumer_User submits a *Cancel.req* after the *Authentication.res* primitive, in which case, the Consumer_PE sends the Cancel Message to the Merchant_PE and then enters the CANCELLED state. This signals the cancellation of the Authentication transaction at the Consumer_PE, which will *discard* any message in the transaction (e.g., the Authentication Status Message) received afterwards.

Although the states of the two TRole_PEs are inconsistent upon transaction termination, M_{163} is expected because IOTP specifies that no trading role can cancel a completed transaction nor resume a cancelled transaction (see Section 3.2.6).

According to the above, we can determine that all four dead markings, M_{20}, M_{37}, M_{106} and M_{163} represent desired terminal states of an Authentication transaction, proving the absence of deadlocks in the transaction with *Config A-1-1*.

Payment-related Transactions

Table 11.3 lists the dead markings of all Payment-related transactions in Configuration Group 1-1, and the corresponding states of each TRole_PE with document exchange information in those dead markings. For brevity, the state values are given in their abbreviation forms: CMP for COMPLETED and CNL for CANCELLED, and the document exchange value is written as a subscript to the state value. A TRole_PE without such information (represented by "-") is not involved, or has not yet started, in the transaction. Details of all dead markings in Table 11.3 are given in Appendix F. Also note that absence of dead markings (represented by "-") under a transaction in Table 11.3 indicates the corresponding termination is not possible for that transaction.

For clarity, we divide the above dead markings into *seven* groups with their group numbers given in the first column of Table 11.3. The dead markings in the first four groups represent the expected terminal markings of the corresponding Payment-related transactions. Since Purchase is the most complicated among the five Payment-related transactions, we shall use it as a representative example for the examination of the dead markings below.

Group 1 represents all possible successful terminations of Payment-related transactions. A successful Deposit (or, Withdrawal or Refund) transaction always terminates at the end

of the (first) Payment document exchange that follows a Brand Dependent Offer or a Brand Independent Offer, respectively, in M_{1065} or M_{538}. A successful Value Exchange transaction terminates at the end of the *second* Payment document exchange in M_{1875} or M_{1272}. A successful Purchase transaction may be completed with 1) the (first) Payment document exchange in M_{1145} or M_{555}; 2) the Payment-and-Delivery document exchange in M_{2144} or M_{1423}; or 3) the Delivery document exchange in M_{2376} or M_{1655}.

Group No.	*Dead marking*			*Trading role protocol entity (TRole_PE)*			
	Purchase	*Deposit/ Withdrawal/ Refund*	*Value Exchange*	*Consumer*	*Merchant*	*PHandler*	*DHandler*
1	*1145, 555*	*1065, 538*	-	CMP_{Pay1}	$CMP_{BDOfr,\ BIOfr}$	CMP_{Pay1}	-
	-	-	*1875, 1272*	CMP_{Pay2}	$CMP_{BDOfr,\ BIOfr}$	CMP_{Pay2}	-
	2144, 1423	-	-	CMP_{PayDlv}	$CMP_{BDOfr,\ BIOfr}$	CMP_{PayDlv}	-
	2376, 1655	-	-	CMP_{Deliv}	$CMP_{BDOfr,\ BIOfr}$	CMP_{Pay1}	CMP_{Deliv}
2	*7*	*7*	*7*	-	CNL_{BIOfr}	-	-
3	*36*	*36*	*36*	CNL_{NoExch}	CNL_{Auth}	-	-
	42	*42*	*42*	CNL_{NoExch}	CNL_{BDOfr}	-	-
	65	*65*	*65*	CNL_{Auth}	CNL_{Auth}	-	-
	80	*80*	*80*	CNL_{BDOfr}	CNL_{BDOfr}	-	-
	152	*151*	*151*	CNL_{BIOfr}	CNL_{BIOfr}	-	-
	472, 453	*458, 439*	*458, 439*	CNL_{Auth}	$CNL_{BDOfr,\ BIOfr}$	-	-
4	*1626, 954*	-	*1888, 1285*	CNL_{Pay1}	$CNL_{BDOfr,\ BIOfr}$	CMP_{Pay1}	-
5	*929, 393*	*878, 385*	*1545, 385*	CNL_{Pay1}	$CMP_{BDOfr,\ BIOfr}$	CNL_{NoExch}	-
	1156, 566	-	-	CNL_{PayDlv}	$CMP_{BDOfr,\ BIOfr}$	CNL_{NoExch}	-
	1142, 552	*1062, 535*	*1068, 535*	CNL_{Pay1}	$CMP_{BDOfr,\ BIOfr}$	CNL_{Pay1}	-
	-	-	*1884, 1281*	CNL_{Pay2}	$CMP_{BDOfr,\ BIOfr}$	CNL_{Pay2}	-
	1638, 966	-	-	CNL_{PayDlv}	$CMP_{BDOfr,\ BIOfr}$	CNL_{PayDlv}	-
	2121, 1400	-	-	CNL_{Deliv}	$CMP_{BDOfr,\ BIOfr}$	CMP_{Pay1}	CNL_{NoExch}
	2374, 1653	-	-	CNL_{Deliv}	$CMP_{BDOfr,\ BIOfr}$	CMP_{Pay1}	CNL_{Deliv}
6	*47*	*47*	*47*	CNL_{NoExch}	CMP_{BIOfr}	-	-
	619	*592*	*592*	CNL_{Auth}	CMP_{BIOfr}	-	-
	382	*375*	*375*	CNL_{BDOfr}	CMP_{BDOfr}	-	-
7	*1378, 748*	*1252, 713*	*1260, 714*	CNL_{Pay1}	$CMP_{BDOfr,\ BIOfr}$	CMP_{Pay1}	-
	-	-	*2066, 1482*	CNL_{Pay2}	$CMP_{BDOfr,\ BIOfr}$	CMP_{Pay2}	-
	2146, 1425	-	-	CNL_{PayDlv}	$CMP_{BDOfr,\ BIOfr}$	CMP_{PayDlv}	-
	2608, 1916	-	-	CNL_{Deliv}	$CMP_{BDOfr,\ BIOfr}$	CMP_{Pay1}	CMP_{Deliv}

Table 11.3: Dead markings for Payment-related transactions in Configuration Group 1-1.

Group 2 contains M_7 that indicates a Payment-related transaction being cancelled during a Brand Independent Offer at the Merchant_PE *only*. Figure 11.4 shows a path leading to M_7 in the state space of the transaction with *Config P-1-1*, and the corresponding time sequence diagram is given in Figure 11.5. The Purchase transaction scenario shown in Figures 11.4 and 11.5 starts with a Brand Independent Offer document exchange. The Merchant_PE creates the Trading Protocol Options (TPO) Message when the local user submits a *TradingOptions.req*. The TPO Message cannot be sent on its own but needs to be combined with the Offer Response Message as a result of a *TradingResult.req* that is expected to be submitted next. However, the Merchant_User cancels before submitting the *TradingResult.req*. No IOTP message has been sent by the Merchant to the Consumer and hence there is no point in sending a Cancel Message. As a result, the Merchant_PE deletes the TPO Message and enters the `CANCELLED` state.

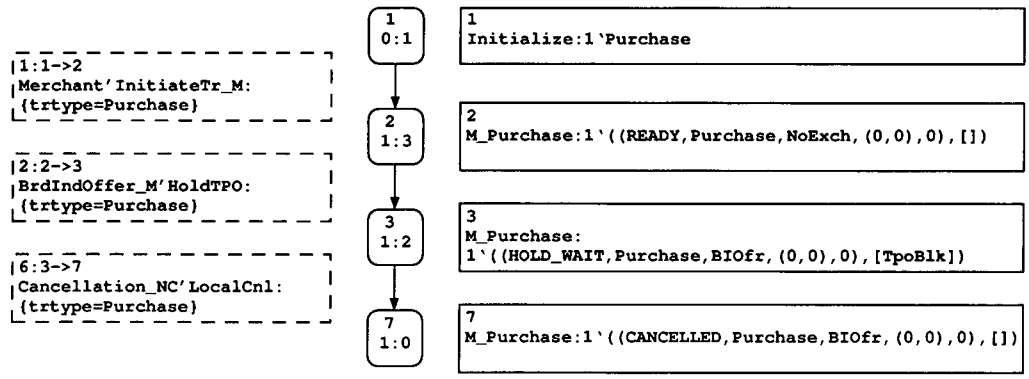


Figure 11.4: Path in the Purchase transaction (*Config P-1-1*) state space leading to M_7.

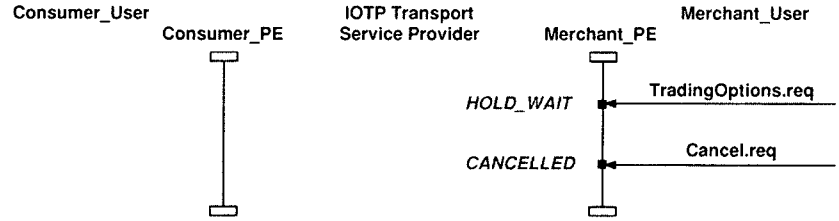


Figure 11.5: Time sequence diagram of the path shown in Figure 11.4.

Group 3 corresponds to the Payment-related transactions being cancelled between the Consumer and Merchant, after the Merchant has initiated an Authentication or Offer document exchange but before the Consumer starts a Payment document exchange. In M_{36} and M_{42}, the transaction is cancelled before the Consumer_PE initiates an Authentication or a Brand Dependent Offer exchange at the beginning of the transaction. In M_{65}, the transaction is cancelled during an Authentication document exchange. In M_{80} and M_{152} (or M_{151}), the transactions are cancelled during an Offer document exchange. In M_{472} (or M_{458}) and M_{453} (or M_{439}), the transactions are cancelled within an Authentication document exchange at the Consumer_PE, whereas they are cancelled within an Offer document exchange at the Merchant_PE. These two markings are expected because of the combination of the Authentication Status Message with the first message in the Offer document exchange at the Merchant_PE. Figure 11.6 shows a path leading to M_{472} in the state space of the transaction with *Config P-1-1*, and the corresponding time sequence diagram is given in Figure 11.7. In the scenario shown in Figures 11.6 and 11.7, the Consumer_PE receives a *Cancel.req* issued by the local user before the receipt of the combined Authentication Status & TPO Message from the Merchant_PE, and therefore still remains in the Authentication document exchange once the transaction has been cancelled.

Group 4 represents the cancellation of a Purchase or Value Exchange transaction between the Consumer and Merchant, after the first Payment document exchange has completed but before the Delivery document exchange (Purchase) or the second Payment document exchange (Value Exchange) starts. This always occurs when the Consumer_User submits a *Cancel.req* at that point. As expected (see Section 3.2.6), the Consumer_PE generates and sends the Cancel Message to the Merchant_PE but not the PHandler_PE. The Merchant_PE enters state `CANCELLED` and delivers a *Cancel.ind* to the Merchant_User,

whereas the PHandler_PE remains in the COMPLETED state. In this case, the Merchant is notified of a cancellation issued by the Consumer, whereas the Payment Handler is not informed because the cancellation occurs after the completion of the Payment document exchange. Since the Merchant is aware of the cancellation once it occurs, the Consumer would be able to receive a refund by initiating a Refund transaction. This could be a refund of the payment for the goods/services to be purchased (Purchase) or for the equivalent value in another currency to be exchanged (Value Exchange).

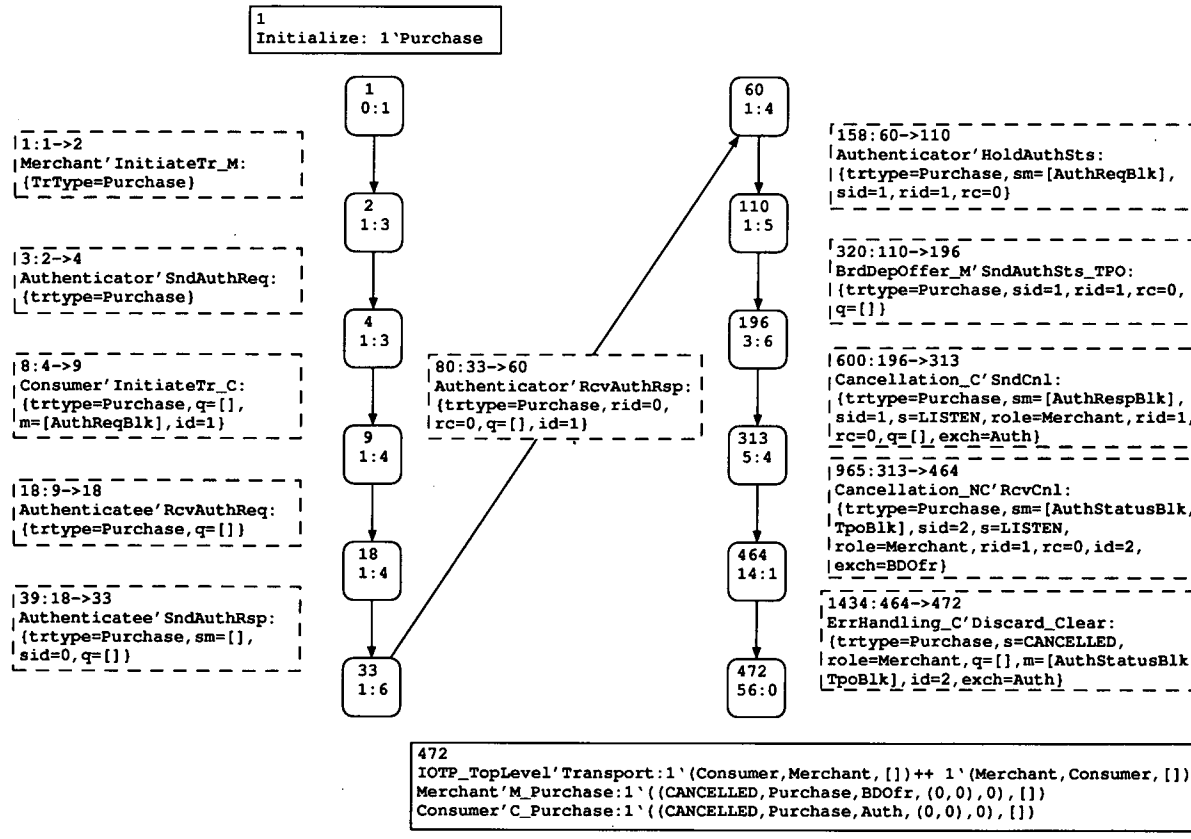


Figure 11.6: Path in the Purchase transaction (*Config P-1-1*) state space leading to M_{472}.

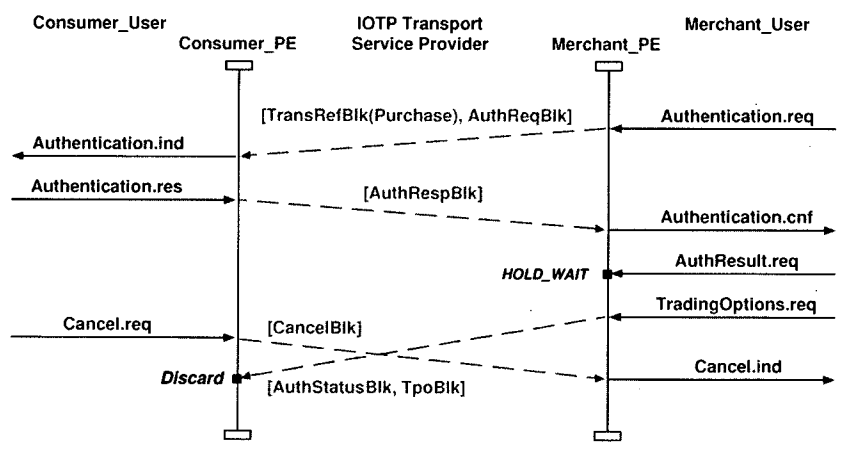


Figure 11.7: Time sequence diagram of the path shown in Figure 11.6.

The dead markings in *Groups 5* to *7* in Table 11.3 reveal deficiencies in the current design of IOTP [36]. We discuss in detail these three groups of dead markings in the next two subsections.

11.3.2 Problem 1: Merchant Not Informed of Cancelled Transaction

In Table 11.3, the dead markings in ***Group 5*** represent the transaction being cancelled *during* the first or the second Payment, the Payment-and-Delivery, or the Delivery document exchange. The Merchant is not aware of such transaction cancellation that only occurs between the Consumer and the Payment Handler or Delivery Handler. These dead markings are expected according to the definition of IOTP in the RFC. Closer inspection however reveals deficiencies related to the corresponding transaction terminations.

Purchase Transaction

We start by analysing the cancellations of a Purchase transaction (*Config P-1-1*) represented by 12 dead markings in Group 5.

Firstly, the transaction is cancelled at the Consumer_PE and PHandler_PE: 1) after the Consumer_PE initiates a Payment or a Payment-and-Delivery document exchange but before it starts at the PHandler_PE in M_{929} and M_{393} (for Pay1) or in M_{1156} and M_{566} (for PayDlv), or 2) after the Payment or the Payment-and-Delivery document exchange has been initiated at both TRole_PEs in M_{1142} and M_{552} (for Pay1) or in M_{1638} and M_{966} (for PayDlv). Similarly, the transaction cancellation may occur at the Consumer_PE and DHandler_PE: 1) after the Consumer_PE initiates a Delivery document exchange but before it starts at the DHandler_PE in M_{2121} and M_{1400}, or 2) after the Delivery document exchange has been initiated at both TRole_PEs in M_{2374} and M_{1653}. In each of these cases, the Merchant may have received the goods from the manufacturer but *unexpectedly* the Consumer does not purchase the goods. Since the Merchant is not informed of the transaction cancellation between the Consumer and Payment Handler, the Merchant still expects the purchase to proceed. This may result in an *inventory problem*, i.e, goods building up in the merchant's warehouse.

Secondly, considering M_{2121}, M_{1400}, M_{2374} and M_{1653} further, neither the Merchant nor the Payment Handler is notified of the cancellation between the Consumer and Delivery Handler. In this case, the Consumer will not receive the goods that was paid for, the Merchant will expect the goods to be delivered, and the Payment Handler will believe that the payment is finalised. This situation may be recovered from the Consumer invoking a Refund transaction where evidence of the payment (e.g., the payment receipt) is provided to the Merchant. However, the Merchant will need to check (somehow) that the goods were not delivered to that Consumer. If the Consumer does not request a refund, then the inventory problem remains.

Other Payment-related Transactions

In IOTP's terminology, the Deposit, Withdrawal and Value Exchange transactions are also referred to as *financial transactions*. Their terminations represented by the corresponding dead markings in Group 5 therefore do not cause an inventory problem. In each financial transaction, the financial institution (such as a bank or a credit card company) takes on the a role of Merchant and is responsible for offering a list of payment methods available and optionally charging fees for transaction processing. All the electronic cash or value for deposit, withdrawal or exchange, together with fee charges if needed, take place between the Consumer and Payment Handler during the Payment document exchange(s).

- In M_{878} (or M_{1545}) and M_{385}, a financial transaction is cancelled before the first Payment document exchange is initiated at the PHandler_PE. In M_{1062} (or M_{1068}) and M_{535}, transaction cancellation occurs during the first Payment document exchange. In both cases, the Merchant is not informed of cancellation. The only consequence is that the transaction record at the Merchant side would not be updated unless the Merchant checks with the Payment Handler at some later point.

- As indicated by M_{1884} or M_{1281}, a Value Exchange transaction may also be cancelled during the second Payment document exchange. In this case, the Consumer has completed the first payment, which could be the upload of a certain amount of electronic value in one currency, but will not receive the second payment, which could be the download of the equivalent value in another currency. Also, this situation may be recovered by the Consumer invoking a Refund transaction with the Merchant, where the Consumer is required to provide evidence of the first payment and the Merchant needs to check that the second payment was not yet made to that Consumer.

Finally, in a Refund transaction that terminates in M_{878} and M_{385}, or M_{1062} and M_{535}, the Consumer is supposed to receive the refund after negotiating with the Merchant in the Offer document exchange, but unfortunately cannot receive the payment from the Payment Handler. Due to this failure, the Consumer needs to re-invoke the Refund transaction with the Merchant. Again, the Merchant needs to check that the payment for the refund was not yet made to that Consumer.

11.3.3 Problem 2: Solely Cancelled Transaction by Consumer

In Table 11.3, the dead markings in the last two groups, ***Group 6*** and ***7***, represent those *undesired* terminations of Payment-related transactions where transaction cancellation is evident at the Consumer side *only*. Below, we examine in detail these two groups of dead markings.

- In Group 6, the transaction involves just two trading roles, the Consumer and Merchant, when it is cancelled at the Consumer side only. A Payment-related transaction, which ends in one of the four terminal states represented by M_{47}, M_{619}, M_{382} and M_{2628}, does no harm to the Consumer since no payment has yet been made, but it may still result in inventory problem as with a Purchase transaction. Further, IOTP specifies that the Merchant should still be able to handle the Cancel Message received from the Consumer even after the Offer document exchange has completed (e.g., the scenario shown in Figures 11.6 and 11.7). With such a specification, we expect that the Consumer and Merchant should be in consistent terminal states, i.e., either both in `COMPLETED` or both in `CANCELLED`, *before* the Payment document exchange starts.

- In Group 7, the transaction is cancelled at the Consumer side only, after the completion of at least one payment. A Payment-related transaction, which ends in one of the terminal states represented by the dead markings in Group 7, results in the possibility of the Consumer not being able to obtain a refund. This is because all the non-Consumer trading roles (especially the Merchant) that are involved in the transaction are in one consistent terminal state `COMPLETED` indicating the successful completion of the transaction.

Example 1: M_{619}

Considering an example of a Purchase transaction that terminates in M_{619}, we illustrate the reason why cancellation is only evident at the Consumer. Figure 11.8 shows two paths leading to M_{619} in the state space of the transaction with *Config P-1-1*. The corresponding time sequence diagrams of these two paths are given in Figures 11.9 (a) and (b).

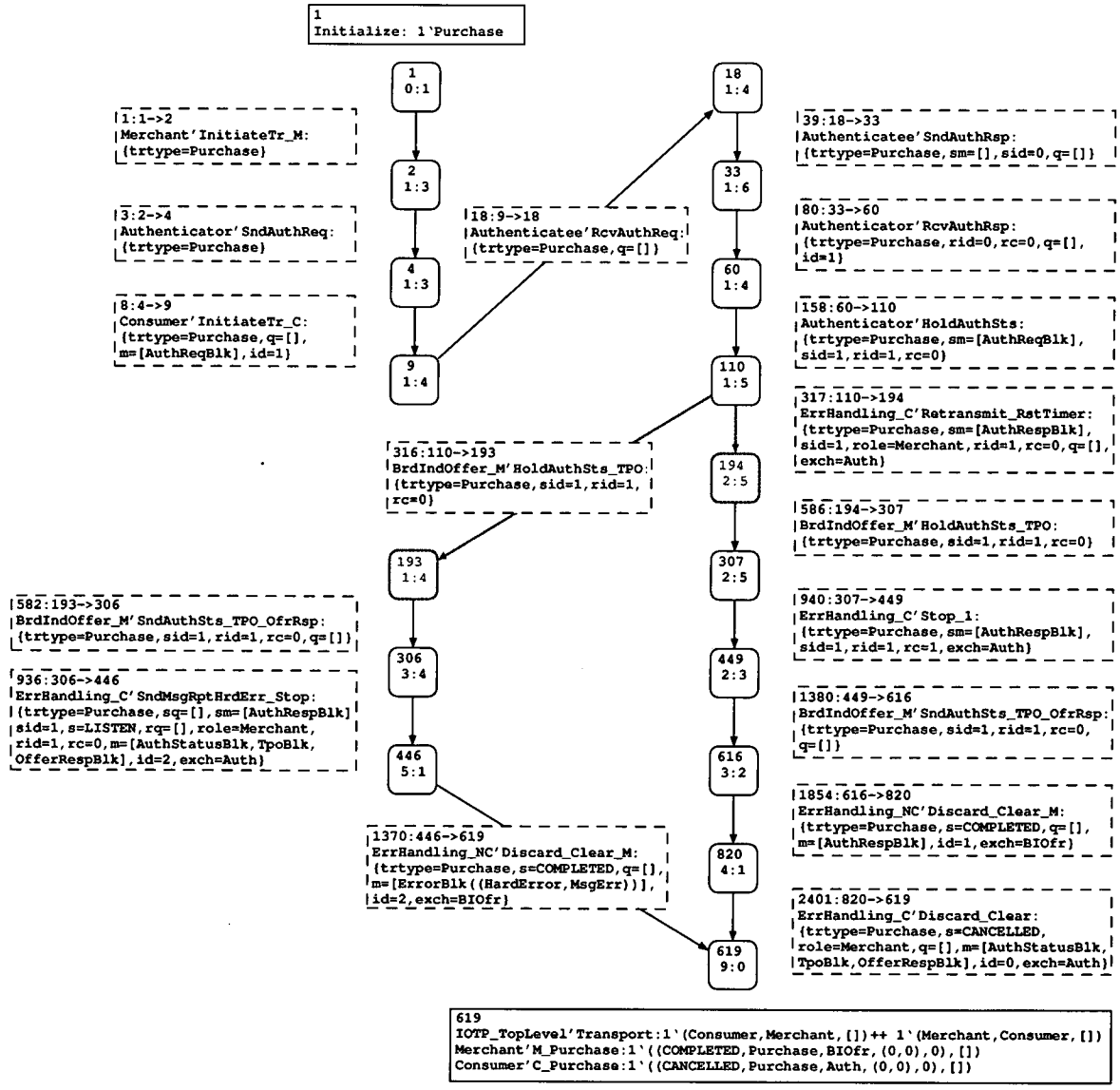


Figure 11.8: Two paths in the Purchase transaction (*Config P-1-1*) state space leading to M_{619}.

The Purchase transaction scenario shown in Figure 11.9 (a) relates to the path involving M_{110} followed by M_{193} on the left part of Figure 11.8. It reveals that the Merchant_PE discards an Error Message reporting a hard error after having completed the Brand Independent Offer document exchange. This is concerned with the error handling procedure defined for IOTP servers. A non-Consumer TRole_PE discards any message reporting an error, after having sent the last message in the document exchange (see Section 3.2.7).

The Purchase transaction scenario shown in Figure 11.9 (b) corresponds to the path involving M_{110} followed by M_{194} on the right part of Figure 11.8. It can be observed that the cancellation occurring at the Consumer is due to the retransmission timer expiring twice before receiving the trading result from the Merchant. In this scenario, the Consumer re-sends the

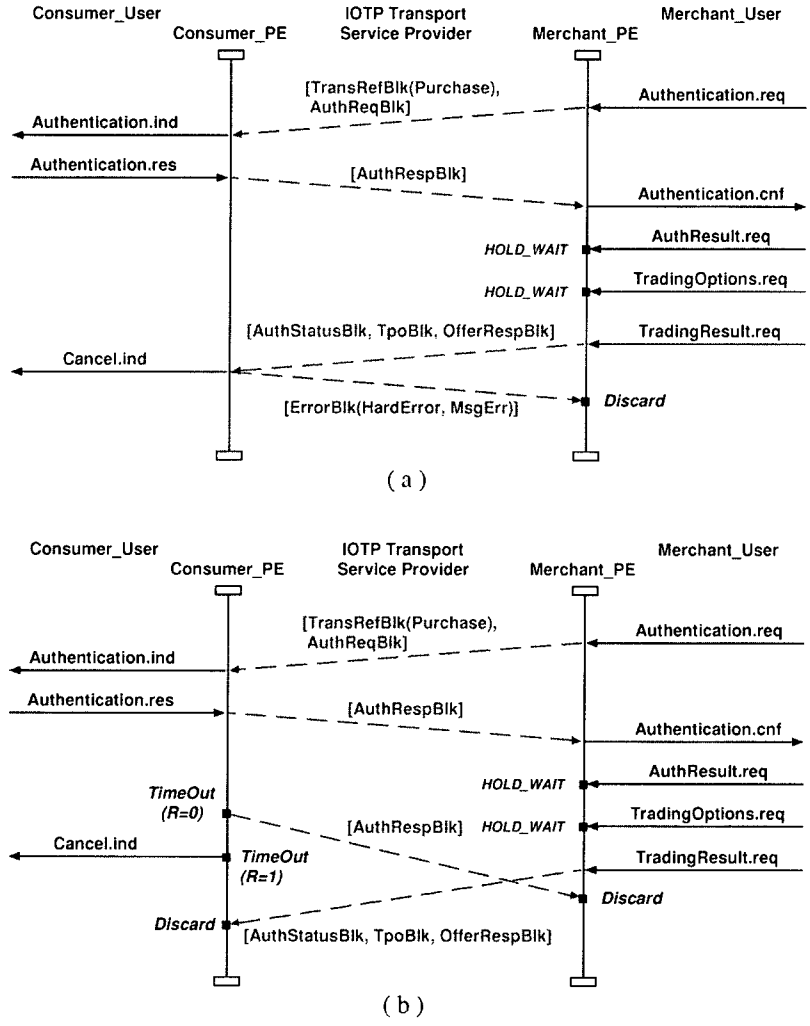


Figure 11.9: Time sequence diagrams of the two paths shown in Figure 11.8: (a) the left path involving M_{110} followed by M_{193} and (b) the right path involving M_{110} followed by M_{194}.

Authentication Response Message upon the first time-out, and cancels the transaction upon the second one given that at most one retransmission is allowed in *Config P-1-1*. Since the Merchant_PE is not aware of this transaction cancellation, it processes the transaction as usual and sends the trading result to the Consumer_PE, completing the Brand Independent Offer following the Authentication document exchange. The Authentication Response Message retransmitted to the Merchant_PE is discarded as a duplicate; and the combined Authentication Status, TPO & Offer Response Message sent to the Consumer_PE is discarded because the Consumer_PE already cancelled the transaction.

Based on the scenarios shown in Figures 11.8 and 11.9, it can be inferred that a Payment-related transaction may end in one of the terminal states represented by the dead markings in Group 7, under one of the following two circumstances:

1. The Consumer_PE sends to the PHandler_PE or DHandler_PE an Error Message reporting a hard error on receiving the last response message in a Payment, a Payment-and-Delivery, or a Delivery document exchange; and

2. The timer has expired and the message retransmission counter has reached its maximum number at the Consumer_PE that is waiting for the last response message from the PHandler_PE or DHandler_PE in the current document exchange.

Example 2: M_{748}

Figure 11.10 shows a path leading to M_{748} in the state space of the transaction with *Config P-1-1*, and Figure 11.11 depicts the corresponding time sequence diagram. The scenario in Figures 11.10 and 11.11 is a representative example of another circumstance under which one of the dead markings in Group 7 can be reached. That is:

3. The Consumer_PE sends to the PHandler_PE or DHandler_PE a Cancel Message (as a result of a *Cancel.req* submitted by the Consumer_User), before receiving the last response message which was already transmitted in the current document exchange.

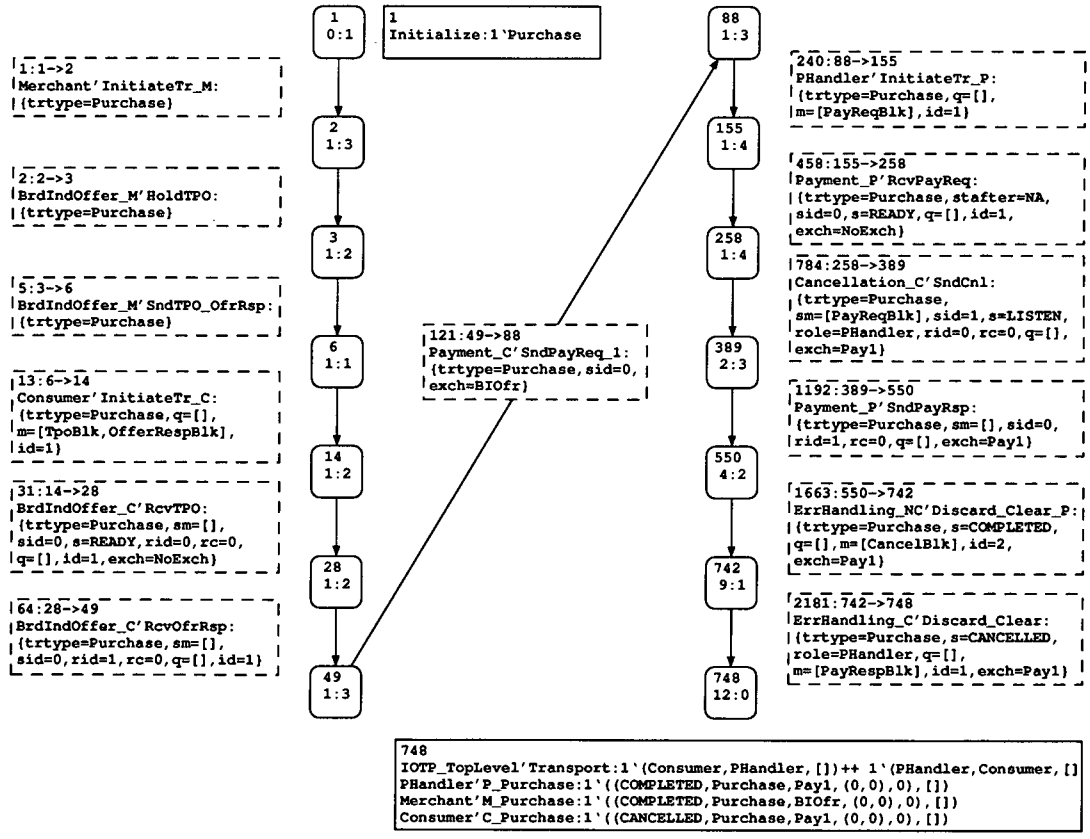


Figure 11.10: Path in the Purchase transaction (*Config P-1-1*) state space leading to M_{748}.

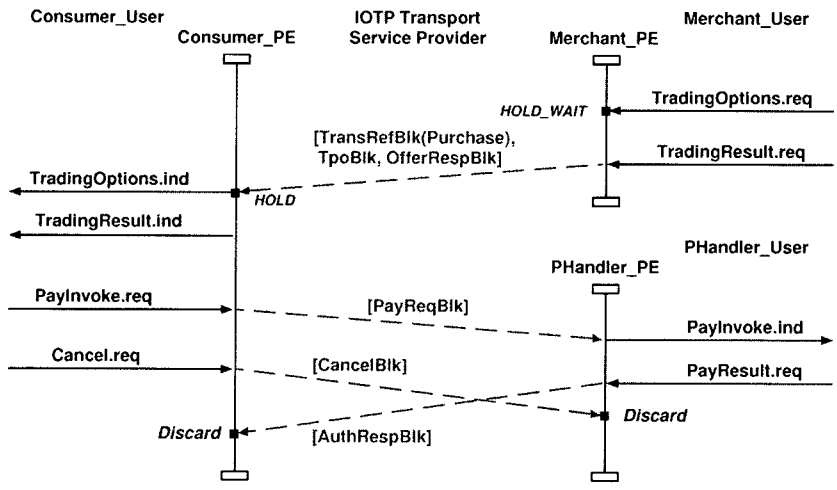


Figure 11.11: Time sequence diagram of the path shown in Figure 11.10.

11.4 Livelocks

The SCC graphs are also generated for the models with configurations in Group 1-1. Table 11.4 lists their statistical information. The number of SCC nodes and arcs and the time used for calculating the SCC graphs are given in the standard state space reports (see Appendix F). The terminal SCCs are obtained using the function ListTerminalSCCs as defined in Appendix F.

Configuration	A-1-1	P-1-1	D-1-1/W-1-1/R-1-1	V-1-1
SCC Nodes	276	3695	2465	2919
SCC Arcs	759	10021	7021	8148
Terminal SCCs	4	37	19	25
Time (sec)	0	1	1	1

Table 11.4: Statistics of SCC graphs for Configuration Group 1-1.

In Configuration Group 1-1, each transaction state space is isomorphic to the SCC graph and contains no self-loops. This can be obtained by comparing Tables 11.2 and 11.4 to see that each transaction state space has the same number of nodes and arcs as its SCC graph. Therefore, the absence of livelocks is proved.

11.5 Dead Transitions

The state space reports (see Appendix F) also include dead transitions in the CPN model. Table 11.5 lists all dead transitions in the activated CPN page instances for each transaction in Configuration Group-1-1. According to Table 11.1, the dead transitions shown in Table 11.5 are expected. The other dead transitions that are not shown in Table 11.1 are also expected, since they are the transitions on those CPN page instances which are never executed in a certain transaction.

Configuration	*Dead Transitions (in the activated CPN page instances)*
A-1-1	Authenticator'HoldAuthSts 1 Cancellation_C'SndCnl_special 1 Cancellation_NC'LocalCnl 1 Cancellation_NC'RcvCnl_special 1 ErrHandling_NC'Discard_Clear_D 1 ErrHandling_NC'Discard_Clear_P 1
P-1-1	Authenticator'SndAuthSts 2 Payment_C'SndPayReq_2 1
D-1-1/W-1-1/R-1-1	Authenticator'SndAuthSts 1 Cancellation_C'SndCnl_special 1 ErrHandling_NC'Discard_Clear_D 1 Payment_C'RcvPayRsp_cont 1 Payment_C'SndPayReq_2 1
V-1-1	Authenticator'SndAuthSts 4 ErrHandling_NC'Discard_Clear_D 4

Table 11.5: Dead transitions in the activated CPN page instances for Configuration Group 1-1.

Chapter 12

Verification Against the Service Specification

A major interest in the verification of IOTP is to see whether IOTP implements the IOTS defined in Chapters 7 to 9. The state space of the IOTP CPN presented in Chapter 11 is used to generate the IOTP language. By comparing this protocol language with the service language of IOTP, we can determine if IOTP provides the required service. Following the approach described in Section 12.1, Sections 12.2 to 12.4 concern the verification of IOTP transactions that are successful and error-free. Section 12.5 focuses on the verification of IOTP transactions that include both error-handling and transaction cancellation. Earlier work on the verification of successful and error-free transactions was published in [177].

12.1 Verification Approach

To handle the inherent complexity of IOTP, an incremental approach is used for its language analysis. This consists of two steps. Firstly, we consider each of the six Authentication and Payment-related transactions under the assumption that they are successful (i.e., without cancellation) and free of *technical* errors. For brevity, we refer to them as *successful transactions.* Secondly, we take into account both transaction cancellation and error handling events. Transactions studied in this step are referred to as *transactions with cancellation.*

It is important to make sure that the protocol and the service are within the same scope for verification. IOTS defines six Authentication phases between four trading roles. The IOTP examined in this thesis only considers the one where the Merchant authenticates the Consumer at the beginning of a transaction (Restriction 10.2). Hence, the other five Authentication phases in IOTS will not be included in the verification of IOTP. Also, authentication retry, payment retry and payment data exchange services are beyond the scope of this work.

Taking advantage of the modular structure of both the IOTS and the IOTP CPNs, the service and protocol languages for each Authentication and Payment-related transaction are obtained following the procedure given in Section 9.1.2, and are represented by minimal FSA. The IOTS CPN is built from a set of modules that specify phases in IOTS. The modules, which model the five *Authentication* phases that are outside our scope, are not used. By disabling the module for the *Cancel* phase, the state space of the successful transaction service can be calculated. The IOTP CPN contains modules for transaction cancellation and error handling, which are disabled for calculating the state space of successful transactions. The details of generating the languages for the verification of IOTP are given in Appendix G.

A comparison between the service and protocol languages determines whether each of the six transactions defined in IOTP conform to the corresponding transaction service specified in IOTS. FSM [11] is used to perform the language comparison between IOTP and IOTS, and the original output data from FSM is included in Appendix G. A protocol is a faithful *refinement* of its service, if the protocol and service languages are equivalent, or if the protocol language is an *acceptable* (or, a valid) subset of the service language. We use the term "missing sequences in the protocol" to refer to sequences in the service but not in the protocol, and the term "illegal sequences in the protocol" for sequences in the protocol but not in the service. Any inconsistency identified in the language comparison aids in further improvement of IOTP, which we shall discuss in detail in Chapter 13.

Finally, due to the current abstraction level in modelling IOTS and IOTP, *Deposit*, *Withdrawal* and *Refund* transactions have the same transaction procedure. Hence, it is sufficient to consider just one of these three transactions.

12.2 Language Comparison for Successful Transactions

Without (technical) error handling, configurations for message retransmission are not considered in the verification of successful transactions. In this section we analyse in detail the results of the language comparison for successful transactions, which forms the basis for their verification in Section 12.4.

12.2.1 Authentication Transaction

Figure 12.1 shows the FSA specifying the service language for a successful Authentication transaction. The protocol language for the transaction is represented by the same FSA. Hence, the service and the protocol languages of a successful Authentication transaction are *identical*, comprising only one sequence of primitives (*Areq Aind Ares Acnf ARSreq ARSind*).

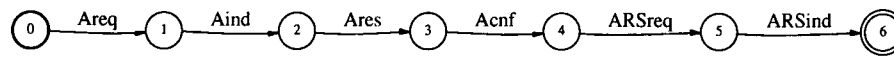


Figure 12.1: The FSA for the service language of a successful Authentication transaction.

12.2.2 Deposit Transaction

Figure 12.2 depicts three FSA specifying the languages obtained for the verification of a successful Deposit transaction. FSA-(a) and FSA-(b) represent the service language and the protocol language for the transaction, respectively. Comparison of the two languages using FSM [11] shows that the protocol language is a subset of the service language. The missing sequences in the protocol are represented by FSA-(c). Below, we examine in detail the sequences accepted by these three FSA.

Service Phases

According to the IOTS definition (in Chapter 7), each trading transaction service is provided via a set of phases. A Deposit comprises an optional Authentication, an Offer and a Payment phase. We use FSA-(a) as an example to describe how the three (or two) phases of a Deposit

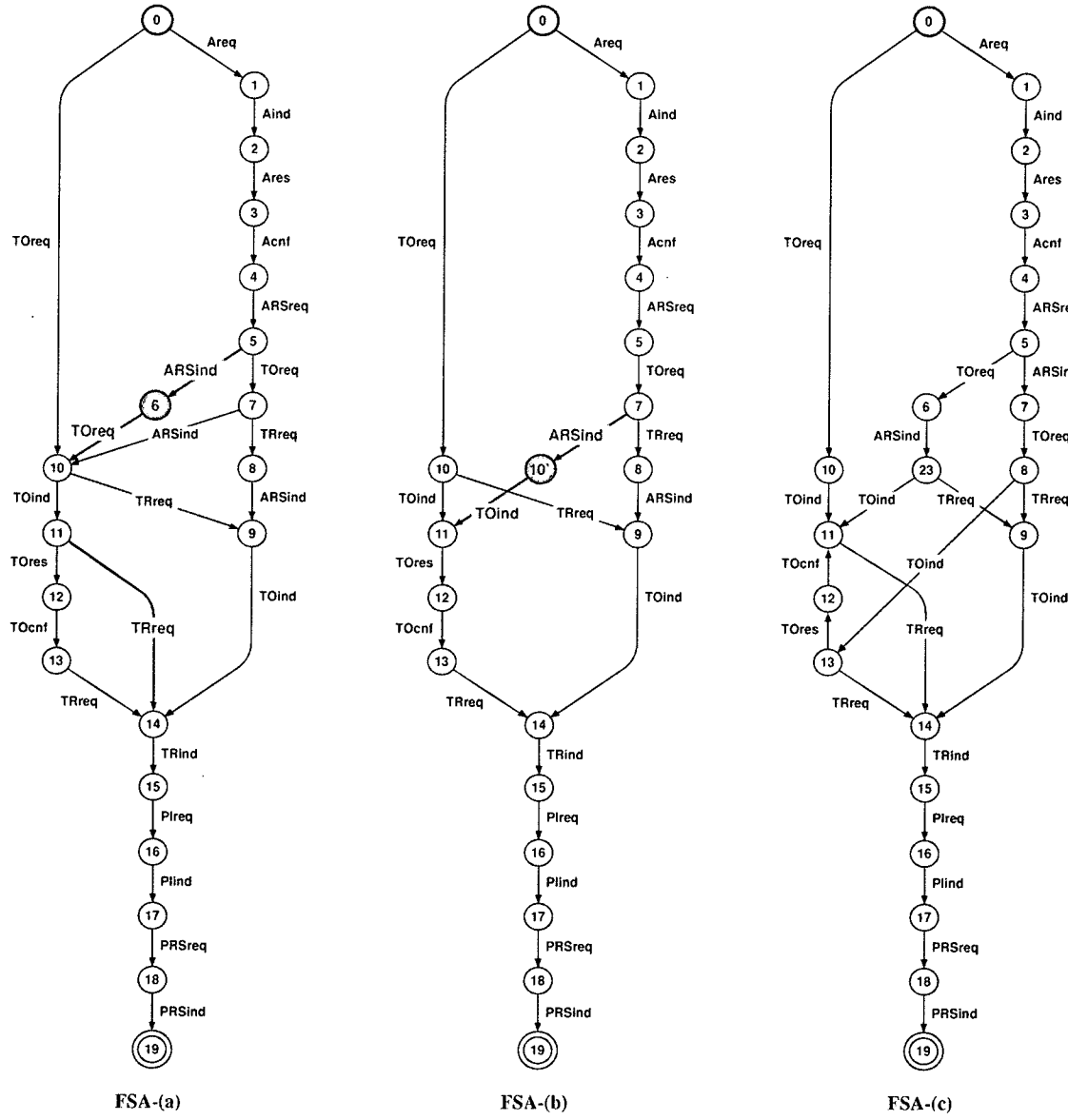


Figure 12.2: Three FSA for a successful Deposit transaction specifying (a) the service language, (b) the protocol language and (c) the missing sequences in the protocol.

are captured in its service language. In the following the word "subsequence" means the subsequence of primitive occurrences.

- The subsequences defined by nodes (0,10,11-15), (0,10,11,14,15) and (0,10,9,14,15) represent the Offer phase at the beginning of the Deposit without Authentication.
- The subsequence defined by nodes 0 to 6 represents the Authentication phase and is followed by the Offer phase subsequences defined by nodes (6,10,11-15), (6,10,11,14,15) and (6,10,9,14,15).
- Each subsequence, starting from nodes 0 to 5 followed by nodes (5,7-9,14,15), (5,7,10,11-15), (5,7,10,11,14,15) or (7,10,9,14,15), represents a overlap between the Authentication and Offer phases, where the primitives *ARSind* (in Authentication), *TOreq* and *TRreq* (*TradingOptions.req* and *TradingResult.req* in Offer) can be interleaved.
- Each subsequence from nodes 15 to 19 represents the Payment phase, which completes the Deposit transaction.

Missing Sequences in the Protocol

Considering Figure 12.2, we can derive FSA-(b) from FSA-(a) by: 1) eliminating node **6** and its associated arcs; 2) introducing a new node **10'**, which duplicates node 10, except that it does not have *TRreq* as an output nor *TOreq* as an input; and 3) removing the arc labelled *TRreq* between nodes 11 and 14 and the arc *ARSind* between node 7 and 10.

Closer investigation of each of the missing sequences shows that they cannot be implemented by IOTP due to the restrictions on the procedures for sending IOTP messages given in RFC 2801. Figure 12.3 depicts two scenarios of how the Consumer_PE and Merchant_PE communicate during an Offer phase when the Merchant_User does not require the Consumer_User to reply with his/her trading selection. The scenario in Figure 12.3 (a) is implemented by a Brand Independent Offer exchange in IOTP, where the RFC specifies that the Merchant_PE must send a combined *TPO & Offer Response Message* after receiving a *TradingOptions.req* (*TOreq*) followed by a *TradingResult.req* (*TRreq*) from the Merchant_User. On receipt of this combined message, the Consumer_PE delivers a *TradingOptions.ind* (*TOind*) and then a *TradingResult.ind* (*TRind*) to the Consumer_User. The subsequence defined by nodes (0,10,9,14,15) in both FSA-(a) and FSA-(b) in Figure 12.2 captures this scenario. In the RFC, the *TPO Message* and the *Offer Response Message* cannot be sent separately in a Brand Independent Offer exchange. Hence, the subsequence defined by nodes (0,10,11,14,15) in both FSA-(a) and FSA-(c), which captures the scenario in Figure 12.3 (b), is not implemented by IOTP.

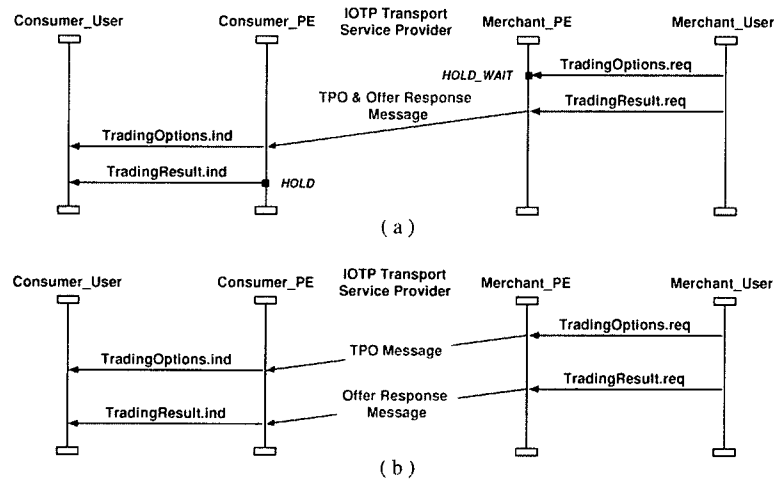


Figure 12.3: Two scenarios in the Offer phase using (a) a combined TPO & Offer Response Message and (b) two separate TPO and Offer Response Messages to implement the *TradingOptions* and *TradingResult* services.

Similarly, the remaining missing subsequences are due to two other combined messages used for communication between the Consumer_PE and Merchant_PE. These are:

- The combined *Authentication Status & TPO Message* in the Authentication and Brand Dependent Offer document exchanges, causing the four subsequences defined by nodes (0-9,10,11,14,15), (0-6,10,9,14,15), (0-5,7,10,11,14,15) and (0-5,7,10,9,14,15) in the service, to be missing in the protocol.
- The combined *Authentication Status & TPO & Offer Response* in the Authentication and Brand Independent Offer document exchanges, resulting in the subsequence defined by nodes (0-6,10,11,12-14,15) in the service to not occur in the protocol.

Primitive Sequence Calculation

Table 12.1 lists the statistics of the three FSA in Figure 12.2. There are: *10* sequences for the service language (FSA-(a)), *4* sequences for the protocol language (FSA-(b)), and thus *6* missing sequences in the protocol (FSA-(c)). The protocol preserves the maximum and minimum sequence length specified in the service. For example, the minimum sequence length is *8* for the sequence defined by nodes (0,10,9,14-19) in each FSA, and the maximum sequence length is *16* for the sequence defined by nodes (0-5,7,10,11,12-19) in FSA-(a) and the one defined by nodes (0-5,7,10',11,12-19) in FSA-(b).

FSA	*States*	*Arcs*	*Final States*	*Sequences*	*Longest*	*Shortest*
FSA-(a)	20	25	1	10	16	8
FSA-(b)	20	23	1	4	16	8
FSA-(c)	21	26	1	6	16	8

Table 12.1: Statistics of the three FSA in Figure 12.2.

12.2.3 Purchase Transaction

Figure 12.4 shows two FSA specifying the service language (FSA-(a)) and protocol language (FSA-(b)) obtained for a successful Purchase transaction. Comparison of the two languages for the Purchase transaction is performed in the same way as for Deposit.

Service Phases

A Purchase comprises an optional Authentication, an Offer, a Payment and a Delivery phase. Each FSA shown in Figure 12.4 has *2* final states, because the Purchase may stop at the end of the Payment phase (resulting in the same procedure as Deposit), or at the end of the Delivery phase that either follows or is combined with the Payment phase.

- The subsequences in the Authentication and the Offer phases are defined by nodes 0-15 in each FSA, which are the same as those defined for a Deposit (Figure 12.2).
- The subsequence defined by nodes 15-19 in each FSA specifies a *standalone* Payment phase, which cannot be combined with but may be followed by a Delivery phase. This subsequence is also the same as the one defined for Deposit.
- The subsequence defined by nodes (19-21,28,29) in each FSA represents a *standalone* Delivery phase that follows Payment.
- The subsequences defined by nodes 15-17 and 22-29 in FSA-(a) represent *a Payment phase combined with a Delivery phase* in IOTS. The subsequence defined by nodes (15,16, 22-26,28,29) in FSA-(b) specifies the corresponding service implemented by IOTP.

Missing Sequences in the Protocol

For a successful Purchase transaction, apart from the Authentication and the Offer phases, only a subset of the service in the combined Payment and Delivery phases is implemented by IOTP. Closer investigation of the missing sequences in the protocol shows that they are implemented due to the use of the two messages: the *Payment Request & Delivery Request Message* and the *Payment Response & Delivery Response Message*. These two combined IOTP

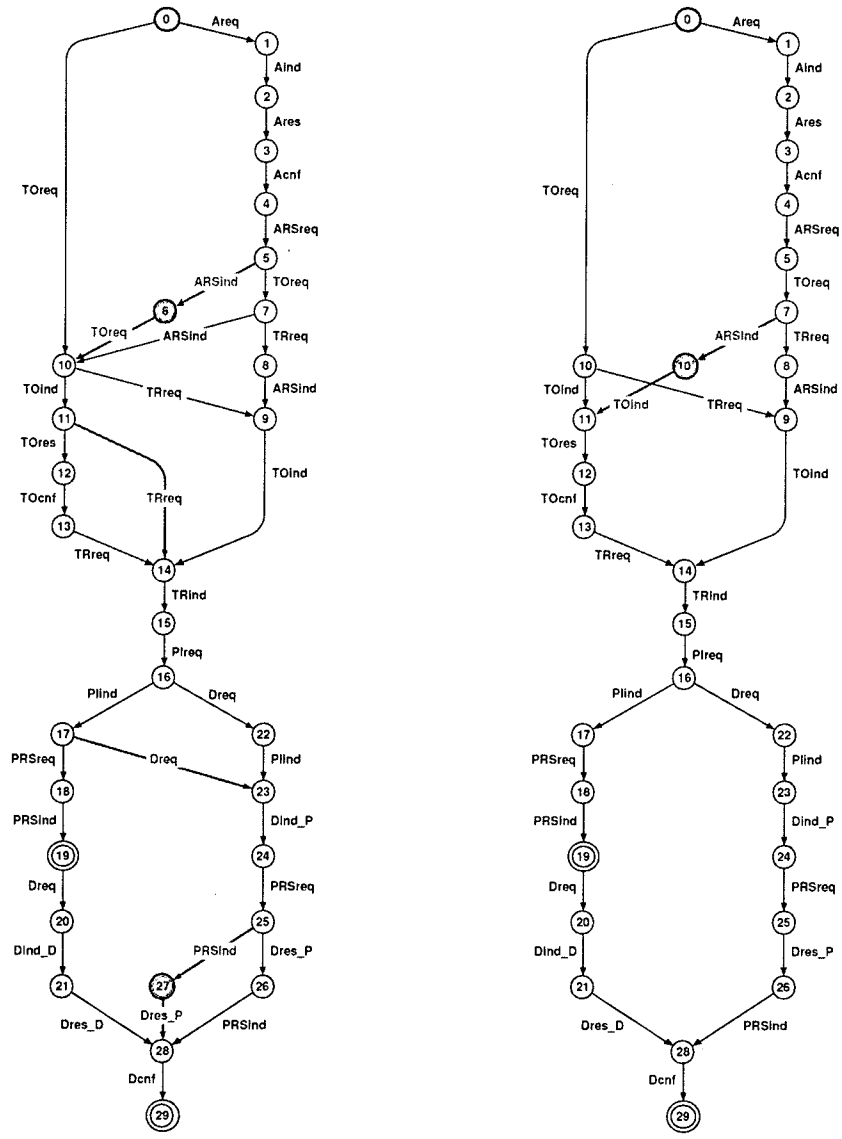


Figure 12.4: Two FSA for a successful Purchase transaction specifying (a) the service language and (b) the protocol language.

messages prevent some subsequences of primitives that occur in the service, from occurring in the protocol. Figure 12.5 depicts two scenarios of how the Consumer_PE and PHandler_PE communicate in a Payment phase combined with a Delivery phase.

In Figure 12.5 (a), a Payment-and-Delivery exchange is used to implement the *PayInvoke*, *PayResult* and *Delivery* services. The Consumer_PE sends a Payment Request & Delivery Request Message, after receiving a *PayInvoke.req* (*PIreq*) followed by a *Delivery.req* (*Dreq*) from the Consumer_User. On receipt of the Payment Request & Delivery Request Message, the PHandler_PE delivers a *PayInvoke.ind* (*PIind*) and then a *Delivery.ind* (*Dind_P*) to the PHandler_User. Similarly, the PHandler_PE sends a Payment Response & Delivery Response Message after receiving a *PayResult.req* (*PRSreq*) followed by a *Delivery.res* (*Dres_P*) from the PHandler_User. On receipt of the Payment Response & Delivery Response Message, the Consumer_PE delivers a *PayResult.ind* (*PRSind*) and then a *Delivery.cnf* (*Dcnf*) to the Consumer_User. The subsequence defined by nodes (15,16,22-26,28,29) in both FSA-(a) and FSA-(b) in Figure 12.4 captures this scenario.

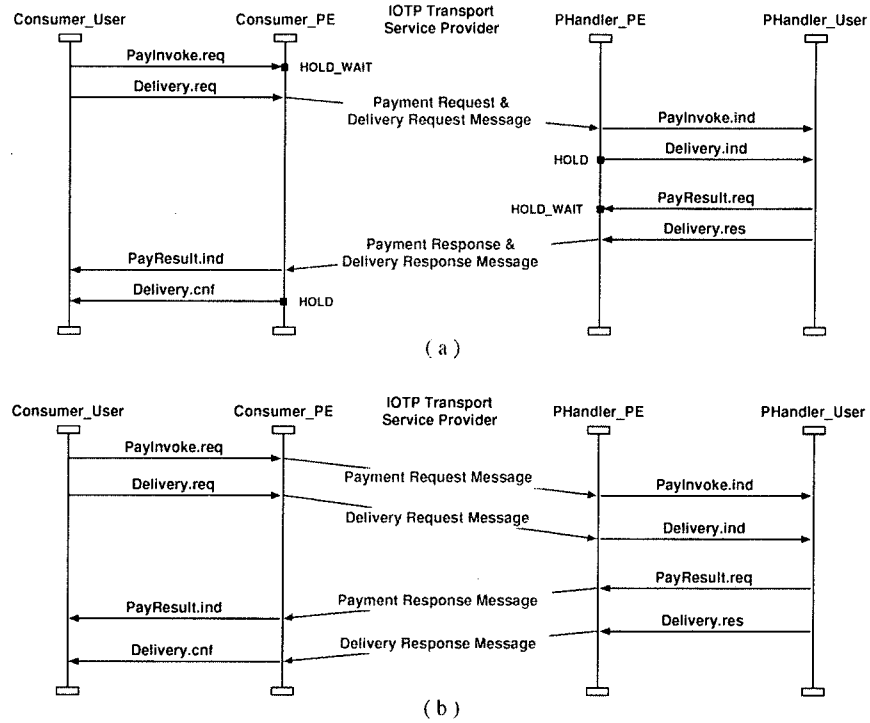


Figure 12.5: Different scenarios in the combined Payment and Delivery phases using (a) a Payment Request & Delivery Request Message and a Payment Response & Delivery Response Message and (b) the separate Payment Request, Delivery Request, Payment Response and Delivery Response Messages.

In Figure 12.5 (b), we allow the Payment Request, the Delivery Request, the Payment Response and the Delivery Response Messages to be sent individually. Due to the use of these four separate messages, occurrences of the *PayInvoke.ind* and the *Delivery.req* are *independent* of each other. This also applies to the *PayResult.ind* and the *Delivery.res*. Hence, the subsequences that involve either 1) *PIind* occurring before *Dreq* (as defined by nodes (16,17,23) in FSA-(a) in Figure 12.4), or 2) *PRSind* occurring before *Dres_P* (as defined by nodes (25,27,28) in FSA-(a) in Figure 12.4) are valid in IOTS but are not implemented by IOTP.

The above facts indicate that the occurrences of those interleaving primitive events make the service language a superset of the protocol language for a successful Purchase transaction.

Primitive Sequence Calculation

Table 12.2 lists the statistics of the two FSA shown in Figure 12.4 and another FSA for the missing sequences in the protocol. The protocol preserves the maximum and minimum sequence length specified in the service. For example, the maximum sequence length is *20* for the sequence defined by nodes (0-5,7,10/10',11,12-21,28,29) in FSA-(a) or FSA-(b). The minimum sequence length is *8* for the sequence defined by nodes (0,10,9,14-19) in both FSA. This is the same as for the Deposit transaction (Figure 12.2), because a Purchase transaction ending with a (standalone) Payment phase has the same transaction procedure as Deposit.

FSA	*States*	*Arcs*	*Final States*	*Sequences*	*Longest*	*Shortest*
Service Language (FSA-(a))	30	37	2	60	20	8
Protocol Language (FSA-(b))	29	32	2	12	20	8
Missing Sequences in the Protocol	44	55	2	48	20	8

Table 12.2: Statistics of FSA in Figure 12.4 and the FSA for the missing sequences in the protocol.

12.2.4 Value Exchange Transaction

The Value Exchange transaction extends the procedure of Deposit with a second (standalone) Payment phase and terminates at the end of that phase. Figure 12.6 depicts three FSA specifying the languages for the verification of a successful Value Exchange transaction. They can be derived from the FSA for a successful Deposit transaction (Figure 12.2) by: 1) updating the final state defined by node 19 to a non-final state; and 2) concatenating to the FSA the group of nodes 19-23 with arcs specifying the subsequences for the second Payment phase.

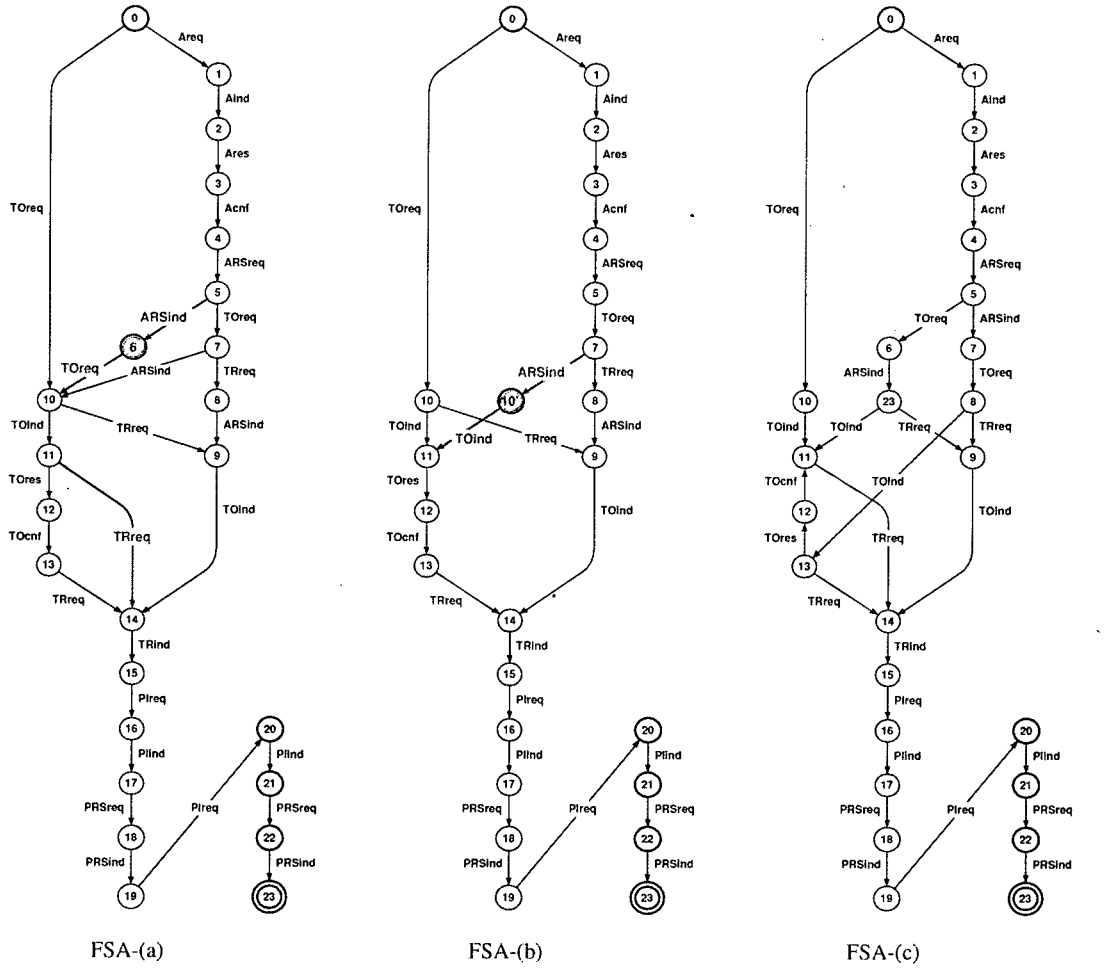


Figure 12.6: Three FSA for a successful Value Exchange transaction specifying (a) the service language, (b) protocol language and (c) the missing sequences in the protocol.

Table 12.3 lists the statistics of the above three FSA for the Value Exchange transaction. It shows that the protocol preserves the maximum and minimum sequence length in the service. The maximum sequence length is *20*, such as the one defined by nodes (0-5,7,10/10',11,12-23) in FSA-(a) or FSA-(b), and the minimum sequence length is *12*, such as the one defined by nodes (0,10,9,14-23) in each FSA.

FSA	*States*	*Arcs*	*Final States*	*Sequences*	*Longest*	*Shortest*
FSA-(a)	24	29	1	10	20	12
FSA-(b)	24	27	1	4	20	12
FSA-(c)	25	30	1	6	20	12

Table 12.3: Statistics of the three FSA in Figure 12.6.

12.3 Valid Implementation of Successful Transaction Service

The language comparison for successful Payment-related transactions in Section 12.2 shows that the protocol language is a subset of the service language due to IOTP combining messages. The sequences defined in IOTS but missing in IOTP result from those subsequences that are not implemented by IOTP during certain phases. These are: 1) a standalone Offer phase, 2) an Authentication phase followed by an Offer phase, and 3) a Payment phase combined with a Delivery phase. A common feature which exists between these phases is that one TRole_User can *successively* submit at least two (non-Cancel) service primitives *without* requiring any confirmation from the IOTS Provider. This causes the inter leavings of the corresponding primitive events to be included in the service definition.

One interesting observation regarding the IOTS is that the interleavings of primitive events during the above phases are just alternative ways of implementing the corresponding service. For example, in a standalone Offer phase, the primitive sequences (*TOreq* ***TOind*** ***TRreq*** *TRind*) and (*TOreq* ***TRreq*** ***TOind*** *TRind*) both specify a Brand Independent Offer service where the Merchant_User does not require the Consumer_User to reply with the trading selection. Hence, occurrences of those interleaving primitive events do not introduce any *new* type of service into the service definition. There exists at least one minimum set of sequences for an acceptable implementation of the service. In the following, we define these minimum acceptable sets of sequences. For clarity, the node-numbers used in each FSA are consistent with those used in the FSA for service languages in Section 12.2.

Requirement 12.1 (Minimum set of sequences for an acceptable implementation of the service in the Offer phase). *A minimum set of sequences for an acceptable implementation of the service in the Offer phase comprises 2 sequences of primitive occurrences, where one is used for the Brand Dependent Offer service and must include TradingOptions.res and TradingOptions.cnf, and the other is for the Brand Independent Offer service and does not involve TradingOptions.res and TradingOptions.cnf.*

In a standalone Offer phase, there is only *1* valid sequence that defines the Brand Dependent Offer service, and *2* specifying the Brand Independent Offer service. According to Requirement 12.1, there exist *two* minimum acceptable sets of sequences for the service in the Offer phase, as denoted by $MASS^1_{Ofr}$ and $MASS^2_{Ofr}$. Figure 12.7 shows the two FSA specifying these two sets of sequences.

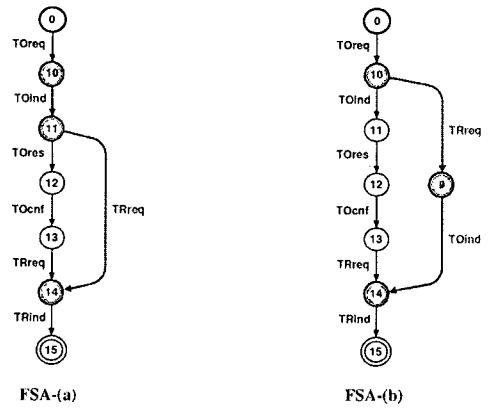


Figure 12.7: Two FSA specifying (a) $MASS^1_{Ofr}$ and (b) $MASS^2_{Ofr}$.

Requirement 12.2 (Minimum set of sequences for an acceptable implementation of the service in the Authentication phase followed by the Offer phase). *A minimum set of sequences for an acceptable implementation of the service in the Authentication phase followed by the Offer phase comprises 2 sequences of primitive occurrences, where one is used for successful Authentication followed by the Brand Dependent Offer service, and the other is for successful Authentication followed by the Brand Independent Offer service.*

With Requirement 12.2, there are five minimum acceptable sets of sequences for the service in the Authentication phase followed by the Offer phase. Figure 12.8 depicts the five FSA specifying these five sets of sequences, as denoted by $MASS^{1}_{AuthOfr}$ to $MASS^{5}_{AuthOfr}$. Also, we use $MASS^{n_1,n_2}_{AuthOfr}$ to denote $MASS^{n_1}_{AuthOfr}$ and $MASS^{n_2}_{AuthOfr}$, where $n_1, n_2 \in \{1,2,3,4,5\}$. In $MASS^{1,2}_{AuthOfr}$, the *TOreq* occurs after *ARSind* to start either the Brand Dependent Offer or the Brand Independent Offer service, whereas the *TOind* and *TRreq* used for a Brand Independent Offer take place in a different order. This is similar in $MASS^{3,4}_{AuthOfr}$ except that the *TOreq* occurs before *ARSind*. In $MASS^{5}_{AuthOfr}$, the *ARSind* directly follows *TOreq* for a Brand Dependent Offer, whereas it occurs after *TRreq* that follows *TOreq* for a Brand Independent Offer.

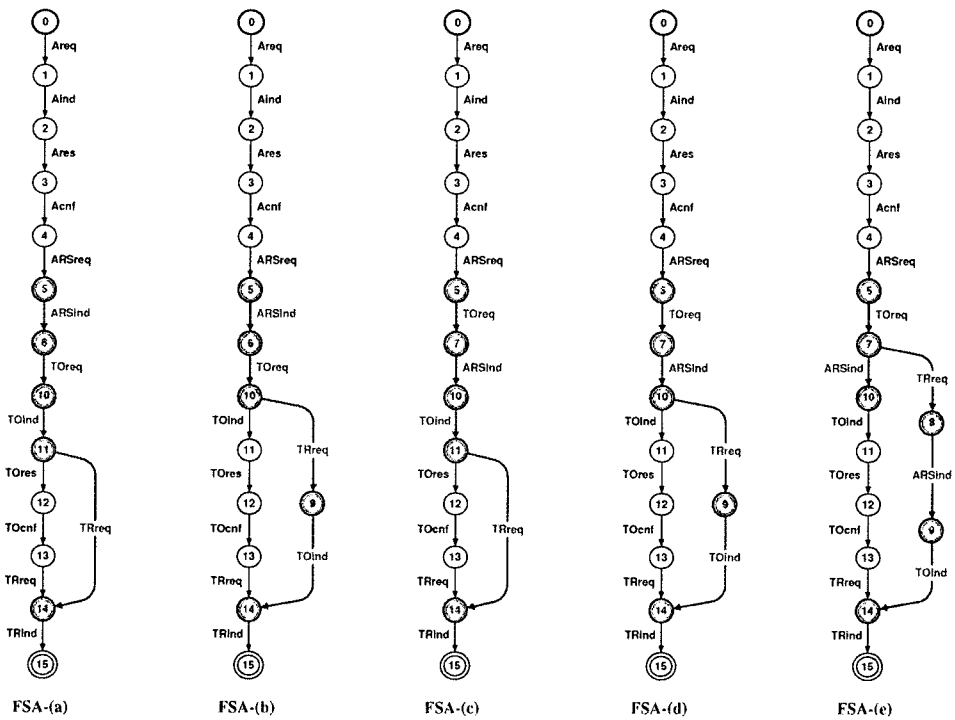


Figure 12.8: Five FSA specifying (a) $MASS^{1}_{AuthOfr}$, (b) $MASS^{2}_{AuthOfr}$, (c) $MASS^{3}_{AuthOfr}$, (d) $MASS^{4}_{AuthOfr}$ and (e) $MASS^{5}_{AuthOfr}$.

Requirement 12.3 (Minimum set of sequences for an acceptable implementation of the service in the Payment phase combined with the Delivery phase). *A minimum set of sequences for an acceptable implementation of the service in the Payment phase combined with the Delivery phase comprises only 1 sequence of primitive occurrences.*

There are *four* minimum acceptable sets of sequences for the combined Payment and Delivery services. Each set comprises a single sequence of primitives. Figure 12.9 illustrates the four FSA specifying these four sets of sequences, as denoted by $MASS^{1}_{PDlv}$ to $MASS^{4}_{PDlv}$.

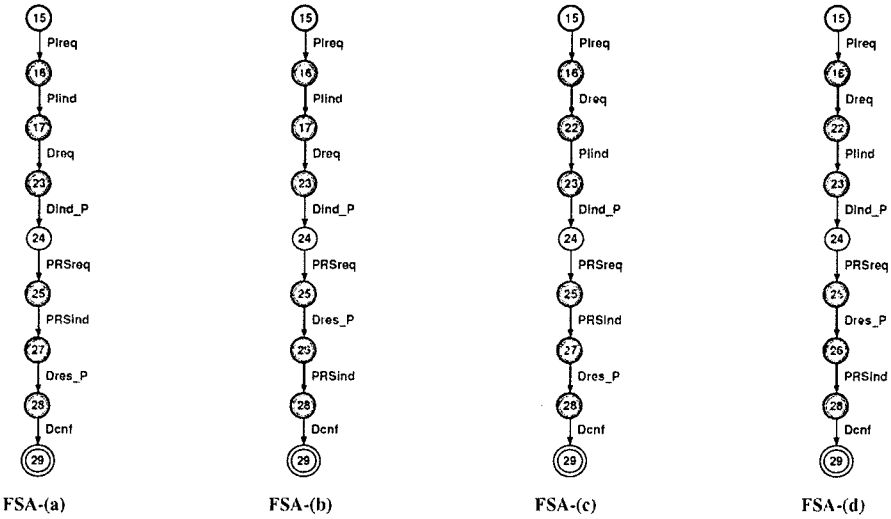


Figure 12.9: Four FSA of (a) $MASS^{1}_{PDlv}$, (b) $MASS^{2}_{PDlv}$, (c) $MASS^{3}_{PDlv}$ and (d) $MASS^{4}_{PDlv}$.

Also, the (standalone) Payment and Delivery phases have no interleaving primitive events. Hence, there is only one (minimum) acceptable set of sequences for implementing the service in each phase, as denoted by $MASS_{Pay}$ for the Payment phase and $MASS_{Dlv}$ for the Delivery phase. Figure 12.10 shows the two FSA specifying these two sets of sequences.



Figure 12.10: Two FSA specifying (a) $MASS_{Pay}$ and (b) $MASS_{Dlv}$.

Now we define the minimum acceptable service languages for successful Payment-related transactions in IOTS. Firstly, we need some notation for the definition:

- let E be the set of events and E^* be the set of finite sequences of events over E; and
- for the sets of sequences A, $B \subseteq E^*$, $A \bullet B = \{\sigma_1\sigma_2 \mid \sigma_1 \in A, \sigma_2 \in B\}$.

Then we define the minimum acceptable service languages (L_{min}) with the essential primitive sequences for each successful Payment-related transaction. These are:

$L^{ij}_{min}(Deposit) = (MASS^{i}_{Ofr} \cup MASS^{j}_{AuthOfr}) \bullet MASS_{Pay}$

$L^{ijk}_{min}(Purchase) = (MASS^{i}_{Ofr} \cup MASS^{j}_{AuthOfr}) \bullet (MASS_{Pay} \cup (MASS_{Pay} \bullet MASS_{Dlv}) \cup MASS^{k}_{PDlv})$

$L^{ij}_{min}(ValueExchange) = (MASS^{i}_{Ofr} \cup MASS^{j}_{AuthOfr}) \bullet MASS_{Pay} \bullet MASS_{Pay}$

where i=1, 2, j=1, 2, 3, 4, 5 and k=1, 2, 3, 4.

Thus the set of minimum acceptable service languages ($MASL$) is given by

$$MASL = \{L^{ij}_{min}(Deposit), L^{ijk}_{min}(Purchase), L^{ij}_{min}(ValueExchange) \mid i \in \{1,2\}, j \in \{1,2,3,4,5\}, k \in \{1,2,3,4\}\}$$

12.4 Verification of Successful Transactions

The verification of successful Payment-related transactions consists of two steps. We firstly determine the minimum set of sequences that must be realised by the protocol for it to be an acceptable (or valid) implementation of IOTS. This has already been done in Section 12.3. With the minimum acceptable subsets defined, we then prove that the subset of sequences implemented by the successful Payment-related transaction in IOTP is an acceptable refinement of the service. The initial idea of applying these two steps to the verification of Payment-related transactions was suggested by Professor Jonathan Billington.

The verification result of successful Authentication and Payment-related transactions in IOTP is given by Theorem 12.1. It can be proved based on: 1) the language comparison results between protocol and service in Section 12.2; and 2) the minimum acceptable implementations of the successful transaction service defined in Section 12.3.

The protocol is a faithful refinement of the service, if 1) the protocol and service languages are equivalent, or 2) the protocol language is an acceptable subset of the service language.

Theorem 12.1 (Successful Transaction Verification). *Subject to the limitations set out in Section 10.2, the successful Authentication and Payment-related transactions specified in IOTP are a faithful refinement of the corresponding services defined in IOTS.* □

Proof: We consider the Authentication and the Payment-related transactions separately.

Firstly, the protocol language of a successful Authentication transaction is equivalent to its service language. This is a direct result from the language comparison in Section 12.2.1.

Secondly, the protocol language of each successful Payment-related transaction (L_p) is a minimum acceptable subset of the corresponding service language defined in IOTS. This is a result from the following facts obtained by using FSM [11]:

1. $L_p(Deposit)$, which is defined by FSA-(b) in Figure 12.2, is identical to $L_{min}^{25}(Deposit)$ (the same is true for Withdrawal and Refund transactions);
2. $L_p(Purchase)$, defined by FSA-(b) in Figure 12.4, is identical to $L_{min}^{254}(Purchase)$; and
3. $L_p(ValueExchange)$, defined by FSA-(b) in Figure 12.6, is identical to $L_{min}^{25}(ValueExchange)$.

Hence, Theorem 12.1 is proved. □

12.5 Verification of Transactions with Cancellation

The verification of successful Authentication and Payment-related transactions is the first step towards verifying IOTP. In this section, we take into account transaction cancellation and error handling procedures in the protocol. As a result, the service will involve a *Cancel* phase, where the transaction may terminate when either a *Cancel.req* or a *Cancel.ind* primitive event occurs at a TRole_User interface. The *Cancel* service, introduces significant complexity into both the service and protocol languages of transactions.

In Section 12.4, we proved that successful Payment-related transactions in IOTP provide an acceptable subset of the corresponding services defined in IOTS. This allows us to simplify the verification of Payment-related transactions with cancellation, by eliminating the sequences for cancellations only occurring to the successful transaction services that cannot be implemented

by IOTP. This is conducted in two steps. Firstly, the IOTS CPN is modified to capture only the acceptable subset of sequences that are implemented by IOTP for successful transactions. Appendix G provides this modified IOTS CPN. Then, by including the module for the Cancel phase into the CPN model, an acceptable subset of the service language for transactions with cancellation is obtained. This subset of sequences defines all possible cancellations that may occur to the successful transaction services implemented by IOTP, and therefore can be used as the service language for the verification of the Payment-related transactions with cancellation.

With error handling, configurations for message retransmission need to be considered for the verification of transactions with cancellation. In Chapter 11, the state space of a transaction is calculated when each RCmax parameter has a value of either *0* or *1*. However, the same protocol language is obtained from the state spaces of the transaction, no matter whether RCmax is set to *0* or *1*. More results are given in Chapter 13.

12.5.1 Authentication Transaction

For Authentication with cancellation, language comparison between the service and protocol shows that there are missing sequences but no illegal sequences in the protocol. Table 12.4 lists the statistics of three FSA representing the service language, the protocol language and the missing sequences in the protocol, for such a transaction. It can be seen that the protocol preserves the maximum and minimum sequence length specified in the service. The minimum sequence length of those occurring in the service but missing in the protocol is *4*, indicating that the protocol preserves *all* the sequences with minimum sequence length of *2* specified in the service. These are: (*Areq, Creq_M*) and (*Areq, Cind_M*), representing that an Authentication transaction is cancelled at the Merchant side without any indication to the Consumer.

FSA	*States*	*Arcs*	*Final States*	*Sequences*	*Longest*	*Shortest*
Service Language	13	36	2	49	6	2
Protocol Language	15	37	3	41	6	2
Missing Sequences in the Protocol	12	18	1	8	6	4

Table 12.4: Statistics of three FSA for an Authentication transaction with cancellation.

Figure 12.11 shows the FSA specifying the *8* missing sequences in the protocol. Figure 12.12 (a)-(d) depict four scenarios representing four of these *8* sequences that are not implemented by the protocol. For brevity, FSA 12.11 refers to the FSA in Figure 12.11 and scenario (a) for the scenario in Figure 12.12 (a) in the following illustrations.

Scenario (a) corresponds to the sequence (*Areq, Aind, Cind_C, Cind_M*) defined by nodes (0-2,5,9) in FSA 12.11. In this scenario, the Consumer_PE enters the state `WAIT`, upon receiving an Authentication Request Message and delivers an *Authentication.ind* to its user. According to the current definition of IOTP [36], no timer is used and thereby no timeout can occur at a TRole_PE which is in the `WAIT` state. As a result, a *Cancel.ind* primitive cannot occur at such a TRole_PE unless a *Cancel Message* is received from the peer TRole_PE. In scenario (a), it is therefore not possible for the *Cancel.ind* to occur after the *Authentication.ind* primitive event at the Consumer_PE. This further implies that any primitive event following the sequence (*Areq, Aind, Cind_C*) will not be implemented by the protocol. One example is the occurrence of a *Cancel.ind* primitive at the Merchant_PE (*Cind_M*) following the *Cind_C*

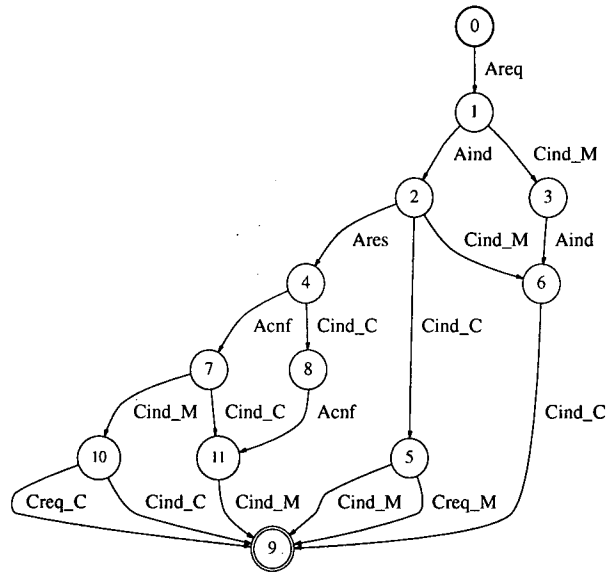


Figure 12.11: The FSA specifying the missing sequences in the protocol for an Authentication transaction with cancellation.

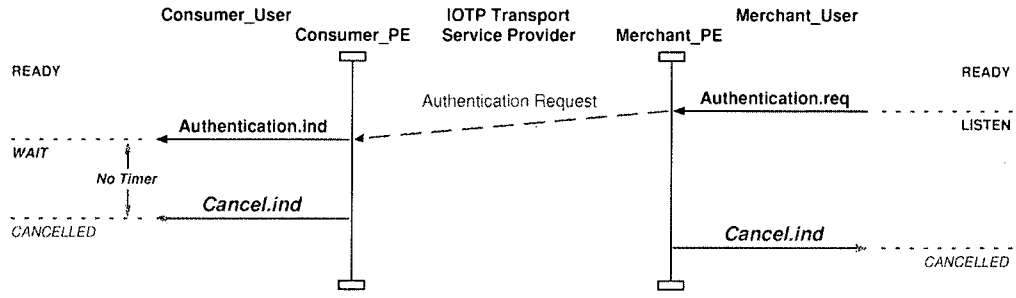
after *Aind* (at the Consumer_PE) as shown in scenario (a). By replacing the *Cind_M* with *Creq_M*, we can obtain the scenario representing another missing sequence in the protocol, which is (*Areq, Aind, Cind_C, Creq_M*) defined by the same group of nodes in FSA 12.11.

Scenario (b) corresponds to the sequence (*Areq, Aind, Cind_M, Cind_C*) defined by nodes (0,1,2,6,9) in FSA 12.11. In this scenario, a *Cind_M* occurs, once the message retransmission timer expires and the retransmission counter reaches its limit at the Merchant_PE. Although the transaction is cancelled by the Merchant_PE, it is not possible for a *Cind_C* to occur following the *Cind_M* under this circumstance. There are two reasons: 1) the Consumer_PE is in the `WAIT` state and no timer is used; and 2) the Consumer is not informed of such cancellation initiated at the Merchant_PE (e.g., via a Cancel Message). Swapping the order of the occurrences of *Aind* and *Cind_M* leads to another missing sequence in the protocol, as defined by nodes (0,1,3,6,9) in FSA 12.11.

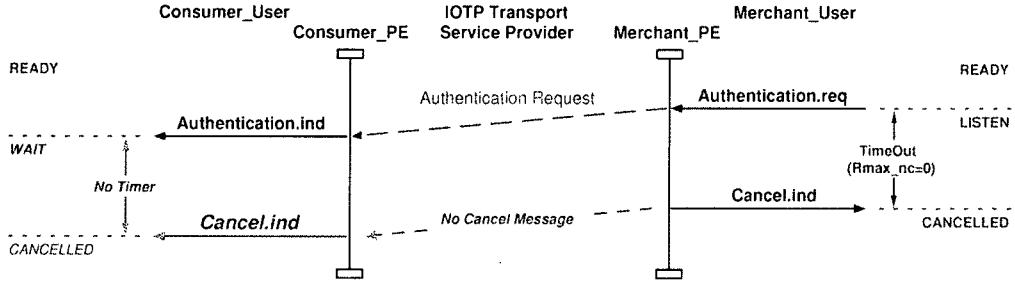
In the next two scenarios (c) and (d), the Merchant_PE enters the state `WAIT` after *Acnf*, and hence it is not possible for a *Cind_M* to occur unless the Merchant_PE receives a Cancel Message from the Consumer_PE. **Scenario (c)** captures the sequence (*Areq, Aind, Ares, Acnf, Cind_C, Cind_M*) defined by nodes (0-2,4,7,11,9) in FSA 12.11. The transaction is cancelled by the Consumer_PE upon a timeout without message retransmission (RCmax_c=0). Another sequence defined by nodes (0-2,4,8,11,9) corresponds to a similar scenario with the only difference being in the order of the occurrences of *Acnf* and *Cind_C*. **Scenario (d)** captures the sequence (*Areq, Aind, Ares, Acnf, Cind_M, Cind_C*) defined by nodes (0-2,4,7,10,9) in FSA 12.11. Together with the other sequence (ending with *Creq_C*) defined by the same group of nodes, both involve a *Cind_M* occurring after *Acnf* and therefore are not preserved by the protocol.

12.5.2 Deposit Transaction

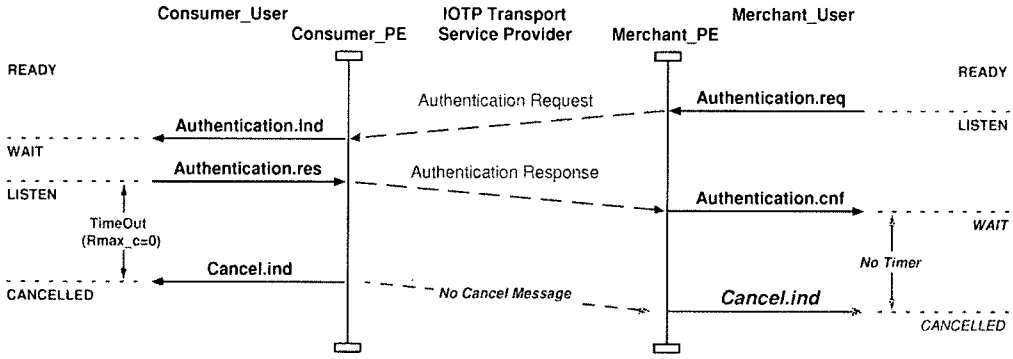
Table 12.5 summarises the statistics of the four FSA representing the service language, the protocol language, the illegal sequences and the missing sequences in the protocol for a Deposit transaction with cancellation.



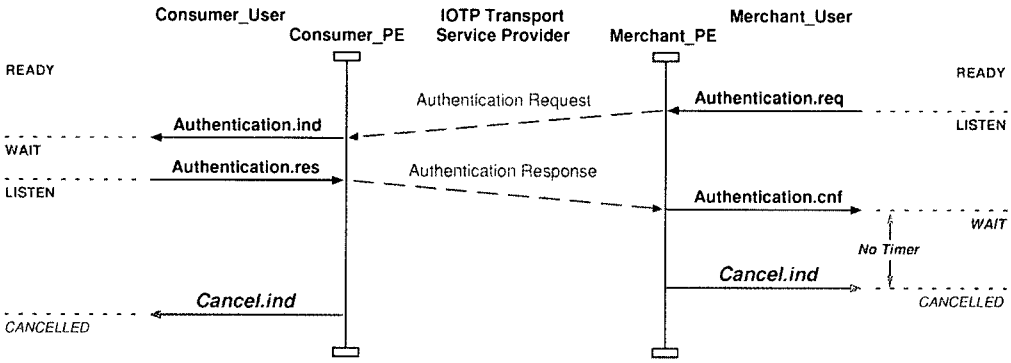
(a) Sequence (*Areq, Aind, Cind_C, Cind_M*)



(b) Sequence (*Areq, Aind, Cind_M, Cind_C*)



(c) Sequence (*Areq, Aind, Ares, Acnf, Cind_C, Cind_M*)



(d) Sequence (*Areq, Aind, Ares, Acnf, Cind_M, Cind_C*)

Figure 12.12: Four time sequence diagrams of the Authentication transaction with cancellation showing four missing sequences in the protocol.

FSA	*States*	*Arcs*	*Final States*	*Sequences*	*Longest*	*Shortest*
Service Language	38	119	3	288	17	2
Protocol Language	55	143	9	258	16	2
Illegal Sequences in the Protocol	33	63	4	92	16	2
Missing Sequences in the Protocol	38	79	1	122	17	4

Table 12.5: Statistics of four FSA for a Deposit transaction with cancellation.

The FSA, which specifies the illegal sequences in the protocol, is shown in Figure 12.13 and is denoted by FSA 12.13. The missing sequences in the protocol for a Deposit transaction are a superset of those for an Authentication transaction, because the Authentication document exchange, which comprises the Authentication transaction, is also part of a Payment-related transaction. As we have already examined the missing sequences in the protocol for Authentication in Section 12.5.1, we can exclude them from those for Deposit. This results in a simplified FSA which specifies the remaining sequences missing in the protocol for Deposit, as shown in Figure 12.14 and denoted by FSA 12.14. To improve readability, the arcs labelled by *Cancel* primitives in each FSA are drawn as dashed arcs. Also, an arc labelled by two *Cancel* primitives which are separated by a comma, represents the occurrence of either primitive [26]. For example, the arc from node 22 to 23 in FSA 12.13 is labelled by *Creq_C, Cind_C*, indicating that either a *Creq_C* or a *Cind_C* primitive occurs.

Illegal Sequences in the Protocol

In FSA 12.13, all sequences, except (*TOreq, TRreq*) defined by nodes (0,1,17), involve either one *Cancel* primitive event at only the Consumer or two *Cancel* primitive events at both the Consumer and the Payment Handler.

- Primitive sequences defined by node sequences excluding node 7 (shown in bold) in FSA 12.13, involve a *Cancel* primitive event occurring at the Consumer only. They represent the solely cancelled Deposit transaction service before the Payment phase starts. Therefore, all these sequences end up with the undesired terminal states given by two dead markings M_{592} and M_{375} of a Deposit transaction in Group 6 in Table 11.3.

- Primitive sequences defined by node sequences containing node 7 in FSA 12.13, either involve a *Cancel* primitive event occurring at the Consumer only, or two *Cancel* primitive events occurring at both the Consumer and Payment Handler. They correspond to the Deposit transaction service cancelled at the Consumer only, or by both the Consumer and Payment Handler (that the Merchant is not informed of), after the Payment phase has started. Therefore, all these sequences end up with the undesired terminal states given by six dead markings of a Deposit transaction in Group 5 and 7 in Table 11.3.

In FSA 12.13, the sequence (*TOreq, TRreq*) indicates a transaction service terminates at the end of an Offer phase (without the Consumer's trading selection) at the Merchant side while nothing happens at the Consumer side. This results from the undesired terminal state specified by M_{47} in Group 6 in Table 11.3. In M_{47}, the transaction is complete at the Merchant_PE at the end of a Brand Independent Offer document exchange, but cancelled at the Consumer_PE without any document exchange being initiated. Figure 12.15 depicts the scenario representing the sequence (*TOreq, TRreq*) that is implemented by IOTP but is illegal in IOTS.

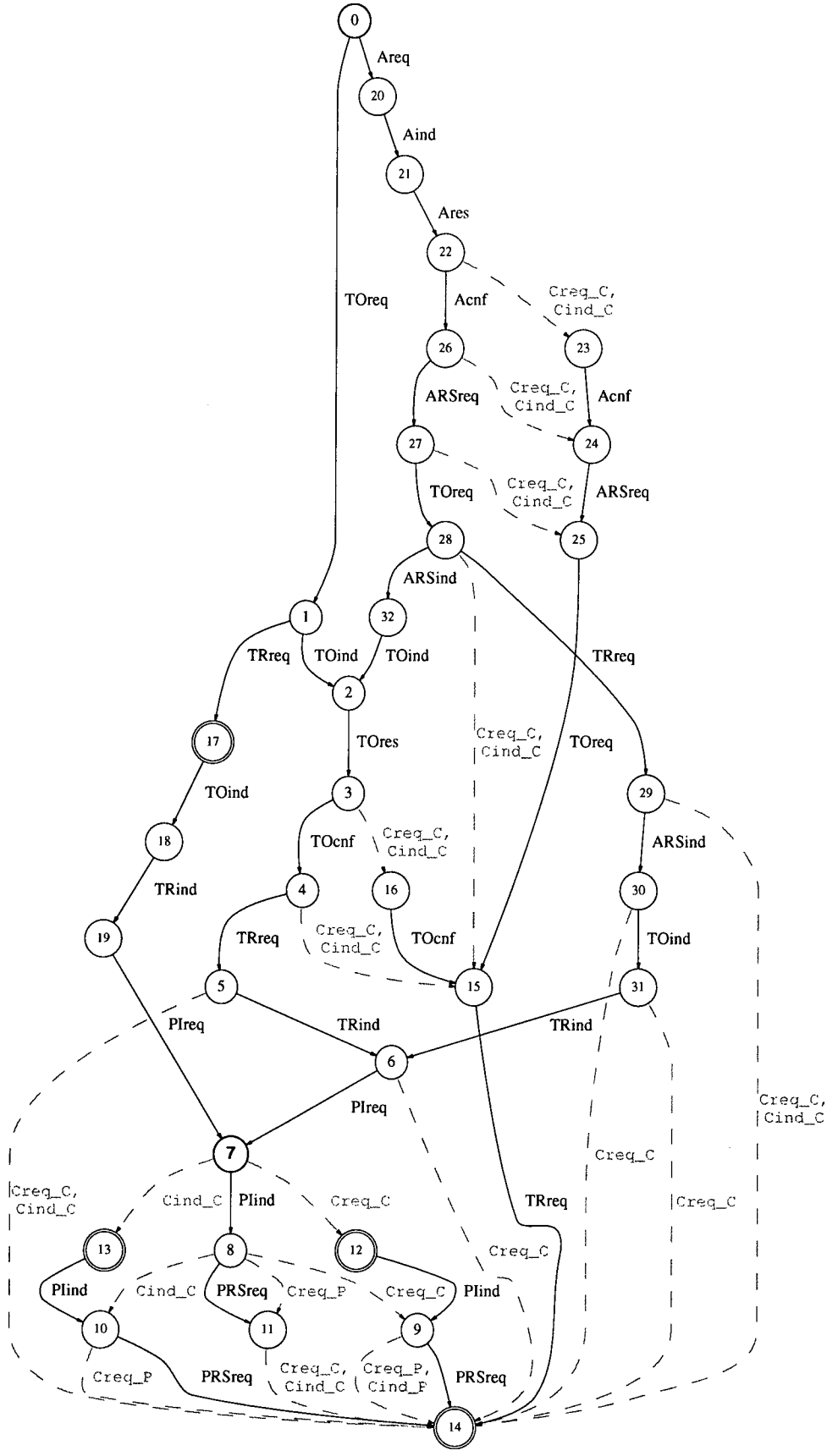


Figure 12.13: The FSA specifying the illegal sequences in the protocol for a Deposit transaction with cancellation.

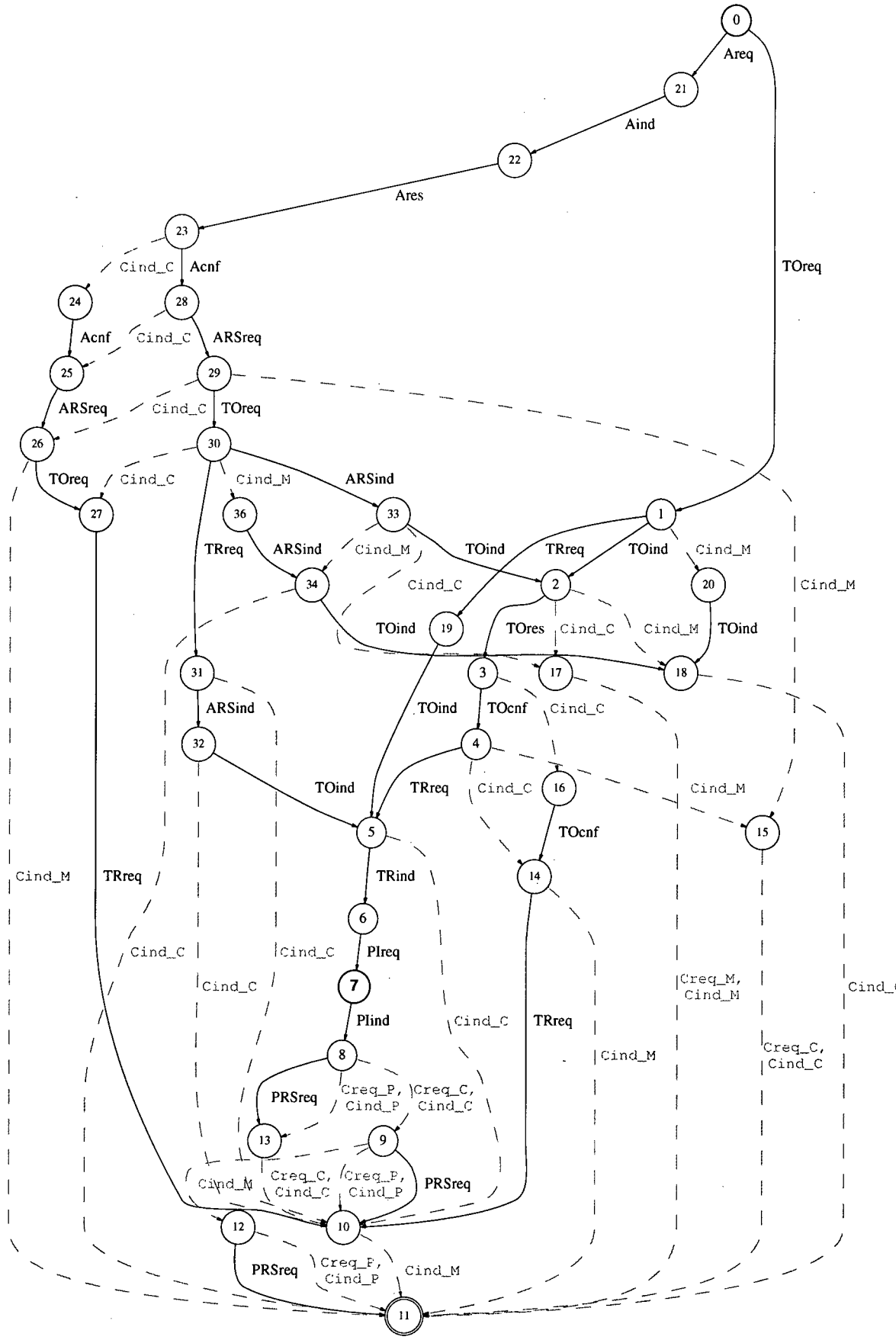


Figure 12.14: The FSA for the missing sequences in the protocol for a Deposit transaction with cancellation excluding those defined by FSA 12.11.

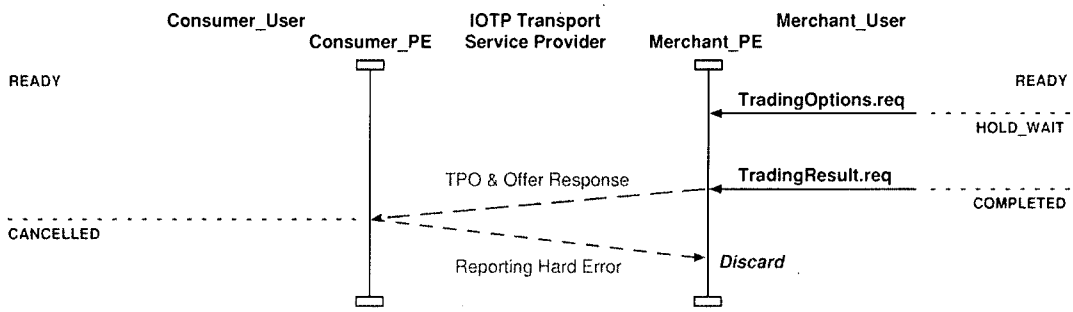


Figure 12.15: Time sequence diagram showing the primitive sequence (*TOreq, TRreq*) that is implemented by IOTP but is not in IOTS.

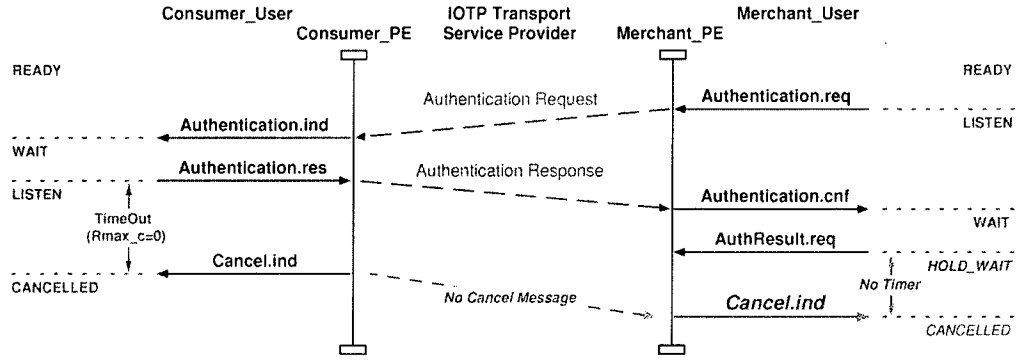
The scenario in Figure 12.15 happens when the Consumer_PE detects a hard error in the TPO & Offer Response message received from the Merchant_PE at the beginning of the transaction and thereby returns an Error Message reporting this error to cancel the transaction. The Consumer_PE enters the `CANCELLED` state, but the *Cancel.ind* primitive will not occur at the Consumer side because the Consumer_User is not aware of the transaction service. According to the definition of IOTP, the Merchant_PE which has entered the `COMPLETED` state, discards the Error Message reporting the hard error from the Consumer_PE (see Section 3.2.7 and also Figure 11.9 (a) in Section 11.3.3).

Missing Sequences in the Protocol

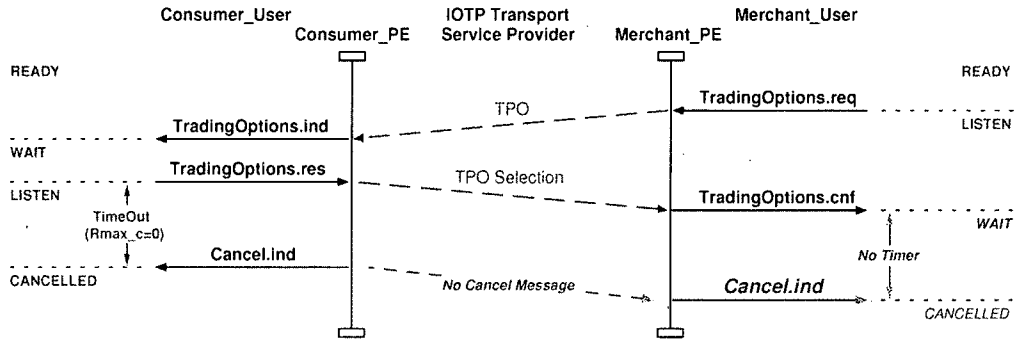
In FSA 12.14, primitive sequences excluding node 7 correspond to the cancelled Deposit transaction service by both the Consumer and Merchant before the Payment phase starts at the Payment Handler. Figure 12.16 depicts six scenarios as representative examples of these missing sequences in the protocol. Using the same convention as in Section 12.5.1, scenario (a) refers to the scenario in Figure 12.12 (a) and similarly for scenario (b) to (f).

- **Scenario (a)** captures the sequence (*Areq, Aind, Ares, Acnf, ARSreq, Cind_C, Cind_M*) defined by nodes (0,21,22,23,28,29,26,11) in FSA 12.14. This is a representative example of the three sequences ending with the primitive event *Cind_M* from node 26 to 11. In this scenario, the Merchant_PE holds on, waiting for *TOreq* after receiving *ARSreq* from the user, to generate a combined Authentication Status & TPO Message. According to the current design of IOTP [36], it is not possible for *Cind_M* to occur when the Merchant_PE is in the `HOLD_WAIT` state, unless the Merchant_PE receives a Cancel Message from the Consumer_PE. The occurrence of *Cind_C* does not generate a Cancel Message to the Merchant and thus will not result in the occurrence of *Cind_M*. The same is true for the two sequences involving *Cind_M* from node 29 to 15 in the FSA.

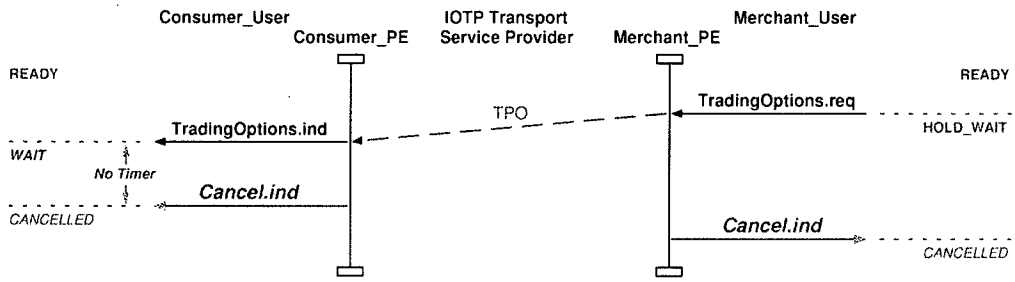
- **Scenario (b)** shows the sequence (*TOreq, TOind, TOres, TOcnf, Cind_C, Cind_M*) defined by nodes (0-4,14,11) in FSA 12.14. In this scenario, the Merchant_PE starts waiting for *TRreq* from the user on occurrence of *TOcnf*. However, once the Merchant has entered the `WAIT` state, either *Cind_M* following *Cind_C* or *Cind_M* being followed by *Creq_C* or *Cind_C*, cannot occur for the same reason as for scenario (a). The corresponding sequences are those ending with *Cind_M* from node 14 to 11, and those involving *Cind_M* from node 4 to 15 in the FSA.



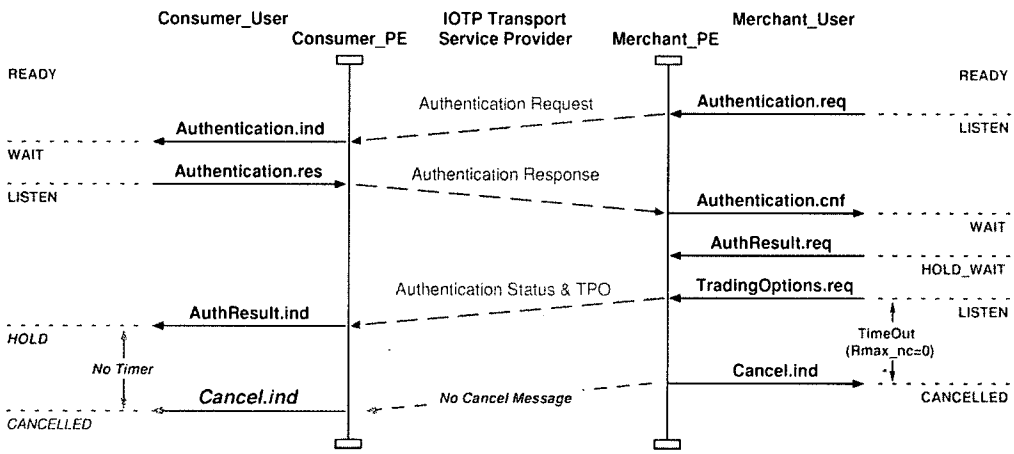
(a) Sequence (*Areq, Aind, Ares, Acnf, ARSreq, Cind_C, Cind_M*)



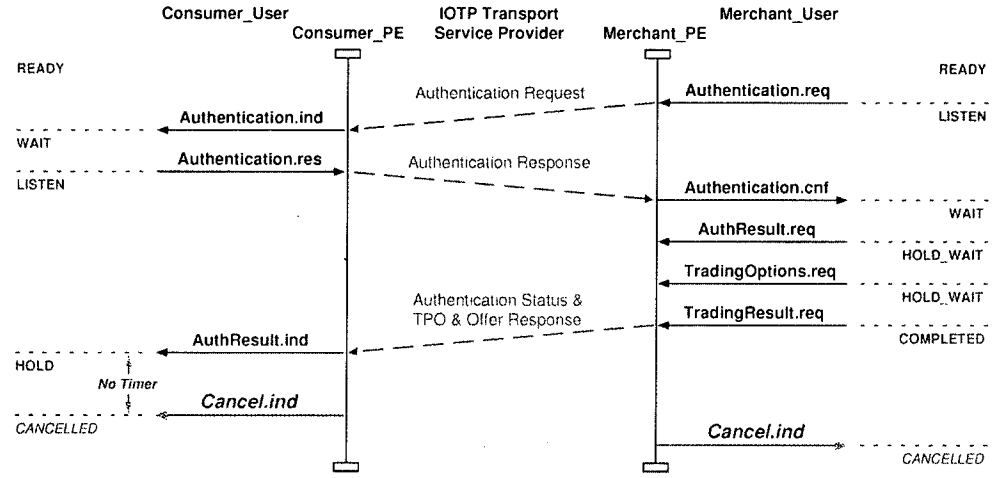
(b) Sequence (*TOreq, TOind, TOres, TOcnf, Cind_C, Cind_M*)



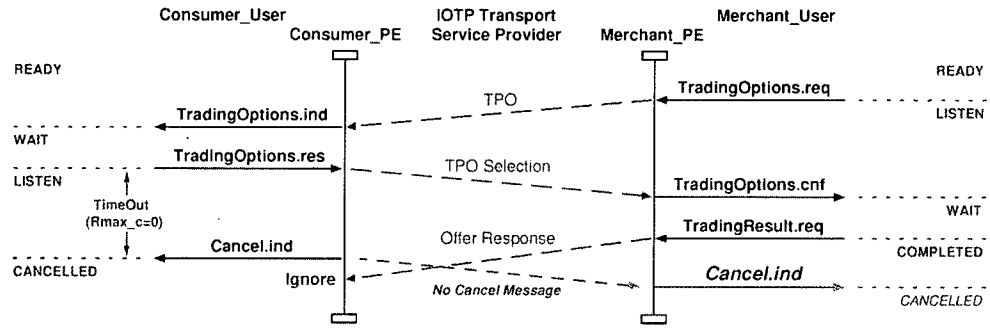
(c) Sequence (*TOreq, TOind, Cind_C, Cind_M*)



(d) Sequence (*Areq, Aind, Ares, Acnf, ARSreq, TOreq, ARSind, Cind_C, Cind_M*)



(e) Sequence (*Areq, Aind, Ares, Acnf, ARSreq, TOreq, TRreq, ARSind, Cind_C, Cind_M*)



(f) Sequence (*TOreq, TOind, TOres, TOcnf, Cind_C, Cind_M*)

Figure 12.16: Six time sequence diagrams of a Deposit transaction with cancellation showing six missing primitive sequences in the protocol.

- **Scenario (c)** corresponds to the sequence (*TOreq, TOind, Cind_C, Cind_M*) defined by nodes (0-2,17,11) in FSA 12.14. In this scenario where a (standalone) TPO Message is used, the Consumer_PE enters the `WAIT` state for *TOres* to be submitted by the user upon occurrence of *TOind*. Similarly, either *Cind_C* following *Cind_M* or *Cind_C* being followed by *Creq_M* or *Cind_M* cannot occur, given that the Consumer_PE is in the `WAIT` state. The corresponding sequences are those ending with *Cind_C* from node 18 to 11, and those involving *Cind_C* from node 2 to 17 in the FSA.

- **Scenario (d)** corresponds to the sequence (*Areq, Aind, Ares, Acnf, ARSreq, TOreq, ARSind, Cind_M, Cind_C*) defined by nodes (0,21-23,28-30,33,34,11) in FSA 12.14. This is a representative example of the two sequences ending with *Cind_C* from node 34 to 11. In this scenario, the Consumer_PE holds on processing the (combined) Authentication Status & TPO Message for delivery of *TOind* after *ARSind* to the user. However, it is not possible for *Cind_C* to occur when the Consumer_PE is in the `HOLD` state, unless the Consumer_PE receives a Cancel Message from the Merchant_PE, which it does not. The same is true for the two sequences involving *Cind_C* from node 33 to 17 in the FSA.

- **Scenario (e)** shows the sequence (*Areq, Aind, Ares, Acnf, ARSreq, TOreq, TRreq, ARSind, Cind_C, Cind_M*) defined by nodes (0,21-23,28-32,10,11) in FSA 12.14. The occurrence of *Cind_C* is not defined in the protocol when the Consumer_PE enters the

HOLD state upon the occurrence of *ARSind*. Also, after the occurrences of *TRreq*, *ARSind* and then *TOind*, the Consumer_PE will remain in the HOLD state for delivery of *TRind* to the user. *Cind_C* cannot occur in this case, and the corresponding sequence is defined by nodes (0,21-23,28-32,5,10,11). Similarly, when no authentication is required, the sequence defined by nodes (0,1,19,5,10,11) is not possible in the protocol.

- **Scenario (f)** captures the sequence (*TOreq, TOind, TOres, TOcnf, TRreq, Cind_C, Cind_M*) defined by nodes (0-5,10,11) in FSA 12.14. In this scenario, the Consumer_PE enters the LISTEN state after the occurrence of *TOres*, waiting for the Offer Response Message from the Merchant_PE. Therefore, *Cind_C* can occur upon a timeout without message retransmission (RCmax_c=0). Since the Merchant_PE is in the COMPLETED state, *Cind_M* cannot occur in the protocol, unless the Merchant_PE receives a Cancel Message from the Consumer_PE, which it does not.

Primitive sequences containing node 7 in FSA 12.14 represent the Deposit transaction service cancelled during the Payment phase. After the Offer phase is complete, the Merchant cannot cancel the transaction but can be notified of transaction cancellation by the Consumer sending a Cancel Message. As a result, only *Cind_M* can occur and it always occurs after *Creq_C* or *Cind_C* in each of these sequences. Also, no *Cancel* primitive event is allowed at the Payment Handler side after a *PRSreq* primitive has been submitted. Below, we divide these sequences into two groups, depending on whether or not *PRSreq* has occurred.

- One group of sequences are those involving *PRSreq* from node 8 to 13, node 9 to 10, or node 12 to 11. These correspond to the Deposit transaction services cancelled between the Consumer and Merchant after *PRSreq* has occurred. IOTP specifies that the Consumer_PE only sends the Cancel Message to the Merchant_PE, but not the PHandler_PE, after being delivered a *PRSind* primitive and before submitting the next non-*Cancel* primitive. Therefore, the sequences in this group are not possible in the protocol.

- The other group, which comprises the rest of the sequences, does not involve *PRSreq*. These correspond to the Deposit transaction service cancelled by both the Consumer and Payment Handler before *PRSreq* occurs, which the Merchant is also aware of. Within the current design of IOTP [36], the Merchant_PE is not informed of transaction cancellation between the Consumer and Payment Handler, and therefore all the sequences in this group are not implemented by the protocol.

12.5.3 Purchase Transaction

Table 12.6 summarises the statistics of the four FSA representing the service language, the protocol language, the illegal sequences and the missing sequences in the protocol for a Purchase transaction with cancellation.

FSA	*States*	*Arcs*	*Final States*	*Sequences*	*Longest*	*Shortest*
Service Language	54	170	4	580	21	2
Protocol Language	77	199	14	510	20	2
Illegal Sequences in the Protocol	55	117	8	332	20	2
Missing Sequences in the Protocol	54	129	1	402	21	4

Table 12.6: Statistics of four FSA for a Purchase transaction with cancellation.

A Purchase transaction without delivery service has the same transaction procedure as a Deposit transaction. Since we have already examined the Deposit transaction in Section 12.5.2, the verification of the Purchase transaction can be simplified by excluding those sequences that do not include delivery. As a result, we obtain two simplified FSA for the illegal sequences and missing sequences in the protocol for the Purchase transaction with delivery and cancellation. The two FSA have a common part which is represented by the FSA shown in Figure 12.17. This FSA, denoted by FSA 12.17, specifies the subsequences of a standalone Offer phase (starting with *Areq* from node 0 to 34) and an Authentication phase followed by an Offer phase (starting with *TOreq* from node 0 to 1) in any successful Payment-related transaction. The remaining subsequences (i.e., those for payment and delivery services) that are part of the illegal sequences or the missing sequences in the protocol for a Purchase transaction with delivery, is represented by the two FSA shown in Figure 12.18 and 12.19, respectively. We use FSA 12.18 and FSA 12.19 to denote these two FSA. Hence, the FSA specifying the illegal sequences in the protocol, can be obtained by concatenating the sequences accepted by FSA 12.17 and those accepted by FSA 12.18, and the FSA for the missing sequences in the protocol is a result of concatenating the sequences accepted by FSA 12.17 and those accepted by FSA 12.19. The following discussion focuses on the subsequences for payment and delivery services, as defined by FSA 12.18 and FSA 12.19.

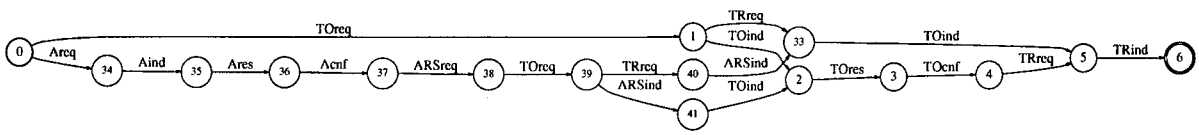


Figure 12.17: The FSA specifying the subsequences of a standalone Offer phase and an Authentication phase followed by an Offer phase in any successful Payment-related transaction.

Illegal Sequences in the Protocol for the Cancelled Purchase with Delivery

The subsequences containing node 11 in FSA 12.18, either involve a *Cancel* primitive event occurring at the Consumer only, or have a *Cancel* primitive event occurring at both the Consumer and the Delivery Handler. They correspond to the Purchase transaction service with delivery provided by the Delivery Handler, which is solely cancelled at the Consumer, or is cancelled between the Consumer and Delivery Handler (that the Merchant is not informed of). Therefore, all these subsequences end up with the undesired terminal states specified by six dead markings of the Purchase transaction, given by M_{2121}, M_{1400}, M_{2374} and M_{1653} (in Group 5), and M_{2608} and M_{1916} (in Group 7) in Table 11.3.

The rest of subsequences defined by FSA 12.18 are those containing node 19. They either involve a *Cancel* primitive event occurring at the Consumer only, or have a *Cancel* primitive event occurring at both the Consumer and the Payment Handler. These represent the Purchase transaction service with delivery provided by the Payment Handler, which is solely cancelled at the Consumer, or is cancelled between the Consumer and Payment Handler (that the Merchant is not informed of). Therefore, all these subsequences end up with the undesired terminal states specified by six dead markings of a Purchase transaction, given by M_{1156}, M_{566}, M_{1638} and M_{966} (in Group 5), and M_{2146} and M_{1425} (in Group 7) in Table 11.3.

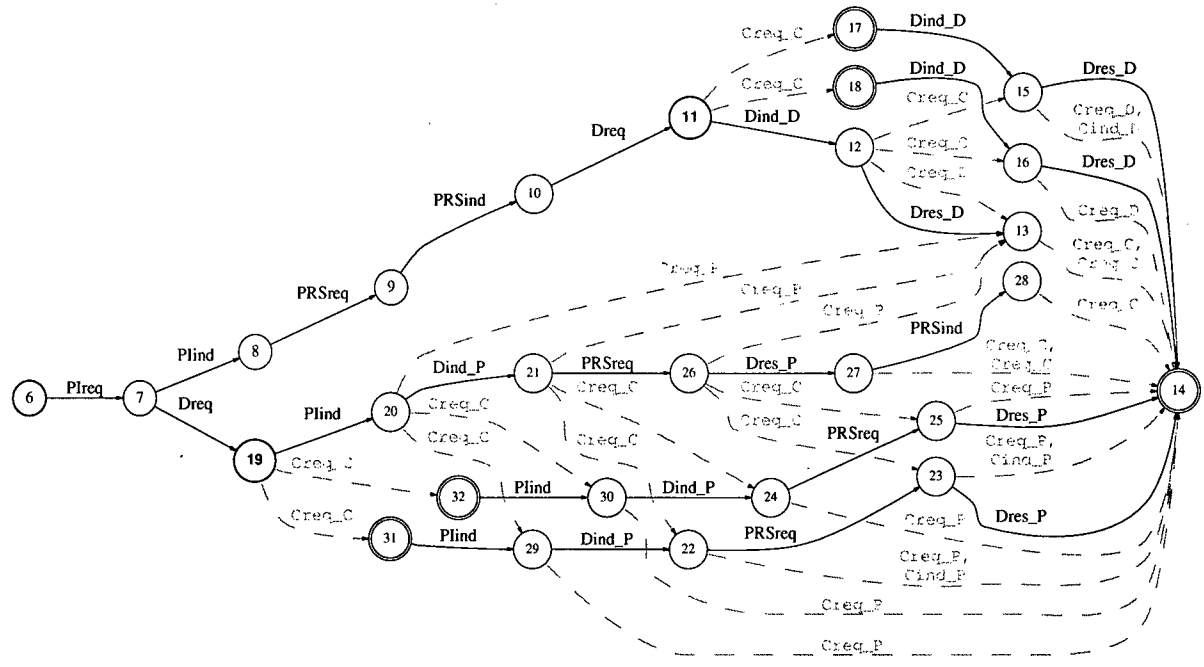


Figure 12.18: The FSA specifying the illegal subsequences for payment and delivery services in the protocol for a Purchase transaction with delivery and cancellation.

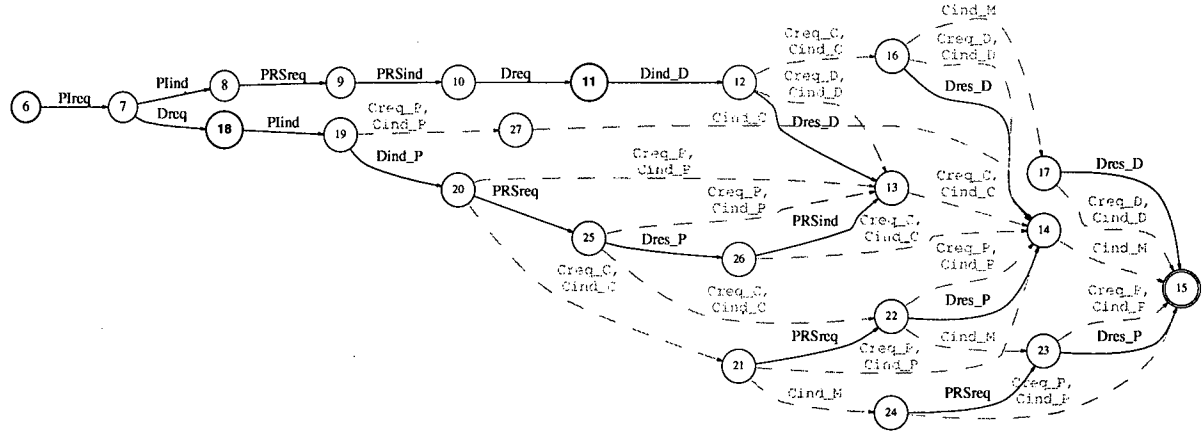


Figure 12.19: The FSA specifying the missing subsequences for payment and delivery services in the protocol for a Purchase transaction with delivery and cancellation.

Missing Sequences in the Protocol for the Cancelled Purchase with Delivery

The subsequences containing node 11 in FSA 12.19 represent the Purchase transaction service cancelled during the Delivery phase where the delivery is provided by the Delivery Handler. These subsequences can be divided into two groups, depending on whether *Dres_D* is involved. In RFC 2801, after *Dres_D* has occurred, no *Cancel* primitive is allowed at the Delivery Handler.

- Subsequences without *Dres_D* correspond to the cancelled transaction service between the Consumer and Delivery Handler that the Merchant is also aware of, *before Dres_D* occurs. In the current version of IOTP [36], the Merchant_PE is not informed of the transaction cancellation between the Consumer and Delivery Handler, resulting in the above subsequences not implemented by the protocol. Also, *Cind_D* cannot occur before *Creq_C* in the protocol, once the DHandler_PE enters the `WAIT` state on occurrence of *Dind_D* from node 11 to 12 in FSA 12.19.

- Subsequences involving *Dres_D* represent the cancelled transaction service between the Consumer and Merchant, *after Dres_D* has occurred. In IOTP [36], the Consumer_PE will not send a Cancel Message to the Merchant_PE after a Delivery document exchange has started, causing the above sequences to be missing in the protocol.

The same arguments apply to the cancelled Purchase transaction service with delivery provided by the Payment Handler. In this case the subsequences must contain node 18 in FSA 12.19 and either involves *Dres_P* or not.

12.5.4 Value Exchange Transaction

Table 12.7 summarises the statistics of the four FSA representing the service language, the protocol language, the illegal sequences and the missing sequences in the protocol for the Value Exchange transaction with cancellation.

FSA	*States*	*Arcs*	*Final States*	*Sequences*	*Longest*	*Shortest*
Service Language	42	130	3	364	21	2
Protocol Language	59	155	9	326	20	2
Illegal Sequences in the Protocol	37	74	4	156	20	2
Missing Sequences in the Protocol	42	89	1	194	21	4

Table 12.7: Statistics of four FSA for a Value Exchange transaction with cancellation.

In terms of transaction procedure, the Value Exchange transaction extends the Deposit transaction with a second Payment phase. The verification of Value Exchange can therefore be simplified by excluding the sequences for Deposit in Section 12.5.2. This results in two simplified FSA for the illegal and the missing sequences in the protocol for the Value Exchange transaction service that is cancelled during the second Payment phase. Again, the two FSA have a common part as represented by FSA 12.17. The remaining subsequences (i.e., those for the cancelled payment service in the second Payment phase) that are part of the illegal sequences or the missing sequences in the protocol, is represented by the FSA shown in Figure 12.20 (FSA 12.20) and the FSA in Figure 12.21 (FSA 12.20), respectively. Also note that the subsequence defined by nodes 6 to 10 in these two FSA represents a successfully completed payment service in the first Payment phase.

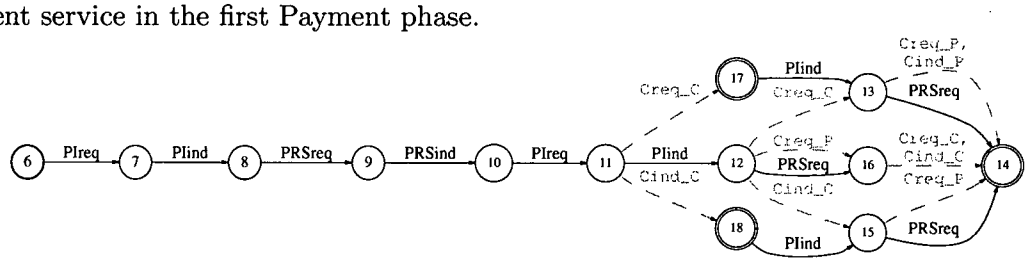


Figure 12.20: The FSA specifying the illegal subsequences for the payment service in the protocol for a Value Exchange transaction service that is cancelled during the second Payment phase.

The FSA for the illegal sequences in the protocol is a result of concatenating FSA 12.17 and FSA 12.20. Inspection of these sequences is similar to the inspection of the sequences containing node 7 in FSA 12.13 for Deposit (see Section 12.5.2). It shows that all the sequences end up with the undesired terminal states represented by four dead markings of a Value Exchange transaction, given by M_{1884} and M_{1281} (in Group 5), and M_{2066} and M_{1482} (in Group 7) in Table 11.3.

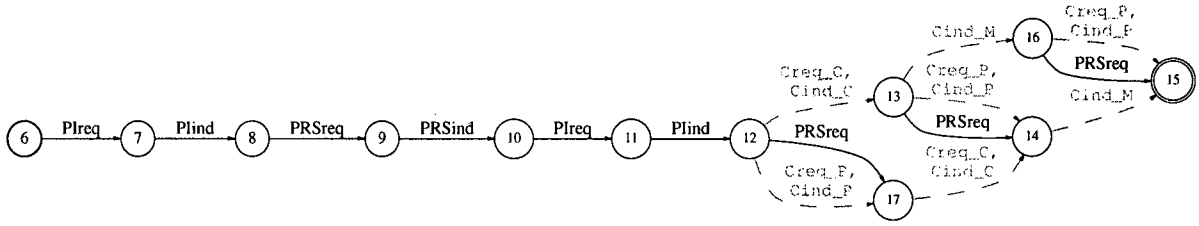


Figure 12.21: The FSA specifying the missing subsequences for the payment service in the protocol for a Value Exchange transaction service that is cancelled during the second Payment phase.

The FSA for the missing sequences in the protocol is a result of concatenating FSA 12.17 and FSA 12.21, and inspection of these sequences is similar to the inspection of the sequences containing node 7 in FSA 12.14 for Deposit.

12.5.5 Discussion of the Results

The language comparison results for a transaction with cancellation show that there are both illegal and missing primitive sequences in IOTP. As discussed in Sections 12.5.2 to 12.5.4, all the illegal sequences in the protocol result from some of the dead markings that represent undesired terminal states of the transactions. This means that they are caused by Problems 1 and 2 stated in Sections 11.3.2 and 11.3.3. As to the missing sequences in the protocol, we have inspected in detail their typical examples for Authentication and Deposit transactions in Sections 12.5.1 and 12.5.2, respectively. Those detected for the rest of the Payment-related transactions can be analysed in a similar way, and are briefly described in Sections 12.5.3 and 12.5.4.

Inspection of the missing sequences in the protocol reveals that no timer is used by a TRole_PE that is waiting for its user to submit the next primitive in IOTP. As a result, a *Cancel.ind* primitive cannot occur at the TRole_PE in the `WAIT` state or the `HOLD_WAIT` state, unless a *Cancel Message* is received from the peer TRole_PE. Furthermore, the peer TRole_PE will only send a Cancel Message on receipt of a *Cancel.req* from its user, whereas no Cancel Message will be sent upon the occurrence of a *Cancel.ind* at the peer TRole_PE. Hence, the occurrence of *Cancel.ind* at a TRole_PE in either of the `WAIT` and `HOLD_WAIT` states, requires the peer TRole_PE to submit a *Cancel.req* primitive. However, we should allow a TRole_PE that is waiting for its user to submit the next primitive (e.g, in the `WAIT` or the `HOLD_WAIT` state), to initiate a *Cancel.ind*, irrespective of whether or not the peer TRole_PE has submitted a *Cancel.req* primitive.

Therefore, the missing sequences in IOTP for transactions with cancellation are necessary in providing the required service defined in IOTS. These are caused by the lack of synchronization upon time-out at a TRole_PE within the current version of IOTP [36]. Given that the underlying medium is reliable, these missing sequences can then be implemented in the protocol, by introducing a timer on user responses and allowing a Cancel Message to be sent when a *Cancel.ind* occurs upon time-out. We shall discuss this in detail in Chapter 13.

Chapter 13

A Revised Internet Open Trading Protocol and its Verification

The state space and language analysis of IOTP in Chapters 11 and 12 revealed incompleteness in the current design of IOTP [36]. In this chapter, we propose solutions to remove the detected deficiencies from IOTP based on our understanding of the protocol operation, and implement the selected solutions by incorporating them into the IOTP CPN model. The analysis of the revised IOTP is another topic of this chapter. It involves using state space and language analysis to verify the properties and the protocol against its service for the revised IOTP. The intent is to show the changes suggested do fix the problems found in IOTP, and that the revised IOTP is functionally correct, for a given set of parameter values.

Section 13.1 summarises the three problems found in IOTP. Sections 13.2 to 13.4 present and evaluate a set of solutions to these three problems. For each problem, only one solution is selected in Section 13.5 to be incorporated into the IOTP CPN model. Section 13.6 provides a brief description of the changes made to the CPN model. Finally, Section 13.7 discusses the analysis results obtained, proves the desired properties and verifies the revised IOTP against IOTS. The revised IOTP CPN and its analysis results are given in Appendix H.

13.1 Summary of Problems in IOTP

We summarise the three problems revealed in IOTP, which are referred to as Problems 1 to 3.

- Problem 1 is the *Merchant's "inventory problem"* caused by the Merchant not being informed that a transaction has been cancelled by the Consumer or Payment/Delivery Handler (see Section 11.3.2);
- Problem 2 is the *Consumer's "non-refundable payment problem"* as a result of a transaction being cancelled only at the Consumer side (see Section 11.3.3); and
- Problem 3 is the lack of a timer being set by a trading role protocol entity when waiting for a response from its user, and the lack of synchronization of cancellation when a time-out occurs that results in cancellation (see Section 12.5.5).

13.2 Solutions to Problem 1 and Discussion

In principle, the Merchant's "inventory problem" can be avoided by the Merchant being informed that the transaction has been cancelled by any other trading role. The following three solutions allow the Merchant to become aware of such transaction cancellation in a timely way.

13.2.1 Solution 1: Use of the Inquiry Transaction

We firstly consider the use of any suitable mechanisms that are available in IOTP. As introduced in Section 3.2.2, one of the Infrastructure transactions, known as *Inquiry*, can be used to check the status of an Authentication or Payment-related transaction. By using Inquiry transactions, the Merchant can obtain the current status of the transaction after the completion of the Offer document exchange. If the Merchant receives a response to the inquiry indicating that the Payment or the Delivery document exchange has been cancelled, the Merchant is then aware that the transaction has been cancelled, and can update its records accordingly.

13.2.2 Solution 2: Notification of the Transaction Status

A second solution is to notify the Merchant about the transaction status, so that the Merchant does not need to use the Inquiry transaction. A straightforward implementation could be to multicast the last IOTP message in a Payment, a Delivery, or a Payment-and-Delivery document exchange, from the sender trading role to the peer trading role and the Merchant. For example, the Payment Handler sends a Payment Response Message or a Cancel Message to both the Consumer and the Merchant, or the Delivery Handler sends a Delivery Response Message or an Error Message reporting a hard error to both the Consumer and the Merchant. With this solution, the Merchant will be notified once the transaction terminates (as the IOTP Transport Service (HTTP) is reliable).

13.2.3 Solution 3: "Back-End" System Operations

We could assume that communication between the Merchant, Payment Handler and Delivery Handler will be handled by *"back-end" systems*, which are outside the scope of IOTP. For example, the Merchant considers a Purchase transaction is complete upon receiving money from the Payment Handler and once the Delivery Handler ships the goods to the consumer that are in stock.

13.2.4 Discussion of Solutions

As a result of each of the above three solutions, the terminal state of the Merchant_PE given in the dead markings of Group 6 in Table 11.3 changes from `COMPLETED` to `CANCELLED`. This removes the inconsistency between the Merchant and Payment/Delivery Handler when the transaction is cancelled at the Consumer and the Payment/Delivery Handler sides, so that the Merchant's records can be updated. Below, we evaluate these three solutions.

An advantage of Solution 1 (the use of an IOTP Inquiry transaction to obtain the transaction's status information), is that it is within the current capabilities of IOTP. Unfortunately it also has two disadvantages concerning *cost* and *synchronization*. There is a cost in terms of complexity and efficiency in initiating a new transaction (Inquiry). The synchronization problem concerns the uncertainty of the timing of the Merchant initiating the Inquiry transaction, which is at what stage the Merchant decides to inquire, e.g., within or after a Payment-related transaction of interest. Also, an Inquiry transaction always occurs between two trading roles, and therefore only guarantees consistency between these two trading roles. Thus it may be necessary for the Merchant to issue two Inquiry transactions (one with the Payment Handler

and the other with the Delivery Handler) to gain a complete picture of what is happening. For these reasons we would recommend that the Inquiry transaction is only used as a recovery option (a fall back position), when other mechanisms fail.

Solution 2, that of conducting reliable multicast notification of the transaction status, appears to be an effective approach, which ensures that both the Consumer and the Merchant activated in the transaction enter the same terminal state upon transaction termination and thus the inconsistencies between them are all removed. The Merchant will now know the status of the transaction at the earliest opportunity. However, multicasting the last IOTP message, such as a Payment Response Message, sent from the Payment Handler to the Merchant, involves server-to-server communication between non-Consumer trading roles, which is currently not supported by IOTP.

Finally, Solution 3 concerns "back-end" system communications which are beyond the scope of Baseline IOTP. This could be the easiest solution because back-end systems are available and may already support the operations needed. If this is the solution tacitly adopted for IOTP, then the effect of these operations on IOTP should be clearly specified in the RFC, as part of its operating environment.

13.3 Solution to Problem 2 and Discussion

To prevent the "non-refundable payment problem" occurring, we need to make sure that the peer non-Consumer trading roles are informed of transaction cancellation initiated by the Consumer. At least the Merchant should be aware of that cancellation to avoid the inventory problem discussed above.

To find solutions, we firstly revisit all possible reasons given in Section 11.3.3 that cause cancellation at the Consumer side only. These can be summarised as:

1. The Consumer_PE detecting a hard error on receipt of the last response message from the PHandler_PE or DHandler_PE;
2. Local transaction cancellation by Consumer_PE upon time-out; and
3. The Consumer_PE sending a Cancel Message before receiving the last response message from the PHandler_PE or DHandler_PE.

Hence, our solutions to keep the Merchant aware of the transaction cancellation issued by the Consumer at any time, can be seen as follows.

1. With respect to the above reasons 1 and 3, we suggest that the Consumer_PE multicast the Error Message reporting a hard error (in reason 1) or the Cancel Message (in reason 3) to both the Merchant_PE and the PHandler_PE (during a Payment or a Payment-and-Delivery document exchange), or to both the Merchant_PE and the DHandler_PE (during a Delivery document exchange). As a result, no matter whether or not the PHandler_PE or DHandler_PE ignores the message, the Merchant_PE being notified about the transaction cancellation is guaranteed.

2. Reason 3 reveals a lack of synchronization. Since the transaction is only cancelled *locally* by the Consumer, it is not possible for any peer non-Consumer_PE (especially a Merchant_PE, see Figure 11.11 (b)) to be aware of such cancellation. Hence, we suggest that

the Consumer_PE generate and multicast a Cancel Message to both the Merchant_PE and the PHandler_PE, or to both the Merchant_PE and the DHandler_PE, once the timer has expired and the maximum number of message retransmissions has been reached.

These two solutions should be considered together to avoid transaction cancellation occurring at the Consumer only. The multicast of an Error Message reporting a hard error or a Cancel Message, is applied to transaction cancellation issued by the Consumer at any time after the Payment document exchange has been initiated. This ensures that the Merchant's inventory problem will not occur whenever the Consumer cancels the transaction.

13.4 Solutions to Problem 3 and Discussion

13.4.1 Solution 1: Use of the Cancel Message

The lack of synchronization relating to cancellation can be removed, by allowing the local trading role entity to send a Cancel Message to the peer entity on the occurrence of *Cancel.ind* as a result of a time-out for message retransmission. With this solution, it is possible for a TRole_PE, which is waiting for its user to submit an IOTS primitive, to receive a Cancel Message and hence deliver a provider initiated *Cancel.ind* to its user, once the transaction has been cancelled at the peer TRole_PE.

13.4.2 Solution 2: Setting Timer for User Response

A second solution is to define a new timer at each TRole_PE to record the time used by that TRole_PE when waiting for the response from its user, i.e., an IOTS primitive to be submitted by the user. When this timer expires, the TRole_PE will initiate a *Cancel.ind* to the user.

13.4.3 Solution 3: Inclusion of Time Value with Message

The last solution is to include a time value with the last retransmitted message from a TRole_PE. This time value specifies when the transaction will be cancelled by that TRole_PE if there is still no response from the peer entity. Hence, with such information, the peer TRole_PE that is waiting for a response from its user, will initiate a *Cancel.ind* if no response is received before the specified time value is reached.

13.4.4 Discussion of Solutions

With each of the above three solutions, it seems possible for *Cancel.ind* to occur at a TRole_PE that is waiting for a response from its user, irrespective of whether or not the peer TRole_PE has submitted a *Cancel.req* primitive. Below, we evaluate these three solutions.

In Solution 1, a Cancel Message is used to synchronize the transaction operation between two TRole_PEs, when a timer for message retransmission expires at one entity. Since the Cancel Message is already defined and used in IOTP, this solution is easy to implement.

In Solution 2, a *Cancel.ind* may occur at a TRole_PE upon time-out for its user's response. The occurrence of *Cancel.ind* at the TRole_PE in such case is independent of the occurrence of *Cancel.ind* at the peer TRole_PE. This solution guarantees that a transaction service can

always terminate at a TRole_PE. However, the implementation of this solution is more complex than that of Solution 1, because we need to set a new timer at each TRole_PE.

With Solution 3, the number of the messages used does not change, and therefore the communication traffic will remain as before. However, this solution could fail if the last retransmitted message has a hard error. Also, it can make the protocol inefficient because of the delay in message delivery. For example, the local TRole_PE sends the response message before the specified time value is reached, but it takes longer than expected for this message to be delivered to the other end. The peer TRole_PE initiates a *Cancel.ind* when the time value is reached and no response is received, and ignores any message received afterwards. Since no message is received, the local TRole_PE keeps on retransmitting the above response message, and will not initiate a *Cancel.ind* until the retransmission counter reaches its maximum value.

13.5 Solutions Chosen

The selected solution to Problem 1 (Merchant's inventory problem) is Solution 2 which is implemented by multicasting the last IOTP message of the ongoing Payment, Delivery, or Payment-and-Delivery document exchange, from the sender trading role to the peer trading role and the Merchant. With this solution, we need to introduce server-to-server communication into IOTP, for supporting message delivery from the Payment Handler or Delivery Handler to the Merchant.

Solution to Problem 2 (Consumer's non-refundable payment problem) has two steps. Firstly, when the timer expired after the message retransmission counter has reached its limit, the Consumer_PE should generate a Cancel Message and send it to the peer non-Consumer_PE(s). Secondly, if the Consumer_PE has generated a Cancel Message or an Error Message reporting a hard error, the message should then be multicast to the Merchant_PE and the PHandler_PE or DHandler_PE that is involved in the ongoing document exchange. This, however, can be considered as one example of the above solution to Problem 1, where the Cancel Message or Error Message is sent by the Consumer as the sender trading role.

We choose Solution 1 to Problem 3 due to its simplicity as well as the possibility of its implementation within the scope of IOTP. Thus, a TRole_PE that initiates *Cancel.ind* to its user upon time-out for message retransmission, must send a Cancel Message to the peer TRole_PE to notify it that the transaction has been cancelled.

13.6 Changes to the IOTP CPN

Based on the selected solutions in Section 13.5, we present a set of changes to the IOTP CPN to reflect the suggested changes to IOTP version 1.0 [36]. The changes are only summarised as the complete revised IOTP CPN is given in Appendix H, where the newly added transitions are drawn with thicker borders and the modified arc inscriptions and guards are shown in bold. We divide the description of these changes into *three* subsections related to the selected solution to each of the three problems identified in IOTP.

13.6.1 Keeping the Merchant Informed of the Transaction Status

The changes made to the IOTP CPN to keep the Merchant informed of the result of a Payment, a Delivery or a Payment-and-Delivery document exchange and the cancellation of the transaction are as follows.

1. Changes on the pages Payment_P (Figure H.6), PayDelivery_P (Figure H.7) and Delivery_D (Figure H.8):

- On page Payment_P, the arc from transition SndPayRsp to place Transport has an inscription using the function Uni_Multicast defined in Table 13.1. It models the PHandler_PE (sndr=PHandler) sending Payment Response (m=[PayRespBlk] with message identifier id given by Add sid) to both the Consumer_PE (rcvr=Consumer) and the Merchant_PE.

```
1  fun Uni_Multicast ( sndr:TradingRole, rcvr:TradingRole, m:IotpMsg, id:MsgId, q:MsgQueue )
2    = if ( sndr=Merchant orelse rcvr=Merchant )
3      then 1'( sndr, rcvr, q^^[ (m, id) ] )
4      else 1'( sndr, rcvr, q^^[ (m, id) ] ) ++ 1'( sndr, Merchant, [ (m, id) ] );
```

Table 13.1: Definition of function Uni_Multicast in the revised IOTP CPN.

- Similarly, on page PayDelivery_P, the arc from transition SndPayDlvRsp to place Transport, models the PHandler_PE (sndr=PHandler) sending Payment Response & Delivery Response (m=[PayRespBlk, DeliveryRespBlk] with id given by Add sid) to both the Consumer_PE (rcvr=Consumer) and the Merchant_PE.

- Similarly, on page Delivery_D, the arc from transition SndDlvRsp to place Transport, models the DHandler_PE (sndr=DHandler) sending Delivery Response (m=[DeliveryRespBlk] with id given by Add sid) to both the Consumer_PE (rcvr=Consumer) and Merchant_PE.

2. Changes on the pages BrdDepOffer_C (Figure H.1), BrdIndOffer_C (Figure H.3), BrdDepOffer_M (Figure H.2), BrdIndOffer_M (Figure H.4) and Payment_C (Figure H.5):

- It is not appropriate for the Merchant_PE to enter the state `COMPLETED` upon sending the Offer Response Message, because the Merchant will not complete the transaction until its final result is received. In this case, the state `FINISHED` (see Table 10.2) will be used to replace the current `COMPLETED` state for the Merchant_PE at the end of the Offer document exchange, and `COMPLETED` will be used when the Merchant receives the *final result* (i.e., the last response message in the transaction) indicating the transaction has been completed successfully. Accordingly, the state value `COMPLETED` in the arc inscriptions on the pages BrdDepOffer_M and BrdIndOffer_M, are changed to FINISHED.

- Table 13.2 defines a new colour set DelivExch (line 1), modelling the *DelivExch* attribute of a *Delivery Component* in an *Offer Response Block* (line 5). It has two values, True or False, indicating whether there is a Delivery document exchange in the transaction. With this information, the Merchant can determine if a Payment Response Message from the Payment Handler is the last response message in the transaction. Accordingly, OfferRespBlk in the arc inscriptions on the pages BrdDepOffer_C, BrdIndOffer_C, BrdDepOffer_M and BrdIndOffer_M, is replaced by OfferRespBlk(dlv). A new place Delivery typed by the colour set DelivExch is added on the pages BrdDepOffer_C, BrdIndOffer_C and Payment_C,

to store the DelivExch information so that the Consumer will know whether to continue or to terminate the transaction at the end of the Payment document exchange (i.e., whether transition RcvPayRsp_cont or RcvPayRsp_stop to occur on page Payment_C).

```
1  color DelivExch = with True | False;
2  var dlv: DelivExch;
3  color TradingBlk = union TransRefBlk:IotpTransType +
4                           AuthReqBlk + AuthRespBlk + AuthStatusBlk +
5                           TpoBlk + TpoSelectionBlk + OfferRespBlk:DelivExch +
6                           PayReqBlk + PayExchBlk + PayRespBlk +
7                           DeliveryReqBlk + DeliveryRespBlk +
8                           ErrorBlk:TwoAttrOfErrComp +
9                           CancelBlk;
```

Table 13.2: Definition of colour set DelivExch in the revised IOTP CPN.

- A new transition, namely RcvPayDlvRsp, is added on both BrdDepOffer_M and BrdInd-Offer_M pages. This models that the Merchant_PE enters the state COMPLETED from the state FINISHED on receipt of the last response message in a Payment-related transaction. The transition RcvPayDlvRsp has a guard defined by the function IsPayDlvRsp which determines if a message is Payment Response, Delivery Response, or combined Payment Response & Delivery Response. The inscription on the arc from RcvPayDlvRsp to place M_BDOfr or M_BIOfr has the function NotLastRsp which determines if a response message is the last message in the transaction. Table 13.3 defines these two functions.

```
1  fun IsPayDlvRsp ( [ PayRespBlk ] ) = true
2    | IsPayDlvRsp ( [ DeliveryRespBlk ] ) = true
3    | IsPayDlvRsp ( [ PayRespBlk, DeliveryRespBlk ] ) = true
4    | IsPayDlvRsp ( _ ) = false
5  fun NotLastRsp ( Purchase, [ PayRespBlk ], [ OfferRespBlk(True) ] ) = true
6    | NotLastRsp ( ValueExchange, [ PayRespBlk ], [ OfferRespBlk(False) ] ) = true
7    | NotLastRsp ( _ , _ , _ ) = false
```

Table 13.3: Functions IsPayDlvRsp and NotLastRsp in the revised IOTP CPN.

3. Changes on the pages Cancellation_C (Figure H.9) and Cancellation_NC (Figure H.10):

   - On page Cancellation_C, the arc from transition SndCnl to place Transport has an inscription using the function Uni_Multicast. It models that: 1) the Consumer_PE sends a Cancel Message to the Merchant_PE when role=Merchant; or otherwise 2) the Consumer_PE multicasts a Cancel message to the Merchant_PE and the PHandler_PE or DHandler_PE represented by role.

   - Similarly, on page Cancellation_NC, the arc from transition SndCnl to place Transport models that: 1) the Merchant_PE (role=Merchant) sends a Cancel Message to the Consumer_PE only; or otherwise 2) the PHandler_PE or DHandler_PE (represented by role) multicasts a Cancel message to both the Consumer_PE and the Merchant_PE.

   - Also, on page Cancellation_NC, receiving a Cancel Message at a non-Consumer_PE is modelled by two transitions: RcvCnl_P_D for the PHandler_PE or DHandler_PE and RcvCnl_M for the Merchant_PE. RcvCnl_P_D cannot occur at the PHandler_PE or DHandler_PE in

the COMPLETED or CANCELLED state, while RcvCnl_M cannot occur at the Merchant in the CANCELLED state.

4. Changes on the pages ErrHandling_C (Figure H.11) and ErrHandling_NC (Figure H.12):
   - On page ErrHandling_C, the arc from transition SndMsgRptHrdErr_Stop to place Transport, with an inscription using the function Uni_Multicast, models that: 1) the Consumer_PE sends an Error Message reporting a hard error to the Merchant_PE when role=Merchant; or otherwise 2) the Consumer_PE multicasts the message to the Merchant_PE and the PHandler_PE or DHandler_PE represented by role.
   - Similarly, on page ErrHandling_NC, the arc from transition SndMsgRptHrdErr_Stop to place Transport models that: 1) the Merchant_PE (role=Merchant) sends a Cancel Message to the Consumer_PE only; or otherwise 2) the PHandler_PE or DHandler_PE (represented by role) multicasts a Cancel message to both the Consumer_PE and the Merchant_PE.
   - Also, on page ErrHandling_NC, receiving an Error Message reporting a hard error at a non-Consumer_PE is modelled by two transitions: Stop_4_P_D for the PHandler_PE or DHandler_PE and Stop_4_M for the Merchant_PE. Stop_4_P_D cannot occur at the PHandler_PE or DHandler_PE in the COMPLETED or CANCELLED state, while Stop_4_M cannot occur at the Merchant in the CANCELLED state.

With the above changes to IOTP, the undesired terminal states represented by the dead markings in ***Group 5*** in Table 11.3, and the illegal sequences ending up with these undesired terminal states in the protocol language, should not be present. Instead, the corresponding terminal states, as defined in Table 13.4, should be introduced. As highlighted, the only change in each terminal state is that the Merchant_PE enters the CANCELLED state instead of the previous COMPLETED state. This indicates that the Merchant will be informed of transaction cancellation between the Consumer and the Payment Handler or Delivery Handler.

Undesired terminal states (dead markings in Group 5 in Table 11.3)				*Desired terminal states (from the IOTP CPN with changes in Section 13.6.1)*			
Consumer	*Merchant*	*PHandler*	*DHandler*	*Consumer*	*Merchant*	*PHandler*	*DHandler*
CNL_{Pay1}	$\mathbf{CMP}_{BDOfr, BIOfr}$	CNL_{NoExch}	-	CNL_{Pay1}	$\mathbf{CNL}_{BDOfr, BIOfr}$	CNL_{NoExch}	-
CNL_{PayDlv}	$\mathbf{CMP}_{BDOfr, BIOfr}$	CNL_{NoExch}	-	CNL_{PayDlv}	$\mathbf{CNL}_{BDOfr, BIOfr}$	CNL_{NoExch}	-
CNL_{Pay1}	$\mathbf{CMP}_{BDOfr, BIOfr}$	CNL_{Pay1}	-	CNL_{Pay1}	$\mathbf{CNL}_{BDOfr, BIOfr}$	CNL_{Pay1}	-
CNL_{Pay2}	$\mathbf{CMP}_{BDOfr, BIOfr}$	CNL_{Pay2}	-	CNL_{Pay2}	$\mathbf{CNL}_{BDOfr, BIOfr}$	CNL_{Pay2}	-
CNL_{PayDlv}	$\mathbf{CMP}_{BDOfr, BIOfr}$	CNL_{PayDlv}	-	CNL_{PayDlv}	$\mathbf{CNL}_{BDOfr, BIOfr}$	CNL_{PayDlv}	-
CNL_{Deliv}	$\mathbf{CMP}_{BDOfr, BIOfr}$	CMP_{Pay1}	CNL_{NoExch}	CNL_{Deliv}	$\mathbf{CNL}_{BDOfr, BIOfr}$	CMP_{Pay1}	CNL_{NoExch}
CNL_{Deliv}	$\mathbf{CMP}_{BDOfr, BIOfr}$	CMP_{Pay1}	CNL_{Deliv}	CNL_{Deliv}	$\mathbf{CNL}_{BDOfr, BIOfr}$	CMP_{Pay1}	CNL_{Deliv}

Table 13.4: Terminal states for revised IOTP updated from those in Group 5 in Table 11.3.

13.6.2 Avoiding Cancellation at the Consumer Only

Secondly, the IOTP CPN is modified to prevent a Payment-related transaction being cancelled at the Consumer side only. With the changes in Section 13.6.1, we only need to update the

ErrHandling_C page (Figure H.11), to include the Consumer_PE sending or multicasting a Cancel Message when the maximum number of time-outs has occurred.

- The arc from transition Stop_1 to place Transport, with an inscription of (Consumer, role, q), models the current communication queue in which the message is delivered from the Consumer_PE to the peer non-Consumer_PE represented by role.
- The arc from place Transport to transition Stop_1, with an inscription using the function Uni_Multicast, models that: 1) the Consumer_PE sends a Cancel Message to the Merchant_PE when role=Merchant; or otherwise 2) the Consumer_PE multicasts a Cancel message to the Merchant_PE and the PHandler_PE or DHandler_PE represented by role.

With the above changes to IOTP, the undesired terminal states represented by the dead markings in ***Group 6*** and ***Group 7*** in Table 11.3, should be correspondingly updated to those defined in Tables 13.5 and 13.6, respectively. Again, the only change in each terminal state is that the Merchant_PE enters the state `CANCELLED` instead of `COMPLETED`. This avoids the transaction being cancelled at the Consumer_PE only. In Table 13.5, the updated terminal states in the last two rows are already defined by the dead markings in ***Group 3*** in Table 11.3. In Table 13.6, the updated terminal state in the first row is defined by the dead markings in ***Group 4*** in Table 11.3. Also, the illegal sequences ending up with the above undesired terminal states should be removed from the protocol language.

Undesired terminal states (dead markings in Group 6 in Table 11.3)			*Desired terminal states (from the IOTP CPN with changes in Section 13.6.2)*		
Consumer	***Merchant***	*P/DHandler*	*Consumer*	***Merchant***	*P/DHandler*
CNL_{NoExch}	$\mathbf{CMP}_{BIOfr}$	-	CNL_{NoExch}	$\mathbf{CNL}_{BIOfr}$	-
CNL_{Auth}	$\mathbf{CMP}_{BIOfr}$	-	CNL_{Auth}	$\mathbf{CNL}_{BIOfr}$	-
$CNL_{BDOfr,\ BIOfr}$	$\mathbf{CMP}_{BDOfr,\ BIOfr}$	-	$CNL_{BDOfr,\ BIOfr}$	$\mathbf{CNL}_{BDOfr,\ BIOfr}$	-

Table 13.5: Terminal states for revised IOTP updated from those in Group 6 in Table 11.3.

Undesired terminal states (dead markings in Group 7 in Table 11.3)				*Desired terminal states (from the IOTP CPN with changes in Section 13.6.2)*			
Consumer	***Merchant***	*PHandler*	*DHandler*	*Consumer*	***Merchant***	*PHandler*	*DHandler*
CNL_{Pay1}	$\mathbf{CMP}_{BDOfr,\ BIOfr}$	CMP_{Pay1}	-	CNL_{Pay1}	$\mathbf{CNL}_{BDOfr,\ BIOfr}$	CMP_{Pay1}	-
CNL_{Pay2}	$\mathbf{CMP}_{BDOfr,\ BIOfr}$	CMP_{Pay2}	-	CNL_{Pay2}	$\mathbf{CNL}_{BDOfr,\ BIOfr}$	CMP_{Pay2}	-
CNL_{PayDlv}	$\mathbf{CMP}_{BDOfr,\ BIOfr}$	CMP_{PayDlv}	-	CNL_{PayDlv}	$\mathbf{CNL}_{BDOfr,\ BIOfr}$	CMP_{PayDlv}	-
CNL_{Deliv}	$\mathbf{CMP}_{BDOfr,\ BIOfr}$	CMP_{Pay1}	CMP_{Deliv}	CNL_{Deliv}	$\mathbf{CNL}_{BDOfr,\ BIOfr}$	CMP_{Pay1}	CMP_{Deliv}

Table 13.6: Terminal states for revised IOTP updated from those in Group 7 in Table 11.3.

13.6.3 Synchronizing Transaction Cancellation upon Time-Out

Thirdly, the IOTP CPN is modified to synchronize transaction cancellation between trading role entities upon time-out for message retransmission. With the changes in Sections 13.6.1 and 13.6.2, we only need to update the ErrHandling_NC page (Figure H.12), to include the non-Consumer_PE sending a Cancel Message to the Consumer_PE when the maximum number of time-outs has occurred. The changes to the ErrHandling_NC page given below are similar to those made on the ErrHandling_C page in Section 13.6.2.

- The arc from transition Stop_1 to place Transport, with an inscription of (role, Consumer, q), models the current communication queue in which the message is delivered from the non-Consumer_PE (represented by role) to the Consumer_PE.
- The arc from place Transport to transition Stop_1, with an inscription using the function Uni_Multicast, models that: 1) the Merchant_PE (role=Merchant) sends a Cancel Message to the Consumer_PE only; or otherwise 2) the PHandler_PE or DHandler_PE (represented by role) multicasts a Cancel message to both the Consumer_PE and the Merchant_PE.

With the above changes to IOTP and those in Sections 13.6.1 and 13.6.2, the missing sequences in the protocol for transactions with cancellation should now be implemented by the revised IOTP.

13.7 State Space and Language Analysis

We analyse the revised IOTP in a similar way to that described in Chapters 11 and 12. Design/CPN [176] and FSM [11] are used to calculate the state space and language analysis results for the revised IOTP CPN, respectively. The configurations analysed are those for all Authentication and Payment-related transactions with all permutations of the message retransmission counter parameters (RCmax_c and RCmax_nc) within the range of *0* to *3*. When RCmax_c or RCmax_nc is increased to *4*, calculating the state spaces of both the Purchase and the Value Exchange transactions exhaust the memory available (512 Mb RAM). This section presents the results of the state space analysis for proving the desired properties of the revised CPN model, and the results of the language analysis to verifying the revised IOTP against IOTS. Appendix H provides the Design/CPN state space reports and the FSM reports which contain the original data for analysing the revised IOTP.

13.7.1 Desired Terminal States and Deadlocks

Table 13.7 gives the statistics of the state space size, in terms of number of nodes, number of arcs and calculation time, for each configuration of the revised CPN analysed. Also included are the results from the inspection of the dead markings (see Appendix H for details), which either represent desired terminal states of the transaction or indicate deadlocks in the transaction. As Table 13.7 shows, all configurations analysed, terminated as desired, therefore proving Property 11.1. Inspection of the dead markings for each transaction also shows that they are independent of the message retransmission counter value within the range of *0* to *3*.

13.7.2 Livelocks

Table 13.8 lists the statistics of the SCC graph, in terms of number of SCC nodes (*SCCs*) and number of arcs, for each configuration. Also included is the number of livelocks (*LL*) present in the state space. A comparison with Table 13.7 shows that, for every configuration analysed, the SCC graph is of the same size as its corresponding state space. As listed in column *LL* in Table 13.8, there are no livelocks in the state space for each analysed configuration of the revised CPN, and thereby Property 11.2 has been proved.

Configuration			*State space size and dead marking inspection*				
Transaction	*RCmax_c*	*RCmax_nc*	*Nodes*	*Arcs*	*Time*	*Desired Terminal States*	*Deadlocks*
Authentication	0	0	39	80	0:00:00	4	0
	0	1	108	281	0:00:01	4	0
	0	2	233	692	0:00:01	4	0
	0	3	437	1417	0:00:03	4	0
	1	0	81	192	0:00:00	4	0
	1	1	220	614	0:00:01	4	0
	1	2	458	1414	0:00:03	4	0
	1	3	832	2764	0:00:06	4	0
	2	0	158	420	0:00:01	4	0
	2	1	409	1215	0:00:02	4	0
	2	2	813	2596	0:00:06	4	0
	2	3	1422	4803	0:00:12	4	0
	3	0	285	824	0:00:02	4	0
	3	1	707	2212	0:00:05	4	0
	3	2	1349	4442	0:00:11	4	0
	3	3	2276	7822	0:00:24	4	0
Purchase	0	0	2144	5243	0:00:13	85	0
	0	1	3790	9789	0:00:25	85	0
	0	2	5795	15695	0:00:44	85	0
	0	3	8328	23735	0:01:15	85	0
	1	0	6788	20235	0:00:57	85	0
	1	1	12695	37947	0:02:15	85	0
	1	2	19609	60342	0:04:23	85	0
	1	3	28272	89934	0:08:17	85	0
	2	0	16829	54500	0:03:36	85	0
	2	1	30933	102289	0:09:31	85	0
	2	2	47931	161437	0:19:34	85	0
	2	3	69068	237692	0:38:52	85	0
	3	0	34645	121293	0:12:15	85	0
	3	1	64025	227384	0:33:30	85	0
	3	2	99185	356279	1:11:31	85	0
	3	3	142499	518910	2:25:38	85	0
Deposit/ Withdrawal/ Refund	0	0	756	1679	0:00:04	39	0
	0	1	1493	3690	0:00:09	39	0
	0	2	2589	7061	0:00:17	39	0
	0	3	4213	12566	0:00:36	39	0
	1	0	2182	5695	0:00:13	39	0
	1	1	4567	12886	0:00:34	39	0
	1	2	8157	24760	0:01:26	39	0
	1	3	13496	43831	0:03:45	39	0
	2	0	4807	13930	0:00:37	39	0
	2	1	10418	32121	0:02:01	39	0
	2	2	18923	61672	0:06:09	39	0
	2	3	31567	108328	0:16:48	39	0
	3	0	9147	28491	0:01:35	39	0
	3	1	20296	66518	0:06:07	39	0
	3	2	37225	127349	0:20:27	39	0
	3	3	62308	221916	0:59:12	39	0
Value Exchange	0	0	1582	3907	0:00:09	63	0
	0	1	2862	7499	0:00:17	63	0
	0	2	4501	12451	0:00:31	63	0
	0	3	6668	19537	0:00:55	63	0
	1	0	6268	19163	0:00:46	63	0
	1	1	11488	36077	0:01:47	63	0
	1	2	17913	57674	0:03:33	63	0
	1	3	26087	86468	0:07:02	63	0
	2	0	17405	60082	0:03:26	63	0
	2	1	31938	111906	0:09:01	63	0
	2	2	49365	175089	0:18:26	63	0
	2	3	70931	255379	0:38:14	63	0
	3	0	39739	148479	0:14:00	63	0
	3	1	72815	274481	0:38:34	63	0
	3	2	111671	423287	1:17:09	63	0
	3	3	158681	605829	2:35:57	63	0

Table 13.7: Statistics for the state space size and dead markings for selected configurations of the revised IOTP CPN model.

RCmax		*Authentication*			*Purchase*			*Deposit/ Withdrawal/ Refund*			*Value Exchange*		
c	*nc*	*SCCs*	*Arcs*	*LL*	*SCCs*	*Arcs*	*LL*	*SCCs*	*Arcs*	*LL*	*SCCs*	*Arcs*	*LL*
0	0	39	80	0	2144	5243	0	756	1679	0	1582	3907	0
0	1	108	281	0	3790	9789	0	1493	3690	0	2862	7499	0
0	2	233	692	0	5795	15695	0	2589	7061	0	4501	12451	0
0	3	437	1417	0	8328	23735	0	4213	12566	0	6668	19537	0
1	0	81	192	0	6788	20235	0	2182	5695	0	6268	19163	0
1	1	220	614	0	12695	37947	0	4567	12886	0	11488	36077	0
1	2	458	1414	0	19609	60342	0	8157	24760	0	17913	57674	0
1	3	832	2764	0	28272	89934	0	13496	43831	0	26087	86468	0
2	0	158	420	0	16829	54500	0	4807	13930	0	17405	60082	0
2	1	409	1215	0	30933	102289	0	10418	32121	0	31938	111906	0
2	2	813	2596	0	47931	161437	0	18923	61672	0	49365	175089	0
2	3	1422	4803	0	69068	237692	0	31567	108328	0	70931	255379	0
3	0	285	824	0	34645	121293	0	9147	28491	0	39739	148479	0
3	1	707	2212	0	64025	227384	0	20296	66518	0	72815	274481	0
3	2	1349	4442	0	99185	356279	0	37225	127349	0	111671	423287	0
3	3	2276	7822	0	142499	518910	0	62308	221916	0	158681	605829	0

Table 13.8: Statistics for the SCC graph and livelocks for selected configurations of the revised IOTP.

13.7.3 Dead Transitions

The revised CPN model includes several changes to transitions in the original CPN model. Table 13.9 lists a number of transitions that are expected to be dead on the activated exchange level pages for each of the six transactions in the revised CPN model, as updated from those shown in Table 11.1,

No.	*Parameter Value*	*Dead Transition*	*Page*
1	Authentication transaction	HoldAuthSts	Authenticator
2	Payment-related transaction	SndAuthSts	
3	Purchase/Deposit/Withdrawal/Refund transaction	SndPayReq_2	Payment_C
4	Deposit/Withdrawal/Refund transaction	RcvPayRsp_cont	
5	Authentication transaction	LocalCnl	Cancellation_NC
6		RcvCnl_P_D	
7	Authentication/Deposit/Withdrawal/Refund transaction	SndCnl_special	Cancellation_C
8	Authentication transaction	Stop_4_P_D	ErrHandling_NC
9		Discard_Clear_P	
10	Authentication/Deposit/Withdrawal/Refund/ Value Exchange transaction	Discard_Clear_D	
11	RCmax_c=0	Retransmit_RstTimer	ErrHandling_C
12		RstTimer	
13		Discard	ErrHandling_NC
14		SndMsgRptTrntErr	
15	RCmax_nc=0	Retransmit_RstTimer	ErrHandling_NC
16		RstTimer	
17		Discard	ErrHandling_C
18		SndMsgRptTrntErr	

Table 13.9: Expected dead transitions under certain parameter values in the revised IOTP CPN.

Table 13.10 lists the dead transitions (DT) for each configuration (recall from Section 11.1.2 that the configuration is given in the order of Transaction, RCmax_c and RCmax_nc). For brevity, the numbers in the header row correspond to those transitions that have their transition

Config	*DT*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*
A-0-0	15	1	0	-	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1
A-0-1	11	1	0	-	-	1	1	1	1	1	1	1	1	1	1	0	0	0	0
A-0-2	11	1	0	-	-	1	1	1	1	1	1	1	1	1	1	0	0	0	0
A-0-3	11	1	0	-	-	1	1	1	1	1	1	1	1	1	1	0	0	0	0
A-1-0	11	1	0	-	-	1	1	1	1	1	1	0	0	0	0	1	1	1	1
A-1-1	7	1	0	-	-	1	1	1	1	1	1	0	0	0	0	0	0	0	0
A-1-2	7	1	0	-	-	1	1	1	1	1	1	0	0	0	0	0	0	0	0
A-1-3	7	1	0	-	-	1	1	1	1	1	1	0	0	0	0	0	0	0	0
A-2-0	11	1	0	-	-	1	1	1	1	1	1	0	0	0	0	1	1	1	1
A-2-1	7	1	0	-	-	1	1	1	1	1	1	0	0	0	0	0	0	0	0
A-2-2	7	1	0	-	-	1	1	1	1	1	1	0	0	0	0	0	0	0	0
A-2-3	7	1	0	-	-	1	1	1	1	1	1	0	0	0	0	0	0	0	0
A-3-0	11	1	0	-	-	1	1	1	1	1	1	0	0	0	0	1	1	1	1
A-3-1	7	1	0	-	-	1	1	1	1	1	1	0	0	0	0	0	0	0	0
A-3-2	7	1	0	-	-	1	1	1	1	1	1	0	0	0	0	0	0	0	0
A-3-3	7	1	0	-	-	1	1	1	1	1	1	0	0	0	0	0	0	0	0
P-0-0	10	0	1	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
P-0-1	6	0	1	1	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0
P-0-2	6	0	1	1	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0
P-0-3	6	0	1	1	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0
P-1-0	6	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
P-1-1	2	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-1-2	2	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-1-3	2	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-2-0	6	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
P-2-1	2	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-2-2	2	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-2-3	2	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-3-0	6	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
P-3-1	2	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-3-2	2	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-3-3	2	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
D/W/R-0-0	13	0	1	1	1	0	0	1	0	0	1	1	1	1	1	1	1	1	1
D/W/R-0-1	9	0	1	1	1	0	0	1	0	0	1	1	1	1	1	0	0	0	0
D/W/R-0-2	9	0	1	1	1	0	0	1	0	0	1	1	1	1	1	0	0	0	0
D/W/R-0-3	9	0	1	1	1	0	0	1	0	0	1	1	1	1	1	0	0	0	0
D/W/R-1-0	9	0	1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1
D/W/R-1-1	5	0	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0
D/W/R-1-2	5	0	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0
D/W/R-1-3	5	0	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0
D/W/R-2-0	9	0	1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1
D/W/R-2-1	5	0	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0
D/W/R-2-2	5	0	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0
D/W/R-2-3	5	0	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0
D/W/R-3-0	9	0	1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1
D/W/R-3-1	5	0	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0
D/W/R-3-2	5	0	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0
D/W/R-3-3	5	0	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0
V-0-0	10	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
V-0-1	6	0	1	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0
V-0-2	6	0	1	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0
V-0-3	6	0	1	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0
V-1-0	6	0	1	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	1
V-1-1	2	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
V-1-2	2	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
V-1-3	2	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
V-2-0	6	0	1	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	1
V-2-1	2	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
V-2-2	2	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
V-2-3	2	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
V-3-0	6	0	1	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	1
V-3-1	2	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
V-3-2	2	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
V-3-3	2	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0

Table 13.10: Dead transitions for selected configurations of the revised IOTP.

numbers as given in Table 13.9. The total number of dead transitions is also listed for each configuration. Closer inspection (see Appendix H) reveals that the dead transitions for each configuration are those expected as given in Table 13.9 (see Appendix H for details), proving Property 11.3 for all configurations analysed.

13.7.4 Language Comparison

Recall from the beginning of Section 12.5, each Payment-related transaction (with cancellation) is verified against an acceptable subset of the transaction service language defined in Section 12.3. Table 13.11 summarises the statistics obtained from Section 12.5 of the four FSA (one specifying the Authentication service language, and the other three representing the acceptable subsets of the service languages) used for the verification of Payment-related transactions.

Transaction Service FSA	*States*	*Arcs*	*Final States*
Authentication	13	36	2
Purchase	54	170	4
Deposit/Withdrawl/Refund	38	119	3
Value Exchange	42	130	3

Table 13.11: Statistics of the FSA specifying the service languages used for the verification of IOTP.

Table 13.12 gives the statistics of the FSA specifying the protocol language of each Authentication and Payment-related transaction in the revised IOTP, and the results of language comparison between protocol and service, for the configurations analysed. These statistics include: the number of states (S), arcs (A) and final states (F) in the minimized FSA; and the number of the illegal sequences (I) and the missing sequences (M) in the protocol. The FSM reports that provide the statistical information are given in Appendix H.

RCmax		*Authentication*					*Purchase*					*Deposit/ Withdrawal/ Refund*					*Value Exchange*				
c	nc	S	A	F	I	M	S	A	F	I	M	S	A	F	I	M	S	A	F	I	M
0	0	13	36	2	0	0	54	170	4	0	0	38	119	3	0	0	42	130	3	0	0
0	1	13	36	2	0	0	54	170	4	0	0	38	119	3	0	0	42	130	3	0	0
0	2	13	36	2	0	0	54	170	4	0	0	38	119	3	0	0	42	130	3	0	0
0	3	13	36	2	0	0	54	170	4	0	0	38	119	3	0	0	42	130	3	0	0
1	0	13	36	2	0	0	54	170	4	0	0	38	119	3	0	0	42	130	3	0	0
1	1	13	36	2	0	0	54	170	4	0	0	38	119	3	0	0	42	130	3	0	0
1	2	13	36	2	0	0	54	170	4	0	0	38	119	3	0	0	42	130	3	0	0
1	3	13	36	2	0	0	54	170	4	0	0	38	119	3	0	0	42	130	3	0	0
2	0	13	36	2	0	0	54	170	4	0	0	38	119	3	0	0	42	130	3	0	0
2	1	13	36	2	0	0	54	170	4	0	0	38	119	3	0	0	42	130	3	0	0
2	2	13	36	2	0	0	54	170	4	0	0	38	119	3	0	0	42	130	3	0	0
2	3	13	36	2	0	0	54	170	4	0	0	38	119	3	0	0	42	130	3	0	0
3	0	13	36	2	0	0	54	170	4	0	0	38	119	3	0	0	42	130	3	0	0
3	1	13	36	2	0	0	54	170	4	0	0	38	119	3	0	0	42	130	3	0	0
3	2	13	36	2	0	0	54	170	4	0	0	38	119	3	0	0	42	130	3	0	0
3	3	13	36	2	0	0	54	170	4	0	0	38	119	3	0	0	42	130	3	0	0

Table 13.12: Statistics on the size of the FSA for selected configurations of the revised IOTP.

In Table 13.12 the language comparison results show that the protocol language of each transaction in the revised IOTP is the same as the corresponding subset of the service language of IOTS (see Table 13.11). For all configurations analysed, the revised IOTP preserves an acceptable subset of the IOTS language defined in Chapter 9, and therefore faithfully refines

the IOTS, in terms of global sequences of service primitives. Also, it can be observed from Table 13.12 that the protocol language is independent of the value of message retransmission counter used at each trading role entity, given all permutations of the counter numbers within the range of *0* to *3*. Hence, we conjecture that this is true for all values of the retransmission counters.

Conjecture 13.1 (Protocol language and message retransmission counter in IOTP). *The protocol language of each Authentication and Payment-related transaction is independent of the value of message retransmission counter used at each trading role entity.* □

Chapter 14

Conclusions

14.1 Contributions

IOTP is an e-commerce protocol developed by IETF to support common trading activities over the Internet. The core of IOTP is a set of financial transactions and therefore it is vitally important that the protocol operates correctly. However, as far as we are aware, there has been no other published work analysing the functional behaviour of IOTP, except for ours [177–182], which are consolidated and expanded on in this thesis.

RFC 2801 [36] contains an informal narrative description of IOTP, so it is ambiguous, difficult to understand and imprecise. Also, IOTP is intended to meet a set of demanding goals, such as the encapsulation of different payment protocols and the support for various trading activities over the Internet, which has made it complex. This thesis presents several key results concerning the formal specification and verification of IOTP using Coloured Petri Nets.

14.1.1 An Initial Assessment of IOTP

RFC 2801 [36], which specifies IOTP, is so far the largest RFC ever developed by IETF. Before verifying IOTP, a critical assessment of the RFC was given in Chapter 4, regarding the conceptual framework and protocol architecture for IOTP. Below, we summarise the problems that have been identified in the assessment together with our suggestions for improvement.

- The RFC defines two building blocks for transactions, known as *trading exchanges* and *document exchanges*. We believe it is unnecessary to define both building blocks, because they attempt to achieve the same purpose and simply group the data exchanges comprising a transaction in different ways. The inclusion of both building blocks makes it more difficult to understand the protocol. Moreover, it could lead implementers to use trading exchanges (introduced early in the RFC) instead of document exchanges, which would result in incompatible implementations, something standardisation is trying to avoid. Hence, we propose to keep the concept of *document exchange* and to remove all parts of the RFC related to *trading exchanges*. This will provide a simpler and more readily understood conceptual framework for IOTP.

- The RFC has unnecessary complexity in specifying authentication within a trading transaction. To remove this complexity while providing the same functionality, we propose that: 1) an *Authentication document exchange* be used in a trading transaction at any

time when authentication is required, and 2) the *Authentication transaction* only be used as a standalone transaction and thus never be invoked as part of any other transaction.

- The lack of a protocol architecture for IOTP makes the RFC difficult to read. When specifying a protocol it is important to understand the relationship of the protocol with its environment. It is also essential to define the intent of the protocol, in terms of the service it provides to its application users, and the characteristics of the underlying transport mechanism that it assumes. IOTP is designed to encapsulate a number of payment systems and hence its interactions with these payment systems need to be clearly defined as well. We have isolated what we believe to be the major components of an architecture for IOTP.

Finally, the overall specification of IOTP given in RFC 2801 is not well structured. For example, definitions of several basic concepts are spread over different sections, and sections specifying major features of the protocol are not organised in a logical order. Accordingly, we have suggested a way to re-structure the RFC to take into account the above conceptual framework and the protocol architecture proposed for IOTP.

14.1.2 IOTS Specification and Validation

The IOTP specification in RFC 2801, as is the case for most of the Internet protocols, does not specify the service provided by the protocol [221]. Developing a service specification represents an important step in developing protocols [21, 26], because the service (of a protocol) defines the requirements of the protocol from a user's perspective and can thus be used as the basis for verifying the operation of the protocol.

In this thesis, we have defined the trading services provided by IOTP in Chapter 7, which we call the Internet Open Trading Service (IOTS). We have formalised IOTS using CPNs to obtain a formal service specification for IOTP in Chapters 8 and 9.

To manage the inherent complexity of IOTS, we have applied a protocol engineering methodology that allows us to define the local interface behaviour before considering the correlations between the events at different interfaces. This methodology consists of five iterative steps for incrementally developing IOTS and is based on OSI standards [114,116] and a protocol verification approach [26]. A contribution to this methodology is the inclusion of a language comparison step to check for consistency between the global service language and the local service language. The purpose of conducting this language comparison is to make sure that the global service specification is consistent with the local service specifications. This has led to the detection of inconsistencies which have been removed from the original CPN model.

14.1.3 IOTP Formal Specification

RFC 2801 describes IOTP in a narrative way, which includes a few diagrams and tables. The informal nature of the IOTP specification and its complexity make it difficult to maintain consistency and clarity. In addition, the document is not well structured and the relationship between several basic concepts is not clear (see Section 14.1.1).

Based on the simplified conceptual framework for IOTP in Chapter 3, a CPN model of IOTP was developed in Chapter 10. The model was tailored towards verification tasks by

introducing a set of simplifications, and focused on the core behaviour of IOTP with three restrictions. The simplifications do not limit the applicability of the analysis results, whereas the restrictions do. The three restrictions are: 1) only Authentication and Payment-related transactions of IOTP version 1.0 [36] are modelled; 2) all six Authentication and Payment-related transactions are independent of each other; and 3) each Payment-related transaction may or may not involve authentication, and if it is required, we only consider the one that occurs between the Consumer and Merchant at the beginning of the transaction. In addition, a set of assumptions were made in the modelling of IOTP when the description in RFC 2801 was ambiguous or incomplete. The resulting model provides a clear, unambiguous and precise definition of the core behaviour of IOTP, i.e., all six Authentication and Payment-related transactions involving both error-handling and transaction cancellations.

This is the first time that a formal model of the IOTP transactions has been created. The model is developed to have a four-level hierarchy that matches the structure of IOTP and allows re-use of modular message exchange mechanisms, known as document exchanges, in IOTP. It is also novel in that it involves communication between 4 entities rather than 2.

14.1.4 Analysis of IOTP and Verification against IOTS

Chapter 11 analyses three desired properties of IOTP: valid transaction termination; absence of livelocks; and absence of un-used protocol operation (indicated by unexpected dead transitions). The analysis results demonstrate that IOTP [36] fails to terminate correctly.

Chapter 12 verifies IOTP against IOTS using language comparison (between IOTP and its service). It defines acceptable subsets of primitive sequences in IOTS, so that IOTP, which preserves a subset of primitive sequences in IOTS, can be verified against these acceptable subsets. This is the first time that the acceptable subsets of service language are defined and used for verification of a protocol against its service.

With the state space and language generated from the IOTP CPN model, inspection of the transaction termination property and comparison with the corresponding IOTS language or acceptable subsets of sequences in IOTS, revealed errors in IOTP. These are: 1) the Merchant not being informed of transaction cancellation between Consumer and Payment Handler or Delivery Handler, which may cause an "inventory problem" for the Merchant; 2) a transaction cancelled at the Consumer side only, which may introduce a "non-refundable payment problem" to the Consumer; and 3) the lack of a timer being set by a trading role protocol entity when waiting for a response from its user, and the lack of synchronization of cancellation when a time-out occurs that results in cancellation.

14.1.5 Revised IOTP and its Verification

Chapter 13 proposes solutions to the errors revealed by the analysis of the CPN model of IOTP version 1.0 [36]. The changes suggested to correct errors in IOTP were implemented, leading to the revised IOTP and its corresponding CPN. To demonstrate the proposed solutions are valid, and increase confidence that the revised IOTP is functionally correct, state space and language analysis of the revised IOTP was performed. The analysis proved that the revised IOTP satisfies the three desired properties (see Section 14.1.4). Further, its language preserves a valid subset of the IOTS language, for the following ranges of initial parameter values:

Transaction is set to one of the six values representing Authentication and Payment-related transactions, and all permutations of RCmax_c and RCmax_nc taking the values from 0 to 3. Based on the above results, we also make the conjecture that the IOTP language is independent of the value of RCmax_c and RCmax_nc.

14.2 Future Work

14.2.1 Extending the Current IOTP CPN

There is still considerable work to be done related to the modelling and specification of IOTP to obtain a complete formal specification. Recall the four restrictions imposed on the current IOTP CPN model to simplify the analysis (see Section 14.1.3). The model can be extended with these restrictions relaxed, and eventually removed. For example, by taking into account the two infrastructure transactions, Ping and Inquiry, a CPN model covering all Baseline IOTP transactions can be obtained.

Based on the development of IOTP version 1.0 since 1998, IOTP version 2.0 is currently being considered with its requirements stated in [75]. The current model can be used as a basis for modelling some of the extensions in IOTP version 2.0. For example, the proposed extensions to support a transaction with multiple deliveries occurring after payment, can be incorporated into the current model by reusing the delivery document exchange modules.

14.2.2 Extending the Results for Larger Values of Parameters

Currently, we have generated the state space of each trading transaction, with the maximum value of the message retransmission counter (*RCmax*) at each trading role entity set to *3*. We expect significantly larger state spaces as *RCmax* is increased. For example, when increasing *RCmax* to *4*, the state spaces of both the Purchase and the Value Exchange transaction become unmanageable within memory limits. Further work includes applying other analysis techniques, such as the *sweep-line method* [48], to alleviate the problems of state space explosion.

The sweep-line method traverses all the states of the full state space, but does not store all the states simultaneously. Properties can be proved using on-the-fly verification. This method is suited to systems that exhibit a particular property, known as progress. A *progress measure* must be defined for each state that gives it a value greater than or equal to its predecessor states. Some example applications are: communication protocols that use counters, sequence numbers or message numbers, systems with time that use a global clock (e.g., timed CPNs [123]) and systems that have an iteration counter. There has been a case study [90] of applying the sweep-line method to the analysis of the Wireless Transaction Protocol. It showed that the use of the sweep-line method achieved reductions in the amount of memory used of approximately 50%.

From the modelling and analysis of IOTP in this thesis, we can see that it is one of those communication protocols that exhibit progress. For example, the message identifier is always incremented or remains constant (for a retransmitted message) during a transaction. Also, the transaction always progresses from one document exchange to another, i.e., a Payment document exchange never happens before an Offer document exchange. Therefore, we believe that the sweep-line method would be well suited to further analysis of IOTP. For example, it

may be useful to use the sweep-line method to give more confidence that the minimized FSA for the IOTP language is independent of $RCmax$, by running values for $RCmax = 4$ to 6 and doing language inclusion checking on-the-fly.

14.2.3 Enhancing the Methodology to Synthesize the Global Service Language from Local Service Languages

In Chapters 7 and 9, we have applied a five-step methodology in the development of the IOTS specification (see Section 14.1.2). It allows us to synthesize all local service automata into a global service CPN, but requires a step of language comparison to check the consistency between the local and the global service languages as this is currently not guaranteed in the above synthesis step.

The methodology could be enhanced by encoding the local service automata as CPNs (called local automata CPNs). These local automata CPNs would then be used directly (or nearly) in the CPN model of the global service, by linking them with places and arcs that model the communication medium between local interfaces. This would eliminate any translation errors from local automata into the global CPN. The next step would be to develop a proof that the local sequences are preserved in the global CPN by construction. As a result, there would be no need for language comparison.

14.2.4 Verification of Other Properties of IOTP

Three desired properties of IOTP have been investigated in this thesis (see Section 14.1.4). There are also other properties, such as security properties, which we would like to include in further analysis of IOTP. For example, to examine the security properties of IOTP, the digital signature and digital certificate should be considered in its formal specification. In addition, we may also need to introduce an "intruder" that acts to cause a malfunction of a transaction.

References

[1] *Proceedings of IFIP WG6.1 International Workshop/Symposiums on Protocol Specification, Testing and Verification (PSTV).* Vol. II (1983) to XIII (1993): North-Holland, Amsterdam Oxford. Vol. XIV (1994) to XVII (1997): Chapman & Hall, London, UK. Vol. XVIII (1998) to XX (2000): Kluwer Academic Publishers, Boston Dordrecht London. Vol. XVI to XX are joint proceedings with *Formal Description Techniques (FORTE).*

[2] Special issue on the rise of protocol engineering. *IEEE Software*, 9(1), January 1992.

[3] *Proceedings of the International Conferences on Applications and Theory of Petri Nets*, Lecture Notes in Computer Science, Berlin, 1993-2002. Springer-Verlag.

[4] *Proceedings of Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, Aarhus, Denmark, 1998,1999,2001,2002. Department of Computer Science, Aarhus University.

[5] R. Aaron, M. Decina, and R. Skillen. Electronic commerce: enablers and implications. *IEEE Communication Magazine*, 37(9):47–52, 1999.

[6] D. Abrazhevich. Classification and characteristics of electronic payment systems. In *Proceedings of 2nd International Conference on Electronic Commerce and Web Technologies*, volume 2115 of *Lecture Notes in Computer Science*, pages 81–90. Spring-Verlag, 2001.

[7] N. R. Adam and Y. Yesha, editors. *Electronic Commerce: Current Research Issues and Applications*, volume 1028 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, Heidelberg, 1996.

[8] L. C. Aiello and F. Massacci. Verifying security protocols as planning in logic programming. *ACM Transactions on Computational Logic*, 2(4):542–580, October 2001.

[9] N. Asokan. *Fairness in Electronic Commerce.* PhD thesis, University of Waterloo, May 1998.

[10] N. Asokan, P.A. Janson, M. Steiner, and M. Waidner. The state of the art in electronic payment systems. *IEEE Computer*, 30(9):28–35, September 1997.

[11] AT&T. FSM Library. URL: `http://www.research.att.com/sw/tools/fsm`.

[12] AT&T. GraphViz. URL: `http://www.research.att.com/sw/tools/graphviz`.

[13] AT&T. Lextools. URL: `http://www.research.att.com/sw/tools/lextools`.

[14] J. B. Ayers, editor. *Making Supply Chain Management Work: Design, Implementation, Partnerships, Technology, and Profits.* Auerbach, Boca Raton, FL, 2002.

[15] W. A. Barret and J. D. Couch. *Compiler Construction: Theory and Practice.* Science Research Associates, second edition, 1986.

[16] G. Bella, F. Massacci, and L. C. Paulson. Verifying the SET registration protocols. *IEEE Journal on Selected Areas in Communications*, 21(1):77–87, 2003.

[17] M. Benerecetti, M. Panti, L. Spalazzi, and S. Tacconi. Verification of payment protocols via multiagent model checking. In *Proceedings of 14th International Conference on Advanced Information Systems Engineering*, volume 2348 of *Lecture Notes in Computer Science*, pages 311–327, Toronto, Ontario, Canada, 27-31 May 2002. Springer-Verlag.

[18] T. Berners-Lee and M. Fischetti. *Weaving the Web: The Past, Present and Future of the World Wide Web by its Investors.* Orion Business, London, 1999.

[19] K. Bhargavan, D. Obradovic, and C. A. Gunter. Formal verification of standards for distance vector routing protocols. *Journal of the ACM*, 49(4):538–576, July 2002.

[20] J. Billington. Abstract specification of the ISO Transport service definition using labelled Numerical Petri Nets. In *Protocol Specification, Testing, and Verification, III*, pages 173–185. Elsevier Science Publishers, Amsterdam, 1983.

[21] J. Billington. Formal specification of protocols: protocol engineering. In *Encyclopedia of Microcomputers*, pages 299–314. Marcel Dekker, New York, 1991.

[22] J. Billington. Protocol specification using P-Graphs, a technique based on Coloured Petri nets. In *Advances in Petri Nets*, pages 293–330. Springer-Verlag, 1999.

[23] J. Billington. Discrete event systems theory and its application to command and control. Technical Report CSEC-14, Computer Systems Engineering Centre, School of Electrical and Information Engineering, University of South Australia, 31 December 2002.

[24] J. Billington, M. Diaz, and G. Rozenberg, editors. *Application of Petri Nets to Communication Networks: Advances in Petri Nets*, volume 1605 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1999.

[25] J. Billington, M. Farrington, and B. Du. Modelling and analysis of multi-agent communication protocols using CP-nets. In *Proceedings of the 3rd Biennial Engineering Mathematics and Applications Conference*, pages 119–122, 1998.

[26] J. Billington, M. C. Wilbur-Ham, and M. Y. Bearman. Automated Protocol Verification. In *Protocol Specification, Testing, and Verification, V*, pages 59–70. Elsevier Science Publishers, 1986.

[27] D. Birch. Shopping Protocols - Buying online is more than just paying. *Journal of Internet Banking and Commerce*, 3(1), January 1998. Available via: `http://www.arraydev.com/commerce/JIBC/9801-9.htm`.

[28] G. V. Bochmann. Finite state description of communication protocols. *Computer Networks*, 2:362–372, 1978.

[29] C. Bolchini and F. A. Schreiber. Smart card embedded information systems: a methodology for privacy oriented architectural design. *Data and Knowledge Engineering*, 41(2-3):159–182, June 2002.

[30] D. Bolignano. Towards the formal verification of electronic commerce protocols. In *Proceedings of 10th Computer Security Foundations Workshop*, pages 133–146, Rockport, MA, USA, 10-12 June 1997. IEEE Computer Society.

[31] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[32] W. Brauer, editor. *Net Theory and Applications*, volume 84 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[33] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. *Extensible Markup Language (XML) 1.0 (Second Edition)*. W3C Recommendation, October 2000.

[34] S. Budkowski and P. Dembinski. A introduction to Estelle: a specification language for distributed systems. *Computer Networks and ISDN Systems*, 14(1):3–23, 1987.

[35] D. Burdett. *The Open Trading Protocol Version 0.9.9*. The OTP Consortium, 1998.

[36] D. Burdett. *Internet Open Trading Protocol - IOTP Version 1.0*. IETF Network Working Group, April 2000. IETF RFC 2801. Available via: `http://www.ietf.org/rfc/rfc2801`.

[37] D. Burdett, D. E. Eastlake, and M. Goncalves. *Internet Open Trading Protocol*. McGraw-Hill, 2000.

[38] A. Burns and G. Davies. *Concurrent Programming*. International Series in Computer Science. Addison-Wesley, Reading, MA, 1993.

[39] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Events Systems*. Kluwer Academic Publishers, Boston Dordrecht, London, 1999.

[40] CEN/ISSS Workshop. *Summaries of some Frameworks, Architectures and Models for Electronic Commerce. Final Draft*, February 2002. [Last accessed: 27 May 2003]. Available via: `http://www.cenorm.be/isss/workshop/ec/Architecture/Arch_Documents.htm`.

[41] Computer Systems Engineering Centre. University of South Australia, URL: `http://www.unisa.edu.au/eie/csec`, February 2004.

[42] CEPSCO. *Commerce Electronic Purse Specifications. Version 2.3. Vol 1-3*, March 2001. Available via: `http://www.cepsco.com/cgi/list_down.pl`.

[43] D. Chaum, A. Fiat, and M. Naor. Untraceable electronic cash. In *Proceedings of Advances in Cryptology - CRYPTO '88*, pages 319–327. Springer-Verlag, 1990.

[44] Q. Chen, S. Bai, J. Wang, S. Zhang, and L. Sui. The problem analysis of SET protocol and its resolution. *Chinese Journal of Software*, 23(2):202–209, 2000.

[45] Q. Chen and M. Hsu. Inter-enterprise collaborative business process management. In *Proceedings of International Conference on Data Engineering*, pages 253–260, Heidelberg, Germany, 2-6 April 2001. IEEE Computer Society.

[46] M. Chesher and R. Kaura. *Electronic Commerce and Business Communications.* Springer, New York, 1998.

[47] S. Christensen, L. M. Kristensen, and T. Mailund. *Design/CPN Sweep-Line Method Library.* Department of Computer Science, Aarhus University, Aarhus, Denmark, 2001.

[48] S. Christensen, L. M. Kristensen, and T. Mailund. A sweep-line method for state space exploration. In *Proceedings of 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2001)*, volume 2031 of *Lecture Notes in Computer Science*, pages 450–464. Springer-Verlag, 2001.

[49] E. Clarke, O. Grümberg, and D. Peled. *Model Checking.* The MIT Press, 1999.

[50] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrenct systems using temporal logic specifications: A practical approach. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 117–127, Austin, Texas, January 1983.

[51] E. M. Clarke and J. M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[52] A. L. Coleman. Java commerce business perspective (white paper). Technical report, Sun Microsystems, 1997. Available via: `http://java.sun.com/products/commerce/docs/business/business.html`.

[53] D. Comer. *Internetworking with TCP/IP.* Prentice-Hall, Upper Saddle River, NJ, fourth edition, 2000.

[54] D. E. Comer. *Computer Networks and Internets: with Internet Applications.* Prentice-Hall, Upper Saddle River, NJ, 2001. Third Edition.

[55] CommerceNet eCo Framework Group. *The eCo Specification*, September 1999. Available via: `http://www.openbuy.org/obi/specs/obiv2.pdf`.

[56] The Open Trading Protocol Consortium. *Internet Open Trading Protocol. Part1: Business Description*, January 1998. Version 0.9.

[57] The SPECS Consortium and J. Bruijning. Evaluation and integration of specification languages. *Computer Networks and ISDN Systems*, 13(2):75–89, 1987.

[58] W3C Consortium. Micropayment Overview.
URL: `http://www.w3c.org/ECommerce/Micropayments`, August 2001.

[59] W3C Consortium. URL: `http://www.w3c.org`, March 2004.

[60] Meta Software Corporation. *Design/CPN Reference Manual for X-Windows. Version 2.0.* Meta Software Corporation, Cambridge, MA, 1993.

[61] A. Corradi, M. Cremonini, R. Montanari, and C. Stefanelli. Mobile agents integrity for electronic commerce applications. *Information Systems*, 24(6):519–533, 1999.

[62] D. Crocker. *MIME Encapsulation of EDI Objects.* IETF Network Working Group, March 1995. IETF RFC 1767. Available via: `http://www.ietf.org/rfc/rfc1767`.

[63] D.H. Crocker. *Standard for the Format of ARPA Internet Text Messages.* Dept. of Electrical Engineering, University of Delaware, Newark, DE 19711, August 1982. IETF RFC 822. Available via: `http://www.ietf.org/rfc/rfc822`.

[64] A. Datta and S. Ghose. Efficient handling of micropayments in electronic commerce. In *Proceedings of 9th Annual Workshop on Information Technologies and Systems*, pages 123–128, Charlotte, NC, USA, 11-12 December 1999.

[65] D. W. Davies and W. L. Price. *Security for Computer Networks: An Introduction to Data Security in Teleprocessing and Electronic Funds Transfer.* Wiley, New York, second edition, 1989.

[66] Version 1.1 Design/CPN Message Sequence Chart Library. Department of Computer Science, University of Aarhus, Denmark, 1998. URL: `http://www.daimi.au.dk/designCPN`.

[67] ASC X12 Standards Development.
URL: `http://www.x12.org/x12org/subcommittees/dev/index.cfm`.

[68] T. Dierks and C. Allen. *The TLS Protocol Version 1.0.* IETF Network Working Group, January 1999. IETF RFC 2246. Available via: `http://www.ietf.org/rfc/rfc2246`.

[69] UN/EDIFACT Standard Directory.
URL: `http://www.unece.org/trade/untdid/welcome.htm`.

[70] P. DongGook, C. Boyd, and E. Dawson. Micropayments for wireless communications. In *Proceedings of 3rd International Conference on Information Security and Cryptology*, volume 2015 of *Lecture Notes in Computer Science*, pages 195–205. Springer-Verlag, 2001.

[71] S. Drees, D. Gomm, H. Plunnecke, W. Reisig, and R. Walter. Bibliography of net theory. In G. Rozenberg, editor, *Advances in Petri Nets 1987*, pages 309–451. Springer-Verlag, Berlin.

[72] H. Dreifus and J. T. Monk. *Smart Cards : A Guide to Building and Managing Smart Card Applications.* Wiley, New York, 1998.

[73] G. N. Drew. *Using SET for Secure Electronic Commerce.* Prentice-Hall, Upper Saddle River, NJ, 1999.

[74] D. Durham and R. Yavatkar. *Inside the Internet's Resource reSerVation Protocol: Foundations for Quality of Service.* John Wiley, New York, 1999.

[75] D. E. Eastlake. *Internet Open Trading Protocol Version 2 Requirements.* IETF Network Working Group, August 2002. IETF RFC 3354. Available via: `http://www.ietf.org/rfc/rfc3354`.

[76] D. E. Eastlake. *Internet Open Trading Protocol Version 1 Errata.* IETF Network Working Group, March 2003. IETF RFC 3504. Available via: `http://www.ietf.org/rfc/rfc3504`.

[77] D. E. Eastlake and C. Smith. *Internet Open Trading Protocol HTTP Supplement.* IETF Network Working Group, September 2000. IETF RFC 2935. Available via: `http://www.ietf.org/rfc/rfc2935`.

[78] M. Deturche et al. Building Blocks for electronic commerce. Technical report, European Board for EDI Electronic Commerce Standardization (EBES), 1997. [Last accessed: 27 May 2003] Available: `http://www.cenorm.be/isss/Workshop/ec/Prb.zip`.

[79] R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1.* IETF Network Working Group, June 1999. IETF RFC 2616. Available via: `http://www.ietf.org/rfc/rfc2616`.

[80] Europay International, MasterCard International and Visa International (EMVCo). *EMV SIntegrated Circuit Card Specifications for Payments. Version 4.0. Vol 1-4*, December 2000. Available via: `http://www.emvco.com`.

[81] International Organization for Standardization. *Information Technology - Open-EDI Reference Model.* ISO/IEC 14662, 1997.

[82] A. Fourati, H. K. B. Ayed, F. Kamoun, and A. Benzekri. Wireless SET: an approach for the deployment of SET over WAP mobile networks. In *Proceedings of IFIP TC6/WG6.2 and WG6.7 Conference on Network Control and Engineering for QoS, Security and Mobility (Net-Con 2002)*, volume 235 of *IFIP Conference Proceedings*, pages 3437–442, Paris, France, 23-25 October 2002. Kluwer.

[83] S. Glassman, M. Manasse, M. Abadi, P. Gauthier, and P. Sobalvarro. The Millicent Protocol for Inexpensive Electronic Commerce. In *Proceedings of 4th International Conference on the World-Wide-Web*, MIT, Boston, December 1995.

[84] S. Glassman, M. Manasse, M. Abadi, P. Gauthier, and P. Sobalvarro. Micropayments revisited. In *Proceedings of International Conference on Topics in Cryptology - CT-RSA 2002*, volume 2271 of *Lecture Notes in Computer Science*, pages 149–163, San Jose, CA, USA, 18-22 February 2002. Springer-Verlag.

[85] Fujitsu Global. URL: `http://www.fujitsu.com`.

[86] R. J. Glushko, J. M. Tenenbaum, and B. Meltzer. An XML framework for agent-based e-commerce. *Communications of the ACM*, 42(3):106–114, 1999.

[87] M. Goncalves. *IP Multicasting.* McGraw-Hill, New York, 1999.

[88] S. Gordon. *Verification of the WAP Transaction Layer using Coloured Petri Nets.* PhD thesis, School of Electrical and Information Engineering, University of South Australia, Adelaide, Australia, August 2001.

[89] S. Gordon and J. Billington. Modelling the WAP Transaction Service using Coloured Petri Nets. In *Proceedings of the First International Conference on Mobile Data Access*, volume 1748 of *Lecture Notes in Computer Science*, pages 105–114, Hong Kong, China, 16-17 December 1999. Springer-Verlag.

[90] S. Gordon, L. M. Kristensen, and J. Billington. Verification of a revised WAP Wireless Transaction Protocol. In *Proceedings of 23rd International Conference on Application and Theory of Petri Nets*, volume 2360 of *Lecture Notes in Computer Science*, pages 182–202, Adelaide, Australia, 24-30 June 2002. Springer-Verlag.

[91] R. Gotzhein. Specifying communication services with temporal logic. In *Protocol Specification, Testing and Verification, X*, pages 295–209, Amsterdam, Oxford, 1990. North-Holland.

[92] P. Grefen, K. Aberer, Y. Hoffner, and H. Ludwig. Crossflow: Cross-organizational workflow management in dynamic virtual enterprises. *Computer Systems Science and Engineering*, 15(5):277–290, 2000.

[93] Jalda API Reference Guide.
URL: `http://www.ericsson.com/mobilityworld/sub/news/SubPages/jalda_api`, May 2003.

[94] D. J. Guinan and A. M. Brown. Commerce JavaBeans (white paper). Technical report, Sun Microsystems, 1999. Available via:
`http://java.sun.com/products/commerce/docs/specifications/commerce_javabeans`.

[95] A. Gupta, D. O. Stahl, and A. B. Whinston. Managing computing resources in intranets: an electronic commerce perspective. *Decision Support Systems*, 24(1):55–69, 1998.

[96] S. Gurgens, J. Lopez, and R. Peralta. Efficient detection of failure modes in electronic commerce protocols. In *Proceedings of 10th International Workshop on Database and Expert Systems Applications*, pages 850–857, Florence, Italy, 1-3 September 1999. IEEE Computer Society.

[97] R. H. Guttman, A. G. Moukas, and P. Maes. Agent-mediated electronic commerce: a survey. *Knowledge Engineering Review*, 13(2):147–159, 1998.

[98] B. Han and J. Billington. Validating TCP connection management. In *Proceedings of Workshop on Software Engineering and Formal Methods*, volume 12 of *Conferences in Research and Practice in Information Technology*, pages 47–55. Australian Computer Society, 2002.

[99] W. Hans, Y. Kawatsura, and M. Hiroya. *Payment API for Internet Open Trading Protocol.* IETF Network Working Group, April 2003. IETF Internet Draft. Available via:
`http://www.ietf.org/internet-drafts/draft-ietf-trade-iotp-v1.0-papi-05`.

[100] P. G. Harrison and N. M. Patel. *Performance Modelling of Communication Networks and Computer Architectures.* Addison-Wesley, Reading, MA, 1993.

[101] N. Heintze, J. D.Tygar, J. Wing, and H. C. Wong. Model checking electronic commerce protocols. In *Proceedings of 2nd USENIX Workshop on Electronic Commerce*, pages 147–164, Oakland, CA, USA, November 18-21 1996.

[102] A. Herzberg and H. Yochai. Mini-pay: Charging per Click on the Web. In *Proceedings of 6th International Conference on the World-Wide-Web*, Santa Clara, CA, USA, April 1997. Available via: `http://decweb.ethz.ch/WWW6/Technical/index.html`.

[103] C. A. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[104] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, Englewood Cliffs, NJ, 1991.

[105] J. E Hopcroft and J. D. Ullman. *Formal Languages and Their Relation to Automata.* Addison-Wesley, Reading, MA, 1969.

[106] C. M. Huang and S. W. Lee. Timed protocol verification for Estelle-specified protocols. *ACM SIGCOMM Computer Communication Review*, 25(3):4–32, July 1995.

[107] J. Hwang and S. Hsueh. Greater protection for credit card holders: a revised SET protocol. *Computer Standards and Interfaces*, 19(1):1–8, 1998.

[108] The Internet Engineering Task Force - IETF. URL: `http://www.ietf.org`.

[109] WAP Forum in Open Mobile Alliance. WAP Wireless Transaction Protocol Specification. Version 1.0. URL: `http://www.wapforum.org/what/technical.htm`, July 2001.

[110] WAP Forum in Open Mobile Alliance. Wireless Application Protocol Architecture Specification. URL: `http://www.wapforum.org/what/technical.htm`, July 2001.

[111] Hitachi Research Topics IOTP.
URL: `http://www.sdl.hitachi.co.jp/english/topics/t_iotp/iotp.htm`, 1999.

[112] ISO. *Information Processing Systems - Open Systems Interconnection - ESTELLE - a formal description technique based on an extended state transition model.* ISO 9074 (Withdrawn 1999-05-06), 1989.

[113] ISO. *Information Processing Systems - Open Systems Interconnection - LOTOS - a formal description technique based on the temporal ordering of observational behaviour.* ISO 8807:1989, 1989.

[114] ITU-T. *Information Technology - Open Systems Interconnection - Basic Reference Model: Conventions for the Definition of OSI Services.* ITU-T Recommendation X.210 | ISO/IEC 10731, November 1993.

[115] ITU-T. *Information Technology - Open Systems Interconnection - Basic Reference Model: The Basic Model.* ITU-T Recommendation X.200 | ISO/IEC 7498, July 1994.

[116] ITU-T. *Information Technology - Open Systems Interconnection - Transport Service Definition.* ITU-T Recommendation X.214 | ISO/IEC 8072, November 1995.

[117] ITU-T. *ITU-T Recommendation H.245, Audiovisual and Multimedia Systems - Control Protocol for Multimedia Communication*, May 1999.

[118] ITU-T. *Message Sequence Chart (MSC).* ITU-T Recommendation Z.120, November 1999.

[119] ITU-T. *Specification and Description Language (SDL).* ITU-T Recommendation Z.100, November 1999.

[120] JOTP Open Trading Protocol Toolkit For Java. [Last accessed: 20 May 2003] URL: `http://www.livebiz.com/jotp/body.html`.

[121] K. Jensen. An introduction to the theoretical aspects of Coloured Petri Nets. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency*, volume 803 of *Lecture Notes in Computer Science*, pages 230–272.

[122] K. Jensen. Coloured Petri Nets and the Invariant-Method. *Theoretical Computer Science*, 14:317–336, 1981.

[123] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts.* Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, second edition, 1997.

[124] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 2, Analysis Methods.* Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, second edition, 1997.

[125] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 3, Practical Use.* Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, 1997.

[126] K. Jensen. Special section on Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):95–191, 1998.

[127] K. Jensen. Special section on the practical use of high-level Petri Nets. *International Journal on Software Tools for Technology Transfer*, 3(4):369–485, 2001.

[128] K. Jensen, S. Christensen, and L. M. Kristensen. *Design/CPN Occurrence Graph Manual. Version 3.0.* Department of Computer Science, Aarhus University, Aarhus, Denmark, 1996.

[129] K. Jensen and G. Rozenberg, editors. *High-Level Petri Nets.* Springer-Verlag, Berlin, Germany, 1991.

[130] J. B Jorgensen and L. M. Kristensen. *Design/CPN OE/OS Graph Manual. Version 1.0.* Department of Computer Science, Aarhus University, Aarhus, Denmark, 1996.

[131] R. J. Linn Jr. Conformance evaluation methodology and protocol testing. *IEEE Journal on Selected Areas on Communications*, 7(7):1143–1158, September 1989.

[132] H. Kahlouche and J. J. Girardot. Design of the ISO Class 0 Tranport Protocol: A stepwise refinement based approach. In *Proceedings of IEEE International Performance, Computing and Communications Conference*, pages 363–370, Phoenix, AZ, February 1997.

[133] R. Kailar. Accountability in electronic commerce protocols. *IEEE Transactions on Software Engineering*, 22(5):313–328, 1996.

[134] Y. Kakuda, Y. Wakahara, and H. Saito. Component-based protocol synthesis. *Transactions of the Institute of Electronics and Communication Engineers of Japan, Part D*, (6):369–378, June 1991.

[135] R. Kalakota and A. B. Whinston. *Frontiers of Electronic Commerce.* Addison-Wesley, Reading, MA, 1996.

[136] Y. Kawatsura. *SET supplement for Internet Open Trading Protocol.* IETF Network Working Group, June 2003. IETF RFC 3538. Available via: `http://www.ietf.org/rfc/rfc3538`.

[137] Y. Kawatsura and K. Davidson. *Digital signatures for Internet Open Trading Protocol.* IETF Network Working Group, April 2000. IETF RFC 2802. Available via: `http://www.ietf.org/rfc/rfc2802`.

[138] D. Kelly. *Automata and Formal Languages: An Introduction.* Prentice-Hall, 1995.

[139] G.C. Kessler and N.T. Pritsky. Payment protocols: Cache on demand. *Information Security Magazine*, October 2000. Available via: `http://www.infosecuritymag.com/articles/ october00/features2.shtml`.

[140] V. Kessler and H. Neumann. A sound logic for analysing electronic commerce protocols. In *Proceedings of 5th European Symposium on Research in Computer Security (ESORICS'98)*, volume 1485 of *Lecture Notes in Computer Science*, pages 345–360, Louvain-la-Neuve, Belgium, September 1998. Springer-Verlag.

[141] P. Kimberley. *Electronic Data Interchange.* McGraw-Hill, New York, 1991.

[142] L. M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.

[143] G. Lacoste, B. Pfitzmann, M. Steiner, and M. Waidner, editors. *SEMPER - Secure Electronic Marketplace for Europe*, volume 1854 of *Lecture Notes in Computer Science.* Springer-Verlag, 2000.

[144] S. S. Lam and A. U. Shankar. A relational notation fro state transition systems. *IEEE Transactions on Software Engineering*, 16(7):755–775, July 1990.

[145] W. Lamersdorf and M. Merz, editors. *Proceedings of International IFIP/GI Working Conference on Trends in Distributed Systems for Electronic Commerce (TREC'98)*, volume 1402 of *Lecture Notes in Computer Science*, Dpunkt Verlag, Heidelberg, Germany, 3-5 June 1998. Springer-Verlag.

[146] C. Larman. *Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design.* Prentice-Hall, Upper Saddle River, N.J., 1998.

[147] H.G. Lee and T. Clark. Impacts of electronic marketplace on transaction cost and market structure. *International Journal of Electronic Commerce*, 1(1):127–149, 1996.

[148] Z. Lee and H. Yu; P. Ku. An analysis and comparison of different types of electronic payment systems. In *Proceedings of Portland International Conference on Management of Engineering and Technology. Technology Management in the Knowledge Era. Selected Papers (IEEE Cat. No.01CH37199)*, volume 2, pages 38–45. IEEE Computer Society, 2001.

[149] M. Lerner, G. Vanecek, N. Vidovic, and D. Vrsalovic. *Middleware Networks: Concept, Design, and Deployment of Internet Infrastructure.* Kluwer Academic, Boston, Mass, 2000.

[150] H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, 1998.

[151] J. Liang, R. Shi, F. Liang, and Z. Gao. Research on WAP clients supports SET payment protocol. *IEEE Wireless Communications*, 9(1):90–95, 2002.

[152] B. Lindstrom and L. Wells. *Design/CPN Performance Tool Manual. Version 1.0.* Department of Computer Science, Aarhus University, Aarhus, Denmark, 1999.

[153] L. Liu and J. Billington. Modelling and analysis of Internet multimedia protocols - methodology and initial results. In *Proceedings of 11th Annual International Symposium of the International Council on System Engineering*, 2001. 8 pages.

[154] M. T. Liu. Protocol engineering. In *Advances in Computers*, volume 29. Academic, New York, NY, 1989.

[155] M. T. Liu. Special issue on protocol engineering. *IEEE Transactions on Computers*, 40(4), April 1991.

[156] K. C. Louden. *Compiler Construction: Principles and Practice.* PWS Publishing Company, 1997.

[157] S. Lu and S. A. Smolka. Model checking the secure electronic transaction (SET) protocol. In *Proceedings of 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'99)*, pages 358–364, College Park MD, USA, 24-27 October 1999.

[158] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceshinis. *Modelling with Generalized Stochastic Petri Nets.* John Wiley & Sons, New York, 1995.

[159] D. Mcelroy and E. Turban. Using smart cards in electronic commerce. *International Journal of Information Management*, 18(1):61–72, 1998.

[160] Microsoft Inc. *Microsoft Windows NT Server Operating System (White Paper): DCOM Technical Overview*, 1996. URL: `http://www.microsoft.com/ntserver/techresources/ appserv/COM/dcomtec.asp`.

[161] Microsoft Inc. *BizTalk Framework 2.0: Document and Message Specification*, 2000. Available via: `http://www.microsoft.com/biztalk/techinfo/framwork20.asp`.

[162] Sun Microsystems. The Java Wallet architecture (white paper). Technical report, 1998. Available via: `http://java.sun.com/products/commerce/docs/whitepapers/arch/archi-tecture.pdf`.

[163] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[164] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML*. The MIT Press, revised edition, 1997.

[165] F. Moller and G. Birtwistle, editors. *Logics for Concurrency: Structure versus Automata*, volume 1043 of *Lecture Notes in Computer Science*. Springer, 1996.

[166] Mondex. URL: `http://www.mondexusa.com`.

[167] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[168] S. L. Murphy and A. U. Shankar. Connection management for the transport layer: service specification and protocol. *IEEE Transactions on Communications*, (12):1762–1775, December 1991.

[169] M. Nakamura, Y. Kakuda, and T. Kikuno. On constructing communication protocols from component-based service specifications. *Computer Communications*, 19(14):1200–1215, December 1996.

[170] S. C. Nash. Format and protocol language (FAPL). *Computer Networks and ISDN Systems*, 14(1):61–77, 1987.

[171] B. Neelakantan and S. V. Raghavan. Protocol conformance testing - A survey. In *Computer Networks, Architecture and Applications*, pages 175–191. Chapman & Hall, London, UK, 1995.

[172] OASIS and UN/CEFACT. *ebXML Technical Specifications*, 2001. Available via: `http://www.ebxml.org/specs/#technical_specifications`.

[173] OBI Consortium. *Open Buying on the Internet (OBI) Specification, Version 2.0*, June 1999. Available via: `http://www.openbuy.org/obi/specs/obiv2.pdf`.

[174] Royal Bank of Canada (RBC). URL: `http://www.royalbank.com`.

[175] National Institute of Standards, Technology, and US Department of Commerce. *Secure Hash Standard.* Federal Information Processing Standards Publication 180-1 (FIPS PUB 180-1), April 1995.

[176] Design/CPN Online. URL: `http://www.daimi.au.dk/designCPN`.

[177] C. Ouyang and J. Billington. On verifying the Internet Open Trading Protocol. In *Proceedings of 4th International Conference on Electronic Commerce and Web Technologies*, volume 2738 of *Lecture Notes in Computer Science*, pages 292–302, Prague, Czech Republic, 1-5 September 2003. Springer-Verlag.

[178] C. Ouyang and J. Billington. An improved formal specification of the Internet Open Trading Protocol. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC 2004)*, pages 779–783, Nicosia, Cyprus, 14-17 March 2004. ACM Press.

[179] C. Ouyang, L. M. Kristensen, and J. Billington. An improved architectural specification of the Internet Open Trading Protocol. In *Proceedings of 3rd Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 119–137, Aarhus, Denmark, 29-31 August 2001. DAIMI PB-554, University of Aarhus, ISSN 0105-8517.

[180] C. Ouyang, L. M. Kristensen, and J. Billington. Towards modelling and analysis of the Internet Open Trading Protocol transactions using Coloured Petri nets. In *Proceedings of 11th Annual International Symposium of the International Council on System Engineering*, Melbourne, Australia, 1-5 July 2001. Paper 6.7.3 on CD-ROM.

[181] C. Ouyang, L. M. Kristensen, and J. Billington. A formal and executable specification of the Internet Open Trading Protocol. In *Proceedings of 3rd International Conference on Electronic Commerce and Web Technologies*, volume 2455 of *Lecture Notes in Computer Science*, pages 377–387, Aix-en-Provence, France, 2-6 September 2002. Springer-Verlag.

[182] C. Ouyang, L. M. Kristensen, and J. Billington. A formal service specification of the Internet Open Trading Protocol. In *Proceedings of 23rd International Conference on Application and Theory of Petri Nets*, volume 2360 of *Lecture Notes in Computer Science*, pages 352–373, Adelaide, Australia, 24-30 June 2002. Springer-Verlag.

[183] EMVCo Overview. URL: `http://www.emvco.com`.

[184] M. Panti, L. Spalazzi, S. Tacconi, and S. Valenti. Automatic verification of security in payment protocols for electronic commerce. In *Proceedings of 4th International Conference on Enterprise Information Systems*, volume 2, pages 968–974, 2002.

[185] M. Papa, O. Bremer, J. Hale, and S. Shenoi. Formal analysis of e-commerce protocols. *IEICE Transactions on Information and Systems*, E84-D(10):1313–1323, 2001.

[186] J. H. Park and S. C. Park. Agent-based merchandise management in business-to-business electronic commerce. *Decision Support Systems*, 35(3):311–333, 2003.

[187] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[188] C. Petersburg, D. C. Yen, L. Binshan, and D. C. Chou. Smart cards in the Internet commerce era. *Journal of Internet Commerce*, 1(1):85–97, 2002.

[189] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[190] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Institut fur Intrumentelle Mathematik, Schriften des IIM Nr. 2, 1962. Second Edition.

[191] T. F. Piatkowski. Protocol engineering. In *Proceedings of IEEE International Conference on Communications (ICC)*, pages 1328–1332, 1983.

[192] J.B. Postel. *Simple Mail Transfer Protocol.* Information Sciences Institute University of Southern California, August 1982. IETF RFC 821. Available via: `http://www.ietf.org/rfc/rfc821`.

[193] DARPA Internet Program. *Transmission Control Protocol.* Information Sciences Institute University of Southern California, September 1981. IETF RFC 793. Available via: `http://www.ietf.org/rfc/rfc793`.

[194] Standard SMart Card Integrated SettLEment System Project SMILE Project. URL: `http://www.ietf.org/proceedings/99mar/slides/trade-smile-99mar`, April 1999.

[195] G. Ramakrishnan. Secure electronic transaction (SET) protocol. *Information Systems Control Journal*, 6:49–51, 2000.

[196] I. Ray and I. Ray. Failure analysis of an e-commerce protocol using model checking. In *Proceedings of 2nd International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems*, pages 176–183, Milipitas, CA, USA, June 2000. IEEE Computer Society.

[197] W. Reisig. *Petri Nets: An Introduction.* EATCS Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, 1985.

[198] T. G. Robertazzi. *Computer Networks and Systems: Queuing Theory and Performance Evaluation.* Springer-Verlag, New York, second edition, 1994.

[199] K. Sabnani. An algorithmic technique for protocol verification. *IEEE Transactions on Communications*, (8):924–931, August 1988.

[200] W. Sadiq and F. Cummins. *Developing Business Systems with CORBA.* Cambridge University Press, UK, 1998.

[201] H. Sakurada and Y. Tsukada. A role-based specification of the SET payment transaction protocol. In *Proceedings of 1st Annual Working Conference on Network Security. IFIP TC1 WG11.4*, pages 1–13, Leuven, Belgium, 26-27 November 2000.

[202] K. Saleh. Special issue on protocol engineering. *Computer Communications*, 19(14), December 1996.

[203] K. Saleh. Synthesis of communication protocols: An annotated bibliography. *Computer Communications*, 26(5):40–59, October 1996.

[204] R. Saracco and P. A. J. Tilanus. CCITT SDL: overview of the languages and its applications. *Computer Networks and ISDN Systems*, 13(2):65–74, 1987.

[205] M. Sayal, F. Casati, U. Dayal, and M. Shan. Integrating workflow management systems with business-to-business interaction standards. In *Proceedings of International Conference on Data Engineering*, pages 287–296, San Jose, CA, USA, 26 February - 1 March 2002. IEEE Computer Society.

[206] C. Schmidt and R. Muller. A framework for micropayment evaluation. *Netnomics*, 1(2):187–200, 1999.

[207] Visa Authenticated Payment Program: $3 - D\ Secure^{TM}$.
URL: `http://international.visa.com/fb/paytech/secure/main.jsp`.

[208] S. Seely. *SOAP: Cross Platform Web Service Development using XML.* Prentice-Hall, Upper Saddle River, NJ, 2002.

[209] SEMPER. Secure Electronic Marketplace for Europe.
URL: `http://www.semper.org`, January 2001.

[210] SET Secure Electronic Transaction LLC (SETCo). URL: `http://www.setco.org`.

[211] K. M. Shelfer and J. D. Procaccino. Smart card evolution. *Communications of the ACM*, 45(7):83–88, July 2002.

[212] S.S.Y. Shim, V.S. Pendyala, M. Sundaram, and J.Z. Gao. Business-to-Business E-commerce Frameworks. *IEEE Computer*, 33(10):40–47, October 2000.

[213] M. Sirbu and J. D. Tygar. Netbill: An Internet commerce system optimized for network delivered services. *IEEE Personal Communications*, 2(4):34–39, 1995.

[214] CEPSCO LLC Web site. URL: `http://www.cepsco.com`.

[215] G. Sivaradje, P. Dananjayan, and D. Saraswady. The characteristics of electronic payment schemes - prospects for the future. *IETE Technical Review*, 17(4):197–202, July-Aug. 2000.

[216] I. Sommerville. *Software Engineering.* Addison-Wesley, Reading, MA, sixth edition, 2000.

[217] B. Spencer. Introduction to the special issue on agent technologies for electronic commerce. *Computational Intelligence*, 18(4):477–481, 2002.

[218] J. M. Spivey, editor. *The Z Notation: A Reference Manual.* Prentice-Hall, second edition, 1992.

[219] W. R. Stevens. *TCP/IP Illustrated.* Addison-Wesley, Reading, MA, 1994.

[220] C. A. Sunshine, editor. *Communication Protocol Modelling.* Artech House, Dedham, MA, 1981.

[221] A. S. Tanenbaum. *Computer Networks.* Prentice-Hall, Englewood Cliffs, NJ, fourth edition, 2003.

[222] J.M. Tenenbaum, T.S. Chowdhry, and K. Hughes. Eco System: An Internet Commerce Architecture. *IEEE Computer*, 30(5):48–55, May 1997.

[223] P. Timmer. *Electronic Commerce: Strategies and Models for Business-to-Business Trading.* Wiley Series in Information Systems. John Wiley & Sons, New York, 1999.

[224] Java Commerce Toolkit. URL: `http://java.sun.com/products/commerce`.

[225] J. D. Tygar. Atomicity in Electronic Commerce. In *Proceedings of 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 8–26, Philadelphia, USA, May 1996.

[226] A. Tyree. *Digital Cash.* Butterworths, Sydney, 1997.

[227] Universal Cardholder Authentication Field ($UCAF^{TM}$). URL: `http://www.mastercardintl.com/newtechnology/ecommercesecurity/spa/ucaf.html`.

[228] J. D. Ullman. *Elements of ML Programming.* Prentice-Hall, 1998.

[229] W. van der Aalst, A. ter Hofstede, and M. Weske, editors. *Proceedings of International Conference on Business Process Management (BPM 2003)*, volume 2678 of *Lecture Note in Computer Science.* Springer-Verlag, June 2003.

[230] W. M. P. van der Aalst. Process-oriented architectures for electronic commerce and interorganizational workflow. *Information Systems*, 24(8):639–671, 1999.

[231] W. M. P. van der Aalst, J. Desel, and A. Oberweis, editors. *Business Process Management: Models, Techniques and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science.* Springer-Verlag, New York, 2000.

[232] U. Varshney, R.J. Vetter, and R. Kalakota. Mobile Commerce: A New Frontier. *IEEE Computer*, 33(10):32–38, October 2000.

[233] S. Venkataraman. Electronic payment methods for e-commerce. *Computing and Information Systems*, 6(2):53–57, May 1999.

[234] M. Verdicchio and P. S. Pietro. Model checking the secure electronic transaction protocol. In *Proceedings of 2003 International Conference on Design, Analysis, and Simulation of Distributed Systems*, pages 165–171, Orlando, FL, USA, 30 March - 3 April 2003. The Society for Modeling and Simulation International.

[235] M. E. Villapol. *Modelling and Analysis of the Resource Reservation Protocol using Coloured Petri Nets.* PhD thesis, School of Electrical and Information Engineering, University of South Australia, Adelaide, Australia, March 2003.

[236] M. E. Villapol and J. Billington. Generation of a service language for the Resource Reservation Protocol using formal methods. In *Proceedings of 11th Annual International Symposium of the International Council on Systems Engineering (INCOSE)*, 2001.

[237] Visa and MasterCard. *SET Secure Electronic Transaction Specification. Version 1.0. Vol 1-3*, May 1997. Available via: `http://www.setco.org/set_specifications.html`.

[238] G. von Bochmann and C. A. Sunshine. Formal methods for protocol specification and validation. In *Computer Networks, Architecture and Applications*, pages 513–531. Plenum Press, NY, second edition, 1989.

[239] Mondex VTP. URL: `http://www.mondexusa.com/html/content/technolo/technolo.htm`.

[240] B. Wang and D. Hutchison. Protocol testing techniques. *Computer Communications*, 10(2):79–87, April 1987.

[241] G. Wang and A. Das. Models and Protocol Structures for Software Agent Based Complex E-Commerce Transactions. In *Proceedings of 2nd International Conference on Electronic Commerce and Web Technologies*, volume 2115 of *Lecture Notes in Computer Science*, pages 8–26, Munich, Germany, 3-7 September 2001. Springer-Verlag.

[242] H. Wang. A conceptual model for virtual markets. *Information Management*, 32(3):147–161, 1997.

[243] J. Wang. *Timed Petri Nets, Theory and Application.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1998.

[244] X. Wang, S. Cheung, and J. Wei. On the modelling of document exchange processes in e-commerce protocols. In *Proceedings of 2nd Asia-Pacific Conference on Quality Software*, pages 387–396, Hong Kong, China, December 2001.

[245] X. Wang, S. Cheung, and J. Wei. A CSP and Z combined modeling of document exchange processes in e-commerce protocols. *Information and Software Technology*, 44(14):875–889, 2002.

[246] D. R. Webber. Introducing XML/EDI frameworks. *Electronic Markets*, 8(1):38–41, 1998.

[247] J. Wei, S. Cheung, and X. Wang. Exploiting automatic analysis of e-commerce protocols. In *Proceedings of 25th Annual International Computer Software and Applications Conference*, pages 55–62, Chicago, Illinois, USA, 8-12 October 2001. IEEE Computer Society.

[248] C. H. West. Generalized technique for communication protocol validation. *IBM Research and Development*, 22(4):393–404, 1978.

[249] J. C. Westland and T. H. K. Clark. *Global Electronic Commerce: Theory and Case Studies.* MIT Press, Cambridge, 1999.

[250] CEN/ISSS Electronic Commerce Workshop.
URL: `http://www.cenorm.be/isss/Workshop/ec`.

[251] Ericsson Mobility World. URL: `http://www.ericsson.com/mobilityworld`.

[252] ANSI ASC X12C. *ASC X12 Reference Model for XML Design.* ANSI ASC X12C Communications and Controls Subcommittee, July 2002. Available via: `http://www.x12.org/x12org/comments/index.cfm`.

[253] P. Zafiropulo. Towards analyzing and synthesizing protocols. *IEEE Transactions on Communications*, (4):651–660, January 1980.

[254] D. Zhou, S. Qing, and Z. Zhou. A new approach for the analysis of electronic commerce protocols. *Journal of Software*, 12(9):1316–1326, 2001.

[255] V. Zwass. Electronic commerce: structures and issues. *International Journal of Electronic Commerce*, 1(1):3–23, 1996.

[256] V. Zwass. Structure and macro-level impacts of electronic commerce: from technological infrastructure to electronic marketplaces. In K. E. Kendall, editor, *Emerging Information Technology*, pages 517–526. Sage Publications, 1999.

Appendix A

A Revised Table of Contents for RFC 2801

Based on our discussion in Section 4.3, this appendix provides a revised table of contents for the IOTP specification given in RFC 2801 [36]. For clarity, the changed sections and subsections in the revised table of contents are written in italics. Note that Appendix A (Brands) is an appendix to Section 2.3.2 (Offer Exchange) and Appendix B (IOTP Data Type Definition) provides the XML definition of IOTP Messages.

1 Background

1.1 Commerce on the Internet, a Different Model

1.2 Benefits of IOTP

1.3 Baseline IOTP

1.4 Objectives of Document

1.5 Scope of Document

1.6 Document Structure

1.7 Intended Readership and Reading Guidelines

2 Introduction

2.1 Trading Roles

2.2 IOTP Messages

2.3 Trading Message Exchanges

2.3.1 Authentication Exchange

2.3.2 Offer Exchange

2.3.3 Payment Exchange

2.3.4 Delivery Exchange

2.4 IOTP Transactions

2.5 IOTP Architecture

2.6 Security Considerations in IOTP

2.7 Scope of Baseline IOTP

3 IOTP Messages

3.1 Overview

3.1.1 IOTP Message Structure

3.1.2 Trading Blocks

3.1.3 Trading Components

3.1.4 XML Document Prolog

3.2 Transaction Reference Block
3.2.1 Transaction Id Component
3.2.2 Message Id Component
3.2.3 Related To Component
3.3 ID Attributes
3.3.1 IOTP Message ID Attribute Definition
3.3.2 Block and Component ID Attribute Definitions
3.3.3 Example of use of ID Attributes
3.4 Element References
3.5 Extending IOTP Messages
3.5.1 Extra XML Elements
3.5.2 Opaque Embedded Data
3.6 Packaged Content Element
3.6.1 Packaging HTML
3.6.2 Packaging XML
3.7 Identifying Languages

4 Trading Blocks
4.1 Trading Protocol Options Block
4.2 TPO Selection Block
4.3 Offer Response Block
4.4 Authentication Request Block
4.5 Authentication Response Block
4.6 Authentication Status Block
4.7 Payment Request Block
4.8 Payment Exchange Block
4.9 Payment Response Block
4.10 Delivery Request Block
4.11 Delivery Response Block
4.12 Inquiry Request Trading Block
4.13 Inquiry Response Trading Block
4.14 Ping Request Block
4.15 Ping Response Block
4.16 Signature Block
4.17 Error Block
4.18 Cancel Block

5 Trading Components
5.1 Protocol Options Component
5.2 Authentication Request Component
5.3 Authentication Response Component
5.4 Trading Role Information Request Component
5.5 Order Component
5.5.1 Order Description Content
5.5.2 OkFrom and OkTo Timestamps

5.6 Organisation Component
5.6.1 Organisation IDs
5.6.2 Trading Role Element
5.6.3 Contact Information Element
5.6.4 Person Name Element
5.6.5 Postal Address Element
5.7 Brand List Component
5.7.1 Brand Element
5.7.2 Protocol Brand Element
5.7.3 Protocol Amount Element
5.7.4 Currency Amount Element
5.7.5 Pay Protocol Element
5.8 Brand Selection Component
5.8.1 Brand Selection Brand Info Element
5.8.2 Brand Selection Protocol Amount Info Element
5.8.3 Brand Selection Currency Amount Info Element
5.9 Payment Component
5.10 Payment Scheme Component
5.11 Payment Receipt Component
5.12 Payment Note Component
5.13 Delivery Component
5.13.1 Delivery Data Element
5.14 Consumer Delivery Data Component
5.15 Delivery Note Component
5.16 Status Component
5.16.1 Offer Completion Codes
5.16.2 Payment Completion Codes
5.16.3 Delivery Completion Codes
5.16.4 Authentication Completion Codes
5.16.5 Undefined Completion Codes
5.16.6 Transaction Inquiry Completion Codes
5.17 Trading Role Data Component
5.17.1 Who Receives a Trading Role Data Component
5.18 Inquiry Type Component
5.19 Signature Component
5.20 Certificate Component
5.21 Error Component
5.21.1 Error Codes
5.21.2 Error Location Element

6 IOTP Transactions
6.1 Authentication and Payment-Related Transactions
6.1.1 Authentication Transaction
6.1.2 Deposit Transaction

6.1.3 Purchase Transaction
6.1.4 Refund Transaction
6.1.5 Withdrawal Transaction
6.1.6 Value Exchange Transaction
6.2 Valid Combinations of Trading Message Exchanges
6.3 Combining Authentication Transactions with other Transactions
6.4 Infrastructure Transactions
6.4.1 Transaction Status Inquiry Transaction
6.4.2 Ping Transaction

7 IOTP Error Handling
7.1 Technical Errors
7.2 Business Errors
7.3 Error Depth
7.3.1 Transport Level
7.3.2 Message Level
7.3.3 Block Level
7.4 Error Processing Guidelines
7.5 Server Role Processing Sequence
7.5.1 Initiating Transactions
7.5.2 Processing Input Messages
7.5.3 Retransmitting Messages
7.6 Client Role Processing Sequence
7.6.1 Initiating Transactions
7.6.2 Processing Input Messages
7.6.3 Retransmitting Messages

8 Transaction Cancellation
8.1 Reasons to Cancel a Transaction
8.2 Cancelling a Transaction
8.2.1 Cancelled by Consumer Role
8.2.2 Cancelled by Non-Consumer Role
8.3 Handling Cancelled Transactions
8.3.1 Consumer Role
8.3.2 Non-Consumer Role

***9* Security Considerations**
9.1 Digital Signatures in IOTP
9.1.1 How IOTP uses Digital Signatures
9.1.2 IOTP Signature Example
9.1.3 Checking a Signature is Correctly Calculated
9.2 Symmetric and Asymmetric Cryptography
9.3 Data Privacy
9.4 Payment Protocol Security
9.5 Secure and Insecure Net Locations

10 Retrieving Logos

10.1 Logo Size

10.2 Logo Color Depth

10.3 Logo Net Location Examples

***11* IANA Considerations**

11.1 Codes Controlled by IANA

11.2 Codes not controlled by IANA

***A* Brands**

A.1 Brand Definitions and Brand Selection

A.1.1 Definition of Payment Instrument

A.1.2 Definition of Brand

A.1.3 Definition of Dual Brand

A.1.4 Definition of Promotional Brand

A.1.5 Identifying Promotional Brands

A.2 Brand List Examples

A.2.1 Simple Credit Card Based Example

A.2.2 Credit Card Brand List Including Promotional Brands

A.2.3 Brand Selection Example

A.2.4 Complex Electronic Cash Based Brand List

***B* IOTP Data Type Definition**

***C* Author's Address**

***D* Full Copyright Statement**

References

Glossary

Appendix B

IOTS Parameter Description

In the Internet Open Trading Service (IOTS) Definition given in Chapter 7, each IOTS primitive has its associated parameters to convey the required control information and user data. This appendix provides a detailed description of all IOTS parameters defined in Chapter 7. For clarity, we divide these parameters into six groups (see Table B.1) and then describe each parameter in the corresponding parameter group. It should be mentioned that the following description of the IOTS parameters does not constitute a *formal* definition of all values for the parameters.

Parameter Group	Parameters
Transaction Parameters	Transaction ID, Transaction Type, Trading Organisation, Process State
Offer-related Parameters	Order Content, Selection Reply
Payment-related Parameters	Payment Method Options, Payment Method Selection, Payment Amount, Payment Type, Payment Number, Payment Instructions, Payment Method-specific Data, Payment Receipt, Payment Note, Payment Error
Delivery-related Parameters	Delivery Type, Delivery Instructions, Delivery Status, Delivery Note, Digital Content Data
Authentication-related Parameters	Authentication Data, Authentication Error
Cancel-related Parameters	Trading Phase, Cancel Reason

Table B.1: IOTS parameter groups.

B.1 Transaction Parameters

These parameters are used to convey information that is common throughout a trading transaction, and are listed below:

- **Transaction ID**[1] uniquely identifies a trading transaction that has been (or is being) carried out. This parameter MUST be present in all IOTS primitives so that each trading role entity involved in a transaction can identify the transaction. To make a transaction traceable, the transaction identifier also needs to be recorded by each trading role entity.
- **Transaction Type**[2] specifies what type of trading transaction is being carried out. This parameter MUST be present[3] since it determines the number and type of the trading

[1]This is mainly based upon Sect.3.3.1 on page 38 of RFC 2801.
[2]This is mainly based upon Sect.3.3.1 on page 38 of RFC 2801.
[3]For brevity, the phrase "this parameter must be present" hereafter should be interpreted as "this parameter must be present in the corresponding primitive(s)".

phases to be involved in the transaction. The valid values for this parameter are: *Purchase*, *Refund*, *Deposit*, *Withdrawal*, *ValueExchange*, and *Authentication*.

- **Trading Organisation**[4] describes each organisation[5] among those that are (to be) involved in a trading transaction. Examples of the information provided are contact details, company name, or postal address of the organisation concerned. Such information is usually given in terms of the corresponding trading role that each organisation acts as.

  This parameter has the following valid values: *Consumer Organisation*, *Merchant Organisation*, *Payment-Handler Organisation*, *Delivery-Handler Organisation*, *Merchant-Customer-Care Organisation*, and *DeliverTo Organisation* that identifies who is to receive the goods or services being delivered. If it is the Consumer, the *DeliverTo Organisation* should refer to the *Consumer Organisation*; otherwise, the recipient of the delivery should be provided by the Consumer at the beginning of the Delivery phase.

  This parameter MUST be present, since each organisation involved in a trading transaction should be clearly aware of the others.

- **Process State**[6] indicates the current state of the process being carried out in a specific trading phase. Examples are the processing of a payment in the Payment phase, or the processing of authentication data in the Authentication phase. This parameter MUST be present and it has the following valid values: *NotYetStarted* indicating that no process is being carried out, *InProgress* indicating that the process has started but is not yet complete, *CompletedOk* indicating that the process has completed successfully, and *Failed* indicating that the process has failed due to some error (e.g., a payment error) being detected. Note that only two values *CompletedOk* and *Failed* can be used when this parameter is conveyed by *PayResult* or *AuthResult* primitives.

B.2 Offer-related Parameters

These parameters convey information that is related to the Offer phase.

- **Order Content**[7] describes in detail the goods or services being purchased. This parameter should be present when the Transaction Type parameter has a value of *Purchase* or *Refund*.

- **Selection Reply** indicates whether or not the Merchant needs to know the Consumer's trading selection. This parameter MUST be present and it has two values: *Yes* and *No*. If *Yes*, the Consumer is required to reply with his/her trading selection; if *No*, the Consumer continues without giving such a reply.

B.3 Payment-related Parameters

There are 10 parameters defined, which convey information that is related to making a payment.

[4]This is mainly based upon Sect.7.6 on pages 102-110 of RFC 2801.

[5]The word "organisation" hereby is used in a broad sense, which means it may also refer to an individual that is participating in the trade, e.g., a customer individual or a sole trader.

[6]This is partially based upon Sect.7.16 on page 135 of RFC 2801.

[7]This is mainly based upon Sect.7.5 on pages 100-102 of RFC 2801.

- **Payment Method Options**[8] contains a list of payment brands (e.g., VISA, MasterCard, American Express, Diners Club, etc.) and currencies that are accepted by the Merchant, and the associated payment protocols (e.g., SET, Mondex VTP, CyberCash, etc.) that can be used to support the payment exchange. This parameter MUST be present to notify the Consumer about the available payment methods offered by the Merchant for making the payment.

- **Payment Method Selection**[9] identifies the payment brand, the currency and the payment protocol that are selected from the list of options provided by the Payment Method Options parameter. This parameter MUST be present to inform the Payment Handler (and the Merchant if the Selection Reply is set to *On*) about the specific payment method being selected by the Consumer.

- **Payment Amount**[10] specifies the exact amount to be paid and which currency is to be used. This parameter MUST be present and it could convey a list of amounts with corresponding currencies that can be used, e.g. 200 in Australian Dollars, 108.22 in Euros, or 106.05 in American Dollars.

- **Payment Type**[11] identifies whether one or two Payment exchanges will be involved in the current trading transaction. This parameter MUST be present so that both Consumer and Payment Handler can be aware of the number of payment exchanges that need to be invoked. The two valid values for this parameter are defined as *one* and *two*. Note that the value *two* is only used when a Value Exchange transaction occurs. Within any other trading transaction (except the Authentication transaction) involving only a single payment exchange, this parameter always has a value of *one*.

- **Payment Number**[12] indicates whether a first or a second payment exchange is being carried out in the current Payment phase. This parameter is present if there are two payment exchanges involved, otherwise it is absent.

- **Payment Instructions**[13] provides the information on how each payment exchange is to be carried out. This contains: 1) the time when the Payment Handler starts or stops to accept the invocation of a payment exchange; 2) whether or not a payment receipt needs to be digitally signed by the Payment Handler; and 3) whether one payment exchange precedes the other within a Value Exchange transaction. This parameter MUST be present, not only to instruct the Consumer to invoke a payment exchange properly, but also to inform the Payment Handler about how to process the payment exchange properly.

- **Payment Method-specific Data**[14] conveys information for a specific payment method that is used in the Payment exchange. For example, when a SET-based payment method is used, each SET message will be carried by this parameter. If this parameter is used in a *PayInvoke* primitive, it MAY or MAY NOT be present, due to the different payment

[8]This is partially based upon Sect.7.7 on pages 111-120 of RFC 2801.
[9]This is mainly based upon Sect.7.8 on pages 120-123 of RFC 2801.
[10]This is mainly based upon Sect.7.7.4 on pages 116-117 of RFC 2801.
[11]This is derived from Sect.9.1.2.5 on page 201 (line 7 from the bottom of the page) of RFC 2801.
[12]This is derived from Sect.7.9 on page 123 (last 6 lines) of RFC 2801.
[13]This is mainly based upon Sect.7.9 on pages 123-124 of RFC 2801.
[14]This is mainly based upon Sect.7.10 on pages 125-126 of RFC 2801.

systems supporting the different payment methods. If this parameter is used in a *PayData* primitive, it MUST be present, because the *PayData* primitive occurs only when payment method-specific information needs to be exchanged between the Consumer and Payment Handler.

- **Payment Receipt**[15] provides a record of the following payment information: the currency amount that has been paid by the Consumer; the date and time when the payment is completed; and the Payment Handler's digital signature if required as indicated within the Payment Instructions parameter. The presence of this parameter is determined by the Process State parameter that indicates whether the current payment exchange has completed successfully (*CompletedOk*) or failed (*Failed*): if *CompletedOk*, the *payment receipt* MUST be present to provide the Consumer with proof of payment; if *Failed*, it is absent.

- **Payment Note**[16] provides additional information about the completed payment, which is not included in the Payment Receipt parameter. For example, it could be information about the remaining balance on the Consumer's account after the current payment has been made. Similarly, this parameter will not be present if the current payment exchange has failed. Otherwise, it MAY be used if the Payment Handler has additional payment information for the Consumer.

- **Payment Error**[17] describes the errors that are detected by the Payment Handler when processing the payment made by the Consumer. This facilitates the error recovery. This parameter MUST be present so that the Consumer will know how to retry the payment, e.g., to re-select a payment method if allowed or to re-enter the card number carefully. Also, the Consumer can decide what to do next, e.g., whether to retry or to cancel the payment. This parameter has the following values:

  - *BrandNotSupp* indicating that the payment brand selected by the Consumer is not supported by the Payment Handler;
  - *CurrNotSupp* indicating that the currency used by the Consumer is not supported by either the payment instrument or the Payment Handler;
  - *InsuffFunds* indicating that insufficient funds are available for the payment to be made;
  - *InstBrandInvalid* indicating that the payment instrument used for the payment is not valid for the payment brand selected, e.g., a Visa credit card is used as the payment instrument while Mastercard (not Visa) is the payment brand that was selected;
  - *InstNotValid* indicating that the payment instrument used by the Consumer is not valid for making the current payment;
  - *BadInstrument* indicating that a problem has been detected with the payment instrument and as a result it cannot be used for the current payment;
  - *AuthError* indicating that an error has been detected during the *payment method-specific* authentication check. Payment method-specific authentication is conducted

[15]This is mainly based upon Sect.7.11 on pages 126-128 of RFC 2801.
[16]This is based upon Sect.7.12 on pages 128-129 of RFC 2801.
[17]This is mainly based upon Sect.7.16.2 on pages 138-139 of RFC 2801.

via a number of payment data exchanges and hence is different from the Authentication transaction defined in IOTP.

B.4 Delivery-related Parameters

There are 5 parameters defined as follows.

- **Delivery Type**[18] identifies whether or not the trading transaction involves a delivery, and if there is a delivery, whether it is provided by the Payment Handler or the Delivery Handler. This parameter MUST be present, so that appropriate action can be taken by the Consumer when requesting a delivery, and by the Payment Handler when the payment is complete. The three values for this parameter are defined as *ndlv*, *pdlv*, and *ddlv*, indicating whether there is no delivery (*ndlv*), a delivery to be provided by the Payment Handler (*pdlv*), or a delivery to be provided by the Delivery Handler (*ddlv*).

- **Delivery Instructions**[19] provides information that is required for the delivery of the goods or services. This contains: 1) the time when the Payment Handler or Delivery Handler starts or stops to accept the processing of a delivery request; 2) the means by which the goods or services will be delivered, e.g., by post or courier, by e-mail, or packaged as *digital content data* (to be described shortly); and 3) where the delivery is to be made (as specified by the *DeliverTo Organisation* of the Trading Organisation parameter). This parameter will not be present if the Delivery Type parameter has a value of *ndlv*. Otherwise, it MUST be present, not only to instruct the Consumer how to request a delivery properly, but also to inform the Payment Handler or Delivery Handler about how to process the delivery properly.

- **Delivery Status**[20] provides the current status information about the processing of a physical delivery. This parameter should be present if there is a physical delivery, and has the following valid values:

  - *DeliveredNoConf* indicating that the goods or services have been delivered but the confirmation of the delivery is awaited;
  - *Confirmed* indicating that the goods or services have been delivered and the confirmation of the delivery has also been received;
  - *DeliverPending* indicating that the goods or services are available and are scheduled for delivery but they have not yet been delivered;
  - *BackOrdered* indicating that the goods or services to be delivered are on order and have not yet been received by the Payment Handler or Delivery Handler, in which case, the delivery will be arranged once the goods or services are received;
  - *TempNotAvail* indicating that the goods or services are temporarily unavailable and will be available afterwards.

[18] This is mainly based upon Sect.7.13 on page 129 of RFC 2801.
[19] This is based upon Sect.7.13.1 on pages 130-132 of RFC 2801.
[20] This is partially based upon Sect.7.16.3 on pages 140-141 of RFC 2801.

- **Delivery Note**[21] contains the exact information about either a physical or electronic delivery of the goods or services, and optionally a delivery identifier specified by the Payment Handler or Delivery Handler. The information contains: the time when the goods will arrive, indication of the means by which the goods is being delivered, notification of a deferred delivery, or a delivery tracking number. This parameter MUST be present, so that the Consumer can use the information to check the delivery.

- **Digital Content Data**[22] comprises the product or service that has been purchased, if electronic, such as a piece of online music or an electronic journal.

B.5 Authentication-related Parameters

Two parameters are defined concerning authentication.

- **Authentication Data**[23] contains the Authenticatee's identity information as being required by the Authenticator. This parameter MUST be present.

- **Authentication Error**[24] describes the errors that are detected by the Authenticator when processing the authentication data. This parameter MUST be present, so that the Authenticatee will know how to retry the authentication (e.g., re-enter the user account carefully), and can decide what to do next (e.g., whether to retry or to cancel the authentication). Two values for this parameter are defined:

  - *NoAuthResp* indicating that no authentication data is provided by the Authenticatee. For example, no information has been provided when the Authenticatee is being asked for a password;
  - *AuthFailed* indicating that authentication failed the Authenticator's verification process. For example, a password provided by the Authenticatee is incorrect.

B.6 Cancel-related Parameters

Two parameters are defined.

- **Trading Phase**[25] indicates the trading phase in which the transaction is cancelled. This parameter MUST be present and has four values: *Offer*, *Payment*, *Delivery* and *Authentication*.

- **Cancel Reason**[26] provides information about transaction status or the error detected in the current trading phase (as specified by the above Trading Phase parameter), which results in the cancellation of the transaction service. If such information is not available, this parameter should at least specify which trading role cancelled the transaction. This parameter MUST be present and has the following values:

[21] This is mainly based upon Sect.7.15 on page 133 of RFC 2801.
[22] This is mainly based upon Sect.7.15 on page 134 of RFC 2801.
[23] This is mainly based upon Sect.7.3 on pages 98-99 of RFC 2801.
[24] This is derived from Sect.7.16.4 on page 143 of RFC 2801.
[25] This is derived from Sect.7.16 on page 134 and Sect.7.18 on page 146 of RFC 2801.
[26] This is partially based upon Sect.7.16.1 on page 137, Sect.7.16.2 on pages 138-139, and Sect.7.16.3 on pages 140-141 of RFC 2801.

- *PayErrNonRcvr* indicating that the error detected during payment processing is not recoverable;
- *DelivPermNotAvail* indicating that the goods or services are permanently unavailable and hence delivery cannot be carried out;
- *AuthErrNonRcvr* indicating that the error detected during authentication processing is not recoverable;
- *TimeOutNonRcvr* indicating that the information expected from a trading role entity has not been provided within the maximum time for the current transaction service to be resumed;
- *ConsCancelled* indicating that the Consumer declines to continue for some reason and cancels the transaction service;
- *MerchCancelled* indicating that the Merchant declines to continue for some reason and cancels the transaction service;
- *PaymtCancelled* indicating that the Payment Handler declines to continue for some reason and cancels the transaction service;
- *DelivCancelled* indicating that the Delivery Handler declines to continue for some reason and cancels the transaction service;
- *ProvdCancelled* indicating that some internal error has occurred that requires the IOTS Provider to cancel the transaction service.

Appendix C

IOTS State Space Analysis Results

C.1 State Space Report for the Original IOTS CPN

```
 Statistics
------------------------------------------------------------------------

  Occurrence Graph
    Nodes:  3056
    Arcs:   6717
    Secs:   16
    Status: Full

  Scc Graph
    Nodes:  2975
    Arcs:   6609
    Secs:   1

 Boundedness Properties
------------------------------------------------------------------------

  Best Integers Bounds    Upper      Lower
  IOTS_Global'C_SAP 1       1          1
  IOTS_Global'M_SAP 1       1          1
  IOTS_Global'P_SAP 1       1          1
  IOTS_Global'D_SAP 1       1          1
  Offer'ConToMer 1          1          0
  Offer'MerToCon 1          1          0
  Offer'C_Prmtrs 1          1          0
  Offer'M_Prmtrs 1          1          0
  Payment'ConToPH 1         1          0
  Payment'PHToCon 1         1          0
  Payment'P_Prmtrs 1        1          0
  Delivery'ConToDH 1        1          0
  Delivery'DHToCon 1        1          0

 Home Properties
------------------------------------------------------------------------

  Home Markings:  None

 Liveness Properties
------------------------------------------------------------------------

  Dead Markings:  799 [996,995,990,989,988,...]
  Dead Transitions Instances: None
  Live Transitions Instances: None
```

C.2 State Space Report for the Revised IOTS CPN

```
 Statistics
------------------------------------------------------------------------

  Occurrence Graph
    Nodes:  3056
    Arcs:   6732
    Secs:   16
    Status: Full

  Scc Graph
    Nodes:  2975
    Arcs:   6624
    Secs:   1

 Boundedness Properties
------------------------------------------------------------------------

  Best Integers Bounds     Upper      Lower
  IOTS_Global'C_SAP 1        1          1
  IOTS_Global'M_SAP 1        1          1
  IOTS_Global'P_SAP 1        1          1
  IOTS_Global'D_SAP 1        1          1
  Offer'ConToMer 1           1          0
  Offer'MerToCon 1           1          0
  Offer'C_Prmtrs 1           1          0
  Offer'M_Prmtrs 1           1          0
  Payment'ConToPH 1          1          0
  Payment'PHToCon 1          1          0
  Payment'P_Prmtrs 1         1          0
  Delivery'ConToDH 1         1          0
  Delivery'DHToCon 1         1          0

 Home Properties
------------------------------------------------------------------------

  Home Markings:  None

 Liveness Properties
------------------------------------------------------------------------

  Dead Markings:  799 [996,995,990,989,988,...]
  Dead Transitions Instances: None
  Live Transitions Instances: None
```

C.3 Validation of Expected Dead Markings

Figure C.1 shows the ML code of the function InValidTerminal for checking the unexpected dead markings in the IOTS state space. It uses: 1) a standard query function ListDeadMarkings to obtain all dead markings in the state space; 2) a ML function ValidTerminal to check whether a marking has the form of one of those expected dead markings, as given in Tables 9.2 and 9.3; and 3) the function PredNode defined in Design/CPN [128] for traversing the state space. Figure C.2 shows the result of checking the unexpected markings on the state spaces of the original and the revised IOTS CPN model. The function InValidTerminal returns the value of an empty list, indicating all dead markings in the state space are expected.

```
(* ---------- To check if a marking n has the form of one of the expected dead markings ---------- *)

fun ValidTerminal n =

    ( (Mark.IOTS_Global'C_SAP 1 n) == 1'STOP andalso (Mark.IOTS_Global'M_SAP 1 n) == 1'OFR_FIN andalso
      (Mark.IOTS_Global'P_SAP 1 n) == 1'STOP andalso (Mark.IOTS_Global'D_SAP 1 n) == 1'IDLE andalso
      ( (Mark.Offer'C_Prmtrs 1 n) == 1'(NA, ndlv) orelse (Mark.Offer'C_Prmtrs 1 n) == 1'(NA, pdlv) orelse
        (Mark.Offer'C_Prmtrs 1 n) == 1'(Pay2, ndlv) ) andalso
      (Mark.Offer'M_Prmtrs 1 n) == empty andalso (Mark.Payment'P_Prmtrs 1 n) == empty andalso
      (Mark.Offer'ConToMer 1 n) == 1'[ ] andalso (Mark.Offer'MerToCon 1 n) == 1'[ ] andalso
      (Mark.Payment'ConToPH 1 n) == 1'[ ] andalso (Mark.Payment'PHToCon 1 n) == 1'[ ] andalso
      (Mark.Delivery'ConToDH 1 n) == 1'[ ] andalso (Mark.Delivery'DHToCon 1 n) == 1'[ ] ) orelse

    ( (Mark.IOTS_Global'C_SAP 1 n) == 1'STOP andalso (Mark.IOTS_Global'M_SAP 1 n) == 1'OFR_FIN andalso
      (Mark.IOTS_Global'P_SAP 1 n) == 1'STOP andalso (Mark.IOTS_Global'D_SAP 1 n) == 1'STOP andalso
      (Mark.Offer'C_Prmtrs 1 n) == 1'(NA, ddlv) andalso (Mark.Offer'M_Prmtrs 1 n) == empty andalso
      (Mark.Payment'P_Prmtrs 1 n) == empty andalso
      (Mark.Offer'ConToMer 1 n) == 1'[ ] andalso (Mark.Offer'MerToCon 1 n) == 1'[ ] andalso
      (Mark.Payment'ConToPH 1 n) == 1'[ ] andalso (Mark.Payment'PHToCon 1 n) == 1'[ ] andalso
      (Mark.Delivery'ConToDH 1 n) == 1'[ ] andalso (Mark.Delivery'DHToCon 1 n) == 1'[ ] ) orelse

    ( (Mark.IOTS_Global'C_SAP 1 n) == 1'IDLE andalso (Mark.IOTS_Global'M_SAP 1 n) == 1'STOP andalso
      (Mark.IOTS_Global'P_SAP 1 n) == 1'IDLE andalso (Mark.IOTS_Global'D_SAP 1 n) == 1'IDLE andalso
      (Mark.Offer'C_Prmtrs 1 n) == empty andalso (Mark.Payment'P_Prmtrs 1 n) == empty andalso
      (Mark.Offer'ConToMer 1 n) == 1'[ ] andalso (Mark.Offer'MerToCon 1 n) == empty andalso
      (Mark.Payment'ConToPH 1 n) == 1'[ ] andalso (Mark.Payment'PHToCon 1 n) == 1'[ ] andalso
      (Mark.Delivery'ConToDH 1 n) == 1'[ ] andalso (Mark.Delivery'DHToCon 1 n) == 1'[ ] ) orelse

    ( (Mark.IOTS_Global'C_SAP 1 n) == 1'STOP andalso (Mark.IOTS_Global'M_SAP 1 n) == 1'STOP andalso
      (Mark.IOTS_Global'P_SAP 1 n) == 1'IDLE andalso (Mark.IOTS_Global'D_SAP 1 n) == 1'IDLE andalso
      (Mark.Payment'P_Prmtrs 1 n) == empty andalso
      ( (Mark.Payment'ConToPH 1 n) == 1'[ ] orelse (Mark.Payment'ConToPH 1 n) == empty ) andalso
      (Mark.Payment'PHToCon 1 n) == 1'[ ] andalso
      (Mark.Delivery'ConToDH 1 n) == 1'[ ] andalso (Mark.Delivery'DHToCon 1 n) == 1'[ ] ) orelse

    ( (Mark.IOTS_Global'C_SAP 1 n) == 1'STOP andalso (Mark.IOTS_Global'M_SAP 1 n) == 1'STOP andalso
      (Mark.IOTS_Global'P_SAP 1 n) == 1'STOP andalso (Mark.IOTS_Global'D_SAP 1 n) == 1'IDLE  andalso
      (Mark.Offer'C_Prmtrs 1 n) <> empty andalso (Mark.Offer'M_Prmtrs 1 n) == empty andalso
      (Mark.Offer'ConToMer 1 n) == empty andalso (Mark.Offer'MerToCon 1 n) == 1'[ ] andalso
      ( (Mark.Delivery'ConToDH 1 n) == 1'[ ] orelse (Mark.Delivery'ConToDH 1 n) == empty ) andalso
      (Mark.Delivery'DHToCon 1 n) == 1'[ ] ) orelse

    ( (Mark.IOTS_Global'C_SAP 1 n) == 1'STOP andalso (Mark.IOTS_Global'M_SAP 1 n) == 1'STOP andalso
      (Mark.IOTS_Global'P_SAP 1 n) == 1'STOP andalso (Mark.IOTS_Global'D_SAP 1 n) == 1'STOP  andalso
      (Mark.Offer'C_Prmtrs 1 n) == 1'(NA, ddlv) andalso (Mark.Offer'M_Prmtrs 1 n) == empty andalso
      (Mark.Payment'P_Prmtrs 1 n) == empty andalso
      (Mark.Offer'ConToMer 1 n) == empty andalso (Mark.Offer'MerToCon 1 n) == 1'[ ] andalso
      (Mark.Payment'ConToPH 1 n) == 1'[ ] andalso (Mark.Payment'PHToCon 1 n) == 1'[ ] );


(* ---------- To check all the unexpected dead markings ---------- *)

fun InValidTerminal ( ) = PredNodes ( ListDeadMarkings ( ),
                                      fn n => not (ValidTerminal n),
                                      NoLimit );
```

Figure C.1: ML code for the InValidTerminal function.

```
InValidTerminal ( );
val it = [] : Node list
```

Figure C.2: Result of checking the unexpected dead markings.

D.2 State Space to FSM Input

Standard ML [228] functions are used in Design/CPN [176] to write the state space to a text file in a format that can be read from FSM [11]. The file format used by FSM is of the form:

```
source_state_1 destination_state_1 transition_1
...
source_state_n destination_state_n transition_n
halt_state_1
...
halt_state_m
```

The source and destination state numbers correspond to the numbers of the markings in the state space. The transition numbers are those assigned to each (primitive or non-primitive) transition in the CPN model (see Section D.1). In addition, the halt states are defined by the dead markings in the state space, which represent all possible endings of primitive sequences.

D.2.1 Mapping State Space to FSA

As a representative example, the ML codes for converting the state space generated from the IOTS CPN model to FSA are listed below. The function be2int takes a binding element as input and returns an integer. The binding elements in the CPN state space represent transition occurrences. Therefore, in the state space of the IOTS CPN, each binding element that represents a primitive transition occurrence is mapped to the integer according to Table D.1, while others (e.g., the transition capture) are assigned a value of 0. Also, a state space comprises nodes and arcs, where nodes represent markings and arcs represent binding elements. The function ArcToBE is defined by Design/CPN to map an arc (a) into its binding element, and hence arc2int maps an arc in the state space to a transition in the FSA. Finally, findhalts, as defined by the function DeadMarking in Design/CPN, determines whether the marking of the specified node (n) is a dead marking in the state space, i.e., a halt state in the FSA.

```
(* For AT&T FSM INPUT -- Mapping function from OG to FSA *)
(* 15-Nov-02: IOTSglobal.sml for obtaining global IOTS language *)

fun be2int (Bind.Offer'TOreq (1,_)) = "1"
  | be2int (Bind.Offer'TOind (1,_)) = "2"
  | be2int (Bind.Offer'TOres (1,_)) = "3"
  | be2int (Bind.Offer'TOcnf (1,_)) = "4"
  | be2int (Bind.Offer'TRreq (1,_)) = "5"
  | be2int (Bind.Offer'TRind (1,_)) = "6"

  | be2int (Bind.Payment'PIreq (1,_)) = "7"
  | be2int (Bind.Payment'PIind (1,_)) = "8"
  | be2int (Bind.Payment'PDreq (1,_)) = "9"
  | be2int (Bind.Payment'PDind (1,_)) = "10"
  | be2int (Bind.Payment'PDres (1,_)) = "11"
  | be2int (Bind.Payment'PDcnf (1,_)) = "12"
  | be2int (Bind.Payment'PRSreq_s (1,_)) = "13"
  | be2int (Bind.Payment'PRSreq_f (1,_)) = "13"
  | be2int (Bind.Payment'PRSind_s (1,_)) = "14"
  | be2int (Bind.Payment'PRSind_f (1,_)) = "14"
  | be2int (Bind.Payment'PRTreq (1,_)) = "15"
  | be2int (Bind.Payment'PRTind (1,_)) = "16"

  | be2int (Bind.Delivery'Dreq_P (1,_)) = "17"
  | be2int (Bind.Delivery'Dreq_D (1,_)) = "17"
```

```
| be2int (Bind.Delivery'Dind_P (1,_)) = "18"
| be2int (Bind.Delivery'Dind_D (1,_)) = "19"
| be2int (Bind.Delivery'Dres_P (1,_)) = "20"
| be2int (Bind.Delivery'Dres_D (1,_)) = "21"
| be2int (Bind.Delivery'Dcnf_P (1,_)) = "22"
| be2int (Bind.Delivery'Dcnf_D (1,_)) = "22"

| be2int (Bind.AuthCM'Areq_C (1,_)) = "23"
| be2int (Bind.AuthCP'Areq_C (1,_)) = "23"
| be2int (Bind.AuthCD'Areq_C (1,_)) = "23"
| be2int (Bind.AuthMC'Aind_C (1,_)) = "24"
| be2int (Bind.AuthPC'Aind_C (1,_)) = "24"
| be2int (Bind.AuthDC'Aind_C (1,_)) = "24"
| be2int (Bind.AuthCM'Ares_C (1,_)) = "25"
| be2int (Bind.AuthCP'Ares_C (1,_)) = "25"
| be2int (Bind.AuthCD'Ares_C (1,_)) = "25"
| be2int (Bind.AuthMC'Acnf_C (1,_)) = "26"
| be2int (Bind.AuthPC'Acnf_C (1,_)) = "26"
| be2int (Bind.AuthDC'Acnf_C (1,_)) = "26"
| be2int (Bind.AuthCM'ARSreq_C (1,_)) = "27"
| be2int (Bind.AuthCP'ARSreq_C (1,_)) = "27"
| be2int (Bind.AuthCD'ARSreq_C (1,_)) = "27"
| be2int (Bind.AuthMC'ARSind_C   (1,_)) = "28"
| be2int (Bind.AuthPC'ARSind_s_C (1,_)) = "28"
| be2int (Bind.AuthPC'ARSind_f_C (1,_)) = "28"
| be2int (Bind.AuthDC'ARSind_C (1,_)) = "28"
| be2int (Bind.AuthCM'ARTreq_C (1,_)) = "29"
| be2int (Bind.AuthCP'ARTreq_C (1,_)) = "29"
| be2int (Bind.AuthCD'ARTreq_C (1,_)) = "29"
| be2int (Bind.AuthMC'ARTind_C (1,_)) = "30"
| be2int (Bind.AuthPC'ARTind_C (1,_)) = "30"
| be2int (Bind.AuthDC'ARTind_C (1,_)) = "30"

| be2int (Bind.AuthMC'Areq_M (1,_)) = "31"
| be2int (Bind.AuthCM'Aind_M (1,_)) = "32"
| be2int (Bind.AuthMC'Ares_M (1,_)) = "33"
| be2int (Bind.AuthCM'Acnf_M (1,_)) = "34"
| be2int (Bind.AuthMC'ARSreq_M (1,_)) = "35"
| be2int (Bind.AuthCM'ARSind_M (1,_)) = "36"
| be2int (Bind.AuthMC'ARTreq_M (1,_)) = "37"
| be2int (Bind.AuthCM'ARTind_M (1,_)) = "38"

| be2int (Bind.AuthPC'Areq_P (1,_)) = "39"
| be2int (Bind.AuthCP'Aind_P (1,_)) = "40"
| be2int (Bind.AuthPC'Ares_P (1,_)) = "41"
| be2int (Bind.AuthCP'Acnf_P (1,_)) = "42"
| be2int (Bind.AuthPC'ARSreq_s_P (1,_)) = "43"
| be2int (Bind.AuthPC'ARSreq_f_P (1,_)) = "43"
| be2int (Bind.AuthCP'ARSind_P (1,_)) = "44"
| be2int (Bind.AuthPC'ARTreq_P (1,_)) = "45"
| be2int (Bind.AuthCP'ARTind_P (1,_)) = "46"

| be2int (Bind.AuthPC'PIind (1,_)) = "8"
| be2int (Bind.AuthPC'Dind_P   (1,_)) = "18"

| be2int (Bind.AuthDC'Areq_D (1,_)) = "47"
| be2int (Bind.AuthCD'Aind_D (1,_)) = "48"
| be2int (Bind.AuthDC'Ares_D (1,_)) = "49"
| be2int (Bind.AuthCD'Acnf_D (1,_)) = "50"
| be2int (Bind.AuthDC'ARSreq_D (1,_)) = "51"
| be2int (Bind.AuthCD'ARSind_D (1,_)) = "52"
| be2int (Bind.AuthDC'ARTreq_D (1,_)) = "53"
| be2int (Bind.AuthCD'ARTind_D (1,_)) = "54"

| be2int (Bind.CnlCM'Creq_C  (1,_)) = "55"
| be2int (Bind.CnlCP'Creq_C  (1,_)) = "55"
```

```
  | be2int (Bind.CnlCD'Creq_C  (1,_)) = "55"
  | be2int (Bind.CnlCM'Cind_C  (1,_)) = "56"
  | be2int (Bind.CnlCP'Cind_C  (1,_)) = "56"
  | be2int (Bind.CnlCD'Cind_C  (1,_)) = "56"
  | be2int (Bind.CnlCM'Cindp_C (1,_)) = "56"
  | be2int (Bind.CnlCP'Cindp_C (1,_)) = "56"
  | be2int (Bind.CnlCD'Cindp_C (1,_)) = "56"

  | be2int (Bind.CnlCM'Creq_M  (1,_)) = "57"
  | be2int (Bind.CnlCM'Cind_M  (1,_)) = "58"
  | be2int (Bind.CnlCM'Cindp_M (1,_)) = "58"

  | be2int (Bind.CnlCP'Creq_P  (1,_)) = "59"
  | be2int (Bind.CnlCP'Cind_P  (1,_)) = "60"
  | be2int (Bind.CnlCP'Cindp_P (1,_)) = "60"

  | be2int (Bind.CnlCD'Creq_D  (1,_)) = "61"
  | be2int (Bind.CnlCD'Cind_D  (1,_)) = "62"
  | be2int (Bind.CnlCD'Cindp_D (1,_)) = "62"

  (* non-primitive transitions are mapped to zero *)

  | be2int (Bind.CnlCM'capture (1,_)) = "0"
  | be2int (Bind.CnlCP'capture (1,_)) = "0"
  | be2int (Bind.CnlCD'capture (1,_)) = "0"

  | be2int (_) = "ERROR";

fun arc2int(a) = be2int(ArcToBE(a));

fun findhalts n = DeadMarking(n);
```

D.2.2 Writing the State Space to FSM Input

The ML code of the two functions for writing the Design/CPN state space to FSM input text files [88] are given below. In brief, og2fsatrans writes binding elements in the state space, line by line, to a text file, and og2fsahalts writes all halt states, one per line, to a text file. A FSM input file is then obtained as a result of concatenating the output file generated from the OG (using og2fsatrans) and that including the halt states from the OG (using og2fsahalts).

```
(* Write OG to FSM input text file, line by line *)
(* in format: Source_State_No Destination_State_No Transition_No *)

fun og2fsatrans filename =
  let val outfile = open_out(filename)
      fun getfsainput (a) = output(
            outfile,
            makestring(SourceNode(a))^" "^makestring(DestNode(a))^" "^arc2int(a)^"\n")
  in
      EvalAllArcs(getfsainput);
      close_out(outfile)
  end;

(* Write the halt states for FSA from OG to FSM input text file, one per line *)
(* in format: Halt_State_No *)

fun og2fsahalts filename =
  let val outfile = open_out(filename)
      fun WriteNodes(n)=output(outfile,st_Node(n)^"\n")
  in
      SearchAllNodes(findhalts, fn n => WriteNodes(n), [], op ::);
      close_out(outfile)
  end;
```

D.3 FSA Minimisation and Comparison

FSM [11] provides the programs for FSA minimisation following the steps below:

1. Converting the input text file, which includes the FSA for the CPN state space, to the binary format used by FSM for manipulating FSA, is performed as:

```
fsmcompile inputfsa.txt > inputfsa.fsa
```

2. Removal of ϵ-transitions from the FSA is performed as:

```
fsmrmepsilon inputfsa.fsa > fsanoeps.fsa
```

3. Determinisation of the FSA is performed as:

```
fsmdeterminize fsanoeps.fsa > fsadeterm.fsa
```

4. Minimisation of the FSA is performed as:

```
fsmminimize fsadeterm.fsa > fsamin.fsa
```

Given that *fsamin1.fsa* and *fsamin2.fsa* are two minimised FSA, the comparison of these two FSA in FSM is performed as:

```
fsmdifference fsamin1.fsa fsamin2.fsa > fsadiff.fsa
```

where the *fsadiff.fsa* defines the primitive sequences that are in *fsamin1.fsa* but not in *fsamin2.fsa.*

D.4 FSA Graphics and Language Text File

FSM provides the program fsmdraw for writing a FSA (e.g., the above *fsadiff.fsa*) into a text file for graphics. This is performed as:

```
fsmdraw -i fsadiff_symbols.txt fsadiff.fsa > fsadiff_draw.txt
```

where the *fsadiff_symbols.txt* file contains the symbols (e.g, primitive events) used to label each arc in the FSA (given by *fsadiff.fsa*).

However, FSM does not have an integrated visualisation facility. GraphViz [12] is used in conjunction with FSM to generate a graphical representation of a FSA. For example, taking the output from fsmdraw (e.g., *fsadiff_draw.txt*), the program dot is used to create a PostScript version of a FSA. This is performed as:

```
dot -Tps -o fsadiff.ps fsadiff_draw.txt
```

If a FSA produced in FSM is acyclic, the complete language represented by the FSA can be written to a text file using lexfsmstring provided by LexTools [13]. This is performed as:

```
lexfsmstring -l fsanocyc_symbols.txt fsanocyc.fsa > fsanocyc_lang.txt
```

where the *fsanocyc.fsa* defines an acyclic FSA.

Appendix E

IOTP CPN Declarations

The complete set of declarations for the IOTP CPN model (created in Chapter 10) is given below. The five constants, Transaction, RCmax_c, RCmax_m, RCmax_p and RCmax_d, are defined at the end of the declarations. The value assigned to Transaction can be any valid values belonging to the colour set IotpTransType, and the values assigned to RCmax_c and RCmax_nc can be any integer.

```
(* Use an improved OG generation algorithm *)
NewOGGeneration:= true;

(* Selected Attributes of Trading Components *)
color IotpTransType = with Authentication | Purchase | Deposit | Withdrawal |
                           Refund | ValueExchange;
color Severity = with TransientError | HardError;
color ErrorCode = with MsgErr | MsgBeingProc;
color TwoAttrOfErrComp = product Severity * ErrorCode;
var trtype: IotpTransType;

(* Trading Blocks and IOTP Messages *)
color TradingBlk = union TransRefBlk:IotpTransType +
                         AuthReqBlk + AuthRespBlk + AuthStatusBlk +
                         TpoBlk + TpoSelectionBlk + OfferRespBlk +
                         PayReqBlk + PayExchBlk + PayRespBlk +
                         DeliveryReqBlk + DeliveryRespBlk +
                         ErrorBlk:TwoAttrOfErrComp +
                         CancelBlk;
color IotpMsg = list TradingBlk;
var m, rm, sm: IotpMsg;

(* IOTP Message with Message Identifier *)
color MsgId = int;
color Message = product IotpMsg * MsgId;
var id, rid, sid: MsgId;

(* Trading Role Communication Queue *)
color MsgQueue = list Message;
color TradingRole = with Consumer | Merchant | PHandler | DHandler;
color TRxTRxMQ = product TradingRole * TradingRole * MsgQueue;
var role: TradingRole;
var q, rq, sq: MsgQueue;
```

```
(* State of TRole_PEs *)
color GeneralState = with READY | COMPLETED | CANCELLED |
                          LISTEN | WAIT | HOLD | HOLD_WAIT | FINISHED;
color Exchange = with Auth | BDOfr | BIOfr | Pay1 | Pay2 | PayDlv | Deliv |
                      NoExch;
color R_Counter = int;      (* Message Retransmission Counter *)
color MIdxRC = product MsgId * R_Counter;
color State = product GeneralState * IotpTransType * Exchange * MIdxRC * MsgId;
color StaxBfr = product State * IotpMsg;
var s: GeneralState;
var exch: Exchange;
var rc: R_Counter;

(* Function to determine the initial state of a TRole_PE *)
fun InitialState(trtype, trtypevalue:IotpTransType)
  = if trtype=trtypevalue
    then 1`((READY, trtype, NoExch, (0, 0), 0), [ ])
    else empty;

(* Function to check a hard-error-reporting message and a Cancel Message *)
fun NotCnl_HrdErr([ErrorBlk(HardError, MsgErr)]) = false
  | NotCnl_HrdErr([CancelBlk]) = false
  | NotCnl_HrdErr(_) = true;

(* Type of IOTP Transaction to be examined *)
val Transaction = Purchase;

(* Maximum Retransmission Counter Values *)
val RCmax_c = 1;
val RCmax_m = 1;
val RCmax_p = 1;
val RCmax_d = 1;
```

Appendix F

IOTP State Space Analysis Results

This appendix provides the state space analysis results for the IOTP CPN model with configurations in Group 1-1 (see Section 11.3), which contain the Design/CPN state space reports and the details of dead markings in each state space. In addition, how to obtain the list of terminal SCCs from a SCC graph is also given.

F.1 State Space Reports

F.1.1 Authentication Transaction (*A-1-1*)

```
 Statistics
------------------------------------------------------------------------

  Occurrence Graph
    Nodes:  276
    Arcs:   759
    Secs:   1
    Status: Full

  Scc Graph
    Nodes:  276
    Arcs:   759
    Secs:   1

 Home Properties
------------------------------------------------------------------------

  Home Markings:  None

 Liveness Properties
------------------------------------------------------------------------

  Dead Markings:  [20,37,106,163]

  Dead Transitions Instances:
  Authenticator'HoldAuthSts 1
  Cancellation_C'SndCnl_special 1
  Cancellation_NC'LocalCnl 1
  Cancellation_NC'RcvCnl_special 1
  ErrHandling_NC'Discard_Clear_D 1
  ErrHandling_NC'Discard_Clear_P 1

  Live Transitions Instances: None
```

F.1.2 Purchase Transaction (*P-1-1*)

```
 Statistics
------------------------------------------------------------------------

  Occurrence Graph
    Nodes:  3695
    Arcs:   10021
    Secs:   28
    Status: Full

  Scc Graph
    Nodes:  3695
    Arcs:   10021
    Secs:   1

 Home Properties
------------------------------------------------------------------------

  Home Markings:  None

 Liveness Properties
------------------------------------------------------------------------

  Dead Markings:  37 [65,748,619,80,7,...]

  Dead Transitions Instances:
  Authenticator'HoldAuthSts 1
  Authenticator'HoldAuthSts 3
  Authenticator'HoldAuthSts 4
  Authenticator'HoldAuthSts 5
  Authenticator'RcvAuthRsp 1
  Authenticator'RcvAuthRsp 3
  Authenticator'RcvAuthRsp 4
  Authenticator'RcvAuthRsp 5
  Authenticator'SndAuthReq 1
  Authenticator'SndAuthReq 3
  Authenticator'SndAuthReq 4
  Authenticator'SndAuthReq 5
  Authenticator'SndAuthSts 1
  Authenticator'SndAuthSts 2
  Authenticator'SndAuthSts 3
  Authenticator'SndAuthSts 4
  Authenticator'SndAuthSts 5
  BrdDepOffer_M'RcvTPOSel 1
  BrdDepOffer_M'RcvTPOSel 3
  BrdDepOffer_M'RcvTPOSel 4
  BrdDepOffer_M'RcvTPOSel 5
  BrdDepOffer_M'SndAuthSts_TPO 1
  BrdDepOffer_M'SndAuthSts_TPO 3
  BrdDepOffer_M'SndAuthSts_TPO 4
  BrdDepOffer_M'SndAuthSts_TPO 5
  BrdDepOffer_M'SndOfrRsp 1
  BrdDepOffer_M'SndOfrRsp 3
  BrdDepOffer_M'SndOfrRsp 4
  BrdDepOffer_M'SndOfrRsp 5
  BrdDepOffer_M'SndTPO 1
  BrdDepOffer_M'SndTPO 3
  BrdDepOffer_M'SndTPO 4
  BrdDepOffer_M'SndTPO 5
```

```
BrdIndOffer_M'HoldAuthSts_TPO 1
BrdIndOffer_M'HoldAuthSts_TPO 3
BrdIndOffer_M'HoldAuthSts_TPO 4
BrdIndOffer_M'HoldAuthSts_TPO 5
BrdIndOffer_M'HoldTPO 1
BrdIndOffer_M'HoldTPO 3
BrdIndOffer_M'HoldTPO 4
BrdIndOffer_M'HoldTPO 5
BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 1
BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 3
BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 4
BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 5
BrdIndOffer_M'SndTPO_OfrRsp 1
BrdIndOffer_M'SndTPO_OfrRsp 3
BrdIndOffer_M'SndTPO_OfrRsp 4
BrdIndOffer_M'SndTPO_OfrRsp 5
Cancellation_NC'LocalCnl 1
Cancellation_NC'LocalCnl 2
Cancellation_NC'LocalCnl 4
Cancellation_NC'LocalCnl 5
Cancellation_NC'LocalCnl 6
Cancellation_NC'LocalCnl 7
Cancellation_NC'RcvCnl 2
Cancellation_NC'RcvCnl 4
Cancellation_NC'RcvCnl 5
Cancellation_NC'RcvCnl 6
Cancellation_NC'RcvCnl_special 1
Cancellation_NC'RcvCnl_special 2
Cancellation_NC'RcvCnl_special 4
Cancellation_NC'RcvCnl_special 5
Cancellation_NC'RcvCnl_special 6
Cancellation_NC'RcvCnl_special 7
Cancellation_NC'SndCnl 2
Cancellation_NC'SndCnl 4
Cancellation_NC'SndCnl 5
Cancellation_NC'SndCnl 6
ErrHandling_NC'Discard 1
ErrHandling_NC'Discard 2
ErrHandling_NC'Discard 4
ErrHandling_NC'Discard 5
ErrHandling_NC'Discard 6
ErrHandling_NC'Discard_Clear_D 2
ErrHandling_NC'Discard_Clear_D 3
ErrHandling_NC'Discard_Clear_D 4
ErrHandling_NC'Discard_Clear_D 5
ErrHandling_NC'Discard_Clear_D 6
ErrHandling_NC'Discard_Clear_D 7
ErrHandling_NC'Discard_Clear_M 1
ErrHandling_NC'Discard_Clear_M 2
ErrHandling_NC'Discard_Clear_M 4
ErrHandling_NC'Discard_Clear_M 5
ErrHandling_NC'Discard_Clear_M 6
ErrHandling_NC'Discard_Clear_M 7
ErrHandling_NC'Discard_Clear_P 1
ErrHandling_NC'Discard_Clear_P 2
ErrHandling_NC'Discard_Clear_P 3
ErrHandling_NC'Discard_Clear_P 4
```

```
  ErrHandling_NC'Discard_Clear_P 5
  ErrHandling_NC'Discard_Clear_P 6
  ErrHandling_NC'Retransmit_RstTimer 1
  ErrHandling_NC'Retransmit_RstTimer 2
  ErrHandling_NC'Retransmit_RstTimer 4
  ErrHandling_NC'Retransmit_RstTimer 5
  ErrHandling_NC'Retransmit_RstTimer 6
  ErrHandling_NC'RstTimer 1
  ErrHandling_NC'RstTimer 2
  ErrHandling_NC'RstTimer 4
  ErrHandling_NC'RstTimer 5
  ErrHandling_NC'RstTimer 6
  ErrHandling_NC'SndMsgRptHrdErr_Stop 2
  ErrHandling_NC'SndMsgRptHrdErr_Stop 4
  ErrHandling_NC'SndMsgRptHrdErr_Stop 5
  ErrHandling_NC'SndMsgRptHrdErr_Stop 6
  ErrHandling_NC'SndMsgRptTrntErr 2
  ErrHandling_NC'SndMsgRptTrntErr 4
  ErrHandling_NC'SndMsgRptTrntErr 5
  ErrHandling_NC'SndMsgRptTrntErr 6
  ErrHandling_NC'Stop_1 1
  ErrHandling_NC'Stop_1 2
  ErrHandling_NC'Stop_1 4
  ErrHandling_NC'Stop_1 5
  ErrHandling_NC'Stop_1 6
  ErrHandling_NC'Stop_4 2
  ErrHandling_NC'Stop_4 4
  ErrHandling_NC'Stop_4 5
  ErrHandling_NC'Stop_4 6
  Payment_C'SndPayReq_2 1

  Live Transitions Instances: None
```

F.1.3 Deposit, Withdrawal or Refund Transaction (*D/W/R-1-1*)

```
 Statistics
------------------------------------------------------------------------

  Occurrence Graph
    Nodes:  2465
    Arcs:   7021
    Secs:   16
    Status: Full

  Scc Graph
    Nodes:  2465
    Arcs:   7021
    Secs:   1

 Home Properties
------------------------------------------------------------------------

  Home Markings:  None

 Liveness Properties
------------------------------------------------------------------------

  Dead Markings:  19 [1065,1062,65,713,592,...]

  Dead Transitions Instances:
```

```
Authenticatee'RcvAuthReq 2
Authenticatee'RcvAuthReq 3
Authenticatee'RcvAuthSts 2
Authenticatee'RcvAuthSts 3
Authenticatee'SndAuthRsp 2
Authenticatee'SndAuthRsp 3
Authenticator'HoldAuthSts 2
Authenticator'HoldAuthSts 3
Authenticator'HoldAuthSts 4
Authenticator'HoldAuthSts 5
Authenticator'RcvAuthRsp 2
Authenticator'RcvAuthRsp 3
Authenticator'RcvAuthRsp 4
Authenticator'RcvAuthRsp 5
Authenticator'SndAuthReq 2
Authenticator'SndAuthReq 3
Authenticator'SndAuthReq 4
Authenticator'SndAuthReq 5
Authenticator'SndAuthSts 1
Authenticator'SndAuthSts 2
Authenticator'SndAuthSts 3
Authenticator'SndAuthSts 4
Authenticator'SndAuthSts 5
BrdDepOffer_C'RcvOfrRsp 2
BrdDepOffer_C'RcvOfrRsp 3
BrdDepOffer_C'RcvTPO 2
BrdDepOffer_C'RcvTPO 3
BrdDepOffer_C'SndTPOSel 2
BrdDepOffer_C'SndTPOSel 3
BrdDepOffer_M'RcvTPOSel 2
BrdDepOffer_M'RcvTPOSel 3
BrdDepOffer_M'RcvTPOSel 4
BrdDepOffer_M'RcvTPOSel 5
BrdDepOffer_M'SndAuthSts_TPO 2
BrdDepOffer_M'SndAuthSts_TPO 3
BrdDepOffer_M'SndAuthSts_TPO 4
BrdDepOffer_M'SndAuthSts_TPO 5
BrdDepOffer_M'SndOfrRsp 2
BrdDepOffer_M'SndOfrRsp 3
BrdDepOffer_M'SndOfrRsp 4
BrdDepOffer_M'SndOfrRsp 5
BrdDepOffer_M'SndTPO 2
BrdDepOffer_M'SndTPO 3
BrdDepOffer_M'SndTPO 4
BrdDepOffer_M'SndTPO 5
BrdIndOffer_C'RcvOfrRsp 2
BrdIndOffer_C'RcvOfrRsp 3
BrdIndOffer_C'RcvTPO 2
BrdIndOffer_C'RcvTPO 3
BrdIndOffer_M'HoldAuthSts_TPO 2
BrdIndOffer_M'HoldAuthSts_TPO 3
BrdIndOffer_M'HoldAuthSts_TPO 4
BrdIndOffer_M'HoldAuthSts_TPO 5
BrdIndOffer_M'HoldTPO 2
BrdIndOffer_M'HoldTPO 3
BrdIndOffer_M'HoldTPO 4
BrdIndOffer_M'HoldTPO 5
```

```
BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 2
BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 3
BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 4
BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 5
BrdIndOffer_M'SndTPO_OfrRsp 2
BrdIndOffer_M'SndTPO_OfrRsp 3
BrdIndOffer_M'SndTPO_OfrRsp 4
BrdIndOffer_M'SndTPO_OfrRsp 5
Cancellation_C'RcvCnl 2
Cancellation_C'RcvCnl 3
Cancellation_C'SndCnl 2
Cancellation_C'SndCnl 3
Cancellation_C'SndCnl_special 1
Cancellation_C'SndCnl_special 2
Cancellation_C'SndCnl_special 3
Cancellation_NC'LocalCnl 2
Cancellation_NC'LocalCnl 3
Cancellation_NC'LocalCnl 4
Cancellation_NC'LocalCnl 5
Cancellation_NC'LocalCnl 6
Cancellation_NC'LocalCnl 7
Cancellation_NC'LocalCnl 8
Cancellation_NC'RcvCnl 2
Cancellation_NC'RcvCnl 3
Cancellation_NC'RcvCnl 4
Cancellation_NC'RcvCnl 5
Cancellation_NC'RcvCnl 7
Cancellation_NC'RcvCnl 8
Cancellation_NC'RcvCnl_special 2
Cancellation_NC'RcvCnl_special 3
Cancellation_NC'RcvCnl_special 4
Cancellation_NC'RcvCnl_special 5
Cancellation_NC'RcvCnl_special 6
Cancellation_NC'RcvCnl_special 7
Cancellation_NC'RcvCnl_special 8
Cancellation_NC'SndCnl 2
Cancellation_NC'SndCnl 3
Cancellation_NC'SndCnl 4
Cancellation_NC'SndCnl 5
Cancellation_NC'SndCnl 7
Cancellation_NC'SndCnl 8
ErrHandling_C'Discard 2
ErrHandling_C'Discard 3
ErrHandling_C'Discard_Clear 2
ErrHandling_C'Discard_Clear 3
ErrHandling_C'Retransmit_RstTimer 2
ErrHandling_C'Retransmit_RstTimer 3
ErrHandling_C'RstTimer 2
ErrHandling_C'RstTimer 3
ErrHandling_C'SndMsgRptHrdErr_Stop 2
ErrHandling_C'SndMsgRptHrdErr_Stop 3
ErrHandling_C'SndMsgRptTrntErr 2
ErrHandling_C'SndMsgRptTrntErr 3
ErrHandling_C'Stop_1 2
ErrHandling_C'Stop_1 3
ErrHandling_C'Stop_4 2
ErrHandling_C'Stop_4 3
```

```
ErrHandling_NC'Discard 2
ErrHandling_NC'Discard 3
ErrHandling_NC'Discard 4
ErrHandling_NC'Discard 5
ErrHandling_NC'Discard 7
ErrHandling_NC'Discard 8
ErrHandling_NC'Discard_Clear_D 1
ErrHandling_NC'Discard_Clear_D 2
ErrHandling_NC'Discard_Clear_D 3
ErrHandling_NC'Discard_Clear_D 4
ErrHandling_NC'Discard_Clear_D 5
ErrHandling_NC'Discard_Clear_D 6
ErrHandling_NC'Discard_Clear_D 7
ErrHandling_NC'Discard_Clear_D 8
ErrHandling_NC'Discard_Clear_M 2
ErrHandling_NC'Discard_Clear_M 3
ErrHandling_NC'Discard_Clear_M 4
ErrHandling_NC'Discard_Clear_M 5
ErrHandling_NC'Discard_Clear_M 6
ErrHandling_NC'Discard_Clear_M 7
ErrHandling_NC'Discard_Clear_M 8
ErrHandling_NC'Discard_Clear_P 1
ErrHandling_NC'Discard_Clear_P 2
ErrHandling_NC'Discard_Clear_P 3
ErrHandling_NC'Discard_Clear_P 4
ErrHandling_NC'Discard_Clear_P 5
ErrHandling_NC'Discard_Clear_P 7
ErrHandling_NC'Discard_Clear_P 8
ErrHandling_NC'Retransmit_RstTimer 2
ErrHandling_NC'Retransmit_RstTimer 3
ErrHandling_NC'Retransmit_RstTimer 4
ErrHandling_NC'Retransmit_RstTimer 5
ErrHandling_NC'Retransmit_RstTimer 7
ErrHandling_NC'Retransmit_RstTimer 8
ErrHandling_NC'RstTimer 2
ErrHandling_NC'RstTimer 3
ErrHandling_NC'RstTimer 4
ErrHandling_NC'RstTimer 5
ErrHandling_NC'RstTimer 7
ErrHandling_NC'RstTimer 8
ErrHandling_NC'SndMsgRptHrdErr_Stop 2
ErrHandling_NC'SndMsgRptHrdErr_Stop 3
ErrHandling_NC'SndMsgRptHrdErr_Stop 4
ErrHandling_NC'SndMsgRptHrdErr_Stop 5
ErrHandling_NC'SndMsgRptHrdErr_Stop 7
ErrHandling_NC'SndMsgRptHrdErr_Stop 8
ErrHandling_NC'SndMsgRptTrntErr 2
ErrHandling_NC'SndMsgRptTrntErr 3
ErrHandling_NC'SndMsgRptTrntErr 4
ErrHandling_NC'SndMsgRptTrntErr 5
ErrHandling_NC'SndMsgRptTrntErr 7
ErrHandling_NC'SndMsgRptTrntErr 8
ErrHandling_NC'Stop_1 2
ErrHandling_NC'Stop_1 3
ErrHandling_NC'Stop_1 4
ErrHandling_NC'Stop_1 5
ErrHandling_NC'Stop_1 7
```

```
  ErrHandling_NC'Stop_1 8
  ErrHandling_NC'Stop_4 2
  ErrHandling_NC'Stop_4 3
  ErrHandling_NC'Stop_4 4
  ErrHandling_NC'Stop_4 5
  ErrHandling_NC'Stop_4 7
  ErrHandling_NC'Stop_4 8
  Payment_C'RcvPayExch 2
  Payment_C'RcvPayExch 3
  Payment_C'RcvPayRsp_cont 1
  Payment_C'RcvPayRsp_cont 2
  Payment_C'RcvPayRsp_cont 3
  Payment_C'RcvPayRsp_stop 2
  Payment_C'RcvPayRsp_stop 3
  Payment_C'SndPayExch 2
  Payment_C'SndPayExch 3
  Payment_C'SndPayReq_1 2
  Payment_C'SndPayReq_1 3
  Payment_C'SndPayReq_2 1
  Payment_C'SndPayReq_2 2
  Payment_C'SndPayReq_2 3
  Payment_P'RcvPayExch 2
  Payment_P'RcvPayExch 3
  Payment_P'RcvPayReq 2
  Payment_P'RcvPayReq 3
  Payment_P'SndPayExch 2
  Payment_P'SndPayExch 3
  Payment_P'SndPayRsp 2
  Payment_P'SndPayRsp 3

  Live Transitions Instances: None
```

F.1.4 Value Exchange Transaction (*V-1-1*)

```
 Statistics
-------------------------------------------------------------------------

  Occurrence Graph
    Nodes:  2919
    Arcs:   8148
    Secs:   20
    Status: Full

  Scc Graph
    Nodes:  2919
    Arcs:   8148
    Secs:   1

 Home Properties
-------------------------------------------------------------------------

  Home Markings:  None

 Liveness Properties
-------------------------------------------------------------------------

  Dead Markings:  25 [1068,65,714,592,80,...]

  Dead Transitions Instances:
  Authenticator'HoldAuthSts 1
```

```
Authenticator'HoldAuthSts 2
Authenticator'HoldAuthSts 3
Authenticator'HoldAuthSts 5
Authenticator'RcvAuthRsp 1
Authenticator'RcvAuthRsp 2
Authenticator'RcvAuthRsp 3
Authenticator'RcvAuthRsp 5
Authenticator'SndAuthReq 1
Authenticator'SndAuthReq 2
Authenticator'SndAuthReq 3
Authenticator'SndAuthReq 5
Authenticator'SndAuthSts 1
Authenticator'SndAuthSts 2
Authenticator'SndAuthSts 3
Authenticator'SndAuthSts 4
Authenticator'SndAuthSts 5
BrdDepOffer_M'RcvTPOSel 1
BrdDepOffer_M'RcvTPOSel 2
BrdDepOffer_M'RcvTPOSel 3
BrdDepOffer_M'RcvTPOSel 5
BrdDepOffer_M'SndAuthSts_TPO 1
BrdDepOffer_M'SndAuthSts_TPO 2
BrdDepOffer_M'SndAuthSts_TPO 3
BrdDepOffer_M'SndAuthSts_TPO 5
BrdDepOffer_M'SndOfrRsp 1
BrdDepOffer_M'SndOfrRsp 2
BrdDepOffer_M'SndOfrRsp 3
BrdDepOffer_M'SndOfrRsp 5
BrdDepOffer_M'SndTPO 1
BrdDepOffer_M'SndTPO 2
BrdDepOffer_M'SndTPO 3
BrdDepOffer_M'SndTPO 5
BrdIndOffer_M'HoldAuthSts_TPO 1
BrdIndOffer_M'HoldAuthSts_TPO 2
BrdIndOffer_M'HoldAuthSts_TPO 3
BrdIndOffer_M'HoldAuthSts_TPO 5
BrdIndOffer_M'HoldTPO 1
BrdIndOffer_M'HoldTPO 2
BrdIndOffer_M'HoldTPO 3
BrdIndOffer_M'HoldTPO 5
BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 1
BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 2
BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 3
BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 5
BrdIndOffer_M'SndTPO_OfrRsp 1
BrdIndOffer_M'SndTPO_OfrRsp 2
BrdIndOffer_M'SndTPO_OfrRsp 3
BrdIndOffer_M'SndTPO_OfrRsp 5
Cancellation_NC'LocalCnl 1
Cancellation_NC'LocalCnl 2
Cancellation_NC'LocalCnl 3
Cancellation_NC'LocalCnl 5
Cancellation_NC'LocalCnl 6
Cancellation_NC'RcvCnl 1
Cancellation_NC'RcvCnl 2
Cancellation_NC'RcvCnl 3
Cancellation_NC'RcvCnl 5
```

```
Cancellation_NC'RcvCnl_special 1
Cancellation_NC'RcvCnl_special 2
Cancellation_NC'RcvCnl_special 3
Cancellation_NC'RcvCnl_special 5
Cancellation_NC'RcvCnl_special 6
Cancellation_NC'SndCnl 1
Cancellation_NC'SndCnl 2
Cancellation_NC'SndCnl 3
Cancellation_NC'SndCnl 5
ErrHandling_NC'Discard 1
ErrHandling_NC'Discard 2
ErrHandling_NC'Discard 3
ErrHandling_NC'Discard 5
ErrHandling_NC'Discard_Clear_D 1
ErrHandling_NC'Discard_Clear_D 2
ErrHandling_NC'Discard_Clear_D 3
ErrHandling_NC'Discard_Clear_D 4
ErrHandling_NC'Discard_Clear_D 5
ErrHandling_NC'Discard_Clear_D 6
ErrHandling_NC'Discard_Clear_M 1
ErrHandling_NC'Discard_Clear_M 2
ErrHandling_NC'Discard_Clear_M 3
ErrHandling_NC'Discard_Clear_M 5
ErrHandling_NC'Discard_Clear_M 6
ErrHandling_NC'Discard_Clear_P 1
ErrHandling_NC'Discard_Clear_P 2
ErrHandling_NC'Discard_Clear_P 3
ErrHandling_NC'Discard_Clear_P 4
ErrHandling_NC'Discard_Clear_P 5
ErrHandling_NC'Retransmit_RstTimer 1
ErrHandling_NC'Retransmit_RstTimer 2
ErrHandling_NC'Retransmit_RstTimer 3
ErrHandling_NC'Retransmit_RstTimer 5
ErrHandling_NC'RstTimer 1
ErrHandling_NC'RstTimer 2
ErrHandling_NC'RstTimer 3
ErrHandling_NC'RstTimer 5
ErrHandling_NC'SndMsgRptHrdErr_Stop 1
ErrHandling_NC'SndMsgRptHrdErr_Stop 2
ErrHandling_NC'SndMsgRptHrdErr_Stop 3
ErrHandling_NC'SndMsgRptHrdErr_Stop 5
ErrHandling_NC'SndMsgRptTrntErr 1
ErrHandling_NC'SndMsgRptTrntErr 2
ErrHandling_NC'SndMsgRptTrntErr 3
ErrHandling_NC'SndMsgRptTrntErr 5
ErrHandling_NC'Stop_1 1
ErrHandling_NC'Stop_1 2
ErrHandling_NC'Stop_1 3
ErrHandling_NC'Stop_1 5
ErrHandling_NC'Stop_4 1
ErrHandling_NC'Stop_4 2
ErrHandling_NC'Stop_4 3
ErrHandling_NC'Stop_4 5

Live Transitions Instances: None
```

F.2 Dead Markings

Figure F.1 to F.3 show details of all dead markings in the Purchase, the Deposit and the Value Exchange transactions state spaces in Configuration Group 1-1. The Withdrawal and the Refund transactions state spaces have the same set of dead markings as the Deposit transaction state space, except for the transaction type information.

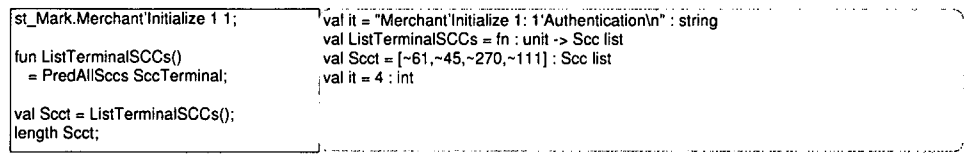
F.3 Terminal SCCs

To calculate all terminal SCCs in the SCC graph, we define the function ListTerminalSCCs as:

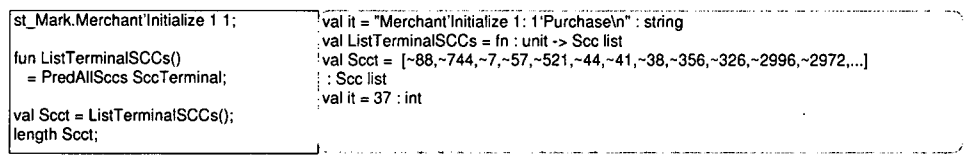
fun ListTerminalSCCs() = PredAllNodes SccTerminal

It uses PredAllSccs to search the entire SCC graph and returns a list of SCC nodes that satisfy the predicate SccTerminal to tell whether a SCC is terminal. Both PredAllSccs and SccTerminal are standard query functions.

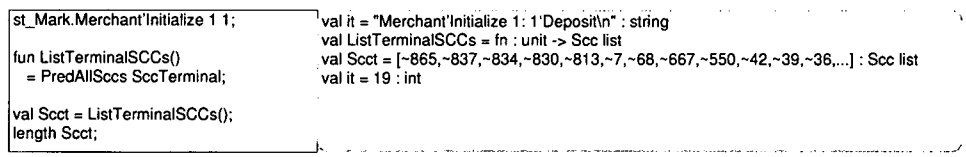
Using ListTerminalSCCs, Figure F.4 shows the result of obtaining all terminal SCCs in the transaction state spaces in Configuration Group 1-1.



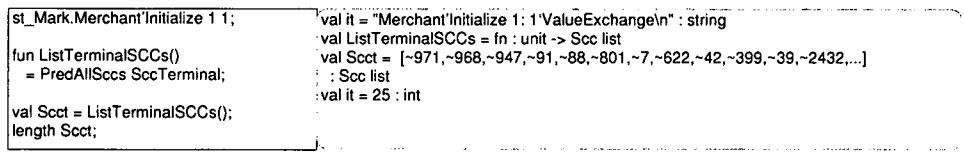
(a)



(b)



(c)



(d)

Figure F.4: Result of calculating terminal SCCs in (a) the Authentication, (b) the Purchase, (c) the Deposit and (d) the Value Exchange transactions state spaces in Configuration Group 1-1.

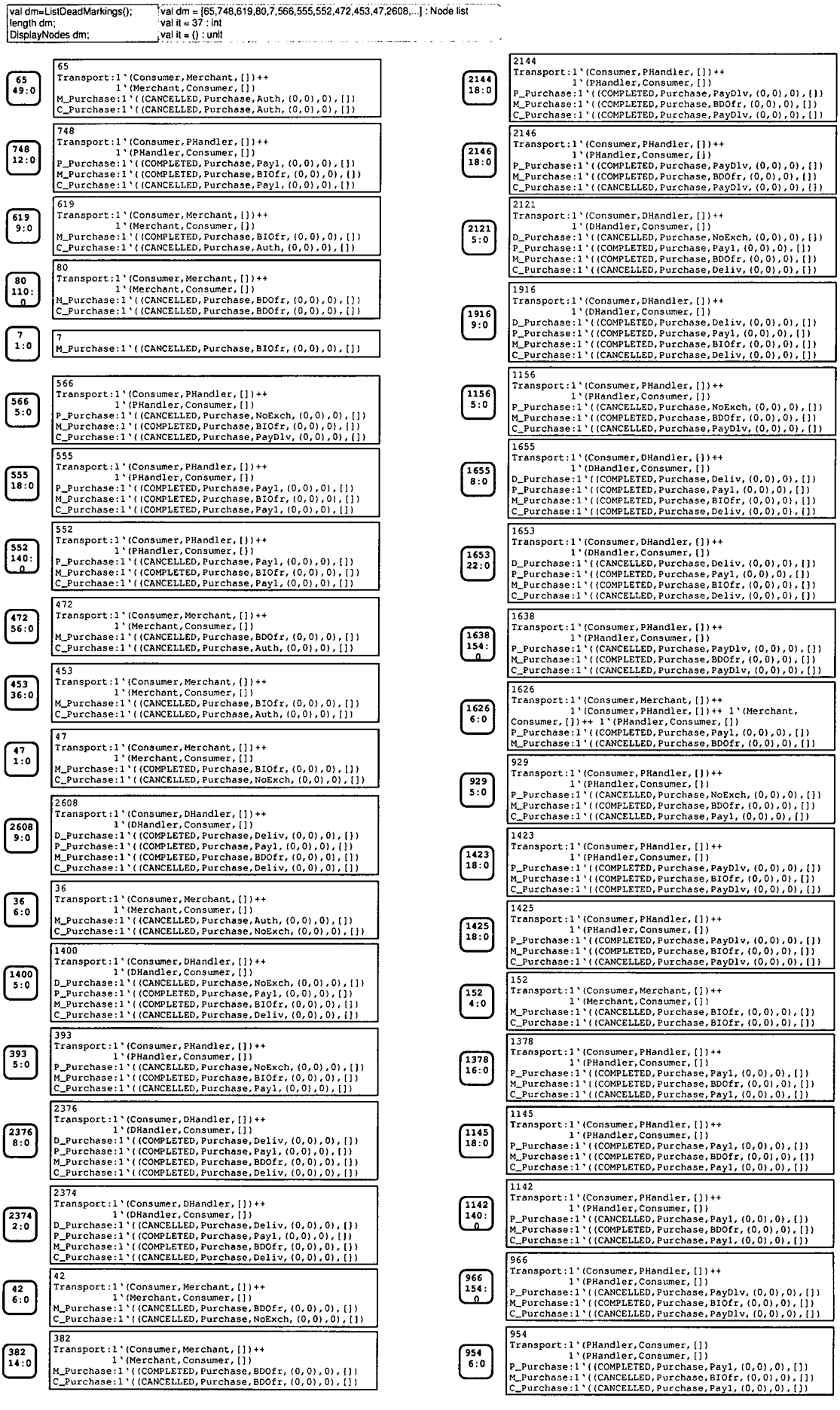


Figure F.1: Dead markings in the state space of the Purchase transaction with *Config P-1-1*.

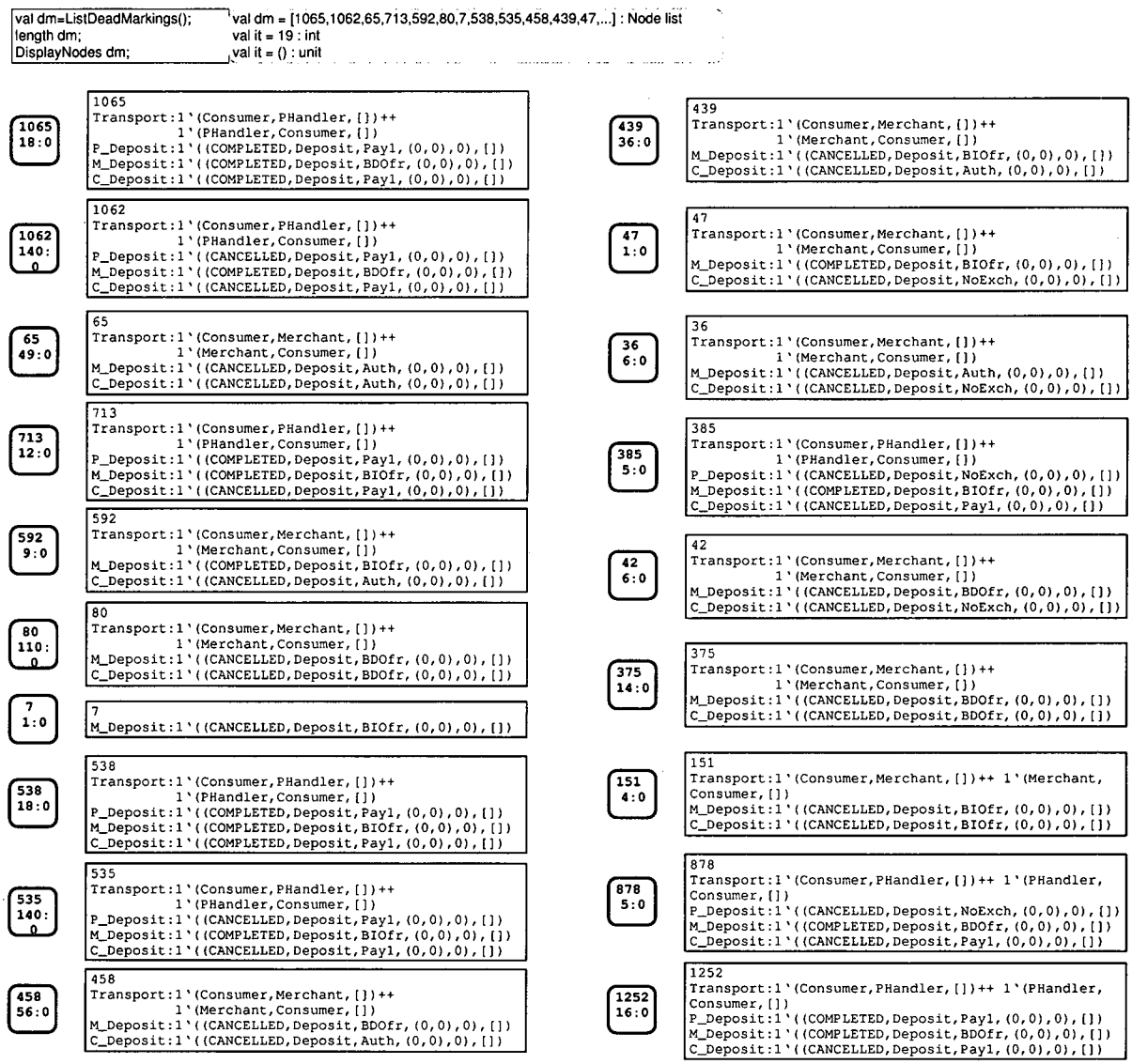


Figure F.2: Dead markings in the state space of the Deposit transaction with *Config D-1-1*.

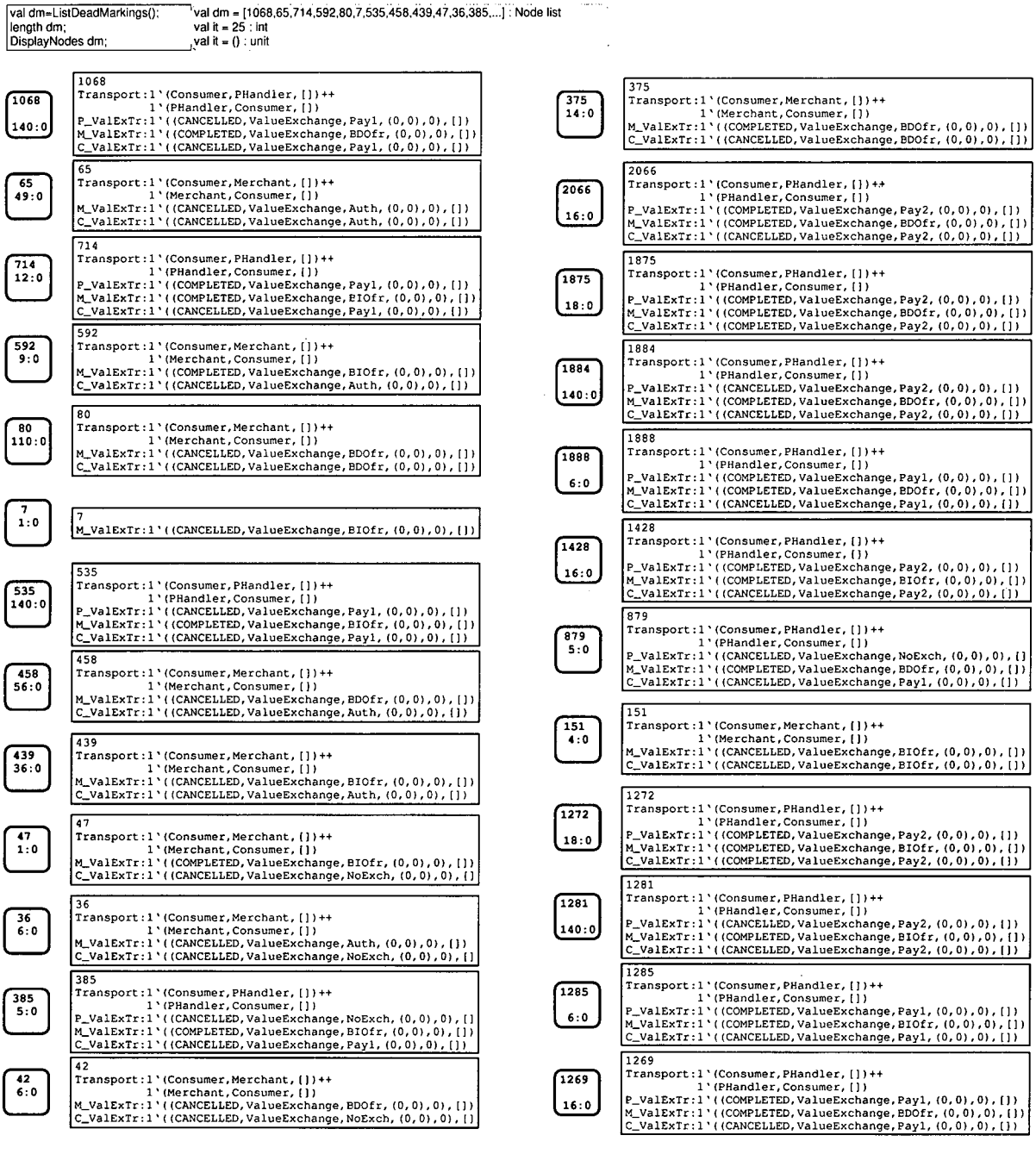


Figure F.3: Dead markings in the state space of the Value Exchange transaction with *Config V-1-1*.

Appendix G

Verification Results of IOTP

In this appendix we describe how the binding elements in the state space of the IOTP CPN model (created in Chapter 10) are mapped to integers representing service primitives in the IOTP language. As discussed in Section 12.5, the IOTS CPN model (created in Chapter 8) needs to be modified for verification of transactions with cancellation in IOTP. Also included are the FSM output for verification of IOTP.

G.1 IOTP CPN Binding Element Map Specification

The protocol language of each Authentication or Payment-related transaction, is obtained from the corresponding transaction state space using the process introduced in Section 6.4.2 and described in detail in Appendix D. Below, we list the function used for mapping the binding elements in the IOTP CPN to integers representing service primitives (as defined in Table D.1). Nearly all of the mappings can be obtained directly from the labels of the transitions in the IOTP CPN. The exceptions are the RcvCnl transitions on pages Cancellation_C (Figure 10.30) and Cancellation_NC (Figure 10.31), and the SndMsgRptHrdErr_Stop transitions on pages ErrHandling_C (Figure 10.33) and ErrHandling_NC (Figure 10.34). The labels on these four transitions indicate that a *Cancel.ind* primitive is not delivered to the TRole_User if the cancellation occurs while the TRole_PE is in the READY state (s=READY). As well as mapping binding elements to service primitives, the function findhalts specifies that all dead markings are mapped to halt states in the FSA.

It should also be mentioned that the following IOTP CPN binding element map specification is used to generate the protocol languages of transactions with cancellation. As for successful transactions, all the mappings from transitions on the four pages Cancellation_C, Cancellation_NC, ErrHandling_C and ErrHandling_NC to integers are not included.

```
(* FOR AT&T FSM INPUT -- Mapping function from OG to FSA *)
(* 19-May-03: IOTPcnlTR.sml for obtaining protocol languages of *)
(*            transactions with cancellation *)

fun be2int (Bind.Consumer'InitiateTr_C (1,_)) = "0"
  | be2int (Bind.Merchant'InitiateTr_M (1,_)) = "0"
  | be2int (Bind.PHandler'InitiateTr_P (1,_)) = "0"
  | be2int (Bind.DHandler'InitiateTr_D (1,_)) = "0"

  | be2int (Bind.Authenticator'SndAuthReq (_,_)) = "31"
  | be2int (Bind.Authenticator'RcvAuthRsp (_,_)) = "34"
  | be2int (Bind.Authenticator'SndAuthSts (_,_)) = "35"
  | be2int (Bind.Authenticator'HoldAuthSts (_,_)) = "35"
```

```
| be2int (Bind.Authenticatee'RcvAuthReq (_,_)) = "24"
| be2int (Bind.Authenticatee'SndAuthRsp (_,_)) = "25"
| be2int (Bind.Authenticatee'RcvAuthSts (_,_)) = "28"

| be2int (Bind.BrdDepOffer_M'SndTPO (_,_)) = "1"
| be2int (Bind.BrdDepOffer_M'SndAuthSts_TPO (_,_)) = "1"
| be2int (Bind.BrdDepOffer_M'RcvTPOSel (_,_)) = "4"
| be2int (Bind.BrdDepOffer_M'SndOfrRsp (_,_)) = "5"

| be2int (Bind.BrdDepOffer_C'RcvTPO (_,_)) = "2"
| be2int (Bind.BrdDepOffer_C'SndTPOSel (_,_)) = "3"
| be2int (Bind.BrdDepOffer_C'RcvOfrRsp (_,_)) = "6"

| be2int (Bind.BrdIndOffer_M'HoldTPO (_,_)) = "1"
| be2int (Bind.BrdIndOffer_M'HoldAuthSts_TPO (_,_)) = "1"
| be2int (Bind.BrdIndOffer_M'SndTPO_OfrRsp (_,_)) = "5"
| be2int (Bind.BrdIndOffer_M'SndAuthSts_TPO_OfrRsp (_,_)) = "5"

| be2int (Bind.BrdIndOffer_C'RcvTPO (_,_)) = "2"
| be2int (Bind.BrdIndOffer_C'RcvOfrRsp (_,_)) = "6"

| be2int (Bind.Payment_C'SndPayReq_1 (_,_)) = "7"
| be2int (Bind.Payment_C'SndPayReq_2 (_,_)) = "7"
| be2int (Bind.Payment_C'RcvPayExch (_,_)) = "10"
| be2int (Bind.Payment_C'SndPayExch (_,_)) = "11"
| be2int (Bind.Payment_C'RcvPayRsp_cont (_,_)) = "14"
| be2int (Bind.Payment_C'RcvPayRsp_stop (_,_)) = "14"

| be2int (Bind.Payment_P'RcvPayReq (_,_)) = "8"
| be2int (Bind.Payment_P'SndPayExch (_,_)) = "9"
| be2int (Bind.Payment_P'RcvPayExch (_,_)) = "12"
| be2int (Bind.Payment_P'SndPayRsp (_,_)) = "13"

| be2int (Bind.PayDelivery_C'HoldPayReq (_,_)) = "7"
| be2int (Bind.PayDelivery_C'SndPayDlvReq (_,_)) = "17"
| be2int (Bind.PayDelivery_C'RcvPayExch (_,_)) = "10"
| be2int (Bind.PayDelivery_C'SndPayExch (_,_)) = "11"
| be2int (Bind.PayDelivery_C'RcvPayRsp (_,_)) = "14"
| be2int (Bind.PayDelivery_C'RcvDlvRsp (_,_)) = "35"

| be2int (Bind.PayDelivery_P'RcvPayReq (_,_)) = "8"
| be2int (Bind.PayDelivery_P'RcvDlvReq (_,_)) = "18"
| be2int (Bind.PayDelivery_P'SndPayExch (_,_)) = "9"
| be2int (Bind.PayDelivery_P'RcvPayExch (_,_)) = "12"
| be2int (Bind.PayDelivery_P'HoldPayRsp (_,_)) = "13"
| be2int (Bind.PayDelivery_P'SndPayDlvRsp (_,_)) = "20"

| be2int (Bind.Delivery_C'SndDlvReq (_,_)) = "17"
| be2int (Bind.Delivery_C'RcvDlvRsp (_,_)) = "22"

| be2int (Bind.Delivery_D'RcvDlvReq (_,_)) = "19"
| be2int (Bind.Delivery_D'SndDlvRsp (_,_)) = "21"

| be2int (Bind.Cancellation_C'SndCnl (_,_)) = "55"
| be2int (Bind.Cancellation_C'SndCnl_special (_,_)) = "55"
| be2int (Bind.Cancellation_C'RcvCnl
          (_,{s=READY,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=_,id=_})) = "0"
| be2int (Bind.Cancellation_C'RcvCnl (_,_)) = "56"

| be2int (Bind.Cancellation_NC'SndCnl
          (_,{s=_,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=Merchant,q=_})) = "57"
| be2int (Bind.Cancellation_NC'SndCnl
          (_,{s=_,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=PHandler,q=_})) = "59"
| be2int (Bind.Cancellation_NC'SndCnl
          (_,{s=_,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=DHandler,q=_})) = "61"
| be2int (Bind.Cancellation_NC'LocalCnl (_,_)) = "58"
```

```
  | be2int (Bind.Cancellation_NC'RcvCnl
           (_,{s=READY,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=_,id=_})) = "0"
  | be2int (Bind.Cancellation_NC'RcvCnl
           (_,{s=_,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=Merchant,id=_})) = "58"
  | be2int (Bind.Cancellation_NC'RcvCnl
           (_,{s=_,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=PHandler,id=_})) = "60"
  | be2int (Bind.Cancellation_NC'RcvCnl
           (_,{s=_,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=DHandler,id=_})) = "62"
  | be2int (Bind.Cancellation_NC'RcvCnl_special (_,_)) = "58"

  | be2int (Bind.ErrHandling_C'Retransmit_RstTimer (_,_)) = "0"
  | be2int (Bind.ErrHandling_C'Stop_1 (_,_)) = "56"
  | be2int (Bind.ErrHandling_C'Discard_Clear (_,_)) = "0"
  | be2int (Bind.ErrHandling_C'Discard (_,_)) = "0"
  | be2int (Bind.ErrHandling_C'SntMsgRptTrntErr (_,_)) = "0"
  | be2int (Bind.ErrHandling_C'Stop_4 (_,_)) = "56"
  | be2int (Bind.ErrHandling_C'RstTimer (_,_)) = "0"

  | be2int (Bind.ErrHandling_C'SndMsgRptHrdErr_Stop
           (_,{s=READY,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,
               role=_,sq=_,m=_,id=_,rq=_})) = "0"
  | be2int (Bind.ErrHandling_C'SndMsgRptHrdErr_Stop (_,_)) = "56"

  | be2int (Bind.ErrHandling_NC'Retransmit_RstTimer (_,_)) = "0"

  | be2int (Bind.ErrHandling_NC'Stop_1
           (_,{trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=Merchant,q=_})) = "58"
  | be2int (Bind.ErrHandling_NC'Stop_1
           (_,{trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=PHandler,q=_})) = "60"
  | be2int (Bind.ErrHandling_NC'Stop_1
           (_,{trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=DHandler,q=_})) = "62"

  | be2int (Bind.ErrHandling_NC'Discard_Clear_M (_,_)) = "0"
  | be2int (Bind.ErrHandling_NC'Discard_Clear_P (_,_)) = "0"
  | be2int (Bind.ErrHandling_NC'Discard_Clear_D (_,_)) = "0"
  | be2int (Bind.ErrHandling_NC'Discard (_,_)) = "0"
  | be2int (Bind.ErrHandling_NC'SntMsgRptTrntErr (_,_)) = "0"

  | be2int (Bind.ErrHandling_NC'Stop_4
           (_,{s=_,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=Merchant,id=_})) = "58"
  | be2int (Bind.ErrHandling_NC'Stop_4
           (_,{s=_,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=PHandler,id=_})) = "60"
  | be2int (Bind.ErrHandling_NC'Stop_4
           (_,{s=_,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=DHandler,id=_})) = "62"

  | be2int (Bind.ErrHandling_NC'RstTimer (_,_)) = "0"

  | be2int (Bind.ErrHandling_NC'SndMsgRptHrdErr_Stop
           (_,{s=READY,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,
               role=_,sq=_,m=_,id=_,rq=_})) = "0"
  | be2int (Bind.ErrHandling_NC'SndMsgRptHrdErr_Stop
           (_,{s=_,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,
               role=Merchant,sq=_,m=_,id=_,rq=_})) = "58"
  | be2int (Bind.ErrHandling_NC'SndMsgRptHrdErr_Stop
           (_,{s=_,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,
               role=PHandler,sq=_,m=_,id=_,rq=_})) = "60"
  | be2int (Bind.ErrHandling_NC'SndMsgRptHrdErr_Stop
           (_,{s=_,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,
               role=DHandler,sq=_,m=_,id=_,rq=_})) = "62"

  | be2int (_) = "ERROR";

fun arc2int(a) = be2int(ArcToBE(a));

fun findhalts n = DeadMarking(n);
```

G.2 Modified IOTS CPN for Verification of IOTP

As mentioned in Section 12.5, the IOTS CPN model given in Chapter 8 is modified to capture only the acceptable subset of sequences that are implemented by IOTP for successful transactions. The changes are made to the Offer, Payment and AuthMC pages, as shown in Figure G.1, to eliminate the interleaving primitive events that are not implemented by IOTP in the Offer, Payment and Authentication (where the Merchant authenticates the Consumer) phases. All the six pages modelling the Authentication phases (see Figure 10.2) except AuthMC are not included for the verification of transactions with cancellation.

In Figure G.1 (a), the new boolean expression added to the guard on transition TOind on page Offer, models that when the Consumer's trading selection is not required (r=No) TOind cannot occur until a trdresult SDU is added to the queue as a result of the occurrence of the transition TRreq. This ensures that TOind always occurs after TRreq when r=No and thus eliminates the sequences which involve TOind occurring before TRreq. The function Is_trdresult(), as defined in Table G.1, determines if an input argument is a trading result SDU.

```
1  fun Is_trdresult ( trdresult( _ , _ ) ) = true
2      | Is_trdresult ( _ ) = false
```

Table G.1: Definition of function Is_trdresult().

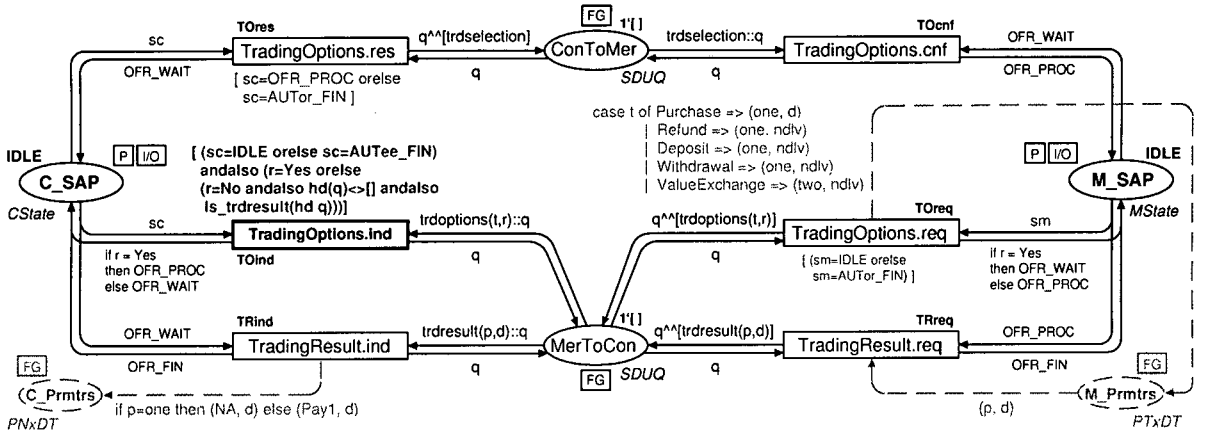
Within the scope of Baseline IOTP (Section 3.3), we have not modelled a failed payment or authentication followed by a payment or authentication retry in the IOTP CPN (Section 10.2.5), and therefore disable the corresponding primitive transitions (drawn in light gray) on the Payment page (Figure G.1 (b)) and the AuthMC page (Figure G.1 (c)). In a similar way to the changes on the Offer page, the guard on the transition PRSind_s on the modified Payment page, ensures that PRSind_s occurs after Dres_P in a Payment phase combined with a Delivery phase. On the modified AuthMC page, the arc from place MerToCon to transition ARSind_C ensures that ARSind_C occurs after TOreq, and the guard on ARSind_C ensures that ARSind_C occurs after TRreq following TOreq when r=No. Finally, due to the abstraction of hiding payment data exchanges in the verification of IOTP, the primitive transitions modelling the *PayData* services are not executed.

G.3 Minimal FSA for Service Languages

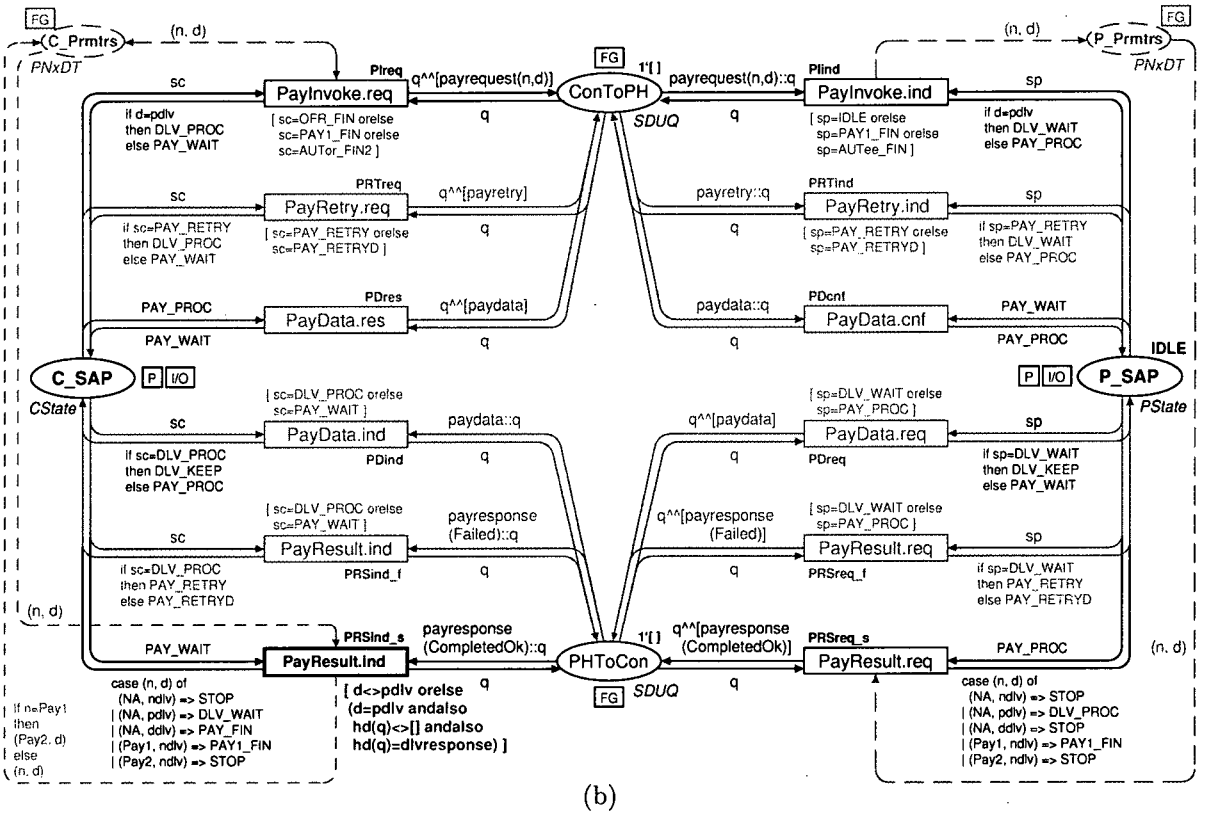
G.3.1 Successful Transactions

FSM report for the minimal FSA for the service language of the successful Authentication transaction:

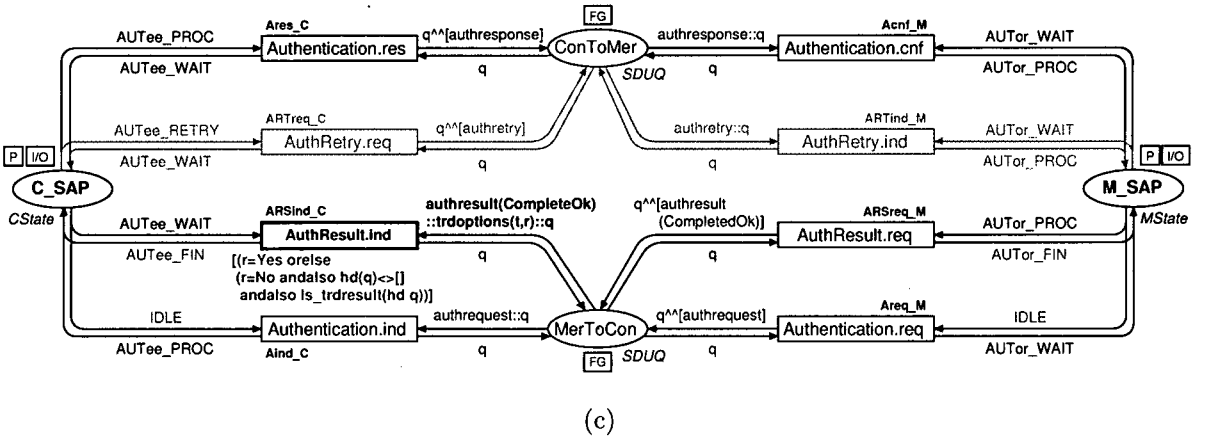
```
class                          basic
transducer                     n
# of states                    7
# of arcs                      6
initial state                  0
# of final states              1
# of eps                       0
# of accessible states         7
# of coaccessible states       7
# of connected states          7
# strongly conn components     7
```



(a)



(b)



(c)

Figure G.1: Three modified IOTS CPN pages: (a) the Offer page, (b) the Payment page and (c) the AuthMC page.

FSM report for the minimal FSA specifying the service language of the successful Deposit transaction:

```
class                          basic
transducer                     n
# of states                    20
# of arcs                      25
initial state                  0
# of final states              1
# of eps                       0
# of accessible states         20
# of coaccessible states       20
# of connected states          20
# strongly conn components     20
```

FSM report for the minimal FSA specifying the service language of the successful Purchase transaction:

```
class                          basic
transducer                     n
# of states                    30
# of arcs                      37
initial state                  0
# of final states              2
# of eps                       0
# of accessible states         30
# of coaccessible states       30
# of connected states          30
# strongly conn components     30
```

FSM report for the minimal FSA for the service language of the successful Value Exchange transaction:

```
class                          basic
transducer                     n
# of states                    24
# of arcs                      29
initial state                  0
# of final states              1
# of eps                       0
# of accessible states         24
# of coaccessible states       24
# of connected states          24
# strongly conn components     24
```

G.3.2 Transactions with Cancellation

FSM report for the minimal FSA for the service language of the Authentication transaction with cancellation:

```
class                          basic
transducer                     n
# of states                    13
# of arcs                      36
initial state                  0
# of final states              2
# of eps                       0
# of accessible states         13
# of coaccessible states       13
# of connected states          13
# strongly conn components     13
```

FSM report for the minimal FSA for the service language of the Deposit transaction with cancellation:

```
class                          basic
transducer                     n
# of states                    38
# of arcs                      119
initial state                  0
# of final states              3
# of eps                       0
# of accessible states         38
# of coaccessible states       38
# of connected states          38
# strongly conn components     38
```

FSM report for the minimal FSA for the service language of the Purchase transaction with cancellation:

```
class                          basic
transducer                     n
# of states                    54
# of arcs                      170
initial state                  0
# of final states              4
# of eps                       0
# of accessible states         54
# of coaccessible states       54
# of connected states          54
# strongly conn components     54
```

FSM report for the minimal FSA for the service language of the Value Exchange transaction with cancellation:

```
class                          basic
transducer                     n
# of states                    42
# of arcs                      130
initial state                  0
# of final states              3
# of eps                       0
# of accessible states         42
# of coaccessible states       42
# of connected states          42
# strongly conn components     42
```

G.4 Minimal FSA for Protocol Languages

G.4.1 Successful Transactions

FSM report for the minimal FSA for the protocol language of the successful Authentication transaction:

```
class                          basic
transducer                     n
# of states                    7
# of arcs                      6
initial state                  0
# of final states              1
# of eps                       0
# of accessible states         7
```

```
# of coaccessible states        7
# of connected states           7
# strongly conn components      7
```

FSM report for the minimal FSA specifying the protocol language of the successful Deposit transaction:

```
class                           basic
transducer                      n
# of states                     20
# of arcs                       23
initial state                   0
# of final states               1
# of eps                        0
# of accessible states          20
# of coaccessible states        20
# of connected states           20
# strongly conn components      20
```

FSM report for the minimal FSA for the protocol language of the successful Purchase transaction:

```
class                           basic
transducer                      n
# of states                     29
# of arcs                       32
initial state                   0
# of final states               2
# of eps                        0
# of accessible states          29
# of coaccessible states        29
# of connected states           29
# strongly conn components      29
```

FSM report for the minimal FSA for the protocol language of the the successful Value Exchange transaction:

```
class                           basic
transducer                      n
# of states                     24
# of arcs                       27
initial state                   0
# of final states               1
# of eps                        0
# of accessible states          27
# of coaccessible states        27
# of connected states           27
# strongly conn components      27
```

G.4.2 Transactions with Cancellation

FSM report for the minimal FSA for the protocol language of the Authentication transaction with cancellation:

```
class                           basic
transducer                      n
# of states                     15
# of arcs                       37
initial state                   0
# of final states               3
# of eps                        0
```

```
# of accessible states          15
# of coaccessible states        15
# of connected states           15
# strongly conn components      15
```

FSM report for the minimal FSA for the protocol language of the Deposit transaction with cancellation:

```
class                           basic
transducer                      n
# of states                     55
# of arcs                       143
initial state                   0
# of final states               9
# of eps                        0
# of accessible states          55
# of coaccessible states        55
# of connected states           55
# strongly conn components      55
```

FSM report for the minimal FSA for the protocol language of the Purchase transaction with cancellation:

```
class                           basic
transducer                      n
# of states                     77
# of arcs                       190
initial state                   0
# of final states               14
# of eps                        0
# of accessible states          77
# of coaccessible states        77
# of connected states           77
# strongly conn components      77
```

FSM report for the minimal FSA for the protocol language of the Value Exchange transaction with cancellation:

```
class                           basic
transducer                      n
# of states                     59
# of arcs                       155
initial state                   0
# of final states               9
# of eps                        0
# of accessible states          59
# of coaccessible states        59
# of connected states           59
# strongly conn components      59
```

G.5 Language Comparison Results

G.5.1 Successful Transactions

FSM report for the minimal FSA for the missing sequences in the protocol for the successful Deposit transaction:

```
class                          basic
transducer                     n
# of states                    21
# of arcs                      26
initial state                  0
# of final states              1
# of eps                       0
# of accessible states         21
# of coaccessible states       21
# of connected states          21
# strongly conn components     21
```

FSM report for the minimal FSA for the missing sequences in the protocol for the successful Purchase transaction:

```
class                          basic
transducer                     n
# of states                    44
# of arcs                      55
initial state                  0
# of final states              2
# of eps                       0
# of accessible states         44
# of coaccessible states       44
# of connected states          44
# strongly conn components     44
```

FSM report for the minimal FSA for the missing sequences in the protocol for the successful Value Exchange transaction:

```
class                          basic
transducer                     n
# of states                    25
# of arcs                      30
initial state                  0
# of final states              1
# of eps                       0
# of accessible states         25
# of coaccessible states       25
# of connected states          25
# strongly conn components     25
```

G.5.2 Transactions with Cancellation

FSM report for the minimal FSA for the missing sequences in the protocol for the Authentication transaction with cancellation:

```
class                          basic
transducer                     n
# of states                    12
# of arcs                      18
initial state                  0
# of final states              1
# of eps                       0
# of accessible states         12
# of coaccessible states       12
# of connected states          12
# strongly conn components     12
```

FSM report for the minimal FSA for the illegal sequences in the protocol for the Deposit transaction with cancellation:

```
class                          basic
transducer                     n
# of states                    33
# of arcs                      63
initial state                  0
# of final states              4
# of eps                       0
# of accessible states         33
# of coaccessible states       33
# of connected states          33
# strongly conn components     33
```

FSM report for the minimal FSA for the missing sequences in the protocol for the Deposit transaction with cancellation:

```
class                          basic
transducer                     n
# of states                    38
# of arcs                      79
initial state                  0
# of final states              1
# of eps                       0
# of accessible states         38
# of coaccessible states       38
# of connected states          38
# strongly conn components     38
```

FSM report for the minimal FSA for the illegal sequences in the protocol for the Purchase transaction with cancellation:

```
class                          basic
transducer                     n
# of states                    55
# of arcs                      117
initial state                  0
# of final states              8
# of eps                       0
# of accessible states         55
# of coaccessible states       55
# of connected states          55
# strongly conn components     55
```

FSM report for the minimal FSA for the missing sequences in the protocol for the Purchase transaction with cancellation:

```
class                          basic
transducer                     n
# of states                    54
# of arcs                      129
initial state                  0
# of final states              1
# of eps                       0
# of accessible states         54
# of coaccessible states       54
# of connected states          54
# strongly conn components     54
```

FSM report for the minimal FSA for the illegal sequences in the protocol for the Value Exchange transaction with cancellation:

```
class                          basic
transducer                     n
# of states                    37
# of arcs                      74
initial state                  0
# of final states              4
# of eps                       0
# of accessible states         37
# of coaccessible states       37
# of connected states          37
# strongly conn components     37
```

FSM report for the minimal FSA for the missing sequences in the protocol for the Value Exchange transaction with cancellation:

```
class                          basic
transducer                     n
# of states                    42
# of arcs                      89
initial state                  0
# of final states              1
# of eps                       0
# of accessible states         42
# of coaccessible states       42
# of connected states          42
# strongly conn components     42
```

Appendix H

Revised IOTP CPN and its Verification Results

This appendix presents the revised IOTP CPN model, its state space analysis results, and the FSM output for its verification against IOTS.

H.1 Revised IOTP CPN Model

The revised IOTP CPN model implements the suggested changes to IOTP that are given in Chapter 13. These changes are made to the global declarations and 12 exchange level pages in the CPN model.

H.1.1 Global Declarations

```
(* Use an improved OG generation algorithm *)
NewOGGeneration:= true;

(* Selected Attributes of Trading Components *)
color IotpTransType = with Authentication | Purchase | Deposit | Withdrawal |
                           Refund | ValueExchange;
color Severity = with TransientError | HardError;
color ErrorCode = with MsgErr | MsgBeingProc;
color TwoAttrOfErrComp = product Severity * ErrorCode;
var trtype: IotpTransType;

color DelivExch = with True | False;  (* Newly Added *)
var dlv: DelivExch;  (* Newly Added *)

(* Trading Blocks and IOTP Messages *)
color TradingBlk = union TransRefBlk:IotpTransType +
                         AuthReqBlk + AuthRespBlk + AuthStatusBlk +
                         TpoBlk + TpoSelectionBlk + OfferRespBlk:DelivExch +
                         PayReqBlk + PayExchBlk + PayRespBlk +
                         DeliveryReqBlk + DeliveryRespBlk +
                         ErrorBlk:TwoAttrOfErrComp +
                         CancelBlk;
color IotpMsg = list TradingBlk;
var m, rm, sm: IotpMsg;

(* IOTP Message with Message Identifier *)
color MsgId = int;
color Message = product IotpMsg * MsgId;
var id, rid, sid: MsgId;
```

```
(* Trading Role Communication Queue *)
color MsgQueue = list Message;
color TradingRole = with Consumer | Merchant | PHandler | DHandler;
color TRxTRxMQ = product TradingRole * TradingRole * MsgQueue;
var role: TradingRole;
var q, rq, sq: MsgQueue;

(* State of TRole_PEs *)
color GeneralState = with READY | COMPLETED | CANCELLED |
                         LISTEN | WAIT | HOLD | HOLD_WAIT | FINISHED;
color Exchange = with Auth | BDOfr | BIOfr | Pay1 | Pay2 | PayDlv | Deliv |
                      NoExch;
color R_Counter = int;      (* Message Retransmission Counter *)
color MIdxRC = product MsgId * R_Counter;
color State = product GeneralState * IotpTransType * Exchange * MIdxRC * MsgId;
color StaxBfr = product State * IotpMsg;
var s: GeneralState;
var exch: Exchange;
var rc: R_Counter;

(* Function to determine the initial state of a TRole_PE *)
fun InitialState(trtype, trtypevalue:IotpTransType)
  = if trtype=trtypevalue
    then 1`((READY, trtype, NoExch, (0, 0), 0), [ ])
    else empty;

(* Function to check a hard-error-reporting message and a Cancel Message *)
fun NotCnl_HrdErr([ErrorBlk(HardError, MsgErr)]) = false
  | NotCnl_HrdErr([CancelBlk]) = false
  | NotCnl_HrdErr(_) = true;

(* New: Function to check a response message in Payment and Delivery exchanges *)
fun IsPayDlvRsp([PayRespBlk]) = true
  | IsPayDlvRsp([DeliveryRespBlk]) = true
  | IsPayDlvRsp([PayRespBlk,DeliveryRespBlk]) = true
  | IsPayDlvRsp( _ ) = false;

(* New: Function to check if a Payment Response is the last message in a transaction *)
fun NotLastRsp(Purchase,[PayRespBlk],[OfferRespBlk(True)]) = true
  | NotLastRsp(ValueExchange,[PayRespBlk],[OfferRespBlk(False)]) = true
  | NotLastRsp( _,_,_ ) = false;

(* New: Function to unicast or multicast a message, always involving Merchant *)
fun Uni_Multicast(sndr:TradingRole,rcvr:TradingRole,m:IotpMsg,id:MsgId,q:MsgQueue)
  = if (sndr=Merchant orelse rcvr=Merchant)
    then 1`(sndr,rcvr,q^^[(m,id)])
    else 1`(sndr,rcvr,q^^[(m,id)])++1`(sndr,Merchant,[(m,id)]);

(* Type of IOTP Transaction to be examined *)
val Transaction = Purchase;

(* Maximum Retransmission Counter Values *)
val RCmax_c = 1;
val RCmax_m = 1;
val RCmax_p = 1;
val RCmax_d = 1;
```

H.1.2 The Revised CPN Pages

Figure H.1 to H.12 show the 12 revised IOTP CPN pages. The arc inscriptions and guards that have been changed are given in bold, and new transitions are outlined by a thick line.

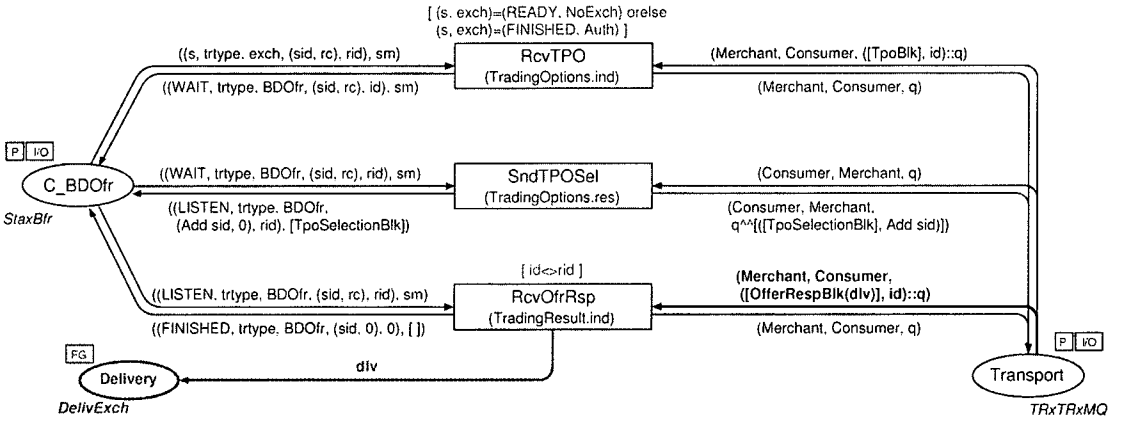


Figure H.1: BrdDepOffer_C page in the revised IOTP CPN.

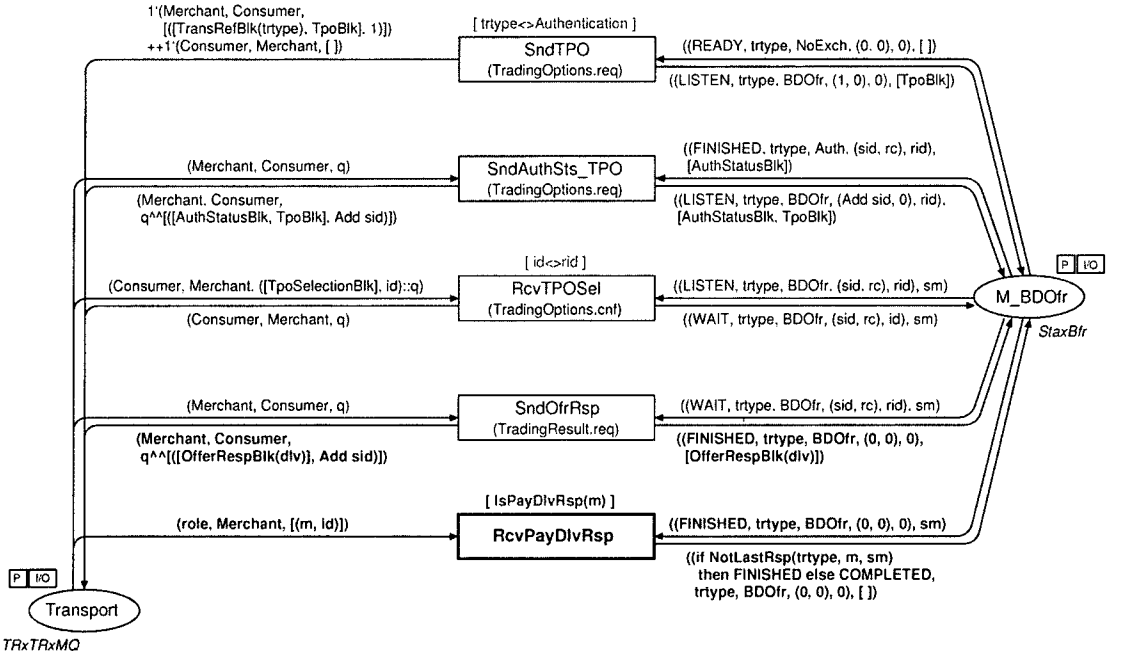


Figure H.2: BrdDepOffer_M page in the revised IOTP CPN.

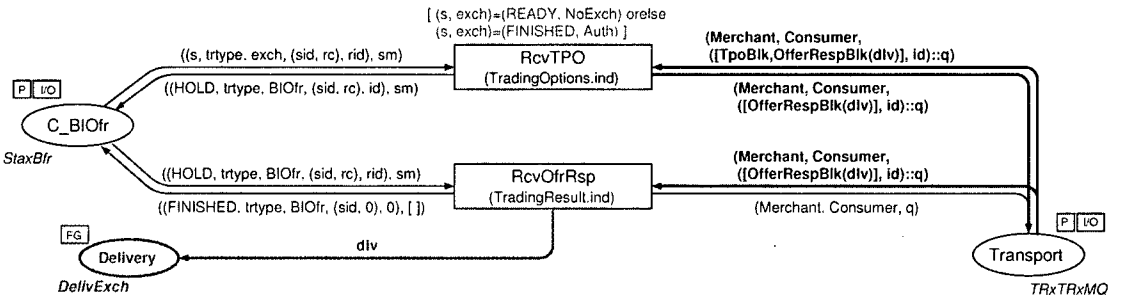


Figure H.3: BrdIndOffer_C page in the revised IOTP CPN.

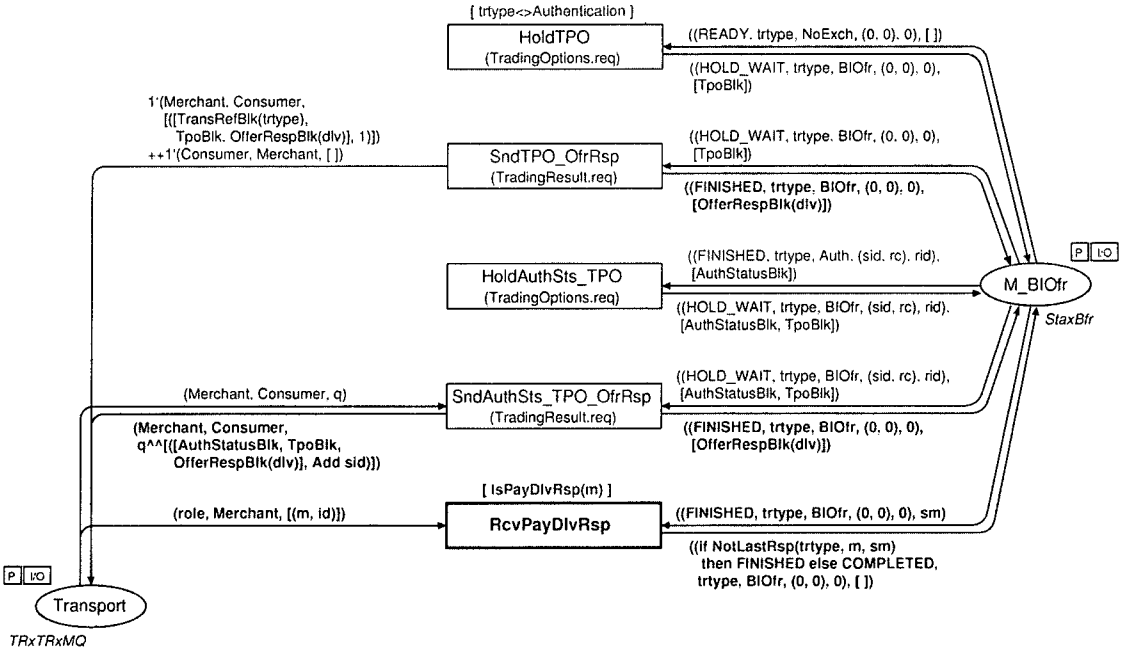


Figure H.4: BrdIndOffer_M page in the revised IOTP CPN.

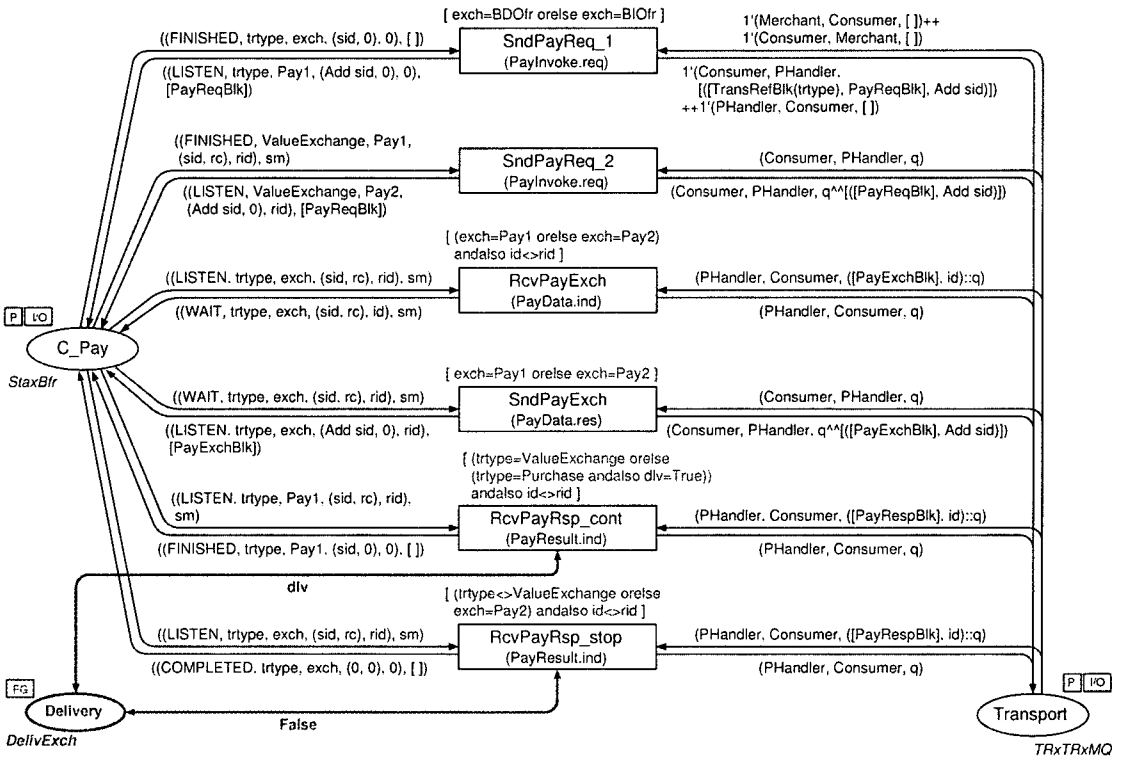


Figure H.5: Payment_C page in the revised IOTP CPN.

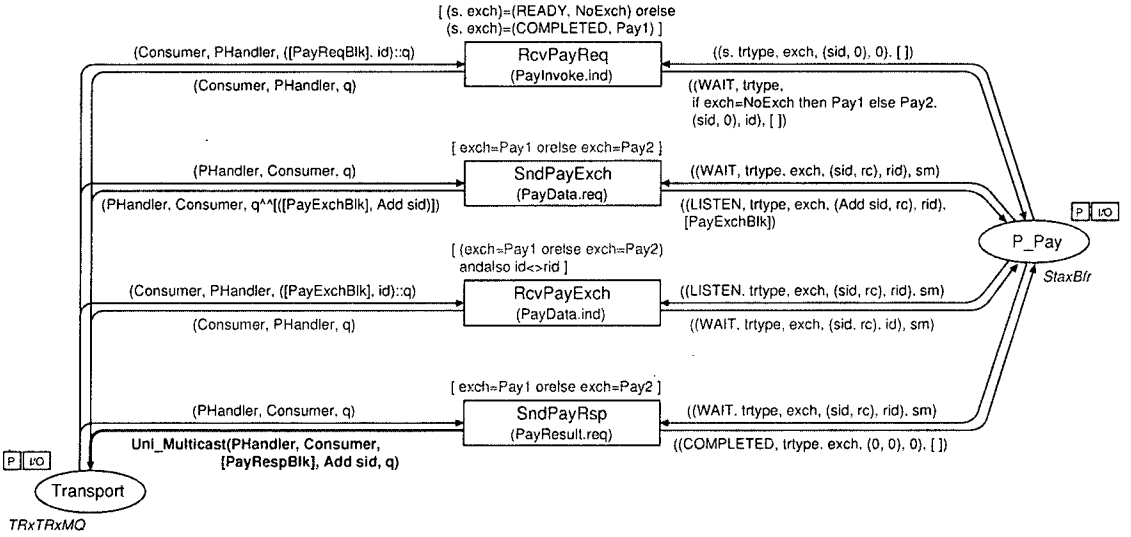


Figure H.6: Payment_P page in the revised IOTP CPN.

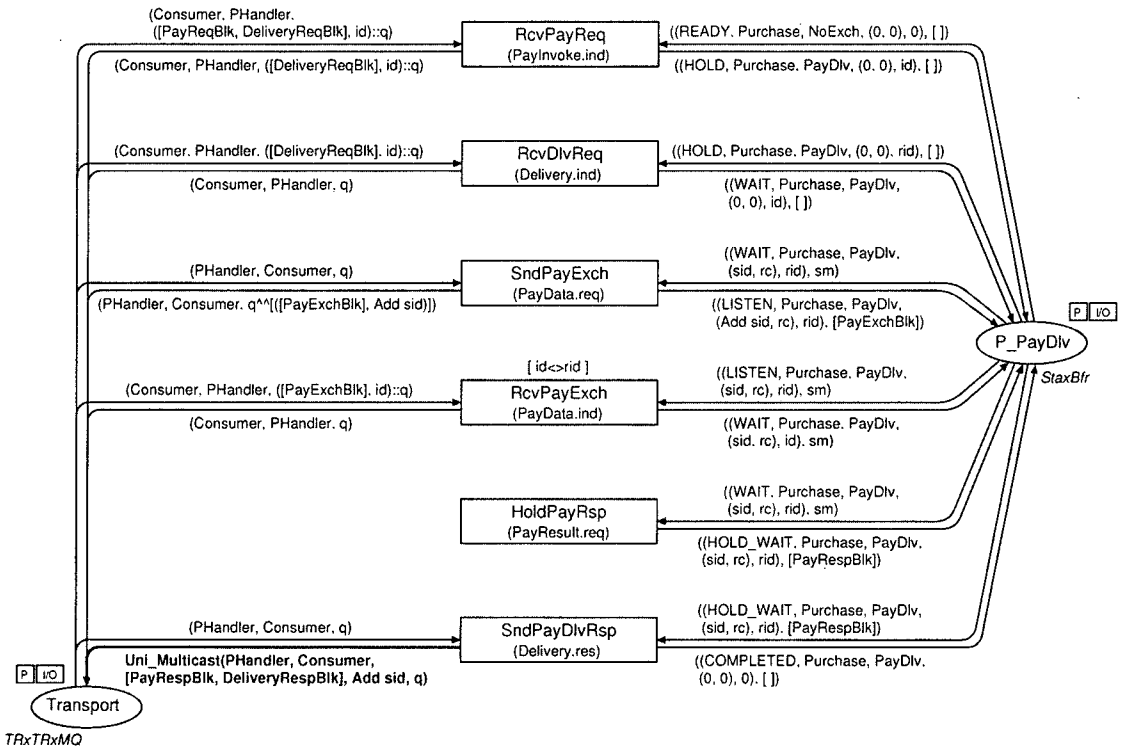


Figure H.7: PayDelivery_P page in the revised IOTP CPN.

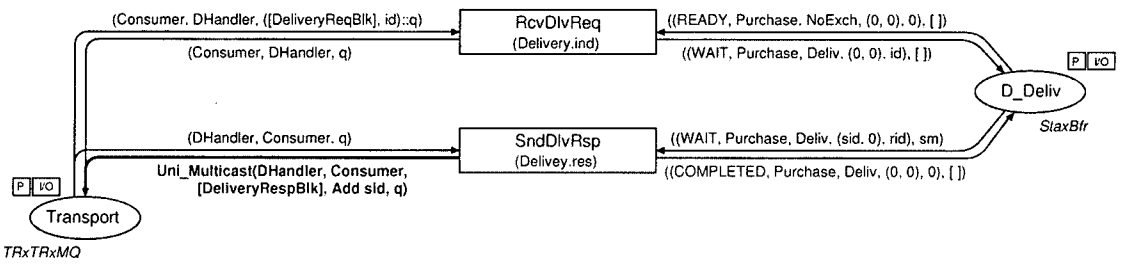


Figure H.8: Delivery_D page in the revised IOTP CPN.

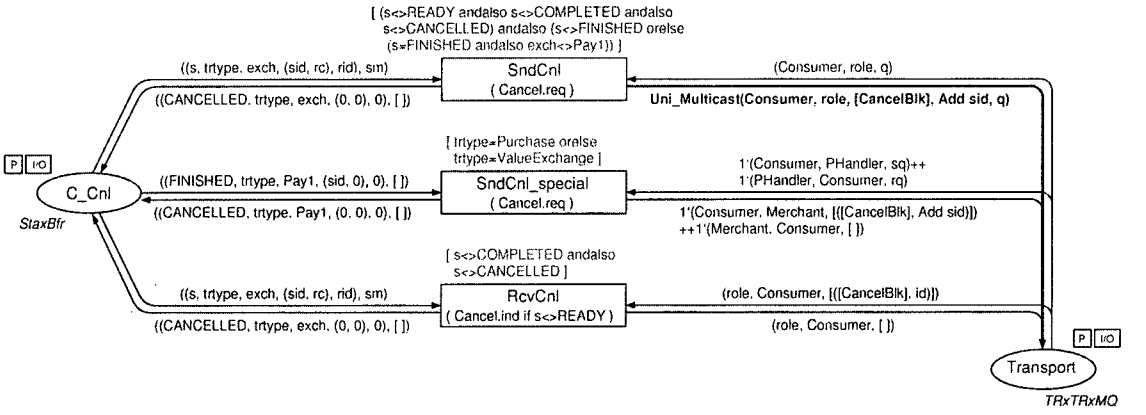


Figure H.9: Cancellation_C page in the revised IOTP CPN.

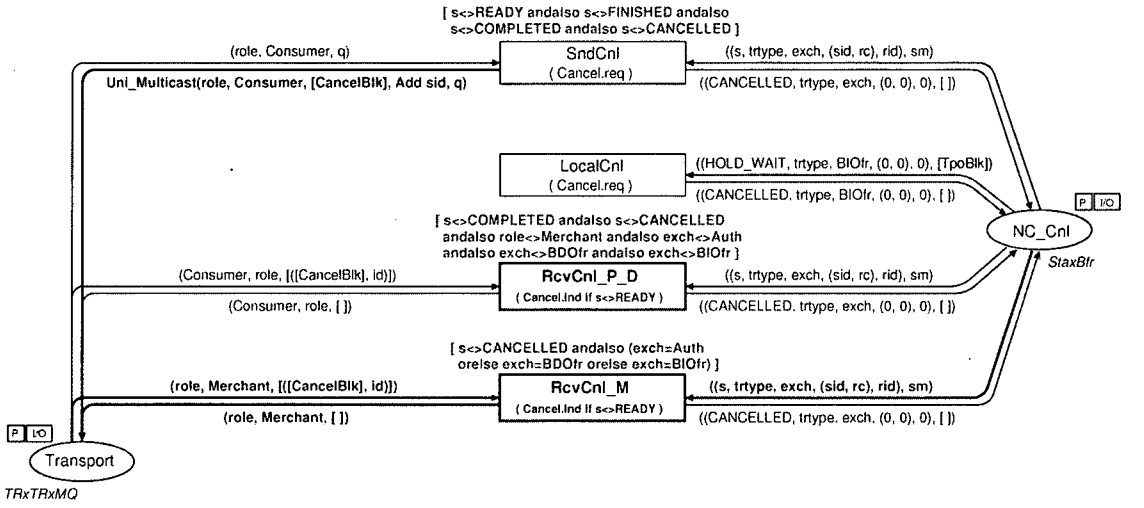


Figure H.10: Cancellation_NC page in the revised IOTP CPN.

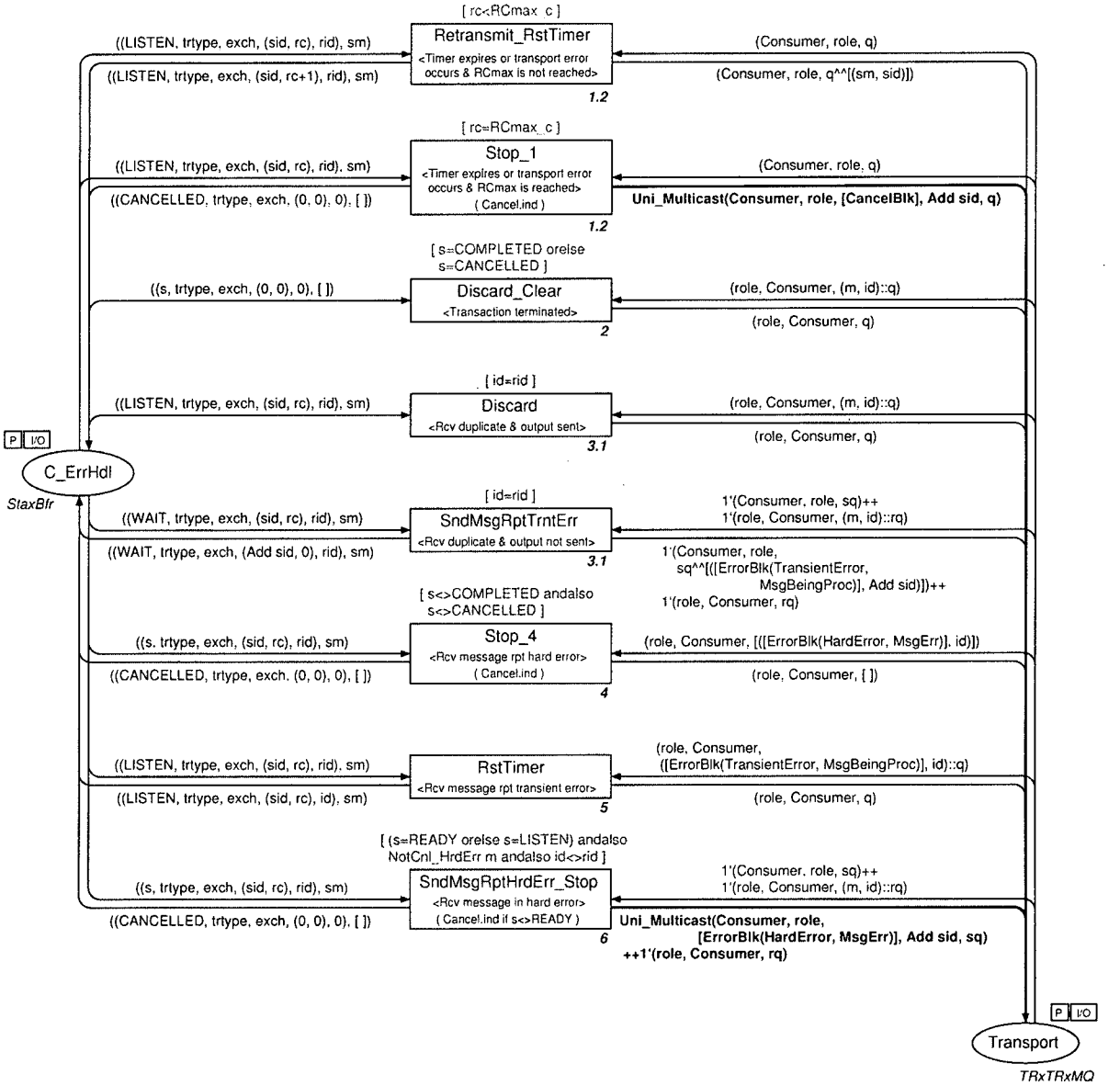


Figure H.11: ErrHandling_C page in the revised IOTP CPN.

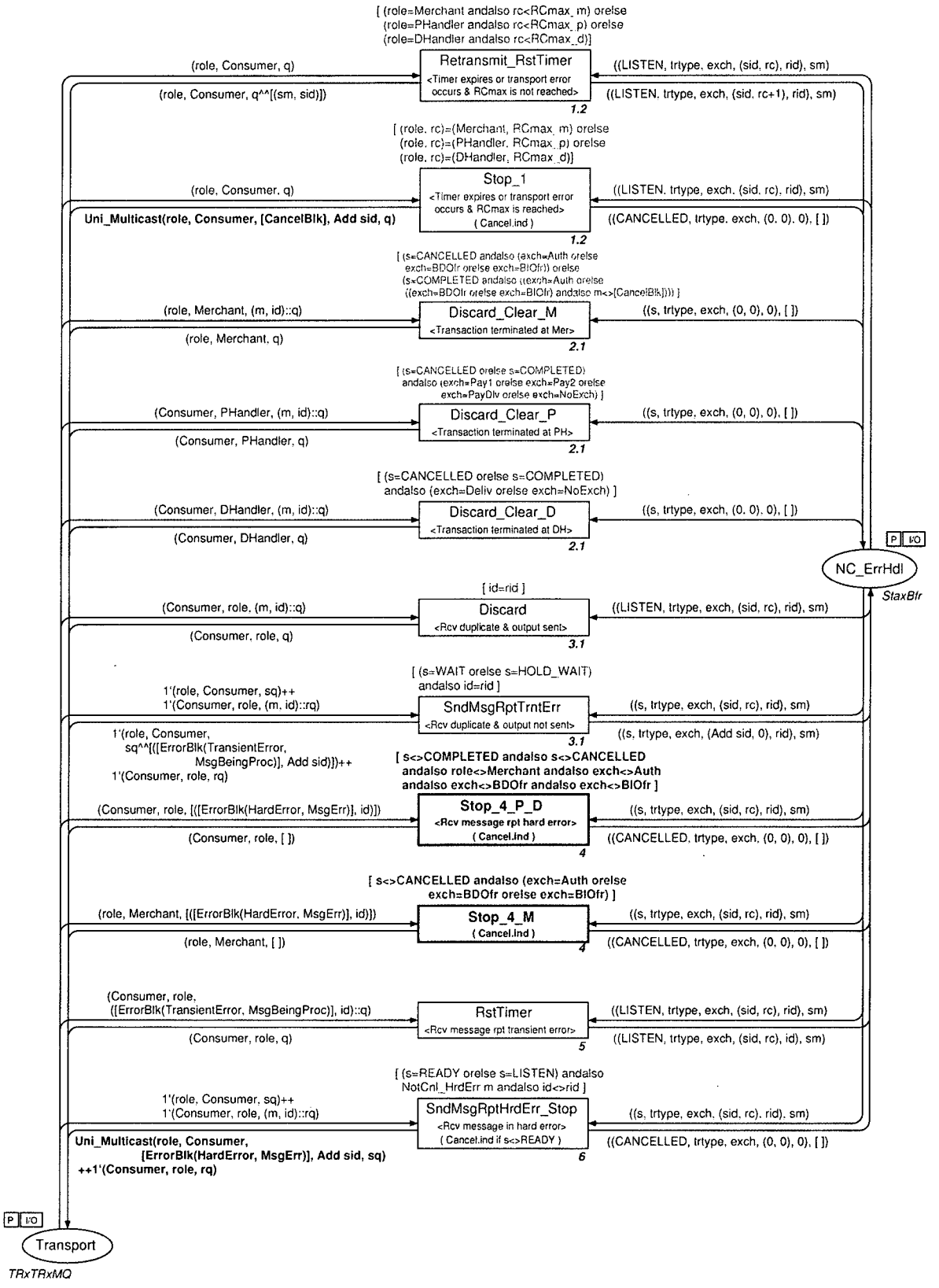


Figure H.12: ErrHandling_NC page in the revised IOTP CPN.

H.2 State Space Analysis Results of the Revised IOTP

The revised IOTP is analysed in a similar way to the analysis of IOTP in Chapter 11. We consider the configurations for all Authentication and Payment-related transactions with all permutations of the message retransmission counter parameters within the range of *0* to *3*. As a representative example, the state space analysis results for the revised IOTP CPN model with configurations in Group 1-1 (see Section 11.3) are given below.

H.2.1 State Space Reports

Authentication Transaction (*A-1-1*)

```
 Statistics
------------------------------------------------------------------------

  Occurrence Graph
    Nodes:  220
    Arcs:   614
    Secs:   1
    Status: Full

  Scc Graph
    Nodes:  220
    Arcs:   614
    Secs:   1

 Home Properties
------------------------------------------------------------------------

  Home Markings:  None

 Liveness Properties
------------------------------------------------------------------------

  Dead Markings:  [18,31,101,142]

  Dead Transitions Instances:
  Authenticator'HoldAuthSts 1
  Cancellation_C'SndCnl_special 1
  Cancellation_NC'LocalCnl 1
  Cancellation_NC'RcvCnl_P_D 1
  ErrHandling_NC'Discard_Clear_D 1
  ErrHandling_NC'Discard_Clear_P 1
  ErrHandling_NC'Stop_4_P_D 1

  Live Transitions Instances: None
```

Purchase Transaction (*P-1-1*)

```
 Statistics
------------------------------------------------------------------------

  Occurrence Graph
    Nodes:  12695
    Arcs:   37947
    Secs:   135
    Status: Full

  Scc Graph
    Nodes:  12695
    Arcs:   37947
    Secs:   7

 Home Properties
------------------------------------------------------------------------

  Home Markings:  None
```

```
Liveness Properties
-------------------------------------------------------------------------
  Dead Markings:  85 [877,873,837,815,811,...]

  Dead Transitions Instances:
  Authenticator'HoldAuthSts 1
  Authenticator'HoldAuthSts 3
  Authenticator'HoldAuthSts 4
  Authenticator'HoldAuthSts 5
  Authenticator'RcvAuthRsp 1
  Authenticator'RcvAuthRsp 3
  Authenticator'RcvAuthRsp 4
  Authenticator'RcvAuthRsp 5
  Authenticator'SndAuthReq 1
  Authenticator'SndAuthReq 3
  Authenticator'SndAuthReq 4
  Authenticator'SndAuthReq 5
  Authenticator'SndAuthSts 1
  Authenticator'SndAuthSts 2
  Authenticator'SndAuthSts 3
  Authenticator'SndAuthSts 4
  Authenticator'SndAuthSts 5
  BrdDepOffer_M'RcvPayDlvRsp 1
  BrdDepOffer_M'RcvPayDlvRsp 3
  BrdDepOffer_M'RcvPayDlvRsp 4
  BrdDepOffer_M'RcvPayDlvRsp 5
  BrdDepOffer_M'RcvTPOSel 1
  BrdDepOffer_M'RcvTPOSel 3
  BrdDepOffer_M'RcvTPOSel 4
  BrdDepOffer_M'RcvTPOSel 5
  BrdDepOffer_M'SndAuthSts_TPO 1
  BrdDepOffer_M'SndAuthSts_TPO 3
  BrdDepOffer_M'SndAuthSts_TPO 4
  BrdDepOffer_M'SndAuthSts_TPO 5
  BrdDepOffer_M'SndOfrRsp 1
  BrdDepOffer_M'SndOfrRsp 3
  BrdDepOffer_M'SndOfrRsp 4
  BrdDepOffer_M'SndOfrRsp 5
  BrdDepOffer_M'SndTPO 1
  BrdDepOffer_M'SndTPO 3
  BrdDepOffer_M'SndTPO 4
  BrdDepOffer_M'SndTPO 5
  BrdIndOffer_M'HoldAuthSts_TPO 1
  BrdIndOffer_M'HoldAuthSts_TPO 3
  BrdIndOffer_M'HoldAuthSts_TPO 4
  BrdIndOffer_M'HoldAuthSts_TPO 5
  BrdIndOffer_M'HoldTPO 1
  BrdIndOffer_M'HoldTPO 3
  BrdIndOffer_M'HoldTPO 4
  BrdIndOffer_M'HoldTPO 5
  BrdIndOffer_M'RcvPayDlvRsp 1
  BrdIndOffer_M'RcvPayDlvRsp 3
  BrdIndOffer_M'RcvPayDlvRsp 4
  BrdIndOffer_M'RcvPayDlvRsp 5
  BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 1
  BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 3
  BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 4
  BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 5
  BrdIndOffer_M'SndTPO_OfrRsp 1
  BrdIndOffer_M'SndTPO_OfrRsp 3
  BrdIndOffer_M'SndTPO_OfrRsp 4
  BrdIndOffer_M'SndTPO_OfrRsp 5
  Cancellation_NC'LocalCnl 1
  Cancellation_NC'LocalCnl 2
  Cancellation_NC'LocalCnl 4
```

```
Cancellation_NC'LocalCnl 5
Cancellation_NC'LocalCnl 6
Cancellation_NC'LocalCnl 7
Cancellation_NC'RcvCnl_M 1
Cancellation_NC'RcvCnl_M 2
Cancellation_NC'RcvCnl_M 4
Cancellation_NC'RcvCnl_M 5
Cancellation_NC'RcvCnl_M 6
Cancellation_NC'RcvCnl_M 7
Cancellation_NC'RcvCnl_P_D 2
Cancellation_NC'RcvCnl_P_D 3
Cancellation_NC'RcvCnl_P_D 4
Cancellation_NC'RcvCnl_P_D 5
Cancellation_NC'RcvCnl_P_D 6
Cancellation_NC'SndCnl 2
Cancellation_NC'SndCnl 4
Cancellation_NC'SndCnl 5
Cancellation_NC'SndCnl 6
ErrHandling_NC'Discard 1
ErrHandling_NC'Discard 2
ErrHandling_NC'Discard 4
ErrHandling_NC'Discard 5
ErrHandling_NC'Discard 6
ErrHandling_NC'Discard_Clear_D 2
ErrHandling_NC'Discard_Clear_D 3
ErrHandling_NC'Discard_Clear_D 4
ErrHandling_NC'Discard_Clear_D 5
ErrHandling_NC'Discard_Clear_D 6
ErrHandling_NC'Discard_Clear_D 7
ErrHandling_NC'Discard_Clear_M 1
ErrHandling_NC'Discard_Clear_M 2
ErrHandling_NC'Discard_Clear_M 4
ErrHandling_NC'Discard_Clear_M 5
ErrHandling_NC'Discard_Clear_M 6
ErrHandling_NC'Discard_Clear_M 7
ErrHandling_NC'Discard_Clear_P 1
ErrHandling_NC'Discard_Clear_P 2
ErrHandling_NC'Discard_Clear_P 3
ErrHandling_NC'Discard_Clear_P 4
ErrHandling_NC'Discard_Clear_P 5
ErrHandling_NC'Discard_Clear_P 6
ErrHandling_NC'Retransmit_RstTimer 1
ErrHandling_NC'Retransmit_RstTimer 2
ErrHandling_NC'Retransmit_RstTimer 4
ErrHandling_NC'Retransmit_RstTimer 5
ErrHandling_NC'Retransmit_RstTimer 6
ErrHandling_NC'RstTimer 1
ErrHandling_NC'RstTimer 2
ErrHandling_NC'RstTimer 4
ErrHandling_NC'RstTimer 5
ErrHandling_NC'RstTimer 6
ErrHandling_NC'SndMsgRptHrdErr_Stop 2
ErrHandling_NC'SndMsgRptHrdErr_Stop 4
ErrHandling_NC'SndMsgRptHrdErr_Stop 5
ErrHandling_NC'SndMsgRptHrdErr_Stop 6
ErrHandling_NC'SndMsgRptTrntErr 2
ErrHandling_NC'SndMsgRptTrntErr 4
ErrHandling_NC'SndMsgRptTrntErr 5
ErrHandling_NC'SndMsgRptTrntErr 6
ErrHandling_NC'Stop_1 1
ErrHandling_NC'Stop_1 2
ErrHandling_NC'Stop_1 4
ErrHandling_NC'Stop_1 5
ErrHandling_NC'Stop_1 6
ErrHandling_NC'Stop_4_M 1
ErrHandling_NC'Stop_4_M 2
```

```
  ErrHandling_NC'Stop_4_M 4
  ErrHandling_NC'Stop_4_M 5
  ErrHandling_NC'Stop_4_M 6
  ErrHandling_NC'Stop_4_M 7
  ErrHandling_NC'Stop_4_P_D 2
  ErrHandling_NC'Stop_4_P_D 3
  ErrHandling_NC'Stop_4_P_D 4
  ErrHandling_NC'Stop_4_P_D 5
  ErrHandling_NC'Stop_4_P_D 6
  Payment_C'SndPayReq_2 1

  Live Transitions Instances: None
```

Deposit, Withdrawal or Refund Transaction (*D/W/R-1-1*)

```
 Statistics
------------------------------------------------------------------------

  Occurrence Graph
    Nodes:  4567
    Arcs:   12886
    Secs:   34
    Status: Full

  Scc Graph
    Nodes:  4567
    Arcs:   12886
    Secs:   2

 Home Properties
------------------------------------------------------------------------

  Home Markings:  None

 Liveness Properties
------------------------------------------------------------------------

  Dead Markings:  39 [79,766,762,728,724,...]

  Dead Transitions Instances:
  Authenticatee'RcvAuthReq 2
  Authenticatee'RcvAuthReq 3
  Authenticatee'RcvAuthSts 2
  Authenticatee'RcvAuthSts 3
  Authenticatee'SndAuthRsp 2
  Authenticatee'SndAuthRsp 3
  Authenticator'HoldAuthSts 2
  Authenticator'HoldAuthSts 3
  Authenticator'HoldAuthSts 4
  Authenticator'HoldAuthSts 5
  Authenticator'RcvAuthRsp 2
  Authenticator'RcvAuthRsp 3
  Authenticator'RcvAuthRsp 4
  Authenticator'RcvAuthRsp 5
  Authenticator'SndAuthReq 2
  Authenticator'SndAuthReq 3
  Authenticator'SndAuthReq 4
  Authenticator'SndAuthReq 5
  Authenticator'SndAuthSts 1
  Authenticator'SndAuthSts 2
  Authenticator'SndAuthSts 3
  Authenticator'SndAuthSts 4
  Authenticator'SndAuthSts 5
  BrdDepOffer_C'RcvOfrRsp 2
  BrdDepOffer_C'RcvOfrRsp 3
  BrdDepOffer_C'RcvTPO 2
  BrdDepOffer_C'RcvTPO 3
  BrdDepOffer_C'SndTPOSel 2
  BrdDepOffer_C'SndTPOSel 3
```

```
BrdDepOffer_M'RcvPayDlvRsp 2
BrdDepOffer_M'RcvPayDlvRsp 3
BrdDepOffer_M'RcvPayDlvRsp 4
BrdDepOffer_M'RcvPayDlvRsp 5
BrdDepOffer_M'RcvTPOSel 2
BrdDepOffer_M'RcvTPOSel 3
BrdDepOffer_M'RcvTPOSel 4
BrdDepOffer_M'RcvTPOSel 5
BrdDepOffer_M'SndAuthSts_TPO 2
BrdDepOffer_M'SndAuthSts_TPO 3
BrdDepOffer_M'SndAuthSts_TPO 4
BrdDepOffer_M'SndAuthSts_TPO 5
BrdDepOffer_M'SndOfrRsp 2
BrdDepOffer_M'SndOfrRsp 3
BrdDepOffer_M'SndOfrRsp 4
BrdDepOffer_M'SndOfrRsp 5
BrdDepOffer_M'SndTPO 2
BrdDepOffer_M'SndTPO 3
BrdDepOffer_M'SndTPO 4
BrdDepOffer_M'SndTPO 5
BrdIndOffer_C'RcvOfrRsp 2
BrdIndOffer_C'RcvOfrRsp 3
BrdIndOffer_C'RcvTPO 2
BrdIndOffer_C'RcvTPO 3
BrdIndOffer_M'HoldAuthSts_TPO 2
BrdIndOffer_M'HoldAuthSts_TPO 3
BrdIndOffer_M'HoldAuthSts_TPO 4
BrdIndOffer_M'HoldAuthSts_TPO 5
BrdIndOffer_M'HoldTPO 2
BrdIndOffer_M'HoldTPO 3
BrdIndOffer_M'HoldTPO 4
BrdIndOffer_M'HoldTPO 5
BrdIndOffer_M'RcvPayDlvRsp 2
BrdIndOffer_M'RcvPayDlvRsp 3
BrdIndOffer_M'RcvPayDlvRsp 4
BrdIndOffer_M'RcvPayDlvRsp 5
BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 2
BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 3
BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 4
BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 5
BrdIndOffer_M'SndTPO_OfrRsp 2
BrdIndOffer_M'SndTPO_OfrRsp 3
BrdIndOffer_M'SndTPO_OfrRsp 4
BrdIndOffer_M'SndTPO_OfrRsp 5
Cancellation_C'RcvCnl 2
Cancellation_C'RcvCnl 3
Cancellation_C'SndCnl 2
Cancellation_C'SndCnl 3
Cancellation_C'SndCnl_special 1
Cancellation_C'SndCnl_special 2
Cancellation_C'SndCnl_special 3
Cancellation_NC'LocalCnl 2
Cancellation_NC'LocalCnl 3
Cancellation_NC'LocalCnl 4
Cancellation_NC'LocalCnl 5
Cancellation_NC'LocalCnl 6
Cancellation_NC'LocalCnl 7
Cancellation_NC'LocalCnl 8
Cancellation_NC'RcvCnl_M 2
Cancellation_NC'RcvCnl_M 3
Cancellation_NC'RcvCnl_M 4
Cancellation_NC'RcvCnl_M 5
Cancellation_NC'RcvCnl_M 6
Cancellation_NC'RcvCnl_M 7
Cancellation_NC'RcvCnl_M 8
Cancellation_NC'RcvCnl_P_D 1
```

```
Cancellation_NC'RcvCnl_P_D 2
Cancellation_NC'RcvCnl_P_D 3
Cancellation_NC'RcvCnl_P_D 4
Cancellation_NC'RcvCnl_P_D 5
Cancellation_NC'RcvCnl_P_D 7
Cancellation_NC'RcvCnl_P_D 8
Cancellation_NC'SndCnl 2
Cancellation_NC'SndCnl 3
Cancellation_NC'SndCnl 4
Cancellation_NC'SndCnl 5
Cancellation_NC'SndCnl 7
Cancellation_NC'SndCnl 8
ErrHandling_C'Discard 2
ErrHandling_C'Discard 3
ErrHandling_C'Discard_Clear 2
ErrHandling_C'Discard_Clear 3
ErrHandling_C'Retransmit_RstTimer 2
ErrHandling_C'Retransmit_RstTimer 3
ErrHandling_C'RstTimer 2
ErrHandling_C'RstTimer 3
ErrHandling_C'SndMsgRptHrdErr_Stop 2
ErrHandling_C'SndMsgRptHrdErr_Stop 3
ErrHandling_C'SndMsgRptTrntErr 2
ErrHandling_C'SndMsgRptTrntErr 3
ErrHandling_C'Stop_1 2
ErrHandling_C'Stop_1 3
ErrHandling_C'Stop_4 2
ErrHandling_C'Stop_4 3
ErrHandling_NC'Discard 2
ErrHandling_NC'Discard 3
ErrHandling_NC'Discard 4
ErrHandling_NC'Discard 5
ErrHandling_NC'Discard 7
ErrHandling_NC'Discard 8
ErrHandling_NC'Discard_Clear_D 1
ErrHandling_NC'Discard_Clear_D 2
ErrHandling_NC'Discard_Clear_D 3
ErrHandling_NC'Discard_Clear_D 4
ErrHandling_NC'Discard_Clear_D 5
ErrHandling_NC'Discard_Clear_D 6
ErrHandling_NC'Discard_Clear_D 7
ErrHandling_NC'Discard_Clear_D 8
ErrHandling_NC'Discard_Clear_M 2
ErrHandling_NC'Discard_Clear_M 3
ErrHandling_NC'Discard_Clear_M 4
ErrHandling_NC'Discard_Clear_M 5
ErrHandling_NC'Discard_Clear_M 6
ErrHandling_NC'Discard_Clear_M 7
ErrHandling_NC'Discard_Clear_M 8
ErrHandling_NC'Discard_Clear_P 1
ErrHandling_NC'Discard_Clear_P 2
ErrHandling_NC'Discard_Clear_P 3
ErrHandling_NC'Discard_Clear_P 4
ErrHandling_NC'Discard_Clear_P 5
ErrHandling_NC'Discard_Clear_P 7
ErrHandling_NC'Discard_Clear_P 8
ErrHandling_NC'Retransmit_RstTimer 2
ErrHandling_NC'Retransmit_RstTimer 3
ErrHandling_NC'Retransmit_RstTimer 4
ErrHandling_NC'Retransmit_RstTimer 5
ErrHandling_NC'Retransmit_RstTimer 7
ErrHandling_NC'Retransmit_RstTimer 8
ErrHandling_NC'RstTimer 2
ErrHandling_NC'RstTimer 3
ErrHandling_NC'RstTimer 4
ErrHandling_NC'RstTimer 5
```

```
ErrHandling_NC'RstTimer 7
ErrHandling_NC'RstTimer 8
ErrHandling_NC'SndMsgRptHrdErr_Stop 2
ErrHandling_NC'SndMsgRptHrdErr_Stop 3
ErrHandling_NC'SndMsgRptHrdErr_Stop 4
ErrHandling_NC'SndMsgRptHrdErr_Stop 5
ErrHandling_NC'SndMsgRptHrdErr_Stop 7
ErrHandling_NC'SndMsgRptHrdErr_Stop 8
ErrHandling_NC'SndMsgRptTrntErr 2
ErrHandling_NC'SndMsgRptTrntErr 3
ErrHandling_NC'SndMsgRptTrntErr 4
ErrHandling_NC'SndMsgRptTrntErr 5
ErrHandling_NC'SndMsgRptTrntErr 7
ErrHandling_NC'SndMsgRptTrntErr 8
ErrHandling_NC'Stop_1 2
ErrHandling_NC'Stop_1 3
ErrHandling_NC'Stop_1 4
ErrHandling_NC'Stop_1 5
ErrHandling_NC'Stop_1 7
ErrHandling_NC'Stop_1 8
ErrHandling_NC'Stop_4_M 2
ErrHandling_NC'Stop_4_M 3
ErrHandling_NC'Stop_4_M 4
ErrHandling_NC'Stop_4_M 5
ErrHandling_NC'Stop_4_M 6
ErrHandling_NC'Stop_4_M 7
ErrHandling_NC'Stop_4_M 8
ErrHandling_NC'Stop_4_P_D 1
ErrHandling_NC'Stop_4_P_D 2
ErrHandling_NC'Stop_4_P_D 3
ErrHandling_NC'Stop_4_P_D 4
ErrHandling_NC'Stop_4_P_D 5
ErrHandling_NC'Stop_4_P_D 7
ErrHandling_NC'Stop_4_P_D 8
Payment_C'RcvPayExch 2
Payment_C'RcvPayExch 3
Payment_C'RcvPayRsp_cont 1
Payment_C'RcvPayRsp_cont 2
Payment_C'RcvPayRsp_cont 3
Payment_C'RcvPayRsp_stop 2
Payment_C'RcvPayRsp_stop 3
Payment_C'SndPayExch 2
Payment_C'SndPayExch 3
Payment_C'SndPayReq_1 2
Payment_C'SndPayReq_1 3
Payment_C'SndPayReq_2 1
Payment_C'SndPayReq_2 2
Payment_C'SndPayReq_2 3
Payment_P'RcvPayExch 2
Payment_P'RcvPayExch 3
Payment_P'RcvPayReq 2
Payment_P'RcvPayReq 3
Payment_P'SndPayExch 2
Payment_P'SndPayExch 3
Payment_P'SndPayRsp 2
Payment_P'SndPayRsp 3

Live Transitions Instances: None
```

Value Exchange Transaction (*V-1-1*)

```
 Statistics
-------------------------------------------------------------------------

  Occurrence Graph
    Nodes:  11488
    Arcs:   36077
```

```
    Secs:   107
    Status: Full

  Scc Graph
    Nodes:  11488
    Arcs:   36077
    Secs:   7

 Home Properties
--------------------------------------------------------------------------
  Home Markings:  None

 Liveness Properties
--------------------------------------------------------------------------
  Dead Markings:  63 [79,771,767,730,726,...]

  Dead Transitions Instances:
  Authenticator'HoldAuthSts 1
  Authenticator'HoldAuthSts 2
  Authenticator'HoldAuthSts 3
  Authenticator'HoldAuthSts 5
  Authenticator'RcvAuthRsp 1
  Authenticator'RcvAuthRsp 2
  Authenticator'RcvAuthRsp 3
  Authenticator'RcvAuthRsp 5
  Authenticator'SndAuthReq 1
  Authenticator'SndAuthReq 2
  Authenticator'SndAuthReq 3
  Authenticator'SndAuthReq 5
  Authenticator'SndAuthSts 1
  Authenticator'SndAuthSts 2
  Authenticator'SndAuthSts 3
  Authenticator'SndAuthSts 4
  Authenticator'SndAuthSts 5
  BrdDepOffer_M'RcvPayDlvRsp 1
  BrdDepOffer_M'RcvPayDlvRsp 2
  BrdDepOffer_M'RcvPayDlvRsp 3
  BrdDepOffer_M'RcvPayDlvRsp 5
  BrdDepOffer_M'RcvTPOSel 1
  BrdDepOffer_M'RcvTPOSel 2
  BrdDepOffer_M'RcvTPOSel 3
  BrdDepOffer_M'RcvTPOSel 5
  BrdDepOffer_M'SndAuthSts_TPO 1
  BrdDepOffer_M'SndAuthSts_TPO 2
  BrdDepOffer_M'SndAuthSts_TPO 3
  BrdDepOffer_M'SndAuthSts_TPO 5
  BrdDepOffer_M'SndOfrRsp 1
  BrdDepOffer_M'SndOfrRsp 2
  BrdDepOffer_M'SndOfrRsp 3
  BrdDepOffer_M'SndOfrRsp 5
  BrdDepOffer_M'SndTPO 1
  BrdDepOffer_M'SndTPO 2
  BrdDepOffer_M'SndTPO 3
  BrdDepOffer_M'SndTPO 5
  BrdIndOffer_M'HoldAuthSts_TPO 1
  BrdIndOffer_M'HoldAuthSts_TPO 2
  BrdIndOffer_M'HoldAuthSts_TPO 3
  BrdIndOffer_M'HoldAuthSts_TPO 5
  BrdIndOffer_M'HoldTPO 1
  BrdIndOffer_M'HoldTPO 2
  BrdIndOffer_M'HoldTPO 3
  BrdIndOffer_M'HoldTPO 5
  BrdIndOffer_M'RcvPayDlvRsp 1
  BrdIndOffer_M'RcvPayDlvRsp 2
  BrdIndOffer_M'RcvPayDlvRsp 3
  BrdIndOffer_M'RcvPayDlvRsp 5
```

```
BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 1
BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 2
BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 3
BrdIndOffer_M'SndAuthSts_TPO_OfrRsp 5
BrdIndOffer_M'SndTPO_OfrRsp 1
BrdIndOffer_M'SndTPO_OfrRsp 2
BrdIndOffer_M'SndTPO_OfrRsp 3
BrdIndOffer_M'SndTPO_OfrRsp 5
Cancellation_NC'LocalCnl 1
Cancellation_NC'LocalCnl 2
Cancellation_NC'LocalCnl 3
Cancellation_NC'LocalCnl 5
Cancellation_NC'LocalCnl 6
Cancellation_NC'RcvCnl_M 1
Cancellation_NC'RcvCnl_M 2
Cancellation_NC'RcvCnl_M 3
Cancellation_NC'RcvCnl_M 5
Cancellation_NC'RcvCnl_M 6
Cancellation_NC'RcvCnl_P_D 1
Cancellation_NC'RcvCnl_P_D 2
Cancellation_NC'RcvCnl_P_D 3
Cancellation_NC'RcvCnl_P_D 4
Cancellation_NC'RcvCnl_P_D 5
Cancellation_NC'SndCnl 1
Cancellation_NC'SndCnl 2
Cancellation_NC'SndCnl 3
Cancellation_NC'SndCnl 5
ErrHandling_NC'Discard 1
ErrHandling_NC'Discard 2
ErrHandling_NC'Discard 3
ErrHandling_NC'Discard 5
ErrHandling_NC'Discard_Clear_D 1
ErrHandling_NC'Discard_Clear_D 2
ErrHandling_NC'Discard_Clear_D 3
ErrHandling_NC'Discard_Clear_D 4
ErrHandling_NC'Discard_Clear_D 5
ErrHandling_NC'Discard_Clear_D 6
ErrHandling_NC'Discard_Clear_M 1
ErrHandling_NC'Discard_Clear_M 2
ErrHandling_NC'Discard_Clear_M 3
ErrHandling_NC'Discard_Clear_M 5
ErrHandling_NC'Discard_Clear_M 6
ErrHandling_NC'Discard_Clear_P 1
ErrHandling_NC'Discard_Clear_P 2
ErrHandling_NC'Discard_Clear_P 3
ErrHandling_NC'Discard_Clear_P 4
ErrHandling_NC'Discard_Clear_P 5
ErrHandling_NC'Retransmit_RstTimer 1
ErrHandling_NC'Retransmit_RstTimer 2
ErrHandling_NC'Retransmit_RstTimer 3
ErrHandling_NC'Retransmit_RstTimer 5
ErrHandling_NC'RstTimer 1
ErrHandling_NC'RstTimer 2
ErrHandling_NC'RstTimer 3
ErrHandling_NC'RstTimer 5
ErrHandling_NC'SndMsgRptHrdErr_Stop 1
ErrHandling_NC'SndMsgRptHrdErr_Stop 2
ErrHandling_NC'SndMsgRptHrdErr_Stop 3
ErrHandling_NC'SndMsgRptHrdErr_Stop 5
ErrHandling_NC'SndMsgRptTrntErr 1
ErrHandling_NC'SndMsgRptTrntErr 2
ErrHandling_NC'SndMsgRptTrntErr 3
ErrHandling_NC'SndMsgRptTrntErr 5
ErrHandling_NC'Stop_1 1
ErrHandling_NC'Stop_1 2
ErrHandling_NC'Stop_1 3
```

```
ErrHandling_NC'Stop_1 5
ErrHandling_NC'Stop_4_M 1
ErrHandling_NC'Stop_4_M 2
ErrHandling_NC'Stop_4_M 3
ErrHandling_NC'Stop_4_M 5
ErrHandling_NC'Stop_4_M 6
ErrHandling_NC'Stop_4_P_D 1
ErrHandling_NC'Stop_4_P_D 2
ErrHandling_NC'Stop_4_P_D 3
ErrHandling_NC'Stop_4_P_D 4
ErrHandling_NC'Stop_4_P_D 5

Live Transitions Instances: None
```

H.2.2 Dead Markings

For many configurations of the revised IOTP CPN examined, we define functions for validating the dead markings in each transaction state space. In the revised IOTP CPN model, a new place Delivery (typed by colour set DelivExch) is added on the BrdDepOffer_C, BrdIndOffer_C and Payment_C pages (see Figures H.1, H.3 and H.5). Upon transaction termination the place Delivery may or may not be empty, which therefore expands the number of the dead markings for the Payment-related transactions in the revised IOTP CPN. However, the marking of the Delivery place does not have any impact on transaction termination. Whether a dead marking represents a desired terminal state of the transaction, is still determined by the terminal states of each of the trading role entities involved. Figure 11.1 in Section 11.3.1 shows the four valid terminal states of the Consumer and Merchant entities upon termination of the Authentication transaction. Table 11.3 in Section 11.3.1, together with Tables 13.4 to 13.6 in Section 13.6 (which update part of Table 11.3), defines all valid terminal states of trading role entities upon termination of each Payment-related transaction.

Authentication Transaction

We define a function InvalidTM_AuthTr to check for any dead markings that represent undesired terminal states of an Authentication transaction. The function InvalidTM_AuthTr uses a function ValidTM_AuthTr which specifies the valid terminal states of the Consumer and Merchant entities upon termination of the Authentication transaction. The ML code for the definitions of these two functions are given below. Figure H.13 shows the result of checking for undesired terminal states in the state space of the Authentication transaction with *Config A-1-1*.

```
fun ValidTM_AuthTr n =

  ((Mark.Consumer'C_AuthTr 1 n)==1`((CANCELLED,Authentication,NoExch,(0,0),0),[]) andalso
   (Mark.Merchant'M_AuthTr 1 n)==1`((CANCELLED,Authentication,Auth,(0,0),0),[])) orelse

  ((Mark.Consumer'C_AuthTr 1 n)==1`((CANCELLED,Authentication,Auth,(0,0),0),[]) andalso
   (Mark.Merchant'M_AuthTr 1 n)==1`((CANCELLED,Authentication,Auth,(0,0),0),[])) orelse

  ((Mark.Consumer'C_AuthTr 1 n)==1`((COMPLETED,Authentication,Auth,(0,0),0),[]) andalso
   (Mark.Merchant'M_AuthTr 1 n)==1`((COMPLETED,Authentication,Auth,(0,0),0),[])) orelse

  ((Mark.Consumer'C_AuthTr 1 n)==1`((CANCELLED,Authentication,Auth,(0,0),0),[]) andalso
   (Mark.Merchant'M_AuthTr 1 n)==1`((COMPLETED,Authentication,Auth,(0,0),0),[]));

fun InvalidTM_AuthTr n = PredNodes ( ListDeadMarkings (),
                                     fn n => not (ValidTM_AuthTr n),
                                     NoLimit );
```

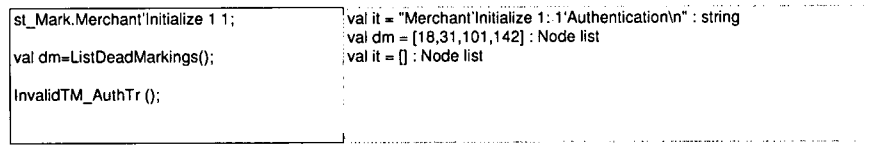


Figure H.13: Result of checking for undesired terminal states in the Authentication transaction state space with *Config A-1-1.*

Purchase Transaction

Validation of the dead markings in the state space of a Purchase transaction is performed in a similar way as what is done for the Authentication transaction. We use two functions ValidTM_Purchase and InvalidTM_Purchase, and their definitions are listed below. Figure H.14 shows the validation result for the Purchase transaction with *Config P-1-1.*

```
fun ValidTM_Purchase n =

  ((Mark.Consumer'C_Purchase 1 n)==1`((COMPLETED,Purchase,Pay1,(0,0),0),[]) andalso
   ((Mark.Merchant'M_Purchase 1 n)==1`((COMPLETED,Purchase,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_Purchase 1 n)==1`((COMPLETED,Purchase,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_Purchase 1 n)==1`((COMPLETED,Purchase,Pay1,(0,0),0),[]) andalso
   (Mark.DHandler'D_Purchase 1 n)==empty) orelse

  ((Mark.Consumer'C_Purchase 1 n)==1`((COMPLETED,Purchase,Pay2,(0,0),0),[]) andalso
   ((Mark.Merchant'M_Purchase 1 n)==1`((COMPLETED,Purchase,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_Purchase 1 n)==1`((COMPLETED,Purchase,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_Purchase 1 n)==1`((COMPLETED,Purchase,Pay2,(0,0),0),[]) andalso
   (Mark.DHandler'D_Purchase 1 n)==empty) orelse

  ((Mark.Consumer'C_Purchase 1 n)==1`((COMPLETED,Purchase,PayDlv,(0,0),0),[]) andalso
   ((Mark.Merchant'M_Purchase 1 n)==1`((COMPLETED,Purchase,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_Purchase 1 n)==1`((COMPLETED,Purchase,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_Purchase 1 n)==1`((COMPLETED,Purchase,PayDlv,(0,0),0),[]) andalso
   (Mark.DHandler'D_Purchase 1 n)==empty) orelse

  ((Mark.Consumer'C_Purchase 1 n)==1`((COMPLETED,Purchase,Deliv,(0,0),0),[]) andalso
   ((Mark.Merchant'M_Purchase 1 n)==1`((COMPLETED,Purchase,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_Purchase 1 n)==1`((COMPLETED,Purchase,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_Purchase 1 n)==1`((COMPLETED,Purchase,Pay1,(0,0),0),[]) andalso
   (Mark.DHandler'D_Purchase 1 n)==1`((COMPLETED,Purchase,Deliv,(0,0),0),[])) orelse

  ((Mark.Consumer'C_Purchase 1 n)==empty andalso
   (Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BIOfr,(0,0),0),[]) andalso
   (Mark.PHandler'P_Purchase 1 n)==empty andalso
   (Mark.DHandler'D_Purchase 1 n)==empty) orelse

  ((Mark.Consumer'C_Purchase 1 n)==1`((CANCELLED,Purchase,NoExch,(0,0),0),[]) andalso
   ((Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,Auth,(0,0),0),[]) orelse
    (Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_Purchase 1 n)==empty andalso
   (Mark.DHandler'D_Purchase 1 n)==empty) orelse

  ((Mark.Consumer'C_Purchase 1 n)==1`((CANCELLED,Purchase,Auth,(0,0),0),[]) andalso
   ((Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,Auth,(0,0),0),[]) orelse
    (Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_Purchase 1 n)==empty andalso
   (Mark.DHandler'D_Purchase 1 n)==empty) orelse

  ((Mark.Consumer'C_Purchase 1 n)==1`((CANCELLED,Purchase,BDOfr,(0,0),0),[]) andalso
   (Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BDOfr,(0,0),0),[]) andalso
   (Mark.PHandler'P_Purchase 1 n)==empty andalso
   (Mark.DHandler'D_Purchase 1 n)==empty) orelse

  ((Mark.Consumer'C_Purchase 1 n)==1`((CANCELLED,Purchase,BIOfr,(0,0),0),[]) andalso
   (Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BIOfr,(0,0),0),[]) andalso
```

```
   (Mark.PHandler'P_Purchase 1 n)==empty andalso
   (Mark.DHandler'D_Purchase 1 n)==empty) orelse

  ((Mark.Consumer'C_Purchase 1 n)==1`((CANCELLED,Purchase,Pay1,(0,0),0),[]) andalso
   ((Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_Purchase 1 n)==1`((COMPLETED,Purchase,Pay1,(0,0),0),[]) andalso
   (Mark.DHandler'D_Purchase 1 n)==empty) orelse

  ((Mark.Consumer'C_Purchase 1 n)==1`((CANCELLED,Purchase,Pay2,(0,0),0),[]) andalso
   ((Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_Purchase 1 n)==1`((COMPLETED,Purchase,Pay2,(0,0),0),[]) andalso
   (Mark.DHandler'D_Purchase 1 n)==empty) orelse

  ((Mark.Consumer'C_Purchase 1 n)==1`((CANCELLED,Purchase,PayDlv,(0,0),0),[]) andalso
   ((Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_Purchase 1 n)==1`((COMPLETED,Purchase,PayDlv,(0,0),0),[]) andalso
   (Mark.DHandler'D_Purchase 1 n)==empty) orelse

  ((Mark.Consumer'C_Purchase 1 n)==1`((CANCELLED,Purchase,Deliv,(0,0),0),[]) andalso
   ((Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_Purchase 1 n)==1`((COMPLETED,Purchase,Pay1,(0,0),0),[]) andalso
   (Mark.DHandler'D_Purchase 1 n)==1`((COMPLETED,Purchase,Deliv,(0,0),0),[])) orelse

  (((Mark.Consumer'C_Purchase 1 n)==1`((CANCELLED,Purchase,Pay1,(0,0),0),[]) orelse
    (Mark.Consumer'C_Purchase 1 n)==1`((CANCELLED,Purchase,PayDlv,(0,0),0),[])) andalso
   ((Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_Purchase 1 n)==1`((CANCELLED,Purchase,NoExch,(0,0),0),[]) andalso
   (Mark.DHandler'D_Purchase 1 n)==empty) orelse

  ((Mark.Consumer'C_Purchase 1 n)==1`((CANCELLED,Purchase,Pay1,(0,0),0),[]) andalso
   ((Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_Purchase 1 n)==1`((CANCELLED,Purchase,Pay1,(0,0),0),[]) andalso
   (Mark.DHandler'D_Purchase 1 n)==empty) orelse

  ((Mark.Consumer'C_Purchase 1 n)==1`((CANCELLED,Purchase,Pay2,(0,0),0),[]) andalso
   ((Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_Purchase 1 n)==1`((CANCELLED,Purchase,Pay2,(0,0),0),[]) andalso
   (Mark.DHandler'D_Purchase 1 n)==empty) orelse

  ((Mark.Consumer'C_Purchase 1 n)==1`((CANCELLED,Purchase,PayDlv,(0,0),0),[]) andalso
   ((Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_Purchase 1 n)==1`((CANCELLED,Purchase,PayDlv,(0,0),0),[]) andalso
   (Mark.DHandler'D_Purchase 1 n)==empty) orelse

  ((Mark.Consumer'C_Purchase 1 n)==1`((CANCELLED,Purchase,Deliv,(0,0),0),[]) andalso
   ((Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_Purchase 1 n)==1`((CANCELLED,Purchase,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_Purchase 1 n)==1`((COMPLETED,Purchase,Pay1,(0,0),0),[]) andalso
   ((Mark.DHandler'D_Purchase 1 n)==1`((CANCELLED,Purchase,NoExch,(0,0),0),[]) orelse
    (Mark.DHandler'D_Purchase 1 n)==1`((CANCELLED,Purchase,Deliv,(0,0),0),[])));

fun InvalidTM_Purchase n = PredNodes ( ListDeadMarkings (),
                                       fn n => not (ValidTM_Purchase n),
                                       NoLimit );
```

```
st_Mark.Merchant'Initialize 1 1;

val dm=ListDeadMarkings();
length dm;

InvalidTM_Purchase ();
```

```
val it = "Merchant'Initialize 1: 1'Purchase\n" : string
val dm = [877,873,837,815,811,803,551,7,663,65,47,1021,...] : Node list
val it = 85 : int
val it = [] : Node list
```

Figure H.14: Result of checking for undesired terminal states in the Purchase transaction state space with *Config P-1-1*.

Deposit Transaction

Similarly, we use two functions ValidTM_Deposit and InvalidTM_Deposit for validation of the dead markings in the state space of a Deposit transaction. The definitions of these two functions are given below. Figure H.15 shows the validation result for the Deposit transaction with *Config D-1-1*.

```
fun ValidTM_Deposit n =

  ((Mark.Consumer'C_Deposit 1 n)==1`((COMPLETED,Deposit,Pay1,(0,0),0),[]) andalso
   ((Mark.Merchant'M_Deposit 1 n)==1`((COMPLETED,Deposit,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_Deposit 1 n)==1`((COMPLETED,Deposit,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_Deposit 1 n)==1`((COMPLETED,Deposit,Pay1,(0,0),0),[])) orelse

  ((Mark.Consumer'C_Deposit 1 n)==empty andalso
   (Mark.Merchant'M_Deposit 1 n)==1`((CANCELLED,Deposit,BIOfr,(0,0),0),[]) andalso
   (Mark.PHandler'P_Deposit 1 n)==empty) orelse

  ((Mark.Consumer'C_Deposit 1 n)==1`((CANCELLED,Deposit,NoExch,(0,0),0),[]) andalso
   ((Mark.Merchant'M_Deposit 1 n)==1`((CANCELLED,Deposit,Auth,(0,0),0),[]) orelse
    (Mark.Merchant'M_Deposit 1 n)==1`((CANCELLED,Deposit,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_Deposit 1 n)==1`((CANCELLED,Deposit,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_Deposit 1 n)==empty) orelse

  ((Mark.Consumer'C_Deposit 1 n)==1`((CANCELLED,Deposit,Auth,(0,0),0),[]) andalso
   ((Mark.Merchant'M_Deposit 1 n)==1`((CANCELLED,Deposit,Auth,(0,0),0),[]) orelse
    (Mark.Merchant'M_Deposit 1 n)==1`((CANCELLED,Deposit,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_Deposit 1 n)==1`((CANCELLED,Deposit,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_Deposit 1 n)==empty) orelse

  ((Mark.Consumer'C_Deposit 1 n)==1`((CANCELLED,Deposit,BDOfr,(0,0),0),[]) andalso
   (Mark.Merchant'M_Deposit 1 n)==1`((CANCELLED,Deposit,BDOfr,(0,0),0),[]) andalso
   (Mark.PHandler'P_Deposit 1 n)==empty) orelse

  ((Mark.Consumer'C_Deposit 1 n)==1`((CANCELLED,Deposit,BIOfr,(0,0),0),[]) andalso
   (Mark.Merchant'M_Deposit 1 n)==1`((CANCELLED,Deposit,BIOfr,(0,0),0),[]) andalso
   (Mark.PHandler'P_Deposit 1 n)==empty) orelse

  ((Mark.Consumer'C_Deposit 1 n)==1`((CANCELLED,Deposit,Pay1,(0,0),0),[]) andalso
   ((Mark.Merchant'M_Deposit 1 n)==1`((CANCELLED,Deposit,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_Deposit 1 n)==1`((CANCELLED,Deposit,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_Deposit 1 n)==1`((COMPLETED,Deposit,Pay1,(0,0),0),[])) orelse

  ((Mark.Consumer'C_Deposit 1 n)==1`((CANCELLED,Deposit,Pay1,(0,0),0),[]) andalso
   ((Mark.Merchant'M_Deposit 1 n)==1`((CANCELLED,Deposit,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_Deposit 1 n)==1`((CANCELLED,Deposit,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_Deposit 1 n)==1`((CANCELLED,Deposit,NoExch,(0,0),0),[])) orelse

  ((Mark.Consumer'C_Deposit 1 n)==1`((CANCELLED,Deposit,Pay1,(0,0),0),[]) andalso
   ((Mark.Merchant'M_Deposit 1 n)==1`((CANCELLED,Deposit,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_Deposit 1 n)==1`((CANCELLED,Deposit,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_Deposit 1 n)==1`((CANCELLED,Deposit,Pay1,(0,0),0),[]));

fun InvalidTM_Deposit n = PredNodes ( ListDeadMarkings (),
                                      fn n => not (ValidTM_Deposit n),
                                      NoLimit );
```

```
st_Mark.Merchant'Initialize 1 1;

val dm=ListDeadMarkings();
length dm;

InvalidTM_Deposit ();
```

```
val it = "Merchant'Initialize 1: 1`Deposit\n" : string
val dm = [79,766,762,728,724,7,55,531,506,42,399,301,...] : Node list
val it = 39 : int
val it = [] : Node list
```

Figure H.15: Result of checking for undesired terminal states in the Deposit transaction state space with *Config D-1-1*.

Value Exchange Transaction

Also, we use two functions ValidTM_ValExTr and InvalidTM_ValExTr to validate the dead markings in the state space of a Value Exchange transaction. These two functions are defined below. Figure H.16 shows the validation result for the Value Exchange transaction with *Config V-1-1*.

```
fun ValidTM_ValExTr n =

  ((Mark.Consumer'C_ValExTr 1 n)==1`((COMPLETED,ValueExchange,Pay1,(0,0),0),[]) andalso
   ((Mark.Merchant'M_ValExTr 1 n)==1`((COMPLETED,ValueExchange,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_ValExTr 1 n)==1`((COMPLETED,ValueExchange,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_ValExTr 1 n)==1`((COMPLETED,ValueExchange,Pay1,(0,0),0),[])) orelse

  ((Mark.Consumer'C_ValExTr 1 n)==1`((COMPLETED,ValueExchange,Pay2,(0,0),0),[]) andalso
   ((Mark.Merchant'M_ValExTr 1 n)==1`((COMPLETED,ValueExchange,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_ValExTr 1 n)==1`((COMPLETED,ValueExchange,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_ValExTr 1 n)==1`((COMPLETED,ValueExchange,Pay2,(0,0),0),[])) orelse

  ((Mark.Consumer'C_ValExTr 1 n)==empty andalso
   (Mark.Merchant'M_ValExTr 1 n)==1`((CANCELLED,ValueExchange,BIOfr,(0,0),0),[]) andalso
   (Mark.PHandler'P_ValExTr 1 n)==empty) orelse

  ((Mark.Consumer'C_ValExTr 1 n)==1`((CANCELLED,ValueExchange,NoExch,(0,0),0),[]) andalso
   ((Mark.Merchant'M_ValExTr 1 n)==1`((CANCELLED,ValueExchange,Auth,(0,0),0),[]) orelse
    (Mark.Merchant'M_ValExTr 1 n)==1`((CANCELLED,ValueExchange,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_ValExTr 1 n)==1`((CANCELLED,ValueExchange,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_ValExTr 1 n)==empty) orelse

  ((Mark.Consumer'C_ValExTr 1 n)==1`((CANCELLED,ValueExchange,Auth,(0,0),0),[]) andalso
   ((Mark.Merchant'M_ValExTr 1 n)==1`((CANCELLED,ValueExchange,Auth,(0,0),0),[]) orelse
    (Mark.Merchant'M_ValExTr 1 n)==1`((CANCELLED,ValueExchange,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_ValExTr 1 n)==1`((CANCELLED,ValueExchange,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_ValExTr 1 n)==empty) orelse

  ((Mark.Consumer'C_ValExTr 1 n)==1`((CANCELLED,ValueExchange,BDOfr,(0,0),0),[]) andalso
   (Mark.Merchant'M_ValExTr 1 n)==1`((CANCELLED,ValueExchange,BDOfr,(0,0),0),[]) andalso
   (Mark.PHandler'P_ValExTr 1 n)==empty) orelse

  ((Mark.Consumer'C_ValExTr 1 n)==1`((CANCELLED,ValueExchange,BIOfr,(0,0),0),[]) andalso
   (Mark.Merchant'M_ValExTr 1 n)==1`((CANCELLED,ValueExchange,BIOfr,(0,0),0),[]) andalso
   (Mark.PHandler'P_ValExTr 1 n)==empty) orelse

  ((Mark.Consumer'C_ValExTr 1 n)==1`((CANCELLED,ValueExchange,Pay1,(0,0),0),[]) andalso
   ((Mark.Merchant'M_ValExTr 1 n)==1`((CANCELLED,ValueExchange,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_ValExTr 1 n)==1`((CANCELLED,ValueExchange,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_ValExTr 1 n)==1`((COMPLETED,ValueExchange,Pay1,(0,0),0),[])) orelse

  ((Mark.Consumer'C_ValExTr 1 n)==1`((CANCELLED,ValueExchange,Pay2,(0,0),0),[]) andalso
   ((Mark.Merchant'M_ValExTr 1 n)==1`((CANCELLED,ValueExchange,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_ValExTr 1 n)==1`((CANCELLED,ValueExchange,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_ValExTr 1 n)==1`((COMPLETED,ValueExchange,Pay2,(0,0),0),[])) orelse

  ((Mark.Consumer'C_ValExTr 1 n)==1`((CANCELLED,ValueExchange,Pay1,(0,0),0),[]) andalso
   ((Mark.Merchant'M_ValExTr 1 n)==1`((CANCELLED,ValueExchange,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_ValExTr 1 n)==1`((CANCELLED,ValueExchange,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_ValExTr 1 n)==1`((CANCELLED,ValueExchange,NoExch,(0,0),0),[])) orelse

  ((Mark.Consumer'C_ValExTr 1 n)==1`((CANCELLED,ValueExchange,Pay1,(0,0),0),[]) andalso
   ((Mark.Merchant'M_ValExTr 1 n)==1`((CANCELLED,ValueExchange,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_ValExTr 1 n)==1`((CANCELLED,ValueExchange,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_ValExTr 1 n)==1`((CANCELLED,ValueExchange,Pay1,(0,0),0),[])) orelse

  ((Mark.Consumer'C_ValExTr 1 n)==1`((CANCELLED,ValueExchange,Pay2,(0,0),0),[]) andalso
   ((Mark.Merchant'M_ValExTr 1 n)==1`((CANCELLED,ValueExchange,BDOfr,(0,0),0),[]) orelse
    (Mark.Merchant'M_ValExTr 1 n)==1`((CANCELLED,ValueExchange,BIOfr,(0,0),0),[])) andalso
   (Mark.PHandler'P_ValExTr 1 n)==1`((CANCELLED,ValueExchange,Pay2,(0,0),0),[]));

fun InvalidTM_ValExTr n = PredNodes ( ListDeadMarkings (),
                                      fn n => not (ValidTM_ValExTr n),
                                      NoLimit );
```

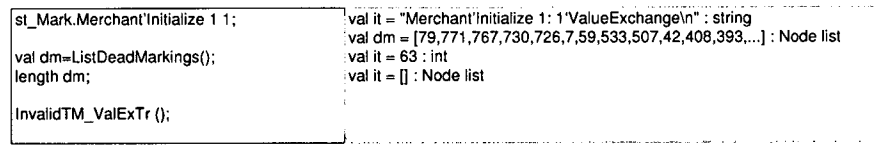


Figure H.16: Result of checking for undesired terminal states in the Value Exchange transaction state space with *Config V-1-1*.

H.2.3 Dead Transitions

Table H.1 lists all dead transitions in the activated CPN page instances for each transaction in the revised IOTP CPN with Configuration Group-1-1. According to Table 13.9, the dead transitions shown in Table H.1 are expected. The other dead transitions that are not shown in Table 13.9 are also expected, since they are the transitions on those CPN page instances which are never executed in a certain transaction.

Configuration	*Dead Transitions (in the activated CPN page instances)*
A-1-1	Authenticator'HoldAuthSts 1 Cancellation_C'SndCnl_special 1 Cancellation_NC'LocalCnl 1 Cancellation_NC'RcvCnl_P_D 1 ErrHandling_NC'Discard_Clear_D 1 ErrHandling_NC'Discard_Clear_P 1 ErrHandling_NC'Stop_4_P_D 1
P-1-1	Authenticator'SndAuthSts 2 Payment_C'SndPayReq_2 1
D-1-1	Authenticator'SndAuthSts 1 Cancellation_C'SndCnl_special 1 ErrHandling_NC'Discard_Clear_D 1 Payment_C'RcvPayRsp_cont 1 Payment_C'SndPayReq_2 1
V-1-1	Authenticator'SndAuthSts 4 ErrHandling_NC'Discard_Clear_D 4

Table H.1: Dead transitions in the activated CPN page instances in the revised IOTP CPN model with configurations in Group 1-1.

H.3 Verification Results of the Revised IOTP

H.3.1 Revised IOTP CPN Binding Element Map Specification

```
(* 07-Jun-03: RvsIOTPcnlTR.sml for obtaining protocol language of *)
(*            transactions with cancellations in the revised IOTP *)

fun be2int (Bind.Consumer'InitiateTr_C (1,_)) = "0"
  | be2int (Bind.Merchant'InitiateTr_M (1,_)) = "0"
  | be2int (Bind.PHandler'InitiateTr_P (1,_)) = "0"
  | be2int (Bind.DHandler'InitiateTr_D (1,_)) = "0"

  | be2int (Bind.Authenticator'SndAuthReq (_,_)) = "31"
  | be2int (Bind.Authenticator'RcvAuthRsp (_,_)) = "34"
  | be2int (Bind.Authenticator'SndAuthSts (_,_)) = "35"
  | be2int (Bind.Authenticator'HoldAuthSts (_,_)) = "35"
```

```
| be2int (Bind.Authenticatee'RcvAuthReq (_,_)) = "24"
| be2int (Bind.Authenticatee'SndAuthRsp (_,_)) = "25"
| be2int (Bind.Authenticatee'RcvAuthSts (_,_)) = "28"

| be2int (Bind.BrdDepOffer_M'SndTPO (_,_)) = "1"
| be2int (Bind.BrdDepOffer_M'SndAuthSts_TPO (_,_)) = "1"
| be2int (Bind.BrdDepOffer_M'RcvTPOSel (_,_)) = "4"
| be2int (Bind.BrdDepOffer_M'SndOfrRsp (_,_)) = "5"
| be2int (Bind.BrdDepOffer_M'RcvPayDlvRsp (_,_)) = "0"  (* Newly Added *)

| be2int (Bind.BrdDepOffer_C'RcvTPO (_,_)) = "2"
| be2int (Bind.BrdDepOffer_C'SndTPOSel (_,_)) = "3"
| be2int (Bind.BrdDepOffer_C'RcvOfrRsp (_,_)) = "6"

| be2int (Bind.BrdIndOffer_M'HoldTPO (_,_)) = "1"
| be2int (Bind.BrdIndOffer_M'HoldAuthSts_TPO (_,_)) = "1"
| be2int (Bind.BrdIndOffer_M'SndTPO_OfrRsp (_,_)) = "5"
| be2int (Bind.BrdIndOffer_M'SndAuthSts_TPO_OfrRsp (_,_)) = "5"
| be2int (Bind.BrdIndOffer_M'RcvPayDlvRsp (_,_)) = "0"  (* Newly Added *)

| be2int (Bind.BrdIndOffer_C'RcvTPO (_,_)) = "2"
| be2int (Bind.BrdIndOffer_C'RcvOfrRsp (_,_)) = "6"

| be2int (Bind.Payment_C'SndPayReq_1 (_,_)) = "7"
| be2int (Bind.Payment_C'SndPayReq_2 (_,_)) = "7"
| be2int (Bind.Payment_C'RcvPayExch (_,_)) = "10"
| be2int (Bind.Payment_C'SndPayExch (_,_)) = "11"
| be2int (Bind.Payment_C'RcvPayRsp_cont (_,_)) = "14"
| be2int (Bind.Payment_C'RcvPayRsp_stop (_,_)) = "14"

| be2int (Bind.Payment_P'RcvPayReq (_,_)) = "8"
| be2int (Bind.Payment_P'SndPayExch (_,_)) = "9"
| be2int (Bind.Payment_P'RcvPayExch (_,_)) = "12"
| be2int (Bind.Payment_P'SndPayRsp (_,_)) = "13"

| be2int (Bind.PayDelivery_C'HoldPayReq (_,_)) = "7"
| be2int (Bind.PayDelivery_C'SndPayDlvReq (_,_)) = "17"
| be2int (Bind.PayDelivery_C'RcvPayExch (_,_)) = "10"
| be2int (Bind.PayDelivery_C'SndPayExch (_,_)) = "11"
| be2int (Bind.PayDelivery_C'RcvPayRsp (_,_)) = "14"
| be2int (Bind.PayDelivery_C'RcvDlvRsp (_,_)) = "35"

| be2int (Bind.PayDelivery_P'RcvPayReq (_,_)) = "8"
| be2int (Bind.PayDelivery_P'RcvDlvReq (_,_)) = "18"
| be2int (Bind.PayDelivery_P'SndPayExch (_,_)) = "9"
| be2int (Bind.PayDelivery_P'RcvPayExch (_,_)) = "12"
| be2int (Bind.PayDelivery_P'HoldPayRsp (_,_)) = "13"
| be2int (Bind.PayDelivery_P'SndPayDlvRsp (_,_)) = "20"

| be2int (Bind.Delivery_C'SndDlvReq (_,_)) = "17"
| be2int (Bind.Delivery_C'RcvDlvRsp (_,_)) = "22"

| be2int (Bind.Delivery_D'RcvDlvReq (_,_)) = "19"
| be2int (Bind.Delivery_D'SndDlvRsp (_,_)) = "21"

| be2int (Bind.Cancellation_C'SndCnl (_,_)) = "55"
| be2int (Bind.Cancellation_C'SndCnl_special (_,_)) = "55"
| be2int (Bind.Cancellation_C'RcvCnl
          (_,{s=READY,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=_,id=_})) = "0"
| be2int (Bind.Cancellation_C'RcvCnl (_,_)) = "56"

| be2int (Bind.Cancellation_NC'SndCnl
          (_,{s=_,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=Merchant,q=_})) = "57"
| be2int (Bind.Cancellation_NC'SndCnl
          (_,{s=_,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=PHandler,q=_})) = "59"
| be2int (Bind.Cancellation_NC'SndCnl
```

```
          (_,{s=_,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=DHandler,q=_})) = "61"
  | be2int (Bind.Cancellation_NC'LocalCnl (_,_)) = "58"

  | be2int (Bind.Cancellation_NC'RcvCnl_P_D  (* Newly Added *)
          (_,{s=READY,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=_,id=_})) = "0"

  | be2int (Bind.Cancellation_NC'RcvCnl_P_D  (* Newly Added *)
          (_,{s=_,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=PHandler,id=_})) = "60"

  | be2int (Bind.Cancellation_NC'RcvCnl_P_D  (* Newly Added *)
          (_,{s=_,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=DHandler,id=_})) = "62"

  | be2int (Bind.Cancellation_NC'RcvCnl_M (_,_)) = "58"  (* Newly Added *)

  | be2int (Bind.ErrHandling_C'Retransmit_RstTimer (_,_)) = "0"
  | be2int (Bind.ErrHandling_C'Stop_1 (_,_)) = "56"
  | be2int (Bind.ErrHandling_C'Discard_Clear (_,_)) = "0"
  | be2int (Bind.ErrHandling_C'Discard (_,_)) = "0"
  | be2int (Bind.ErrHandling_C'SntMsgRptTrntErr (_,_)) = "0"
  | be2int (Bind.ErrHandling_C'Stop_4 (_,_)) = "56"
  | be2int (Bind.ErrHandling_C'RstTimer (_,_)) = "0"
  | be2int (Bind.ErrHandling_C'SndMsgRptHrdErr_Stop
          (_,{s=READY,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,
              role=_,sq=_,m=_,id=_,rq=_})) = "0"
  | be2int (Bind.ErrHandling_C'SndMsgRptHrdErr_Stop (_,_)) = "56"

  | be2int (Bind.ErrHandling_NC'Retransmit_RstTimer (_,_)) = "0"
  | be2int (Bind.ErrHandling_NC'Stop_1
          (_,{trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=Merchant,q=_})) = "58"
  | be2int (Bind.ErrHandling_NC'Stop_1
          (_,{trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=PHandler,q=_})) = "60"
  | be2int (Bind.ErrHandling_NC'Stop_1
          (_,{trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=DHandler,q=_})) = "62"
  | be2int (Bind.ErrHandling_C'Discard_Clear_M (_,_)) = "0"
  | be2int (Bind.ErrHandling_C'Discard_Clear_P (_,_)) = "0"
  | be2int (Bind.ErrHandling_C'Discard_Clear_D (_,_)) = "0"
  | be2int (Bind.ErrHandling_C'Discard (_,_)) = "0"
  | be2int (Bind.ErrHandling_C'SntMsgRptTrntErr (_,_)) = "0"

  | be2int (Bind.ErrHandling_C'Stop_4_M (_,_)) = "58"  (* Newly Added *)

  | be2int (Bind.ErrHandling_C'Stop_4_P_D  (* Newly Added *)
          (_,{s=_,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=PHandler,id=_})) = "60"

  | be2int (Bind.ErrHandling_C'Stop_4_P_D  (* Newly Added *)
          (_,{s=_,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,role=DHandler,id=_})) = "62"

  | be2int (Bind.ErrHandling_C'RstTimer (_,_)) = "0"
  | be2int (Bind.ErrHandling_C'SndMsgRptHrdErr_Stop
          (_,{s=READY,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,
              role=_,sq=_,m=_,id=_,rq=_})) = "0"
  | be2int (Bind.ErrHandling_C'SndMsgRptHrdErr_Stop
          (_,{s=_,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,
              role=Merchant,sq=_,m=_,id=_,rq=_})) = "58"
  | be2int (Bind.ErrHandling_C'SndMsgRptHrdErr_Stop
          (_,{s=_,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,
              role=PHandler,sq=_,m=_,id=_,rq=_})) = "60"
  | be2int (Bind.ErrHandling_C'SndMsgRptHrdErr_Stop
          (_,{s=_,trtype=_,exch=_,sid=_,rc=_,rid=_,sm=_,
              role=DHandler,sq=_,m=_,id=_,rq=_})) = "62"

  | be2int (_) = "ERROR";

fun arc2int(a) = be2int(ArcToBE(a));

fun findhalts n = DeadMarking(n);
```

H.3.2 Minimal FSA for the Revised IOTP Languages

FSM reports for the minimal FSA for the protocol language of the Authentication transaction in the revised IOTP:

```
class                          basic
transducer                     n
# of states                    13
# of arcs                      36
initial state                  0
# of final states              2
# of eps                       0
# of accessible states         13
# of coaccessible states       13
# of connected states          13
# strongly conn components     13
```

FSM report for the minimal FSA for the protocol language of the Deposit transaction in the revised IOTP:

```
class                          basic
transducer                     n
# of states                    38
# of arcs                      119
initial state                  0
# of final states              3
# of eps                       0
# of accessible states         38
# of coaccessible states       38
# of connected states          38
# strongly conn components     38
```

FSM report for the minimal FSA for the protocol language of the Purchase transaction in the revised IOTP:

```
class                          basic
transducer                     n
# of states                    54
# of arcs                      170
initial state                  0
# of final states              4
# of eps                       0
# of accessible states         54
# of coaccessible states       54
# of connected states          54
# strongly conn components     54
```

FSM report for the minimal FSA for the protocol language of the Value Exchange transaction in the revised IOTP:

```
class                          basic
transducer                     n
# of states                    42
# of arcs                      130
initial state                  0
# of final states              3
# of eps                       0
# of accessible states         42
# of coaccessible states       42
# of connected states          42
# strongly conn components     42
```

Appendix I

Publications

C. Ouyang, L. M. Kristensen and J. Billington. Towards Modelling and Analysis of Internet Open Trading Protocol Transactions using Coloured Petri Nets. In *Proceedings of 11th Annual International Symposium of the International Council of Systems Engineering (INCOSE2001)*, Paper 6.7.3 on CD-ROM (8 pages), Melbourne, Australia, 1-5 July 2001.

Abstract. Electronic Commerce (E-Commerce) is a rapidly growing area with new companies entering the market place and existing companies moving their commercial operations onto the Internet. The Internet Open Trading Protocol (IOTP) is a standard being developed by the Internet Engineering Task Force (IETF) aiming at providing an interoperable framework for E-Commerce over the Internet. In this paper we use Coloured Petri Nets (CP-nets or CPNs) to construct an executable model of the IOTP Deposit Transaction. Due to the formal semantics of CP-nets, this CPN model constitutes an unambiguous specification of the Deposit Transaction and allows us to conduct a formal analysis of the Deposit Transaction. This initial analysis has already revealed minor deficiencies in the Deposit Transaction procedures demonstrating the benefit of applying formal methods for the specification and development of complex communication protocols

C. Ouyang, L. M. Kristensen and J. Billington. An Improved Architectural Specification of the Internet Open Trading Protocol. In *Proceedings of 3rd Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'01)*, pages 115-133, Aarhus, Denmark, 29-31 August 2001. Department of Computer Science, University of Aarhus. DAIMI PB-554, ISSN 0105-8517.

Abstract. The Internet Open Trading Protocol (IOTP) is an international standard (RFC 2801) currently being developed by the Internet Engineering Task Force. IOTP aims at providing an interoperable framework for electronic commerce (e-commerce) over the Internet. IOTP is expected to evolve into one of the central building blocks for the developing of next generation e-commerce on the Internet. We apply Coloured Petri Nets and Design/CPN for modelling and analysing IOTP, focusing on its protocol architecture. The contribution of this paper is the construction of a CPN specification of RFC 2801. Based on the constructed CPN model, we propose a complete and simplified architecture of IOTP compared with the partial architectural specification given in the RFC. We apply simulation, message sequence charts, and state space analysis to validate the CPN models of IOTP exchanges and transactions, and to validate that the suggested protocol architecture conforms to the specification of IOTP given in the RFC.

C. Ouyang, L. M. Kristensen and J. Billington. A Formal Service specification for the Internet Open Trading Protocol. In *Proceedings of 23rd International Conference on Application and Theory of Petri Nets (ICATPN 2002)*, pages 352-373, Adelaide, Australia, 24-28 June 2002, Volume 2360 of Lecture Notes in Computer Science, Springer-Verlag, ISBN 3-540-43787-8.

Abstract. This paper presents our service specification for the Internet Open Trading Protocol (IOTP) developed using Coloured Petri Nets. To handle IOTP's complexity, we apply a protocol engineering methodology based on Open Systems Interconnection (OSI) principles consisting of five iterative steps: the definition of service primitives and parameters; the creation of an automaton specifying the local service language for each of the four trading roles of IOTP; the development of a CPN model synthesizing the local automata into a specification of the global service capturing the correlations between the service primitives at the distributed trading roles; the generation of the occurrence graph representing the global service language; and lastly a new step, language comparison to ensure the consistency between the specifications of the local service language and the global service language. The outcome is a proposed formal service specification for IOTP.

C. Ouyang, L. M. Kristensen and J. Billington. A Formal and Executable Specification of the Internet Open Trading Protocol. In *Proceedings of 3rd International Conference on E-Commerce and Web Technologies (EC-Web 2002)*, pages 377-387, Aix-en-Provence, France, 2-6 September 2002, Volume 2455 of Lecture Notes in Computer Science, Springer-Verlag, ISBN 3-540-44137-9.

Abstract. The Internet Open Trading Protocol (IOTP) is being developed by the Internet Engineering Task Force for electronic commerce (e-commerce) over the Internet. The core of IOTP is a set of trading transactions that reflects the most common trading activities in the real world. We apply the formal method of Coloured Petri Nets (CP-nets) to construct an abstract executable specification of IOTP's trading transaction protocols. The formal semantics of CP-nets allows us to investigate the termination properties of the transactions using state space techniques. This investigation has revealed deficiencies in the termination of IOTP trading transactions, demonstrating the benefit of applying formal methods to the specification and verification of e-commerce protocols.

C. Ouyang and J. Billington. On Verifying the Internet Open Trading Protocol. In *Proceedings of 4th International Conference on E-Commerce and Web Technologies (EC-Web 2003)*, pages 292-302, Prague, Czech Republic, 1-5 September 2002, Volume 2738 of Lecture Notes in Computer Science, Springer-Verlag, ISBN 3-540-40808-8.

Abstract. The Internet Open Trading Protocol (IOTP) is designed to support a set of electronic transactions that capture common trading activities. The protocol provides reliable trading transaction services that are payment-system independent. To verify IOTP, we use a protocol verification methodology that compares IOTP's service and protocol languages. The service language is generated from the service model specifying only the external behaviour of IOTP, whereas the protocol language is obtained from the protocol model by hiding internal operations of IOTP. Comparing these two languages indicates whether IOTP formally conforms

to its service. The initial verification results show that IOTP transactions that are successful, error-free and independent of each other, implement a subset of the service language. We conclude that this subset is a valid implementation of the service and is due to the way IOTP combines messages in some circumstances.

C. Ouyang and J. Billington. An Improved Formal Specification of the Internet Open Trading Protocol. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC 2004)*, pages 779-783, Nicosia, Cyprus, 14-17 March 2004, ACM Press, ISBN 1-58113-812-1.

Abstract. The Internet Open Trading Protocol (IOTP) is an electronic commerce protocol being developed by the Internet Engineering Task Force as Request For Comments (RFC) 2801. The core of IOTP is a set of electronic transactions that reflect the most common trading activities in the real world, such as purchasing goods or depositing funds. We use Coloured Petri Nets (CPNs) to construct an improved specification of IOTP by refining the structure of our previous specifications to better reflect IOTP's definition and to improve modularity and re-use. We also include IOTP's procedures for error handling and arbitrary transaction cancellation, which are essential for a complete specification. The improved specification conforms to the narrative description of IOTP in RFC 2801 and allows us to conduct a thorough analysis of the protocol.

C. Ouyang and J. Billington. Formal Analysis of the Internet Open Trading Protocol. Accepted for publication in *Proceedings of the 1st International Workshop on Theory Building and Formal Methods in Electronic/Mobile Commerce (TheFormEMC)*, Toledo, Spain, 1-2 October 2004. In series of Lecture Notes in Computer Science, Springer-Verlag, in press.

Abstract. The Internet Open Trading Protocol (IOTP) is an electronic commerce protocol developed by the Internet Engineering Task Force (IETF) to support common trading activities over the Internet. The core of IOTP is a set of financial transactions and therefore it is vitally important that the protocol operates correctly. An informal specification of IOTP is published as Request For Comments (RFC) 2801. We have applied the formal method of Coloured Petri Nets (CPNs) to obtain a formal specification of IOTP. Based on the IOTP CPN specification, this paper presents a detailed investigation of a set of behavioural properties of IOTP using state space techniques. The analysis reveals deficiencies in the termination of IOTP transactions, demonstrating the benefit of applying formal methods to the specification and verification of e-commerce protocols.